Intel® Math Kernel Library

Reference Manual

March 2007

Disclaimer and Legal Information

Document Number: 630813-025US

World Wide Web: http://developer.intel.com

Version	Version Information	Date
-001	Original Issue.	11/94
-002	Added functions crotg, zrotg. Documented versions of functions ?her2k, ?symm, ?syrk, and ?syr2k not previously described. Pagination revised.	5/95
-003	Changed the title; former title: "Intel BLAS Library for the Pentium® Processor Reference Manual." Added functions ?rotm, ?rotmg and updated Appendix C.	1/96
-004	Documents Intel® Math Kernel library (Intel® MKL) release 2.0 with the parallelism capability. Information on parallelism has been added in Chapter 1 and in section "BLAS Level 3 Routines" in Chapter 2.	11/96
-005	Two-dimensional FFTs have been added. C interface has been added to both one- and two-dimensional FFTs.	8/97
-006	Documents Intel Math Kernel Library release 2.1. Sparse BLAS section has been added in Chapter 2.	1/98
-007	Documents Intel Math Kernel Library release 3.0. Descriptions of LAPACK routines (Chapters 4 and 5) and CBLAS interface (Appendix C) have been added. Quick Reference has been excluded from the manual; MKL 3.0 Quick Reference is now available in HTML format.	1/99
-008	Documents Intel Math Kernel Library release 3.2. Description of FFT routines have been revised. In Chapters 4 and 5 NAG names for LAPACK routines have been excluded.	6/99
-009	New LAPACK routines for eigenvalue problems have been added in chapter 5.	11/99
-010	Documents Intel Math Kernel Library release 4.0. Chapter 6 describing the VML functions has been added.	06/00
-011	Documents Intel Math Kernel Library release 5.1. LAPACK section has been extended to include the full list of computational and driver routines.	04/01
-6001	Documents Intel Math Kernel Library release 6.0 beta. New DFT interface and Vector Statistical Library functions have been added.	07/02
-6002	Documents Intel Math Kernel Library 6.0 beta update. DFT functions description has been updated. CBLAS interface description was extended.	12/02
-6003	Documents Intel Math Kernel Library 6.0 gold. DFT functions have been updated. Auxiliary LAPACK routines' descriptions were added to the manual.	03/03
-6004	Documents Intel Math Kernel Library release 6.1.	07/03
-6005	Documents Intel Math Kernel Library release 7.0 beta. Includes ScaLAPACK and sparse solver descriptions.	11/03
-017	Documents Intel MKL and Intel® Cluster MKL release 7.0 gold. Auxiliary ScaLAPACK and alternative sparse solver interface were added.	04/04

Version	Version Information	Date
-018	Documents Intel MKL and Intel® Cluster MKL release 8.0 beta. Sparse BLAS and DFTI sections were extended. New functionality was added: Sparse BLAS, Cluster DFTI, iterative sparse solver, multiple-precision arithmetic, interval linear solver, and convolution/correlation. Fortran95 interface to LAPACK functions was added.	03/05
-019	Documents Intel MKL and Intel® Cluster MKL release 8.0 gold. Fortran95 interface to BLAS and Sparse BLAS functions has been added.	08/05
-020	Documents Intel MKL and Intel Cluster MKL release 8.0.2. PARDISO functionality description has been extended with indefinite symmetric matrices pivoting.	03/06
-021	Documents Intel MKL and Intel Cluster MKL release 8.1 gold. Chapter 13 on Trigonometric Transform functions has been added. Information on specific features of Fortran-95 implementation for LAPACK routines has been reflected in a new Appendix E and the relevant subsection of Chapter 3.	03/06
-022	Documents Intel MKL and Intel Cluster MKL release 9.0 beta. Statistical Functions, Fourier Transform Functions (Cluster DFT) descriptions have been updated. Description of new VML functions, RCI Sparse Solvers, and Poisson solver have been added. Chapter 13 has been renamed to PDE Support. Code examples have been expanded.	05/06
-023	Documents Intel MKL and Intel Cluster MKL release 9.0 gold. Complex Interval Solvers have been added. Description of the old deprecated FFT functions have been removed.	09/06
-024	Documents Intel MKL and Intel Cluster MKL release 9.1 beta. Optimization Solvers Routines and Support Functions chapters have been added. Chapters on Sparse Solvers and Partial Differential Equations Support have been extended. LAPACK chapters have been partially updated to reflect LAPACK 3.1 version.	01/07
-025	Documents Intel MKL and Intel Cluster MKL release 9.1 gold. BLACS chapter has been added. Chapters on BLAS and FFT have been extended. LAPACK chapters have been additionally updated to reflect LAPACK 3.1 version. Description of BSR format has been added to Appendix A. New FFT examples have been added to Appendix C.	03/07

Contents

Chapter 5: LAPACK Auxiliary and Utility Routines

Chapter 10: Statistical Functions

Chapter 13: Partial Differential Equations Support

Chapter 14: Optimization Solvers Routines

Chapter 15: Support Functions

Chapter 16: BLACS Routines

Appendix A: Linear Solvers Basics

Appendix B: Routine and Function Arguments

Appendix C: Code Examples

1 Overview

The Intel® Math Kernel Library (Intel® MKL) provides Fortran routines and functions that perform a wide variety of operations on vectors and matrices including sparse matrices and interval matrices. The library also includes discrete Fourier transform routines, as well as vector mathematical and vector statistical functions with Fortran and C interfaces.

The version of the library named Intel® Cluster MKL is a superset of Intel MKL and includes also ScaLAPACK software and Cluster DFT software for solving respective computational problems on distributed-memory parallel computers.

The Intel MKL enhances performance of the application programs that use it because the library has been optimized for latest generations of Intel® processors.

This chapter introduces the Intel Math Kernel Library and provides information about the organization of this manual.

About This Software

The Intel® Math Kernel Library includes the following groups of routines:

- Basic Linear Algebra Subprograms (BLAS):
  - vector operations
  - matrix-vector operations
  - matrix-matrix operations
- Sparse BLAS Level 1, 2, and 3 (basic operations on sparse vectors and matrices)
- LAPACK routines for solving systems of linear equations
- LAPACK routines for solving least-squares problems, eigenvalue and singular value problems, and Sylvester's equations
- Auxiliary and utility LAPACK routines
- ScaLAPACK computational, driver and auxiliary routines (for Intel Cluster MKL only)
- Direct and Iterative Sparse Solver routines
- Vector Mathematical Library (VML) functions for computing core mathematical functions on vector arguments (with Fortran and C interfaces)
- Vector Statistical Library (VSL): functions for generating vectors of pseudorandom numbers with different types of statistical distributions and for performing convolution and correlation computations

- General Discrete Fourier Transform Functions (DFT), providing fast computation of DFT via the Fast Fourier Transform (FFT) algorithms and having Fortran and C interfaces
- Cluster DFT functions (for Intel Cluster MKL only)
- Interval Solver routines for solving systems of interval linear equations
- Tools for solving partial differential equations - trigonometric transform routines and Poisson solver
- Optimization Solver routines for solving nonlinear least squares problems through the Trust-Region (TR) algorithms and computing Jacobi matrix by central differences
- GMP arithmetic functions.

For specific issues on using the library, please refer to the MKL Release Notes.

Technical Support

Intel MKL provides a product web site that offers timely and comprehensive product information, including product features, white papers, and technical articles. For the latest information, check: http://developer.intel.com/software/products/

Intel also provides a support web site that contains a rich repository of self help information, including getting started tips, known product issues, product errata, license information, user forums, and more (visit http://support.intel.com/support/performancetools/libraries/mkl).

Registering your product entitles you to one year of technical support and product updates through Intel® Premier Support. Intel Premier Support is an interactive issue management and communication web site providing these services:

- Submit issues and review their status.
- Download product updates anytime of the day.

To register your product, contact Intel, or seek product support, please visit: http://developer.intel.com/software/products/support.

BLAS Routines

BLAS routines and functions are divided into the following groups according to the operations they perform:

- BLAS Level 1 Routines and Functions perform operations of both addition and reduction on vectors of data. Typical operations include scaling and dot products.
- BLAS Level 2 Routines perform matrix-vector operations, such as matrix-vector multiplication, rank-1 and rank-2 matrix updates, and solution of triangular systems.

- BLAS Level 3 Routines perform matrix-matrix operations, such as matrix-matrix multiplication, rank-k update, and solution of triangular systems.

Starting from release 8.0, Intel® MKL also supports Fortran-95 interface to BLAS routines.

Sparse BLAS Routines

Sparse BLAS Level 1 Routines and Functions and Sparse BLAS Level 2 Routines and Level 3 routines and functions operate on sparse vectors and matrices. These routines perform vector operations similar to BLAS Level 1, 2, and 3 routines. Sparse BLAS routines take advantage of vector and matrix sparsity: they allow you to store only non-zero elements of vectors and matrices. Intel MKL also supports Fortran-95 interface to Sparse BLAS routines.

LAPACK Routines

The Intel® Math Kernel Library fully supports LAPACK 3.0 set of computational, driver, auxiliary and utility routines and partially supports LAPACK 3.1 routines - one computational routine (`?stemr`) and a number of auxiliary and utility routines.

The original versions of LAPACK from which that part of Intel MKL was derived can be obtained from http://www.netlib.org/lapack/index.html. The authors of LAPACK are E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen.

The LAPACK routines can be divided into the following groups according to the operations they perform:

- Routines for solving systems of linear equations, factoring and inverting matrices, and estimating condition numbers (see Chapter 3).
- Routines for solving least-squares problems, eigenvalue and singular value problems, and Sylvester's equations (see Chapter 4).
- Auxiliary and utility routines used to perform certain subtasks, common low-level computation or related tasks (see Chapter 5).

Starting from release 8.0, Intel MKL also supports Fortran-95 interface to LAPACK computational and driver routines. This interface provides an opportunity for simplified calls of LAPACK routines with fewer required arguments.

ScaLAPACK Routines

ScaLAPACK package (included with Intel® Cluster MKL only, see Chapter 6 and Chapter 7) runs on distributed-memory architectures and includes routines for solving systems of linear equations, solving linear least-squares problems, eigenvalue and singular value problems, as well as performing a number of related computational tasks.

The original versions of ScaLAPACK from which that part of Intel Cluster MKL was derived can be obtained from http://www.netlib.org/scalapack/index.html. The authors of ScaLAPACK are L. Blackford, J. Choi, A.Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K.Stanley, D. Walker, and R. Whaley.

Intel Cluster MKL version of ScaLAPACK is optimized for Intel® processors and uses MPICH version of MPI as well as Imtel MPI.

Sparse Solver Routines

Direct sparse solver routines in Intel MKL (see Chapter 8) solve symmetric and symmetrically-structured sparse matrices with real or complex coefficients. For symmetric matrices, these Intel MKL subroutines can solve both positive definite and indefinite systems. Intel MKL includes the PARDISO* sparse solver interface as well as an alternative set of user callable direct sparse solver routines.

If you use the sparse solver PARDISO* from Intel MKL, please cite:

O.Schenk and K.Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. J. of Future Generation Computer Systems, 20(3):475- 487, 2004.

Intel MKL provides also an iterative sparse solver (see Chapter 8) that uses sparse BLAS level 2 and 3 routines and works with different sparse data formats.

VML Functions

Vector Mathematical Library (VML) functions (see Chapter 9) include a set of highly optimized implementations of certain computationally expensive core mathematical functions (power, trigonometric, exponential, hyperbolic etc.) that operate on vectors of real and complex numbers.

Application programs that might significantly improve performance with VML include nonlinear programming software, integrals computation, and many others. VML provides interfaces both for FORTRAN and C languages.

Statistical Functions

Vector Statistical Library (VSL) contains two sets of functions (see Chapter 10):

- The first set includes a collection of pseudo- and quasi-random number generator subroutines implementing basic continuous and discrete distributions. To provide best performance, VSL subroutines use calls to highly optimized Basic Random Number Generators and a library of vector mathematical functions.
- The second set includes a collection of routines that implement a wide variety of convolution and correlation operations.

Fourier Transform Functions

The Intel® MKL multidimensional Discrete Fourier Transform functions with mixed radix support (see Chapter 11) provide uniformity of DFT computation and combine functionality with ease of use. Both Fortran and C interface specification are given. There is also a cluster version of DFT functions, which runs on distributed-memory architectures and is provided with Intel Cluster MKL package.

DFT functions provide fast computation via the Fast Fourier Transform (FFT) algorithms not only for lengths that are powers of 2 but also for 3, 5, 7, 11, and other radices.

Interval Solver Routines

Interval Solver routines included into Intel® MKL (see Chapter 12) can be used to solve interval systems of linear equations and related problems.

Partial Differential Equations Support

Intel® MKL provides tools for solving Partial Differential Equations (PDE) (see Chapter 13). These tools are Trigonometric Transform interface routines and Poisson Library.

The Trigonometric Transform routines may be helpful to users who implement their own solvers similar to the solver that the Poisson Library provides. The users can improve performance of their solvers by using fast sine, cosine, and staggered cosine transforms implemented in the Trigonometric Transform interface.

Poisson Library is designed for fast solving of simple Helmholtz, Poisson, and Laplace problems. The Trigonometric Transform interface, which underlies the solver, is based on Intel MKL DFT interface (refer to Chapter 11), optimized for Intel® processors.

Optimization Solvers Routines

Intel® MKL provides Optimization Solvers routines (see Chapter 14) that can be used to solve nonlinear least squares problems with or without linear (bound) constraints through the Trust-Region (TR) algorithms and compute Jacobi matrix by central differences.

See also Appendix A Optimization Solvers Basics for description of the basic notions of optimization solvers, nonlinear least square problems, and the TR algorithm.

Support Functions

Intel® MKL support functions (see Chapter 15) are used to support the operation of Intel MKL software and provide basic information on the library and library operation, such as the current library version, timing, setting and measuring of CPU frequency, error handling, and memory allocation.

BLACS Routines

Intel® Math Kernel Library implements routines from the BLACS (Basic Linear Algebra Communication Subprograms) package (see Chapter 16) that are used to support a linear algebra oriented message passing interface that may be implemented efficiently and uniformly across a large range of distributed memory platforms.

The original versions of BLACS from which that part of Intel MKL was derived can be obtained from http://www.netlib.org/blacs/index.html. The authors of BLACS are Jack Dongarra and R. Clint Whaley.

GMP Arithmetic Functions

Intel® MKL implementation of GMP arithmetic functions includes arbitrary precision arithmetic operations on integer numbers. The interfaces of such functions fully match the GNU Multiple Precision (GMP) Arithmetic Library.

Performance Enhancements

The Intel® Math Kernel Library has been optimized by exploiting both processor and system features and capabilities. Special care has been given to those routines that most profit from cache-management techniques. These especially include matrix-matrix operation routines such as `dgemm()`.

In addition, code optimization techniques have been applied to minimize dependencies of scheduling integer and floating-point units on the results within the processor.

The major optimization techniques used throughout the library include:

- Loop unrolling to minimize loop management costs.
- Blocking of data to improve data reuse opportunities.
- Copying to reduce chances of data eviction from cache.

- Data prefetching to help hide memory latency.
- Multiple simultaneous operations (for example, dot products in `dgemm`) to eliminate stalls due to arithmetic unit pipelines.
- Use of hardware features such as the SIMD arithmetic units, where appropriate.

These are techniques from which the arithmetic code benefits the most.

Parallelism

In addition to the performance enhancements discussed above, the Intel® MKL offers performance gains through parallelism provided by the symmetric multiprocessing performance (SMP) feature. You can obtain improvements from SMP in the following ways:

- One way is based on user-managed threads in the program and further distribution of the operations over the threads based on data decomposition, domain decomposition, control decomposition, or some other parallelizing technique. Each thread can use any of the Intel MKL functions because the library has been designed to be thread-safe.
- Another method is to use the FFT and BLAS level 3 routines. They have been parallelized and require no alterations of your application to gain the performance enhancements of multiprocessing. Performance using multiple processors on the level 3 BLAS shows excellent scaling. Since the threads are called and managed within the library, the application does not need to be recompiled thread-safe (see also Fortran-95 Interface Conventions in Chapter 2).
- Yet another method is to use `tuned LAPACK routines`. Currently these include the single- and double precision flavors of routines for `QR` factorization of general matrices, triangular factorization of general and symmetric positive-definite matrices, solving systems of equations with such matrices, as well as solving symmetric eigenvalue problems.

For instructions on setting the number of available processors for the BLAS level 3 and LAPACK routines, see the Intel® MKL User's Guide.

Platforms Supported

The Intel® Math Kernel Library includes Fortran routines and functions optimized for Intel® processor-based computers running operating systems that support multiprocessing. In addition to the Fortran interface, the Intel MKL includes a C-language interface for the Discrete Fourier transform functions, as well as for the Vector Mathematical Library and Vector Statistical Library functions. For hardware and software requirements to use Intel MKL, see MKL Release Notes.

About This Manual

This manual describes the routines and functions of the Intel® MKL and Intel® Cluster MKL. Each reference section describes a routine group typically consisting of routines used with four basic data types: single-precision real, double-precision real, single-precision complex, and double-precision complex.

Each routine group is introduced by its name, a short description of its purpose, and the calling sequence, or syntax, for each type of data with which each routine of the group is used. The following sections are also included:

Description	Describes the operation performed by routines of the group based on one or more equations. The data types of the arguments are defined in general terms for the group.
Input Parameters	Defines the data type for each parameter on entry, for example: *a* `REAL` for `saxpy` `DOUBLE PRECISION` for `daxpy`
Output Parameters	Lists resultant parameters on exit.

Audience for This Manual

The manual addresses programmers proficient in computational mathematics and assumes a working knowledge of the principles and vocabulary of linear algebra, mathematical statistics, and Fourier transforms.

Manual Organization

The manual contains the following chapters and appendixes:

Chapter 1	Overview. Introduces the Intel Math Kernel Library software, provides information on manual organization, and explains notational conventions.
Chapter 2	BLAS and Sparse BLAS Routines. Provides descriptions of BLAS and Sparse BLAS functions and routines.
Chapter 3	LAPACK Routines: Linear Equations. Provides descriptions of LAPACK routines for solving systems of linear equations and performing a number of related computational tasks: triangular factorization, matrix inversion, estimating the condition number of matrices.

Chapter 4	LAPACK Routines: Least Squares and Eigenvalue Problems. Provides descriptions of LAPACK routines for solving least-squares problems, standard and generalized eigenvalue problems, singular value problems, and Sylvester's equations.
Chapter 5	LAPACK Auxiliary and Utility Routines. Describes auxiliary and utility LAPACK routines that perform certain subtasks or common low-level computation.
Chapter 6	ScaLAPACK Routines. Describes ScaLAPACK computational and driver routines (software included with Intel Cluster MKL only).
Chapter 7	ScaLAPACK Auxiliary and Utility Routines. Describes ScaLAPACK auxiliary routines (software included with Intel Cluster MKL only).
Chapter 8	Sparse Solver Routines. Describes direct sparse solver routines that solve symmetric and symmetrically-structured sparse matrices. Also describes the iterative sparse solver routines.
Chapter 9	Vector Mathematical Functions. Provides descriptions of VML functions for computing elementary mathematical functions on vector arguments.
Chapter 10	Statistical Functions. Provides descriptions of VSL functions for generating vectors of pseudorandom numbers and for performing convolution and correlation operations.
Chapter 11	Fourier Transform Functions. Describes multidimensional functions for computing the Discrete Fourier Transform and cluster DFT functions (software included with Intel Cluster MKL only).
Chapter 12	Interval Linear Solvers. Describes routines that can be used to solve interval systems of linear equations and related problems.
Chapter 13	Partial Differential Equations Support. Describes Trigonometric Transform interface routines and Poisson Library implemented for solving Partial Differential Equations (PDE).
Chapter 14	Optimization Solvers Routines. Describes routines for solving optimization problems. These routines solve nonlinear least squares problems through the Trust-Region (TR) algorithms and compute Jacobi matrix by central differences.
Chapter 15	Support Functions. Describes functions that are used to support the operation of Intel MKL software, such as status information functions, timing and error handling functions.
Chapter 16	BLACS Routines. Describes Basic Linear Algebra Communication Subprograms that are used to support a linear algebra oriented message passing interface.
Appendix A	Linear Solvers Basics. Briefly describes the basic definitions and approaches used in linear algebra for solving systems of linear equations. Describes sparse data storage formats, as well as basic concepts of interval arithmetic.

Appendix B	Routine and Function Arguments. Describes the major arguments of the BLAS routines and VML functions: vector and matrix arguments.
Appendix C	Code Examples. Provides code examples of calling various Intel MKL functions and routines (BLAS, PARDISO, Direct and Iterative Sparse Solver, DFT, Cluster DFT, Intreval Linear Solvers, Trigonometric Transforms, and Poisson Library).
Appendix D	CBLAS Interface to the BLAS. Provides the C interface to the BLAS.
Appendix E	Specific Features of Fortran-95 Interfaces for LAPACK Routines. Provides the features of Intel MKL Fortran-95 interfaces for LAPACK routines in comparison with Netlib implementation.
Appendix F	Optimization Solvers Basics. Briefly describes the basic notions of optimization solvers, nonlinear least square problem, and Trust Region algorithm.

The manual also includes a Bibliography, Glossary and an Index.

Notational Conventions

This manual uses the following notational conventions:

- Routine name shorthand (`?ungqr` instead of `cungqr`/`zungqr`).
- Font conventions used for distinction between the text and the code.

Routine Name Shorthand

For shorthand, character codes are represented by a question mark "`?`" names of routine groups. The question mark is used to indicate any or all possible varieties of a function; for example:

`?swap`	Refers to all four data types of the vector-vector `?swap` routine: `sswap`, `dswap`, `cswap`, and `zswap`.

Font Conventions

The following font conventions are used:

`UPPERCASE COURIER`	Data type used in the discussion of input and output parameters for Fortran interface. For example, `CHARACTER*1`.
`lowercase courier`	Code examples: `a(k+i,j) = matrix(i,j)` and data types for C interface, for example, `const float*`

`lowercase courier mixed with UpperCase courier`	Function names for C interface, for example, `vmlSetMode`
`lowercase courier italic`	Variables in arguments and parameters discussion. For example, *`incx`*.
`*`	Used as a multiplication symbol in code examples and equations and where required by the Fortran syntax.

BLAS and Sparse BLAS Routines 2

This chapter contains descriptions of the BLAS and Sparse BLAS routines of the Intel® Math Kernel Library. The routine descriptions are arranged in several sections according to the BLAS level of operation:

- BLAS Level 1 Routines and Functions (vector-vector operations)
- BLAS Level 2 Routines (matrix-vector operations)
- BLAS Level 3 Routines (matrix-matrix operations)
- Sparse BLAS Level 1 Routines and Functions (vector-vector operations).
- Sparse BLAS Level 2 and Level 3 (matrix-vector and matrix-matrix operations)

Each section presents the routine and function group descriptions in alphabetical order by routine or function group name; for example, the `?asum` group, the `?axpy` group. The question mark in the group name corresponds to different character codes indicating the data type (`s`, `d`, `c`, and `z` or their combination); see Routine Naming Conventions.

When BLAS or Sparse BLAS routines encounter an error, they call the error reporting routine `xerbla`. To be able to view error reports, you must include `xerbla` in your code. A copy of the source code for `xerbla` is included in the library.

In BLAS Level 1 groups `i?amax` and `i?amin`, an "`i`" is placed before the character code and corresponds to the index of an element in the vector. These groups are placed in the end of the BLAS Level 1 section.

Routine Naming Conventions

BLAS routine names have the following structure:

<character code> <name> <mod> ( )

The <*character code*> is a character that indicates the data type:

`s`	real, single precision
`c`	complex, single precision
`d`	real, double precision
`z`	complex, double precision

Some routines and functions can have combined character codes, such as `sc` or `dz`.

For example, the function `scasum` uses a complex input array and returns a real value.

The <name> field, in BLAS level 1, indicates the operation type. For example, the BLAS level 1 routines ?dot, ?rot, ?swap compute a vector dot product, vector rotation, and vector swap, respectively.

In BLAS level 2 and 3, <name> reflects the matrix argument type:

ge	general matrix
gb	general band matrix
sy	symmetric matrix
sp	symmetric matrix (packed storage)
sb	symmetric band matrix
he	Hermitian matrix
hp	Hermitian matrix (packed storage)
hb	Hermitian band matrix
tr	triangular matrix
tp	triangular matrix (packed storage)
tb	triangular band matrix.

The <mod> field, if present, provides additional details of the operation. BLAS level 1 names can have the following characters in the <mod> field:

c	conjugated vector
u	unconjugated vector
g	Givens rotation.

BLAS level 2 names can have the following characters in the <mod> field:

mv	matrix-vector product
sv	solving a system of linear equations with matrix-vector operations
r	rank-1 update of a matrix
r2	rank-2 update of a matrix.

BLAS level 3 names can have the following characters in the <mod> field:

mm	matrix-matrix product
sm	solving a system of linear equations with matrix-matrix operations
rk	rank-k update of a matrix
r2k	rank-2k update of a matrix.

The examples below illustrate how to interpret BLAS routine names:

ddot	<d> <dot>: double-precision real vector-vector dot product
cdotc	<c> <dot> <c>: complex vector-vector dot product, conjugated
scasum	<sc> <asum>: sum of magnitudes of vector elements, single precision real output and single precision complex input
cdotu	<c> <dot> <u>: vector-vector dot product, unconjugated, complex
sgemv	<s> <ge> <mv>: matrix-vector product, general matrix, single precision
ztrmm	<z> <tr> <mm>: matrix-matrix product, triangular matrix, double-precision complex.

Sparse BLAS naming conventions are similar to those of BLAS level 1. For more information, see "Naming Conventions".

Fortran-95 Interface Conventions

Fortran-95 interface to BLAS and Sparse BLAS Level 1 routines is implemented through wrappers that call respective Fortran-77 routines. This interface uses such features of Fortran-95 as assumed-shape arrays and optional arguments to provide simplified calls to BLAS and Sparse BLAS Level 1 routines with fewer arguments.

The main conventions that are used in Fortran-95 interface are as follows:

- The names of arguments used in Fortran-95 call are typically the same as for the respective generic (Fortran-77) interface. However, to reduce the number of argument names used in the library, the following identity settings of formal argument names were made:

Generic Argument Name	Fortran-95 Argument Name
ap	*a*

  Note that these name changes of formal arguments have no impact on program semantics and follow the conventions of unification names.

- Input arguments such as array dimensions are not required in Fortran-95 and are skipped from the calling sequence. Array dimensions are reconstructed from the user data that must exactly follow the required array shape.

  Also, an argument can be skipped if its value is completely defined by the presence or absence of another argument in the calling sequence, and the restored value is the only meaningful value for the skipped argument.

- Arguments *incx* and *incy* are skipped. In all cases their values are assumed to be 1. One can obtain the effect of the values of *incx* and *incy* not being equal to 1 by using corresponding Fortran-95 feature: index incrementing may be directly established in actual arguments. Other possibility to obtain this effect is to use Fortran-77 call.
- Some generic arguments are declared as optional in Fortran-95 interface and may or may not be present in the calling sequence. An argument can be declared optional if it satisfies one of the following conditions:

  1. If an input argument can take only a few possible values, it can be declared as optional. The default value of such argument is typically set as the first value in the list and all exceptions to this rule are explicitly stated in the routine description.
  2. If an input argument has a natural default value, it can be declared as optional. The default value of such optional argument is set to its natural default value.

- Optional arguments are given in square brackets in Fortran-95 call syntax.

The concrete rules used for reconstructing the values of omitted optional parameters are specific for each routine and are detailed in the respective "Fortran-95 Notes" subsection given at the end of routine specification section. If this subsection is omitted, the Fortran-95 interface for the given routine does not differ from the corresponding Fortran-77 interface.

Note that this interface is not implemented in the current version of Sparse BLAS Level 2 and Level 3 routines. Fortran-95 interfaces for each these routines is given in the "Interfaces - Fortran-95" subsection at the end of the respective routine specification section.

Matrix Storage Schemes

Matrix arguments of BLAS routines can use the following storage schemes:

- Full storage: a matrix *A* is stored in a two-dimensional array *a*, with the matrix element a^{ij} stored in the array element *a*(*i*,*j*).
- Packed storage scheme allows you to store symmetric, Hermitian, or triangular matrices more compactly: the upper or lower triangle of the matrix is packed by columns in a one-dimensional array.
- Band storage: a band matrix is stored compactly in a two-dimensional array: columns of the matrix are stored in the corresponding columns of the array, and diagonals of the matrix are stored in rows of the array.

For more information on matrix storage schemes, see "Matrix Arguments" in Appendix B.

BLAS Level 1 Routines and Functions

BLAS Level 1 includes routines and functions, which perform vector-vector operations. Table 2-1 lists the BLAS Level 1 routine and function groups and the data types associated with them.

Table 2-1 BLAS Level 1 Routine Groups and Their Data Types

Routine or Function Group	Data Types	Description
?asum	s, d, sc, dz	Sum of vector magnitudes (functions)
?axpy	s, d, c, z	Scalar-vector product (routines)
?copy	s, d, c, z	Copy vector (routines)
?dot	s, d	Dot product (functions)
?sdot	sd, d	Dot product with extended precision (functions)
?dotc	c, z	Dot product conjugated (functions)
?dotu	c, z	Dot product unconjugated (functions)
?nrm2	s, d, sc, dz	Vector 2-norm (Euclidean norm) a normal or null vector(functions)
?rot	s, d, cs, zd	Plane rotation of points (routines)
?rotg	s, d, c, z	Givens rotation of points (routines)
?rotm	s, d	Modified plane rotation of points
?rotmg	s, d	Givens modified plane rotation of points
?scal	s, d, c, z, cs, zd	Vector scaling (routines)
?swap	s, d, c, z	Vector-vector swap (routines)
i?amax	s, d, c, z	Vector maximum value, absolute largest element of a vector, where *i* is an index to this value in the vector array (functions)

Routine or Function Group	Data Types	Description
i?amin	s, d, c, z	Vector minimum value, absolute smallest element of a vector, where *i* is an index to this value in the vector array (functions)
dcabs1	d	Absolute value of a double complex number *z*.

?asum

Computes the sum of magnitudes of the vector elements.

Syntax

Fortran 77:

```
res = sasum(n, x, incx )
res = scasum(n, x, incx )
res = dasum(n, x, incx )
res = dzasum(n, x, incx )
```

Fortran 95:

```
res = asum(x)
```

Description

The function ?asum function computes the sum of the magnitudes of elements of a real vector, or the sum of magnitudes of the real and imaginary parts of elements of a complex vector:

res = |*Rex* (*1*)| + |*Imx* (1)| + |*Rex*(2) | + |*Imx* (2)|+ ... + |*Rex* (*n*)| + |*Imx* (*n*)|

where *x* is a vector of order *n*.

Input Parameters

n	INTEGER. Specifies the order of vector *x*.
x	REAL for sasum DOUBLE PRECISION for dasum

COMPLEX for scasum
DOUBLE COMPLEX for dzasum

Array, DIMENSION at least (1 + (*n*-1)*abs(*incx*)).

incx INTEGER. Specifies the increment for the elements of *x*.

Output Parameters

res REAL for sasum
DOUBLE PRECISION for dasum
REAL for scasum
DOUBLE PRECISION for dzasum
Contains the sum of magnitudes of real and imaginary parts of all elements of the vector.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine asum interface are the following:

x Holds the array of size(*n*).

?axpy

Computes a vector-scalar product and adds the result to a vector.

Syntax

Fortran 77:

```
call saxpy(n, a, x, incx, y, incy)
call daxpy(n, a, x, incx, y, incy)
call caxpy(n, a, x, incx, y, incy)
call zaxpy(n, a, x, incx, y, incy)
```

Fortran 95:

```
call axpy(x, y [,a])
```

Description

The ?axpy routines perform a vector-vector operation defined as

y := *a***x* + *y*

where:

a is a scalar

x and *y* are vectors of order *n*.

Input Parameters

n	INTEGER. Specifies the order of vectors *x* and *y*.
a	REAL for saxpy DOUBLE PRECISION for daxpy COMPLEX for caxpy DOUBLE COMPLEX for zaxpy Specifies the scalar *a*.
x	REAL for saxpy DOUBLE PRECISION for daxpy COMPLEX for caxpy DOUBLE COMPLEX for zaxpy Array, DIMENSION at least (1 + (*n*-1)*abs(*incx*)).
incx	INTEGER. Specifies the increment for the elements of *x*.
y	REAL for saxpy DOUBLE PRECISION for daxpy COMPLEX for caxpy DOUBLE COMPLEX for zaxpy Array, DIMENSION at least (1 + (*n*-1)*abs(*incy*)).
incy	INTEGER. Specifies the increment for the elements of *y*.

Output Parameters

y	Contains the updated vector *y*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `axpy` interface are the following:

x	Holds the array of size(*n*).
y	Holds the array of size (*n*).
a	The default value is 1.

?copy

Copies vector to another vector.

Syntax

Fortran 77:

```
call scopy(n, x, incx, y, incy)
call dcopy(n, x, incx, y, incy)
call ccopy(n, x, incx, y, incy)
call zcopy(n, x, incx, y, incy)
```

Fortran 95:

```
call copy(x, y)
```

Description

The `?copy` routines perform a vector-vector operation defined as

y = *x*,

where *x* and *y* are vectors.

Input Parameters

n	`INTEGER`. Specifies the order of vectors *x* and *y*.

x	`REAL` for `scopy`
	`DOUBLE PRECISION` for `dcopy`
	`COMPLEX` for `ccopy`

	DOUBLE COMPLEX for zcopy Array, DIMENSION at least (1 + (*n*-1)*abs(*incx*)).
incx	INTEGER. Specifies the increment for the elements of *x*.
y	REAL for scopy DOUBLE PRECISION for dcopy COMPLEX for ccopy DOUBLE COMPLEX for zcopy Array, DIMENSION at least (1 + (*n*-1)*abs(*incy*)).
incy	INTEGER. Specifies the increment for the elements of *y*.

Output Parameters

y	Contains a copy of the vector *x* if *n* is positive. Otherwise, parameters are unaltered.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine copy interface are the following:

x	Holds the vector of length(*n*).
y	Holds the vector of length (*n*).

?dot

Computes a vector-vector dot product.

Syntax

Fortran 77:

```
res = sdot(n, x, incx, y, incy)
res = ddot(n, x, incx, y, incy)
```

Fortran 95:

```
res = dot(x, y)
```

Description

The ?dot functions perform a vector-vector reduction operation defined as

$$res = \sum (x * y),$$

where x and y are vectors.

Input Parameters

n	INTEGER. Specifies the order of vectors *x* and *y*.
x	REAL for sdot DOUBLE PRECISION for ddot Array, DIMENSION at least (1+(*n*-1)*abs(*incx*)).
incx	INTEGER. Specifies the increment for the elements of *x*.
y	REAL for sdot DOUBLE PRECISION for ddot Array, DIMENSION at least (1+(*n*-1)*abs(*incy*)).
incy	INTEGER. Specifies the increment for the elements of *y*.

Output Parameters

res	REAL for sdot DOUBLE PRECISION for ddot Contains the result of the dot product of *x* and *y*, if *n* is positive. Otherwise, *res* contains 0.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine dot interface are the following:

x	Holds the vector of length(*n*).
y	Holds the vector of length (*n*).

?sdot

Computes a vector-vector dot product with extended precision.

Syntax

Fortran 77:

```
res = sdsdot(n, sb, sx, incx, s y, incy)
res = dsdot(n, s x, incx, s y, incy)
```

Fortran 95:

```
res = sdot(sx, sy)
res = sdot(sx, sy, sb)
```

Description

The ?sdot functions compute the inner product of two vectors with extended precision. Both functions use extended precision accumulation of the intermediate results, but the function sdsdot outputs the final result in single precision, whereas the function dsdot outputs the double precision result. The function sdsdot also adds scalar value *sb* to the inner product.

Input Parameters

n	INTEGER. Specifies the number of elements in the input vectors *sx* and *sy*.
sb	REAL. Single precision scalar to be added to inner product (for the function sdsdot only).
sx, *sy*	REAL. Arrays, DIMENSION at least (1+(*n* –1)*abs(*incx*)) and (1+(*n*–1)*abs(*incy*)), respectively. Contain the input single precision vectors.
incx	INTEGER. Specifies the increment for the elements of *sx*.
incy	INTEGER. Specifies the increment for the elements of *sy*.

Output Parameters

res	REAL for sdsdot

DOUBLE PRECISION for dsdot
Contains the result of the dot product of *sx* and *sy* (with *sb* added for sdsdot), if *n* is positive. Otherwise, *res* contains *sb* for sdsdot and 0 for dsdot.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine sdot interface are the following:

sx	Holds the vector of length(*n*).
sy	Holds the vector of length(*n*).

NOTE. Note that scalar parameter *sb* is declared as a required parameter in Fortran-95 interface for the function sdot to distinguish between function flavors that output final result in different precision.

?dotc

Computes a dot product of a conjugated vector with another vector.

Syntax

Fortran 77:

```
res = cdotc(n, x, incx, y, incy )
res = zdotc(n, x, incx, y, incy )
```

Fortran 95:

```
res = dotc(x, y)
```

Description

The ?dotC functions perform a vector-vector operation defined as

$$res = \sum (conjg(x) * y))$$

where x and y are n-element vectors.

Input Parameters

n	INTEGER. Specifies the order of vectors x and y.
x	COMPLEX for cdotc DOUBLE COMPLEX for zdotc Array, DIMENSION at least (1 + (*n* –1)*abs(*incx*)).
incx	INTEGER. Specifies the increment for the elements of x.
y	COMPLEX for cdotc DOUBLE COMPLEX for zdotc Array, DIMENSION at least (1 + (*n* –1)*abs(*incy*)).
incy	INTEGER. Specifies the increment for the elements of y.

Output Parameters

res	COMPLEX for cdotc DOUBLE COMPLEX for zdotc Contains the result of the dot product of the conjugated x and unconjugated y, if n is positive. Otherwise, *res* contains 0.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine dotc interface are the following:

x	Holds the vector of length(n).
y	Holds the vector of length (n).

?dotu

Computes a vector-vector dot product.

Syntax

Fortran 77:

```
res = cdotu(n, x, incx, y, incy )
res = zdotu(n, x, incx, y, incy )
```

Fortran 95:

```
res = dotu(x, y)
```

Description

The ?dotu functions perform a vector-vector reduction operation defined as

$$res = \sum (x * y),$$

where x and y are n-element complex vectors.

Input Parameters

n	INTEGER. Specifies the order of vectors *x* and *y*.
x	COMPLEX for cdotu DOUBLE COMPLEX for zdotu Array, DIMENSION at least (1 + (*n* –1)*abs(*incx*)).
incx	INTEGER. Specifies the increment for the elements of *x*.
y	COMPLEX for cdotu DOUBLE COMPLEX for zdotu Array, DIMENSION at least (1 + (*n* –1)*abs(*incy*)).
incy	INTEGER. Specifies the increment for the elements of *y*.

Output Parameters

res	COMPLEX for cdotu DOUBLE COMPLEX for zdotu Contains the result of the dot product of *x* and *y*, if *n* is positive. Otherwise, *res* contains 0.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `dotu` interface are the following:

x	Holds the vector of length(*n*).
y	Holds the vector of length (*n*).

?nrm2

Computes the Euclidean norm of a vector.

Syntax

Fortran 77:

```
res = snrm2(n, x, incx )
res = dnrm2(n, x, incx )
res = scnrm2(n, x, incx )
res = dznrm2(n, x, incx )
```

Fortran 95:

```
res = nrm2(x)
```

Description

The `?nrm2` functions perform a vector reduction operation defined as

```
res  = | |x| |,
```

where:

x is a vector

res is a value containing the Euclidean norm of the elements of *x*.

Input Parameters

n	`INTEGER`. Specifies the order of vector *x*.
x	`REAL` for `snrm2`

DOUBLE PRECISION for dnrm2
COMPLEX for scnrm2
DOUBLE COMPLEX for dznrm2
Array, DIMENSION at least (1 + (*n* -1)*abs (*incx*)).

incx INTEGER. Specifies the increment for the elements of *x*.

Output Parameters

res REAL for snrm2
DOUBLE PRECISION for dnrm2
REAL for scnrm2
DOUBLE PRECISION for dznrm2
Contains the Euclidean norm of the vector *x*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine nrm2 interface are the following:

x Holds the vector of length(*n*).

?rot

Performs rotation of points in the plane.

Syntax

Fortran 77:

```
call srot(n, x, incx, y, incy, c, s )
call drot(n, x, incx, y, incy, c, s )
call csrot(n, x, incx, y, incy, c, s )
call zdrot(n, x, incx, y, incy, c, s )
```

Fortran 95:

```
call rot(x, y [,c] [,s])
```

Description

Given two complex vectors *x* and *y*, each vector element of these vectors is replaced as follows:

```
x(i) = c*x(i) + s*y(i)
y(i) =  c*y(i) -  s*x(i)
```

Input Parameters

n	INTEGER. Specifies the order of vectors *x* and *y*.
x	REAL for srot DOUBLE PRECISION for drot COMPLEX for csrot DOUBLE COMPLEX for zdrot Array, DIMENSION at least (1 + (*n* -1)*abs(*incx*)).
incx	INTEGER. Specifies the increment for the elements of *x*.
y	REAL for srot DOUBLE PRECISION for drot COMPLEX for csrot DOUBLE COMPLEX for zdrot Array, DIMENSION at least (1 + (*n* -1)*abs(*incy*)).
incy	INTEGER. Specifies the increment for the elements of *y*.
c	REAL for srot DOUBLE PRECISION for drot REAL for csrot DOUBLE PRECISION for zdrot A scalar.
s	REAL for srot DOUBLE PRECISION for drot REAL for csrot DOUBLE PRECISION for zdrot A scalar.

Output Parameters

x	Each element is replaced by *c***x* + *s***y*.
y	Each element is replaced by *c***y* - *s***x*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `rot` interface are the following:

x	Holds the vector of length(*n*).
y	Holds the vector of length (*n*).
c	The default value is 1.
s	The default value is 1.

?rotg

Computes the parameters for a Givens rotation.

Syntax

Fortran 77:

```
call srotg(a, b, c, s )
call drotg(a, b, c, s )
call crotg(a, b, c, s )
call zrotg(a, b, c, s )
```

Fortran 95:

```
call rotg(a, b, c, s)
```

Description

Given the cartesian coordinates `(a, b)` of a point `p`, these routines return the parameters *a*, *b*, *c*, and *s* associated with the Givens rotation that zeros the *y*–coordinate of the point.

See a more accurate LAPACK version `?lartg`.

Input Parameters

a	`REAL` for `srotg`
	`DOUBLE PRECISION` for `drotg`
	`COMPLEX` for `crotg`

DOUBLE COMPLEX for zrotg
Provides the *x*-coordinate of the point p.

b
REAL for srotg
DOUBLE PRECISION for drotg
COMPLEX for crotg
DOUBLE COMPLEX for zrotg
Provides the *y*-coordinate of the point p.

Output Parameters

a
Contains the parameter *r* associated with the Givens rotation.

b
Contains the parameter *z* associated with the Givens rotation.

c
REAL for srotg
DOUBLE PRECISION for drotg
REAL for crotg
DOUBLE PRECISION for zrotg
Contains the parameter *c* associated with the Givens rotation.

s
REAL for srotg
DOUBLE PRECISION for drotg
COMPLEX for crotg
DOUBLE COMPLEX for zrotg
Contains the parameter *s* associated with the Givens rotation.

?rotm

Performs rotation of points in the modified plane.

Syntax

Fortran 77:

```
call srotm(n, x, incx, y, incy, param )
call drotm(n, x, incx, y, incy, param )
```

Fortran 95:

```
call rotm(x, y [,param])
```

Description

Given two complex vectors *x* and *y*, each vector element of these vectors is replaced as follows:

*x(i) = H*x(i) + H*y(i)*

*y(i) = H*y(i) - H*x(i)*

where:

H is a modified Givens transformation matrix whose values are stored in the *param(2)* through *param(5)* array. See discussion on the *param* argument.

Input Parameters

n	INTEGER. Specifies the order of vectors *x* and *y*.
x	REAL for srotm DOUBLE PRECISION for drotm Array, DIMENSION at least (1 + (*n* -1)*abs(*incx*)).
incx	INTEGER. Specifies the increment for the elements of *x*.
y	REAL for srotm DOUBLE PRECISION for drotm Array, DIMENSION at least (1 + (*n* -1)*abs(*incy*)).
incy	INTEGER. Specifies the increment for the elements of *y*.
param	REAL for srotm DOUBLE PRECISION for drotm Array, DIMENSION 5. The elements of the *param* array are: *param(1)* contains a switch, *flag*. *param(2-5)* contain *h11*, *h21*, *h12*, and *h22*, respectively, the components of the array *H*. Depending on the values of *flag*, the components of *H* are set as follows:

$$flag = -1. : H = \begin{bmatrix} h11 & h12 \\ h21 & h22 \end{bmatrix}$$

$$flag = 0. : H = \begin{bmatrix} 1. & h12 \\ h21 & 1. \end{bmatrix}$$

$$flag = 1. : H = \begin{bmatrix} h11 & 1. \\ -1 & h22 \end{bmatrix}$$

$$flag = -2. : H = \begin{bmatrix} 1. & 0. \\ 0. & 1. \end{bmatrix}$$

In the above cases, the matrix entries of 1., -1., and 0. are assumed based on the last three values of *flag* and are not actually loaded into the *param* vector.

Output Parameters

x	Each element is replaced by *h11*x + h12*y*.
y	Each element is replaced by *h21*x + h22*y*.
H	Givens transformation matrix updated.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `rotm` interface are the following:

x	Holds the vector of length(*n*).
y	Holds the vector of length (*n*).
param	The default value for *param*(1) is -2.

?rotmg

Computes the modified parameters for a Givens rotation.

Syntax

Fortran 77:

```
call srotmg(d1, d2, x1, y1, param)
call drotmg(d1, d2, x1, y1, param)
```

Fortran 95:

```
call rotmg(x1, y1, param [, d1] [d2])
```

Description

Given cartesian coordinates (*x1*, *y1*) of an input vector, these routines compute the components of a modified Givens transformation matrix *H* that zeros the *y*-component of the resulting vector:

$$\begin{bmatrix} x \\ 0 \end{bmatrix} = H \begin{bmatrix} x1 \\ y1 \end{bmatrix}$$

Input Parameters

d1	REAL for srotmg DOUBLE PRECISION for drotmg Provides the updated scaling factor for the *x*-coordinate of the input vector (sqrt(*d1*)*x1*).
d2	REAL for srotmg DOUBLE PRECISION for drotmg

Provides the updated scaling factor for the *y*-coordinate of the input vector `(sqrt(d2)*y1)`.

x1
`REAL` for `srotmg`
`DOUBLE PRECISION` for `drotmg`
Provides the rotated *x*-coordinate of the input vector.

y1
`REAL` for `srotmg`
`DOUBLE PRECISION` for `drotmg`
Provides the *y*-coordinate of the input vector.

Output Parameters

param
`REAL` for `srotmg`
`DOUBLE PRECISION` for `drotmg`
Array, `DIMENSION` 5.
The elements of the *param* array are:
param(1) contains a switch, *flag*. *param(2-5)* contain *h11*, *h21*, *h12*, and *h22*, respectively, the components of the array *H*.
Depending on the values of *flag*, the components of *H* are set as follows:

$$flag = -1. : H = \begin{bmatrix} h11 & h12 \\ h21 & h22 \end{bmatrix}$$

$$flag = 0. : H = \begin{bmatrix} 1. & h12 \\ h21 & 1. \end{bmatrix}$$

$$flag = 1. : H = \begin{bmatrix} h11 & 1. \\ -1 & h22 \end{bmatrix}$$

$$flag = -2. : H = \begin{bmatrix} 1. & 0. \\ 0. & 1. \end{bmatrix}$$

In the above cases, the matrix entries of 1., -1., and 0. are assumed based on the last three values of *flag* and are not actually loaded into the *param* vector.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `rotmg` interface are the following:

d1	The default value is 1.
d2	The default value is 1.

?scal

Computes a vector by a scalar product.

Syntax

Fortran 77:

```
call sscal(n, a, x, incx )
call dscal(n, a, x, incx )
call cscal(n, a, x, incx )
call zscal(n, a, x, incx )
call csscal(n, a, x, incx )
call zdscal(n, a, x, incx )
```

Fortran 95:

```
call scal(x, a)
```

Description

The ?scal routines perform a vector-vector operation defined as

x = *a***x*

where:

a is a scalar, *x* is an *n*-element vector.

Input Parameters

n	INTEGER. Specifies the order of vector *x*.
a	REAL for sscal and csscal DOUBLE PRECISION for dscal and zdscal COMPLEX for cscal DOUBLE COMPLEX for zscal Specifies the scalar *a*.
x	REAL for sscal DOUBLE PRECISION for dscal COMPLEX for cscal and csscal DOUBLE COMPLEX for zscal and zdscal Array, DIMENSION at least (1 + (*n* –1)*abs(*incx*)).
incx	INTEGER. Specifies the increment for the elements of *x*.

Output Parameters

x	Overwritten by the updated vector x.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine scal interface are the following:

x	Holds the vector of length(*n*).

NOTE. Note that scalar parameter *a* is declared as a required parameter in Fortran-95 interface for the routine scal to distinguish between routine flavors that operate on different data types.

?swap

Swaps a vector with another vector.

Syntax

Fortran 77:

```
call sswap(n, x, incx, y, incy )
call dswap(n, x, incx, y, incy )
call cswap(n, x, incx, y, incy )
call zswap(n, x, incx, y, incy )
```

Fortran 95:

```
call swap(x, y)
```

Description

Given the two complex vectors *x* and *y*, the ?swap routines return vectors *y* and *x* swapped, each replacing the other.

Input Parameters

n	INTEGER. Specifies the order of vectors *x* and *y*.
x	REAL for sswap DOUBLE PRECISION for dswap COMPLEX for cswap DOUBLE COMPLEX for zswap Array, DIMENSION at least (1 + (*n* -1)*abs(*incx*)).
incx	INTEGER. Specifies the increment for the elements of *x*.
y	REAL for sswap DOUBLE PRECISION for dswap COMPLEX for cswap DOUBLE COMPLEX for zswap Array, DIMENSION at least (1 + (*n* -1)*abs(*incy*)).
incy	INTEGER. Specifies the increment for the elements of *y*.

Output Parameters

x	Contains the resultant vector *x*.
y	Contains the resultant vector *y*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `swap` interface are the following:

x	Holds the vector of length(*n*).
y	Holds the vector of length (*n*).

i?amax

Finds the element of a vector that has the largest absolute value.

Syntax

Fortran 77:

```
res = isamax(n, x, incx )
index = idamax(n, x, incx )
index = icamax(n, x, incx )
index = izamax(n, x, incx )
```

Fortran 95:

```
res = iamax(x)
```

Description

Given a vector *x*, the `i?amax` functions return the position of the vector element `x(i)` that has the largest absolute value for real flavors, or the largest sum `|Rex(i)|+|Imx(i)|` for complex flavors.

If *n* is not positive, `0` is returned.

If more than one vector element is found with the same largest absolute value, the index of the first one encountered is returned.

Input Parameters

n	INTEGER. Specifies the order of the vector *x*.
x	REAL for isamax DOUBLE PRECISION for idamax COMPLEX for icamax DOUBLE COMPLEX for izamax Array, DIMENSION at least (1+(*n*-1)*abs(*incx*)).
incx	INTEGER. Specifies the increment for the elements of *x*.

Output Parameters

index	INTEGER. Contains the position of vector element *x* that has the largest absolute value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine amax interface are the following:

x	Holds the vector of length (*n*).

i?amin

Finds the element of a vector that has the smallest absolute value.

Syntax

Fortran 77:

```
index = isamin(n, x, incx )
index = idamin(n, x, incx )
index = icamin(n, x, incx )
index = izamin(n, x, incx )
```

Fortran 95:

```
res = iamin(x)
```

Description

Given a vector *x*, the `i?amin` functions return the position of the vector element *x*(*i*) that has the smallest absolute value for real dlavors, or the smallest sum |Re*x*(*i*)|+|Im*x*(*i*)| for complex flavors.

If *n* is not positive, 0 is returned.

If more than one vector element is found with the same smallest absolute value, the index of the first one encountered is returned.

Input Parameters

n	INTEGER. On entry, *n* specifies the order of the vector *x*.
x	REAL for `isamin` DOUBLE PRECISION for `idamin` COMPLEX for `icamin` DOUBLE COMPLEX for `izamin` Array, DIMENSION at least `(1+(n-1)*abs(incx))`.
incx	INTEGER. Specifies the increment for the elements of *x*.

Output Parameters

index	INTEGER. Contains the position of vector element *x* that has the smallest absolute value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `amin` interface are the following:

x	Holds the vector of length (*n*).

dcabs1

Computes absolute value of double complex number.

Syntax

Fortran 77:

```
res = dcabs1(z)
```

Fortran 95:

```
res = dcabs1(z)
```

Description

The `dcabs1` is an auxiliary routine for a few BLAS Level 1 routines. This function performs an operation defined as

res=|Re(*z*)|+|Im(*z*)|,

where *z* is a scalar and *res* is a value containing the absolute value of a double complex number *z*.

Input Parameters

z	`DOUBLE COMPLEX` scalar.

Output Parameters

res	`DOUBLE PRECISION`. Contains the absolute value of a double complex number *z*.

BLAS Level 2 Routines

This section describes BLAS Level 2 routines, which perform matrix-vector operations. Table 2-2 lists the BLAS Level 2 routine groups and the data types associated with them.

Table 2-2 BLAS Level 2 Routine Groups and Their Data Types

Routine Groups	Data Types	Description
?gbmv	s, d, c, z	Matrix-vector product using a general band matrix

Routine Groups	Data Types	Description
?gemv	s, d, c, z	Matrix-vector product using a general matrix
?ger	s, d	Rank-1 update of a general matrix
?gerc	c, z	Rank-1 update of a conjugated general matrix
?geru	c, z	Rank-1 update of a general matrix, unconjugated
?hbmv	c, z	Matrix-vector product using a Hermitian band matrix
?hemv	c, z	Matrix-vector product using a Hermitian matrix
?her	c, z	Rank-1 update of a Hermitian matrix
?her2	c, z	Rank-2 update of a Hermitian matrix
?hpmv	c, z	Matrix-vector product using a Hermitian packed matrix
?hpr	c, z	Rank-1 update of a Hermitian packed matrix
?hpr2	c, z	Rank-2 update of a Hermitian packed matrix
?sbmv	s, d	Matrix-vector product using symmetric band matrix
?spmv	s, d	Matrix-vector product using a symmetric packed matrix
?spr	s, d	Rank-1 update of a symmetric packed matrix
?spr2	s, d	Rank-2 update of a symmetric packed matrix
?symv	s, d	Matrix-vector product using a symmetric matrix
?syr	s, d	Rank-1 update of a symmetric matrix
?syr2	s, d	Rank-2 update of a symmetric matrix
?tbmv	s, d, c, z	Matrix-vector product using a triangular band matrix
?tbsv	s, d, c, z	Linear solution of a triangular band matrix

Routine Groups	Data Types	Description
?tpmv	s, d, c, z	Matrix-vector product using a triangular packed matrix
?tpsv	s, d, c, z	Linear solution of a triangular packed matrix
?trmv	s, d, c, z	Matrix-vector product using a triangular matrix
?trsv	s, d, c, z	Linear solution of a triangular matrix

?gbmv

Computes a matrix-vector product using a general band matrix

Syntax

Fortran 77:

```
call sgbmv(trans, m, n, kl, ku, alpha, a, lda, x, inxc, beta, y, incy )
call dgbmv(trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy )
call cgbmv(trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy )
call zgbmv(trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy )
```

Fortran 95:

```
call gbmv(a, x, y [,kl][,m] [,alpha][,beta] [,trans])
```

Description

The ?gbmv routines perform a matrix-vector operation defined as

```
y := alpha*A*x + beta*y
```

or

```
y := alpha*A'*x + beta*y,
```

or

```
y := alpha *conjg(A')*x + beta*y,
```

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*n* band matrix, with *kl* sub-diagonals and *ku* super-diagonals.

Input Parameters

trans

CHARACTER*1. Specifies the operation to be performed, as follows:

***trans* value**	**Operation to be Performed**
N or n	*y* := *alpha***A***x* + *beta***y*
T or t	*y* := *alpha***A*'**x* + *beta***y*
C or c	*y* := *alpha* *conjg(*A*')**x* + *beta***y*

m

INTEGER. Specifies the number of rows of the matrix *A*. The value of *m* must be at least zero.

n

INTEGER. Specifies the number of columns of the matrix *A*. The value of *n* must be at least zero.

kl

INTEGER. Specifies the number of sub-diagonals of the matrix *A*.

The value of *kl* must satisfy $0 \le kl$.

ku

INTEGER. Specifies the number of super-diagonals of the matrix *A*.

The value of *ku* must satisfy $0 \le ku$.

alpha

REAL for sgbmv
DOUBLE PRECISION for dgbmv
COMPLEX for cgbmv
DOUBLE COMPLEX for zgbmv
Specifies the scalar *alpha*.

a

REAL for sgbmv
DOUBLE PRECISION for dgbmv
COMPLEX for cgbmv
DOUBLE COMPLEX for zgbmv
Array, DIMENSION (*lda*, *n*).
Before entry, the leading (*kl* + *ku* + 1) by *n* part of the array *a* must contain the matrix of coefficients. This matrix must be supplied column-by-column, with the leading

diagonal of the matrix in row (*ku* + 1) of the array, the first super-diagonal starting at position 2 in row *ku*, the first sub-diagonal starting at position 1 in row (*ku* + 2), and so on. Elements in the array *a* that do not correspond to elements in the band matrix (such as the top left *ku* by *ku* triangle) are not referenced.
The following program segment transfers a band matrix from conventional full matrix storage to band storage:

```
do 20, j = 1, n
 k = ku + 1 - j
   do 10, i = max(1, j-ku), min(m,
j+kl)
   a(k+i, j) = matrix(i,j)
 10 continue
20 continue
```

lda — INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least (*kl* + *ku* + 1).

x — REAL for sgbmv
DOUBLE PRECISION for dgbmv
COMPLEX for cgbmv
DOUBLE COMPLEX for zgbmv
Array, DIMENSION at least (1 + (*n* – 1)*abs(*incx*)) when *trans* = 'N' or 'n', and at least (1 + (*m* – 1)*abs(*incx*)) otherwise. Before entry, the incremented array *x* must contain the vector *x*.

incx — INTEGER. Specifies the increment for the elements of *x*. *incx* must not be zero.

beta — REAL for sgbmv
DOUBLE PRECISION for dgbmv
COMPLEX for cgbmv
DOUBLE COMPLEX for zgbmv
Specifies the scalar beta. When *beta* is equal to zero, then *y* need not be set on input.

y — REAL for sgbmv

	DOUBLE PRECISION for dgbmv COMPLEX for cgbmv DOUBLE COMPLEX for zgbmv Array, DIMENSION at least (1 +(*m* – 1)*abs(*incy*)) when *trans* = 'N' or 'n' and at least (1 +(*n* – 1)*abs(*incy*)) otherwise. Before entry, the incremented array *y* must contain the vector *y*.
incy	INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y	Overwritten by the updated vector *y*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gbmv interface are the following:

a	Holds the array *a* of size (*kl+ku+1*,*n*).
x	Holds the vector of length (*rx*), where *rx = n* if *trans = 'N'*, *rx = m* otherwise.
y	Holds the vector of length (*ry*), where *ry = m* if *trans = 'N'*, *ry = n* otherwise.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
kl	If omitted, assumed *kl= ku*.
ku	Restored as *ku= lda-kl-1*.
m	If omitted, assumed *m= n*.
alpha	The default value is 1.
beta	The default value is 1.

?gemv

Computes a matrix-vector product using a general matrix

Syntax

Fortran 77:

```
call sgemv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy )
call dgemv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy )
call cgemv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy )
call zgemv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy )
```

Fortran 95:

```
call gemv(a, x, y [,alpha][,beta] [,trans])
```

Description

The ?gemv routines perform a matrix-vector operation defined as

y := *alpha***A***x* + *beta***y*,

or

y := *alpha***A*'**x* + *beta***y*,

or

y := *alpha**conjg(*A*')**x* + *beta***y*,

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*n* matrix.

Input Parameters

trans CHARACTER*1. Specifies the operation to be performed, as follows:

***trans* value**	**Operation to be Performed**
N or n	*y* := *alpha***A***x* + *beta***y*

	`T` or `t` *y* := *alpha***A*'**x* + *beta***y* `C` or `c` *y* := *alpha* *conj(*A*')**x* + *beta***y*
m	`INTEGER`. Specifies the number of rows of the matrix *A*. The value of *m* must be at least zero.
n	`INTEGER`. Specifies the number of columns of the matrix *A*. The value of *n* must be at least zero.

alpha	`REAL` for `sgemv`
	`DOUBLE PRECISION` for `dgemv`
	`COMPLEX` for `cgemv`
	`DOUBLE COMPLEX` for `zgemv`
	Specifies the scalar *alpha*.

a	`REAL` for `sgemv`
	`DOUBLE PRECISION` for `dgemv`
	`COMPLEX` for `cgemv`
	`DOUBLE COMPLEX` for `zgemv`
	Array, `DIMENSION (`*lda*`, `*n*`)`. Before entry, the leading *m*-by-*n* part of the array *a* must contain the matrix of coefficients.

lda	`INTEGER`. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least `max(1, `*m*`)`.

x	`REAL` for `sgemv`
	`DOUBLE PRECISION` for `dgemv`
	`COMPLEX` for `cgemv`
	`DOUBLE COMPLEX` for `zgemv`
	Array, `DIMENSION` at least `(1+(`*n*`-1)*abs(`*incx*`))` when *trans* = `'N'` or `'n'` and at least `(1+(`*m* `- 1)*abs(`*incx*`))` otherwise. Before entry, the incremented array *x* must contain the vector *x*.

incx	`INTEGER`. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

beta	`REAL` for `sgemv`
	`DOUBLE PRECISION` for `dgemv`
	`COMPLEX` for `cgemv`
	`DOUBLE COMPLEX` for `zgemv`

	Specifies the scalar *beta*. When *beta* is set to zero, then *y* need not be set on input.
y	REAL for sgemv DOUBLE PRECISION for dgemv COMPLEX for cgemv DOUBLE COMPLEX for zgemv Array, DIMENSION at least (1 +(*m* – 1)*abs(*incy*)) when *trans* = 'N' or 'n' and at least (1 +(*n* – 1)*abs(*incy*)) otherwise. Before entry with non-zero *beta*, the incremented array *y* must contain the vector *y*.
incy	INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y	Overwritten by the updated vector *y*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gemv interface are the following:

a	Holds the matrix A of size (*m*,*n*).
x	Holds the vector of length (*rx*) where *rx* = *n* if *trans* = 'N', *rx* = *m* otherwise.
y	Holds the vector of length (*ry*) where *ry* = *m* if *trans* = 'N', *ry* = *n* otherwise.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
alpha	The default value is 1.
beta	The default value is 1.

?ger

Performs a rank-1 update of a general matrix.

Syntax

Fortran 77:

```
call sger(m, n, alpha, x, incx, y, incy, a, lda )
call dger(m, n, alpha, x, incx, y, incy, a, lda )
```

Fortran 95:

```
call ger(a, x, y [,alpha])
```

Description

The ?ger routines perform a matrix-vector operation defined as

*A := alpha*x*y'+ A*,

where:

alpha is a scalar,

x is an *m*-element vector,

y is an *n*-element vector,

A is an *m*-by-*n* matrix.

Input Parameters

m	INTEGER. Specifies the number of rows of the matrix *A*. The value of *m* must be at least zero.
n	INTEGER. Specifies the number of columns of the matrix *A*. The value of *n* must be at least zero.
alpha	REAL for sger DOUBLE PRECISION for dger Specifies the scalar *alpha*.
x	REAL for sger DOUBLE PRECISION for dger

	Array, DIMENSION at least (1 + (*m* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *m*-element vector *x*.
incx	INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.
y	REAL for sger DOUBLE PRECISION for dger Array, DIMENSION at least (1 + (*n* - 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.
incy	INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.
a	REAL for sger DOUBLE PRECISION for dger Array, DIMENSION (*lda*, *n*). Before entry, the leading *m*-by-*n* part of the array *a* must contain the matrix of coefficients.
lda	INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *m*).

Output Parameters

a	Overwritten by the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ger interface are the following:

a	Holds the matrix A of size (*m*,*n*).
x	Holds the vector of length (*m*).
y	Holds the vector of length (*n*).
alpha	The default value is 1.

?gerc

Performs a rank-1 update (conjugated) of a general matrix.

Syntax

Fortran 77:

```
call cgerc(m, n, alpha, x, incx, y, incy, a, lda )
call zgerc(m, n, alpha, x, incx, y, incy, a, lda )
```

Fortran 95:

```
call gerc(a, x, y [,alpha])
```

Description

The ?gerc routines perform a matrix-vector operation defined as

```
A := alpha*x*conjg(y') + A,
```

where:

alpha is a scalar,

x is an *m*-element vector,

y is an *n*-element vector,

A is an *m*-by-*n* matrix.

Input Parameters

m	INTEGER. Specifies the number of rows of the matrix *A*. The value of *m* must be at least zero.
n	INTEGER. Specifies the number of columns of the matrix *A*. The value of *n* must be at least zero.
alpha	SINGLE PRECISION COMPLEX for cgerc DOUBLE PRECISION COMPLEX for zgerc Specifies the scalar *alpha*.
x	SINGLE PRECISION COMPLEX for cgerc DOUBLE PRECISION COMPLEX for zgerc

	Array, DIMENSION at least (1 + (*m* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *m*-element vector *x*.
incx	INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.
y	COMPLEX for cgerc DOUBLE COMPLEX for zgerc Array, DIMENSION at least (1 + (*n* - 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.
incy	INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.
a	COMPLEX for cgerc DOUBLE COMPLEX for zgerc Array, DIMENSION (*lda*, *n*). Before entry, the leading *m*-by-*n* part of the array *a* must contain the matrix of coefficients.
lda	INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *m*).

Output Parameters

a	Overwritten by the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gerc interface are the following:

a	Holds the matrix A of size (*m*,*n*).
x	Holds the vector of length (*m*).
y	Holds the vector of length (*n*).
alpha	The default value is 1.

?geru

Performs a rank-1 update (unconjugated) of a general matrix.

Syntax

Fortran 77:

```
call cgeru(m, n, alpha, x, incx, y, incy, a, lda )
call zgeru(m, n, alpha, x, incx, y, incy, a, lda )
```

Fortran 95:

```
call geru(a, x, y [,alpha])
```

Description

The ?geru routines perform a matrix-vector operation defined as

*A := alpha*x*y ' + A*,

where:

alpha is a scalar,

x is an *m*-element vector,

y is an *n*-element vector,

A is an *m*-by-*n* matrix.

Input Parameters

m	INTEGER. Specifies the number of rows of the matrix *A*. The value of *m* must be at least zero.
n	INTEGER. Specifies the number of columns of the matrix *A*. The value of *n* must be at least zero.
alpha	COMPLEX for cgeru DOUBLE COMPLEX for zgeru Specifies the scalar *alpha*.
x	COMPLEX for cgeru DOUBLE COMPLEX for zgeru

	Array, DIMENSION at least (1 + (*m* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *m*-element vector *x*.
incx	INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.
y	COMPLEX for cgeru DOUBLE COMPLEX for zgeru Array, DIMENSION at least (1 + (*n* - 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.
incy	INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.
a	COMPLEX for cgeru DOUBLE COMPLEX for zgeru Array, DIMENSION (*lda*, *n*). Before entry, the leading *m*-by-*n* part of the array *a* must contain the matrix of coefficients.
lda	INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *m*).

Output Parameters

a	Overwritten by the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine geru interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
x	Holds the vector of length (*m*).
y	Holds the vector of length (*n*).
alpha	The default value is 1.

?hbmv

Computes a matrix-vector product using a Hermitian band matrix.

Syntax

Fortran 77:

```
call chbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy )
call zhbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy )
```

Fortran 95:

```
call hbmv(a, x, y [,uplo][,alpha] [,beta])
```

Description

The ?hbmv routines perform a matrix-vector operation defined as

y := *alpha***A***x* + *beta***y*,

where:

alpha and *beta* are scalars,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* Hermitian band matrix, with *k* super-diagonals.

Input Parameters

uplo — CHARACTER*1. Specifies whether the upper or lower triangular part of the band matrix *A* is being supplied, as follows:

***uplo* value**	**Part of Matrix *A* Supplied**
U or u	The upper triangular part of matrix *A* is being supplied.
L or l	The lower triangular part of matrix *A* is being supplied.

n — INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

k INTEGER. Specifies the number of super-diagonals of the matrix *A*.

The value of *k* must satisfy $0 \le k$.

alpha COMPLEX for chbmv
DOUBLE COMPLEX for zhbmv
Specifies the scalar *alpha*.

a COMPLEX for chbmv
DOUBLE COMPLEX for zhbmv
Array, DIMENSION (*lda*, *n*).
Before entry with *uplo* = 'U' or 'u', the leading (*k* + 1) by *n* part of the array *a* must contain the upper triangular band part of the Hermitian matrix. The matrix must be supplied column-by-column, with the leading diagonal of the matrix in row (*k* + 1) of the array, the first super-diagonal starting at position 2 in row *k*, and so on. The top left *k* by *k* triangle of the array *a* is not referenced.
The following program segment transfers the upper triangular part of a Hermitian band matrix from conventional full matrix storage to band storage:

```
do 20, j = 1, n
 m = k + 1 - j
 do 10, i = max(1, j - k), j
  a(m + i, j) = matrix(i, j)
 10 continue
20 continue
```

Before entry with *uplo* = 'L' or 'l', the leading (*k* + 1) by *n* part of the array *a* must contain the lower triangular band part of the Hermitian matrix, supplied column-by-column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right *k* by *k* triangle of the array *a* is not referenced.

The following program segment transfers the lower triangular part of a Hermitian band matrix from conventional full matrix storage to band storage:

```
do 20, j = 1, n
 m = 1 - j
 do 10, i = j, min( n, j + k )
  a( m + i, j ) = matrix( i, j )
 10 continue
20 continue
```

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

lda — INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least (*k* + 1).

x — COMPLEX for chbmv
DOUBLE COMPLEX for zhbmv
Array, DIMENSION at least (1 + (*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the vector *x*.

incx — INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

beta — COMPLEX for chbmv
DOUBLE COMPLEX for zhbmv
Specifies the scalar *beta*.

y — COMPLEX for chbmv
DOUBLE COMPLEX for zhbmv
Array, DIMENSION at least (1 + (*n* – 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the vector *y*.

incy — INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y — Overwritten by the updated vector *y*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `hbmv` interface are the following:

a	Holds the array *a* of size (*k+1,n*).
x	Holds the vector of length (*n*).
y	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
alpha	The default value is 1.
beta	The default value is 1.

?hemv

Computes a matrix-vector product using a Hermitian matrix.

Syntax

Fortran 77:

```
call chemv(uplo, n, alpha, a, lda, x, incx, beta, y, incy )
call zhemv(uplo, n, alpha, a, lda, x, incx, beta, y, incy )
```

Fortran 95:

```
call hemv(a, x, y [,uplo][,alpha] [,beta])
```

Description

The `?hemv` routines perform a matrix-vector operation defined as

*y := alpha*A*x + beta*y*,

where:

alpha and *beta* are scalars,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* Hermitian matrix.

Input Parameters

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the array *a* is to be referenced, as follows:

uplo value	Part of Array *a* To Be Referenced
U or u	The upper triangular part of array *a* is to be referenced.
L or l	The lower triangular part of array *a* is to be referenced.

n

INTEGER. Specifies the order of the matrix A. The value of *n* must be at least zero.

alpha

COMPLEX for chemv
DOUBLE COMPLEX for zhemv
Specifies the scalar *alpha*.

a

COMPLEX for chemv
DOUBLE COMPLEX for zhemv
Array, DIMENSION (*lda*, *n*).
Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *a* is not referenced. Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of *a* is not referenced.
The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

lda

INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *n*).

x

COMPLEX for chemv
DOUBLE COMPLEX for zhemv
Array, DIMENSION at least (1 + (*n* – 1)*abs(*incx*)).
Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx

INTEGER. Specifies the increment for the elements of *x*.

	The value of *incx* must not be zero.
beta	COMPLEX for chemv DOUBLE COMPLEX for zhemv Specifies the scalar *beta*. When *beta* is supplied as zero then *y* need not be set on input.
y	COMPLEX for chemv DOUBLE COMPLEX for zhemv Array, DIMENSION at least (1 + (*n* - 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.
incy	INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y	Overwritten by the updated vector *y*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hemv interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
x	Holds the vector of length (*n*).
y	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
alpha	The default value is 1.
beta	The default value is 1.

?her

Performs a rank-1 update of a Hermitian matrix.

Syntax

Fortran 77:

```
call cher(uplo, n, alpha, x, incx, a, lda )
call zher(uplo, n, alpha, x, incx, a, lda )
```

Fortran 95:

```
call her(a, x [,uplo] [, alpha])
```

Description

The ?her routines perform a matrix-vector operation defined as

A := *alpha***x**conjg(*x*') + *A*,

where:

alpha is a real scalar,

x is an *n*-element vector,

A is an *n*-by-*n* Hermitian matrix.

Input Parameters

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the array *a* is to be referenced, as follows:

***uplo* value**	**Part of Array *a* To Be Referenced**
U or u	The upper triangular part of array *a* is to be referenced.
L or l	The lower triangular part of array *a* is to be referenced.

n

INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

alpha

REAL for cher
DOUBLE PRECISION for zher
Specifies the scalar *alpha*.

x
COMPLEX for cher
DOUBLE COMPLEX for zher
Array, dimension at least (1 + (*n* - 1)*abs(*incx*)).
Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx
INTEGER. Specifies the increment for the elements of *x*.
The value of *incx* must not be zero.

a
COMPLEX for cher
DOUBLE COMPLEX for zher
Array, DIMENSION (*lda*, *n*).
Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *a* is not referenced.
Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of *a* is not referenced.
The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

lda
INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *n*).

Output Parameters

a
With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated matrix.
With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated matrix.
The imaginary parts of the diagonal elements are set to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `her` interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
x	Holds the vector of length (*n*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
alpha	The default value is 1.

?her2

Performs a rank-2 update of a Hermitian matrix.

Syntax

Fortran 77:

```
call cher2(uplo, n, alpha, x, incx, y, incy, a, lda )
call zher2(uplo, n, alpha, x, incx, y, incy, a, lda )
```

Fortran 95:

```
call her2(a, x, y [,uplo][,alpha])
```

Description

The `?her2` routines perform a matrix-vector operation defined as

A := *alpha* *x*conjg(*y*') + conjg(*alpha*)**y* *conjg(*x*') + *A*,

where:

alpha is a scalar,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* Hermitian matrix.

Input Parameters

uplo
CHARACTER*1. Specifies whether the upper or lower triangular part of the array *a* is to be referenced, as follows:

uplo value	Part of Array *a* To Be Referenced
U or u	The upper triangular part of array *a* is to be referenced.
L or l	The lower triangular part of array *a* is to be referenced.

n
INTEGER. Specifies the order of the matrix A. The value of *n* must be at least zero.

alpha
COMPLEX for cher2
DOUBLE COMPLEX for zher2
Specifies the scalar *alpha*.

x
COMPLEX for cher2
DOUBLE COMPLEX for zher2
Array, DIMENSION at least (1 + (*n* – 1)*abs(*incx*)).
Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx
INTEGER. Specifies the increment for the elements of *x*.
The value of *incx* must not be zero.

y
COMPLEX for cher2
DOUBLE COMPLEX for zher2
Array, DIMENSION at least (1 + (*n* – 1)*abs(*incy*)).
Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy
INTEGER. Specifies the increment for the elements of *y*.
The value of *incy* must not be zero.

a
COMPLEX for cher2
DOUBLE COMPLEX for zher2
Array, DIMENSION (*lda*, *n*).
Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *a* is not referenced.

Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of *a* is not referenced.
The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

lda — INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *n*).

Output Parameters

a — With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated matrix.
With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated matrix.
The imaginary parts of the diagonal elements are set to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine her2 interface are the following:

a — Holds the matrix A of size (*n*,*n*).

x — Holds the vector of length (*n*).

y — Holds the vector of length (*n*).

uplo — Must be 'U' or 'L'. The default value is 'U'.

alpha — The default value is 1.

?hpmv

Computes a matrix-vector product using a Hermitian packed matrix.

Syntax

Fortran 77:

```
call chpmv(uplo, n, alpha, ap, x, incx, beta, y, incy )
call zhpmv(uplo, n, alpha, ap, x, incx, beta, y, incy )
```

Fortran 95:

```
call hpmv(a, x, y [,uplo][,alpha] [,beta])
```

Description

The ?hpmv routines perform a matrix-vector operation defined as

y := *alpha***A***x* + *beta***y*,

where:

alpha and *beta* are scalars,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* Hermitian matrix, supplied in packed form.

Input Parameters

uplo — CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix *A* is supplied in the packed array *ap*, as follows:

***uplo* value**	**Part of Matrix *A* Supplied**
U or u	The upper triangular part of matrix *A* is supplied in *ap*.
L or l	The lower triangular part of matrix *A* is supplied in *ap*.

n — INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

alpha — COMPLEX for chpmv

	DOUBLE COMPLEX for zhpmv Specifies the scalar *alpha*.
ap	COMPLEX for chpmv DOUBLE COMPLEX for zhpmv Array, DIMENSION at least ((*n**(*n* + 1))/2). Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, 1), *ap*(2) and *ap*(3) contain *a*(1, 2) and *a*(2, 2) respectively, and so on. Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, 1), *ap*(2) and *ap*(3) contain *a*(2, 1) and *a*(3, 1) respectively, and so on. The imaginary parts of the diagonal elements need not be set and are assumed to be zero.
x	COMPLEX for chpmv DOUBLE PRECISION COMPLEX for zhpmv Array, DIMENSION at least (1 +(*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.
incx	INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.
beta	COMPLEX for chpmv DOUBLE COMPLEX for zhpmv Specifies the scalar *beta*. When *beta* is equal to zero then *y* need not be set on input.
y	COMPLEX for chpmv DOUBLE COMPLEX for zhpmv Array, DIMENSION at least (1 + (*n* – 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.
incy	INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y	Overwritten by the updated vector *y*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `hpmv` interface are the following:

a	Holds the array *a* of size (*n*(n+1)/2*).
x	Holds the vector of length (*n*).
y	Holds the vector of length (*n*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
alpha	The default value is 1.
beta	The default value is 1.

?hpr

Performs a rank-1 update of a Hermitian packed matrix.

Syntax

Fortran 77:

```
call chpr(uplo, n, alpha, x, incx, ap)
call zhpr(uplo, n, alpha, x, incx, ap)
```

Fortran 95:

```
call hpr(a, x [,uplo] [, alpha])
```

Description

The `?hpr` routines perform a matrix-vector operation defined as

A := *alpha*x**conjg(*x*') + *A*,

where:

alpha is a real scalar,

x is an *n*-element vector,

A is an *n*-by-*n* Hermitian matrix, supplied in packed form.

Input Parameters

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix *A* is supplied in the packed array *ap*, as follows:

***uplo* value**	**Part of Matrix *A* Supplied**
U or u	The upper triangular part of matrix *A* is supplied in *ap*.
L or l	The lower triangular part of matrix *A* is supplied in *ap*.

n

INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

alpha

REAL for chpr
DOUBLE PRECISION for zhpr
Specifies the scalar *alpha*.

x

COMPLEX for chpr
DOUBLE COMPLEX for zhpr
Array, DIMENSION at least (1 + (*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx

INTEGER. Specifies the increment for the elements of *x*. *incx* must not be zero.

ap

COMPLEX for chpr
DOUBLE COMPLEX for zhpr
Array, DIMENSION at least ((*n**(*n* + 1))/2). Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, 1), *ap*(2) and *ap*(3) contain *a*(1, 2) and *a*(2, 2) respectively, and so on.

Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, 1), *ap*(2) and *ap*(3) contain *a*(2, 1) and *a*(3, 1) respectively, and so on.
The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

Output Parameters

ap — With *uplo* = 'U' or 'u', overwritten by the upper triangular part of the updated matrix.
With *uplo* = 'L' or 'l', overwritten by the lower triangular part of the updated matrix.
The imaginary parts of the diagonal elements are set to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hpr interface are the following:

a — Holds the array *a* of size (*n**(n+1)/2).

x — Holds the vector of length (*n*).

uplo — Must be 'U' or 'L'. The default value is 'U'.

alpha — The default value is 1.

?hpr2

Performs a rank-2 update of a Hermitian packed matrix.

Syntax

Fortran 77:

```
call chpr2(uplo, n, alpha, x, incx, y, incy, ap )
call zhpr2(uplo, n, alpha, x, incx, y, incy, ap )
```

Fortran 95:

```
call hpr2(a, x, y [,uplo][,alpha])
```

Description

The ?hpr2 routines perform a matrix-vector operation defined as

A := *alpha***x**conjg(*y*') + conjg(*alpha*)**y**conjg(*x*') + *A*,

where:

alpha is a scalar,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* Hermitian matrix, supplied in packed form.

Input Parameters

uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix *A* is supplied in the packed array *ap*, as follows

***uplo* value**	**Part of Matrix *A* Supplied**
U or u	The upper triangular part of matrix *A* is supplied in *ap*.
L or l	The lower triangular part of matrix *A* is supplied in *ap*.

n	INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.
alpha	COMPLEX for chpr2 DOUBLE COMPLEX for zhpr2 Specifies the scalar *alpha*.
x	COMPLEX for chpr2 DOUBLE COMPLEX for zhpr2 Array, dimension at least (1 +(*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.
incx	INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.
y	COMPLEX for chpr2

DOUBLE COMPLEX for zhpr2
Array, DIMENSION at least (1 +(*n* - 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy — INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

ap — COMPLEX for chpr2
DOUBLE COMPLEX for zhpr2
Array, DIMENSION at least ((*n**(*n* + 1))/2). Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1,1), *ap*(2) and *ap*(3) contain *a*(1,2) and *a*(2,2) respectively, and so on.
Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1,1), *ap*(2) and *ap*(3) contain *a*(2,1) and *a*(3,1) respectively, and so on.
The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

Output Parameters

ap — With *uplo* = 'U' or 'u', overwritten by the upper triangular part of the updated matrix.
With *uplo* = 'L' or 'l', overwritten by the lower triangular part of the updated matrix.
The imaginary parts of the diagonal elements need are set to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hpr2 interface are the following:

a — Holds the array *a* of size (*n**(*n*+1)/2).

x	Holds the vector of length (*n*).
y	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
alpha	The default value is 1.

?sbmv

Computes a matrix-vector product using a symmetric band matrix.

Syntax

Fortran 77:

```
call ssbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy )
call dsbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy )
```

Fortran 95:

```
call sbmv(a, x, y [,uplo][,alpha] [,beta])
```

Description

The ?sbmv routines perform a matrix-vector operation defined as

y := *alpha***A***x* + *beta***y*,

where:

alpha and *beta* are scalars,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* symmetric band matrix, with *k* super-diagonals.

Input Parameters

uplo — CHARACTER*1. Specifies whether the upper or lower triangular part of the band matrix *A* is being supplied, as follows:

uplo value	Part of Matrix *A* Supplied
U or u	The upper triangular part of matrix *A* is supplied.

	`L` or `l` The lower triangular part of matrix *A* is supplied.
n	`INTEGER`. Specifies the order of the matrix *A*. The value of *n* must be at least zero.
k	`INTEGER`. Specifies the number of super-diagonals of the matrix *A*. The value of *k* must satisfy $0 \le k$.

alpha	`REAL` for `ssbmv`
	`DOUBLE PRECISION` for `dsbmv`
	Specifies the scalar *alpha*.

a	`REAL` for `ssbmv`
	`DOUBLE PRECISION` for `dsbmv`
	Array, `DIMENSION (lda, n)`. Before entry with *uplo* = `'U'` or `'u'`, the leading `(k + 1)` by *n* part of the array *a* must contain the upper triangular band part of the symmetric matrix, supplied column-by-column, with the leading diagonal of the matrix in row `(k + 1)` of the array, the first super-diagonal starting at position 2 in row *k*, and so on. The top left *k* by *k* triangle of the array *a* is not referenced.
	The following program segment transfers the upper triangular part of a symmetric band matrix from conventional full matrix storage to band storage:
	`do 20, j = 1, n`
	`m = k + 1 - j`
	`do 10, i = max( 1, j - k ), j`
	`a( m + i, j ) = matrix( i, j )`
	`10 continue`
	`20 continue`
	Before entry with *uplo* = `'L'` or `'l'`, the leading `(k + 1)` by *n* part of the array *a* must contain the lower triangular band part of the symmetric matrix, supplied column-by-column, with the leading diagonal of the matrix

in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right *k* by *k* triangle of the array *a* is not referenced.
The following program segment transfers the lower triangular part of a symmetric band matrix from conventional full matrix storage to band storage:

```
do 20, j = 1, n
  m = 1 - j
  do 10, i = j, min( n, j + k )
    a( m + i, j ) = matrix( i, j )
  10 continue
20 continue
```

lda — INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least (*k* + 1).

x — REAL for ssbmv
DOUBLE PRECISION for dsbmv
Array, DIMENSION at least (1 + (*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the vector *x*.

incx — INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

beta — REAL for ssbmv
DOUBLE PRECISION for dsbmv
Specifies the scalar *beta*.

y — REAL for ssbmv
DOUBLE PRECISION for dsbmv
Array, DIMENSION at least (1 + (*n* – 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the vector *y*.

incy — INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y	Overwritten by the updated vector *y*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `sbmv` interface are the following:

a	Holds the array *a* of size (*k+1*,*n*).
x	Holds the vector of length (*n*).
y	Holds the vector of length (*n*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
alpha	The default value is 1.
beta	The default value is 1.

?spmv

Computes a matrix-vector product using a symmetric packed matrix.

Syntax

Fortran 77:

```
call sspmv(uplo, n, alpha, ap, x, incx, beta, y, incy )
call dspmv(uplo, n, alpha, ap, x, incx, beta, y, incy )
```

Fortran 95:

```
call spmv(a, x, y [,uplo][,alpha] [,beta])
```

Description

The `?spmv` routines perform a matrix-vector operation defined as

*y := alpha*A*x + beta*y*,

where:

alpha and *beta* are scalars,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* symmetric matrix, supplied in packed form.

Input Parameters

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix *A* is supplied in the packed array *ap*, as follows:

***uplo* value**	**Part of Matrix *A* Supplied**
U or u	The upper triangular part of matrix *A* is supplied in *ap*.
L or l	The lower triangular part of matrix *A* is supplied in *ap*.

n

INTEGER. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

alpha

REAL for sspmv
DOUBLE PRECISION for dspmv
Specifies the scalar *alpha*.

ap

REAL for sspmv
DOUBLE PRECISION for dspmv
Array, DIMENSION at least ((n*(n + 1))/2).
Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1,1), *ap*(2) and *ap*(3) contain *a*(1,2) and *a*(2, 2) respectively, and so on. Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1,1), *ap*(2) and *ap*(3) contain *a*(2,1) and *a*(3,1) respectively, and so on.

x

REAL for sspmv
DOUBLE PRECISION for dspmv
Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)).
Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx	INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.
beta	REAL for sspmv DOUBLE PRECISION for dspmv Specifies the scalar *beta*. When *beta* is supplied as zero, then *y* need not be set on input.
y	REAL for sspmv DOUBLE PRECISION for dspmv Array, DIMENSION at least (1 + (*n* - 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.
incy	INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y	Overwritten by the updated vector *y*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine spmv interface are the following:

a	Holds the array *a* of size (*n*(n+1)/2).
x	Holds the vector of length (*n*).
y	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
alpha	The default value is 1.
beta	The default value is 1.

?spr

Performs a rank-1 update of a symmetric packed matrix.

Syntax

Fortran 77:

```
call sspr(uplo, n, alpha, x, incx, ap )
call dspr(uplo, n, alpha, x, incx, ap )
```

Fortran 95:

```
call spr(a, x [,uplo] [, alpha])
```

Description

The ?spr routines perform a matrix-vector operation defined as

a:= *alpha***x***x*'+ *A*,

where:

alpha is a real scalar,

x is an *n*-element vector,

A is an *n*-by-*n* symmetric matrix, supplied in packed form.

Input Parameters

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix *A* is supplied in the packed array *ap*, as follows:

***uplo* value**	**Part of Matrix *A* Supplied**
U or u	The upper triangular part of matrix *A* is supplied in *ap*.
L or l	The lower triangular part of matrix *A* is supplied in *ap*.

n

INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

alpha

REAL for sspr

	DOUBLE PRECISION for dspr Specifies the scalar *alpha*.
x	REAL for sspr DOUBLE PRECISION for dspr Array, DIMENSION at least (1 + (*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.
incx	INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.
ap	REAL for sspr DOUBLE PRECISION for dspr Array, DIMENSION at least ((*n**(*n* + 1))/2). Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1,1), *ap*(2) and *ap*(3) contain *a*(1,2) and *a*(2,2) respectively, and so on. Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1,1), *ap*(2) and *ap*(3) contain *a*(2,1) and *a*(3,1) respectively, and so on.

Output Parameters

ap	With *uplo* = 'U' or 'u', overwritten by the upper triangular part of the updated matrix. With *uplo* = 'L' or 'l', overwritten by the lower triangular part of the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine spr interface are the following:

a	Holds the array *a* of size (*n**(*n*+1)/2).
x	Holds the vector of length (*n*).

uplo	Must be 'U' or 'L'. The default value is 'U'.
alpha	The default value is 1.

?spr2

Performs a rank-2 update of a symmetric packed matrix.

Syntax

Fortran 77:

```
call sspr2(uplo, n, alpha, x, incx, y, incy, ap )
call dspr2(uplo, n, alpha, x, incx, y, incy, ap )
```

Fortran 95:

```
call spr2(a, x, y [,uplo][,alpha])
```

Description

The ?spr2 routines perform a matrix-vector operation defined as

A:= *alpha***x***y*'+ *alpha***y***x*' + *A*,

where:

alpha is a scalar,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* symmetric matrix, supplied in packed form.

Input Parameters

uplo — CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix *A* is supplied in the packed array *ap*, as follows:

uplo value	Part of Matrix *A* Supplied
U or u	The upper triangular part of matrix *A* is supplied in *ap*.
L or l	The lower triangular part of matrix *A* is supplied in *ap*.

n INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

alpha REAL for sspr2
DOUBLE PRECISION for dspr2
Specifies the scalar *alpha*.

x REAL for sspr2
DOUBLE PRECISION for dspr2
Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

y REAL for sspr2
DOUBLE PRECISION for dspr2
Array, DIMENSION at least (1 + (*n* - 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

ap REAL for sspr2
DOUBLE PRECISION for dspr2
Array, DIMENSION at least ((*n**(*n* + 1))/2). Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1,1), *ap*(2) and *ap*(3) contain *a*(1,2) and *a*(2,2) respectively, and so on.
Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1,1), *ap*(2) and *ap*(3) contain *a* (2,1) and *a*(3,1) respectively, and so on.

Output Parameters

ap With *uplo* = 'U' or 'u', overwritten by the upper triangular part of the updated matrix.

With *uplo* = 'L' or 'l', overwritten by the lower triangular part of the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine spr2 interface are the following:

a	Holds the array *a* of size (*n*(n+1)/2*).
x	Holds the vector of length (*n*).
y	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
alpha	The default value is 1.

?symv

Computes a matrix-vector product for a symmetric matrix.

Syntax

Fortran 77:

```
call ssymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy )
call dsymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy )
```

Fortran 95:

```
call symv(a, x, y [,uplo][,alpha] [,beta])
```

Description

The ?symv routines perform a matrix-vector operation defined as

*y := alpha*A*x + beta*y*,

where:

alpha and *beta* are scalars,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* symmetric matrix.

Input Parameters

uplo — CHARACTER*1. Specifies whether the upper or lower triangular part of the array *a* is to be referenced, as follows:

***uplo* value**	**Part of Array *a* To Be Referenced**
U or u	The upper triangular part of array *a* is to be referenced.
L or l	The lower triangular part of array *a* is to be referenced.

n — INTEGER. Specifies the order of the matrix *a*. The value of *n* must be at least zero.

alpha — REAL for ssymv
DOUBLE PRECISION for dsymv
Specifies the scalar *alpha*.

a — REAL for ssymv
DOUBLE PRECISION for dsymv
Array, DIMENSION (*lda*, *n*).
Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix A and the strictly lower triangular part of *a* is not referenced. Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix A and the strictly upper triangular part of *a* is not referenced.

lda — INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *n*).

x — REAL for ssymv
DOUBLE PRECISION for dsymv
Array, DIMENSION at least (1 + (*n* – 1)*abs(*incx*)).
Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx — INTEGER. Specifies the increment for the elements of *x*.

	The value of *incx* must not be zero.
beta	REAL for ssymv DOUBLE PRECISION for dsymv Specifies the scalar *beta*. When *beta* is supplied as zero, then *y* need not be set on input.
y	REAL for ssymv DOUBLE PRECISION for dsymv Array, DIMENSION at least (1 + (*n* – 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.
incy	INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y	Overwritten by the updated vector *y*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine symv interface are the following:

a	Holds the matrix A of size (*n*,*n*).
x	Holds the vector of length (*n*).
y	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
alpha	The default value is 1.
beta	The default value is 1.

?syr

Performs a rank-1 update of a symmetric matrix.

Syntax

Fortran 77:

```
call ssyr(uplo, n, alpha, x, incx, a, lda )
call dsyr(uplo, n, alpha, x, incx, a, lda )
```

Fortran 95:

```
call syr(a, x [,uplo] [, alpha])
```

Description

The ?syr routines perform a matrix-vector operation defined as

A := *alpha***x***x*' + *A* ,

where:

alpha is a real scalar,

x is an *n*-element vector,

A is an *n*-by-*n* symmetric matrix.

Input Parameters

uplo — CHARACTER*1. Specifies whether the upper or lower triangular part of the array *a* is to be referenced, as follows:

***uplo* value**	**Part of Array *a* To Be Referenced**
U or u	The upper triangular part of array *a* is to be referenced.
L or l	The lower triangular part of array *a* is to be referenced.

n — INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

alpha — REAL for ssyr
DOUBLE PRECISION for dsyr

	Specifies the scalar *alpha*.
x	REAL for ssyr DOUBLE PRECISION for dsyr Array, DIMENSION at least (1 + (*n*–1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.
incx	INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.
a	REAL for ssyr DOUBLE PRECISION for dsyr Array, DIMENSION (*lda*, *n*). Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix *A* and the strictly lower triangular part of *a* is not referenced. Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix *A* and the strictly upper triangular part of *a* is not referenced.
lda	INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *n*).

Output Parameters

a	With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated matrix. With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine syr interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
x	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
alpha	The default value is 1.

?syr2

Performs a rank-2 update of symmetric matrix.

Syntax

Fortran 77:

```
call ssyr2(uplo, n, alpha, x, incx, y, incy, a, lda )
call dsyr2(uplo, n, alpha, x, incx, y, incy, a, lda )
```

Fortran 95:

```
call syr2(a, x, y [,uplo][,alpha])
```

Description

The ?syr2 routines perform a matrix-vector operation defined as

*A := alpha*x*y'+ alpha*y*x' + A*,

where:

alpha is a scalar,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* symmetric matrix.

Input Parameters

uplo — CHARACTER*1. Specifies whether the upper or lower triangular part of the array *a* is to be referenced, as follows:

uplo value	Part of Array *a* To Be Referenced
U or u	The upper triangular part of array *a* is to be referenced.

	`L` or `l` The lower triangular part of array *a* is to be referenced.
n	`INTEGER`. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

alpha	`REAL` for `ssyr2`
	`DOUBLE PRECISION` for `dsyr2`
	Specifies the scalar *alpha*.

x	`REAL` for `ssyr2`
	`DOUBLE PRECISION` for `dsyr2`
	Array, `DIMENSION` at least `(1 + (n - 1)*abs(incx))`.
	Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx	`INTEGER`. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

y	`REAL` for `ssyr2`
	`DOUBLE PRECISION` for `dsyr2`
	Array, `DIMENSION` at least `(1 + (n - 1)*abs(incy))`.
	Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy	`INTEGER`. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

a	`REAL` for `ssyr2`
	`DOUBLE PRECISION` for `dsyr2`
	Array, `DIMENSION (lda, n)`.
	Before entry with *uplo* = `'U'` or `'u'`, the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *a* is not referenced.
	Before entry with *uplo* = `'L'` or `'l'`, the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *a* is not referenced.

lda	`INTEGER`. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least `max(1, n)`.

Output Parameters

a	With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated matrix. With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine syr2 interface are the following:

a	Holds the matrix A of size (*n*,*n*).
x	Holds the vector *x* of length (*n*).
y	Holds the vector *y* of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
alpha	The default value is 1.

?tbmv

Computes a matrix-vector product using a triangular band matrix.

Syntax

Fortran 77:

```
call stbmv(uplo, trans, diag, n, k, a, lda, x, incx )
call dtbmv(uplo, trans, diag, n, k, a, lda, x, incx )
call ctbmv(uplo, trans, diag, n, k, a, lda, x, incx )
call ztbmv(uplo, trans, diag, n, k, a, lda, x, incx )
```

Fortran 95:

```
call tbmv(a, x [,uplo] [, trans] [,diag])
```

Description

The ?tbmv routines perform one of the matrix-vector operations defined as

x := A*x, or x := A'*x, or x := conjg(A')*x,

where:

x is an n-element vector,

A is an n-by-n unit, or non-unit, upper or lower triangular band matrix, with (k +1) diagonals.

Input Parameters

uplo

CHARACTER*1. Specifies whether the matrix is an upper or lower triangular matrix, as follows:

***uplo* value**	**Matrix *A***
U or u	An upper triangular matrix.
L or l	A lower triangular matrix.

trans

CHARACTER*1. Specifies the operation to be performed, as follows:

***trans* value**	**Operation to be Performed**
N or n	x := A*x
T or t	x := A'*x
C or c	x := conjg(A')*x

diag

CHARACTER*1. Specifies whether or not A is unit triangular, as follows:

***diag* value**	**Matrix *a***
U or u	Matrix a is a unit triangular.
N or n	Matrix a is not a unit triangular.

n

INTEGER. Specifies the order of the matrix A. The value of n must be at least zero.

k

INTEGER. On entry with *uplo* = 'U' or 'u', k specifies the number of super-diagonals of the matrix A. On entry with *uplo* = 'L' or 'l', k specifies the number of sub-diagonals of the matrix a.

The value of k must satisfy $0 \le k$.

a

REAL for stbmv
DOUBLE PRECISION for dtbmv
COMPLEX for ctbmv
DOUBLE COMPLEX for ztbmv
Array, DIMENSION (*lda*, *n*).
Before entry with *uplo* = 'U' or 'u', the leading (*k* + 1) by *n* part of the array *a* must contain the upper triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row (*k* + 1) of the array, the first super-diagonal starting at position 2 in row *k*, and so on. The top left *k* by *k* triangle of the array *a* is not referenced. The following program segment transfers an upper triangular band matrix from conventional full matrix storage to band storage:

```
              do 20, j = 1, n
               m = k + 1 - j
do 10, i = max(1, j - k), j
               a(m + i, j) = matrix(i, j)
          10 continue
         20 continue
```

Before entry with *uplo* = 'L' or 'l', the leading (*k* + 1) by *n* part of the array *a* must contain the lower triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right *k* by *k*

triangle of the array *a* is not referenced. The following program segment transfers a lower triangular band matrix from conventional full matrix storage to band storage:

```
do 20, j = 1, n
 m = 1 - j
      do 10, i = j, min(n, j + k)
   a(m + i, j) = matrix (i, j)
10 continue
20 continue
```

Note that when *diag* = 'U' or 'u', the elements of the array *a* corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity.

lda — INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least (*k* + 1).

x — REAL for stbmv
DOUBLE PRECISION for dtbmv
COMPLEX for ctbmv
DOUBLE COMPLEX for ztbmv
Array, DIMENSION at least (1 + (*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx — INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

Output Parameters

x — Overwritten with the transformed vector *x*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine tbmv interface are the following:

a	Holds the array *a* of size (*k+1*,*n*).
x	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
diag	Must be 'N' or 'U'. The default value is 'N'.

?tbsv

Solves a system of linear equations whose coefficients are in a triangular band matrix.

Syntax

Fortran 77:

```
call stbsv(uplo, trans, diag, n, k, a, lda, x, incx )
call dtbsv(uplo, trans, diag, n, k, a, lda, x, incx )
call ctbsv(uplo, trans, diag, n, k, a, lda, x, incx )
call ztbsv(uplo, trans, diag, n, k, a, lda, x, incx )
```

Fortran 95:

```
call tbsv(a, x [,uplo] [, trans] [,diag])
```

Description

The ?tbsv routines solve one of the following systems of equations:

*A*x = b*, or *A'*x = b*, or conjg(*A'*)*x = b*,

where:

b and *x* are *n*-element vectors,

A is an *n*-by-*n* unit, or non-unit, upper or lower triangular band matrix, with (*k* + 1) diagonals.

The routine does not test for singularity or near-singularity.

Such tests must be performed before calling this routine.

Input Parameters

uplo

CHARACTER*1. Specifies whether the matrix *A* is an upper or lower triangular matrix, as follows:

***uplo* value**	**Matrix *A***
U or u	An upper triangular matrix.
L or l	A lower triangular matrix.

trans

CHARACTER*1. Specifies the operation to be performed, as follows:

***trans* value**	**Operation to be Performed**
N or n	*A***x* = *b*
T or t	*A*'**x* = *b*
C or c	conjg(*A*')**x* = *b*

diag

CHARACTER*1. Specifies whether or not *A* is unit triangular, as follows:

***diag* value**	**Matrix *A***
U or u	Matrix *A* is a unit triangular.
N or n	Matrix *A* is not a unit triangular.

n

INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

k

INTEGER. On entry with *uplo* = 'U' or 'u', *k* specifies the number of super-diagonals of the matrix *A*. On entry with *uplo* = 'L' or 'l', *k* specifies the number of sub-diagonals of the matrix *A*.

The value of *k* must satisfy 0 ≤ *k*.

a

REAL for stbsv
DOUBLE PRECISION for dtbsv
COMPLEX for ctbsv
DOUBLE COMPLEX for ztbsv
Array, DIMENSION (*lda*, *n*).
Before entry with *uplo* = 'U' or 'u', the leading (*k* + 1) by *n* part of the array *a* must contain the upper triangular band part of the matrix of coefficients, supplied

column-by-column, with the leading diagonal of the matrix in row (*k* + 1) of the array, the first super-diagonal starting at position 2 in row *k*, and so on. The top left *k* by *k* triangle of the array *a* is not referenced.
The following program segment transfers an upper triangular band matrix from conventional full matrix storage to band storage:

```
do 20, j = 1, n
  m = k + 1 - j
  do 10, i = max(1, j - k), jl
    a(m + i, j) = matrix (i, j)
   10 continue
20 continue
```

Before entry with *uplo* = 'L' or 'l', the leading (*k* + 1) by *n* part of the array *a* must contain the lower triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right *k* by *k* triangle of the array *a* is not referenced.
The following program segment transfers a lower triangular band matrix from conventional full matrix storage to band storage:

```
do 20, j = 1, n
   m = 1 - j
   do 10, i = j, min(n, j + k)
    a(m + i, j) = matrix (i, j)
  10 continue
20 continue
```

When *diag* = 'U' or 'u', the elements of the array *a* corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity.

lda	INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least (*k* + 1).
x	REAL for stbsv DOUBLE PRECISION for dtbsv COMPLEX for ctbsv DOUBLE COMPLEX for ztbsv Array, DIMENSION at least (1 + (*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element right-hand side vector *b*.
incx	INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

Output Parameters

x	Overwritten with the solution vector *x*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine tbsv interface are the following:

a	Holds the array *a* of size (*k+1*,*n*).
x	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
diag	Must be 'N' or 'U'. The default value is 'N'.

?tpmv

Computes a matrix-vector product using a triangular packed matrix.

Syntax

Fortran 77:

```
call stpmv(uplo, trans, diag, n, ap, x, incx )
call dtpmv(uplo, trans, diag, n, ap, x, incx )
call ctpmv(uplo, trans, diag, n, ap, x, incx )
call ztpmv(uplo, trans, diag, n, ap, x, incx )
```

Fortran 95:

```
call tpmv(a, x [,uplo] [, trans] [,diag])
```

Description

The ?tpmv routines perform one of the matrix-vector operations defined as

x := *A***x*, or *x* := *A*'**x*, or *x* := conjg(*A*')**x*,

where:

x is an *n*-element vector,

A is an *n*-by-*n* unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

Input Parameters

uplo — CHARACTER*1. Specifies whether the matrix *A* is an upper or lower triangular matrix, as follows:

uplo value	Matrix *A*
U or u	An upper triangular matrix.
L or l	A lower triangular matrix.

trans CHARACTER*1. Specifies the operation to be performed, as follows:

trans value	Operation To Be Performed
N or n	*x* := *A***x*
T or t	*x* := *A*'**x*
C or c	*x* := conjg(*A*')**x*

diag CHARACTER*1. Specifies whether or not *A* is unit triangular, as follows:

diag value	Matrix *A*
U or u	Matrix *A* is assumed to be unit triangular.
N or n	Matrix *A* is not assumed to be unit triangular.

n INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

ap REAL for stpmv
DOUBLE PRECISION for dtpmv
COMPLEX for ctpmv
DOUBLE COMPLEX for ztpmv
Array, DIMENSION at least ((*n**(*n* + 1))/2). Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1,1), *ap*(2) and *ap*(3) contain *a*(1,2) and *a*(2,2) respectively, and so on. Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1,1), *ap*(2) and *ap*(3) contain *a*(2,1) and *a*(3,1) respectively, and so on. When *diag* = 'U' or 'u', the diagonal elements of *a* are not referenced, but are assumed to be unity.

x	REAL for stpmv DOUBLE PRECISION for dtpmv COMPLEX for ctpmv DOUBLE COMPLEX for ztpmv Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.
incx	INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

Output Parameters

x	Overwritten with the transformed vector *x*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine tpmv interface are the following:

a	Holds the array *a* of size (*n**(n+1)/2).
x	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
diag	Must be 'N' or 'U'. The default value is 'N'.

?tpsv

Solves a system of linear equations whose coefficients are in a triangular packed matrix.

Syntax

Fortran 77:

```
call stpsv(uplo, trans, diag, n, ap, x, incx )
call dtpsv(uplo, trans, diag, n, ap, x, incx )
call ctpsv(uplo, trans, diag, n, ap, x, incx )
call ztpsv(uplo, trans, diag, n, ap, x, incx )
```

Fortran 95:

```
call tpsv(a, x [,uplo] [, trans] [,diag])
```

Description

The ?tpsv routines solve one of the following systems of equations

*A*x = b*, or *A'*x = b*, or conj(*A'*)*x = b*,

where:

b and *x* are *n*-element vectors,

A is an *n*-by-*n* unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

This routine does not test for singularity or near-singularity.

Such tests must be performed before calling this routine.

Input Parameters

uplo — CHARACTER*1. Specifies whether the matrix *A* is an upper or lower triangular matrix, as follows:

***uplo* value**	**Matrix *A***
U or u	An upper triangular matrix.
L or l	A lower triangular matrix.

trans — CHARACTER*1. Specifies the operation to be performed, as follows:

trans value	Operation To Be Performed
N or n	*A***x* = *b*
T or t	*A*'**x* = *b*
C or c	conjg(*A*')**x* = *b*

diag CHARACTER*1. Specifies whether or not the matrix *A* is unit triangular, as follows:

diag value	Matrix *A*
U or u	Matrix *A* is assumed to be unit triangular.
N or n	Matrix *A* is not assumed to be unit triangular.

n INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

ap REAL for stpsv
DOUBLE PRECISION for dtpsv
COMPLEX for ctpsv
DOUBLE COMPLEX for ztpsv
Array, DIMENSION at least ((*n**(*n* + 1))/2). Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, +1), *ap*(2) and *ap*(3) contain *a*(1, 2) and *a*(2, 2) respectively, and so on.
Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, +1), *ap*(2) and *ap*(3) contain *a*(2, +1) and *a*(3, +1) respectively, and so on.
When *diag* = 'U' or 'u', the diagonal elements of *a* are not referenced, but are assumed to be unity.

x REAL for stpsv
DOUBLE PRECISION for dtpsv
COMPLEX for ctpsv
DOUBLE COMPLEX for ztpsv

	Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element right-hand side vector *b*.
incx	INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

Output Parameters

x	Overwritten with the solution vector *x*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine tpsv interface are the following:

a	Holds the array *a* of size (*n**(n+1)/2).
x	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
diag	Must be 'N' or 'U'. The default value is 'N'.

?trmv

Computes a matrix-vector product using a triangular matrix.

Syntax

Fortran 77:

```
call strmv(uplo, trans, diag, n, a, lda, x, incx )
call dtrmv(uplo, trans, diag, n, a, lda, x, incx )
call ctrmv(uplo, trans, diag, n, a, lda, x, incx )
call ztrmv(uplo, trans, diag, n, a, lda, x, incx )
```

Fortran 95:

```
call trmv(a, x [,uplo] [, trans] [,diag])
```

Description

The ?trmv routines perform one of the following matrix-vector operations defined as

x := *A***x*, or *x* := A'**x*, or *x* := conjg(*A*')**x*,

where:

x is an *n*-element vector,

A is an *n*-by-*n* unit, or non-unit, upper or lower triangular matrix.

Input Parameters

uplo — CHARACTER*1. Specifies whether the matrix *A* is an upper or lower triangular matrix, as follows:

***uplo* value**	**Matrix *A***
U or u	An upper triangular matrix.
L or l	A lower triangular matrix.

trans — CHARACTER*1. Specifies the operation to be performed, as follows:

***trans* value**	**Operation To Be Performed**
N or n	*x* := *A***x*
T or t	*x* := *A*'**x*
C or c	*x* := conjg(*A*')**x*

diag — CHARACTER*1. Specifies whether or not *A* is unit triangular, as follows:

***diag* value**	**Matrix *A***
U or u	Matrix *A* is assumed to be unit triangular.
N or n	Matrix *A* is not assumed to be unit triangular.

n — INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

a — REAL for strmv

	DOUBLE PRECISION for dtrmv COMPLEX for ctrmv DOUBLE COMPLEX for ztrmv Array, DIMENSION (*lda*, *n*). Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular matrix and the strictly lower triangular part of *a* is not referenced. Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular matrix and the strictly upper triangular part of *a* is not referenced. When *diag* = 'U' or 'u', the diagonal elements of *a* are not referenced either, but are assumed to be unity.
lda	INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *n*).
x	REAL for strmv DOUBLE PRECISION for dtrmv COMPLEX for ctrmv DOUBLE COMPLEX for ztrmv Array, DIMENSION at least (1 + (*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.
incx	INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

Output Parameters

x	Overwritten with the transformed vector *x*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine trmv interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
x	Holds the vector of length (*n*).

uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
diag	Must be 'N' or 'U'. The default value is 'N'.

?trsv

Solves a system of linear equations whose coefficients are in a triangular matrix.

Syntax

Fortran 77:

```
call strsv(uplo, trans, diag, n, a, lda, x, incx )
call dtrsv(uplo, trans, diag, n, a, lda, x, incx )
call ctrsv(uplo, trans, diag, n, a, lda, x, incx )
call ztrsv(uplo, trans, diag, n, a, lda, x, incx )
```

Fortran 95:

```
call trsv(a, x [,uplo] [, trans] [,diag])
```

Description

The ?trsv routines solve one of the systems of equations:

*A*x = b*, or *A'*x = b*, or conjg(*A'*)*x = b*,

where:

b and *x* are *n*-element vectors,

A is an *n*-by-*n* unit, or non-unit, upper or lower triangular matrix.

The routine does not test for singularity or near-singularity.

Such tests must be performed before calling this routine.

Input Parameters

uplo	CHARACTER*1. Specifies whether the matrix is an upper or lower triangular matrix, as follows:

	***uplo* value**	**Matrix *A***
	`U` or `u`	An upper triangular matrix.
	`L` or `l`	A lower triangular matrix.

trans `CHARACTER*1`. Specifies the operation to be performed, as follows:

***trans* value**	**Operation To Be Performed**
`N` or `n`	*A***x* = *b*
`T` or `t`	*A*'**x* = *b*
`C` or `c`	conjg(*A*')**x* = *b*

diag `CHARACTER*1`. Specifies whether or not *A* is unit triangular, as follows:

***diag* value**	**Matrix *a***
`U` or `u`	Matrix *a* is a unit triangular.
`N` or `n`	Matrix *a* is not a unit triangular.

n `INTEGER`. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

a `REAL` for `strsv`
`DOUBLE PRECISION` for `dtrsv`
`COMPLEX` for `ctrsv`
`DOUBLE COMPLEX` for `ztrsv`
Array, `DIMENSION (`*lda*`,`*n*`)`. Before entry with *uplo* = `'U'` or `'u'`, the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular matrix and the strictly lower triangular part of *a* is not referenced. Before entry with *uplo* = `'L'` or `'l'`, the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular matrix and the strictly upper triangular part of *a* is not referenced. When *diag* = `'U'` or `'u'`, the diagonal elements of *a* are not referenced either, but are assumed to be unity.

lda `INTEGER`. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least `max(1,` *n*`)`.

x `REAL` for `strsv`

	DOUBLE PRECISION for dtrsv COMPLEX for ctrsv DOUBLE COMPLEX for ztrsv Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element right-hand side vector *b*.
incx	INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

Output Parameters

x	Overwritten with the solution vector *x*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine trsv interface are the following:

a	Holds the matrix *a* of size (*n*,*n*).
x	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
diag	Must be 'N' or 'U'. The default value is 'N'.

BLAS Level 3 Routines

BLAS Level 3 routines perform matrix-matrix operations. Table 2-3 lists the BLAS Level 3 routine groups and the data types associated with them.

Table 2-3 BLAS Level 3 Routine Groups and Their Data Types

Routine Group	Data Types	Description
?gemm	s, d, c, z	Matrix-matrix product of general matrices
?hemm	c, z	Matrix-matrix product of Hermitian matrices

Routine Group	Data Types	Description
?herk	c, z	Rank-k update of Hermitian matrices
?her2k	c, z	Rank-2k update of Hermitian matrices
?symm	s, d, c, z	Matrix-matrix product of symmetric matrices
?syrk	s, d, c, z	Rank-k update of symmetric matrices
?symm	s, d, c, z	Rank-2k update of symmetric matrices
?trmm	s, d, c, z	Matrix-matrix product of triangular matrices
?trsm	s, d, c, z	Linear matrix-matrix solution for triangular matrices

Symmetric Multiprocessing Version of Intel® MKL

Many applications spend considerable time for executing BLAS level 3 routines. This time can be scaled by the number of processors available on the system through using the symmetric multiprocessing(SMP) feature built into the Intel MKL Library. The performance enhancements based on the parallel use of the processors are available without any programming effort on your part.

To enhance performance, the library uses the following methods:

- The operation of BLAS level 3 matrix-matrix functions permits to restructure the code in a way which increases the localization of data reference, enhances cache memory use, and reduces the dependency on the memory bus.
- Once the code has been effectively blocked as described above, one of the matrices is distributed across the processors to be multiplied by the second matrix. Such distribution ensures effective cache management which reduces the dependency on the memory bus performance and brings good scaling results.

?gemm

Computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product.

Syntax

Fortran 77:

```
call sgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call dgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call cgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call zgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
```

Fortran 95:

```
call gemm(a, b, c [,transa][,transb] [,alpha][,beta])
```

Description

The ?gemm routines perform a matrix-matrix operation with general matrices. The operation is defined as

`C := alpha*op(A)*op(B) + beta*C`,

where:

`op(x)` is one of `op(x) = x`, or `op(x) = x'`, or `op(x) = conjg(x')`,

alpha and *beta* are scalars,

A, *B* and *C* are matrices:

`op(A)` is an *m*-by-*k* matrix,

`op(B)` is a *k*-by-*n* matrix,

C is an *m*-by-*n* matrix.

Input Parameters

transa	CHARACTER*1. Specifies the form of `op(A)` to be used in the matrix multiplication as follows:

transa value	Form of op(*a*)
N or n	op(*A*) = *A*
T or t	op(*A*) = *A*'
C or c	op(*A*) = conjg(*A*')

transb CHARACTER*1. Specifies the form of op(*B*) to be used in the matrix multiplication as follows:

transb value	Form of op(*b*)
N or n	op(*B*) = *B*
T or t	op(*B*) = *B*'
C or c	op(*B*) = conjg(*B*')

m INTEGER. Specifies the number of rows of the matrix op(*A*) and of the matrix *C*. The value of *m* must be at least zero.

n INTEGER. Specifies the number of columns of the matrix op(*B*) and the number of columns of the matrix *C*. The value of *n* must be at least zero.

k INTEGER. Specifies the number of columns of the matrix op(*A*) and the number of rows of the matrix op(*B*). The value of *k* must be at least zero.

alpha REAL for sgemm
DOUBLE PRECISION for dgemm
COMPLEX for cgemm
DOUBLE COMPLEX for zgemm
Specifies the scalar *alpha*.

a REAL for sgemm
DOUBLE PRECISION for dgemm
COMPLEX for cgemm
DOUBLE COMPLEX for zgemm
Array, DIMENSION (*lda*, *ka*), where *ka* is *k* when *transa*= 'N' or 'n', and is *m* otherwise. Before entry with *transa*= 'N' or 'n', the leading *m*-by-*k* part of the array *a* must contain the matrix *A*, otherwise the leading *k*-by-*m* part of the array *a* must contain the matrix *A*.

lda INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. When *transa*= 'N' or 'n', then *lda* must be at least max(1, *m*), otherwise *lda* must be at least max(1, *k*).

b REAL for sgemm
DOUBLE PRECISION for dgemm
COMPLEX for cgemm
DOUBLE COMPLEX for zgemm
Array, DIMENSION (*ldb*, *kb*), where *kb* is *n* when *transb* = 'N' or 'n', and is *k* otherwise. Before entry with *transb* = 'N' or 'n', the leading *k*-by-*n* part of the array *b* must contain the matrix *B*, otherwise the leading *n*-by-*k* part of the array *b* must contain the matrix *B*.

ldb INTEGER. Specifies the first dimension of *b* as declared in the calling (sub)program. When *transb* = 'N' or 'n', then *ldb* must be at least max(1, *k*), otherwise *ldb* must be at least max(1, *n*).

beta REAL for sgemm
DOUBLE PRECISION for dgemm
COMPLEX for cgemm
DOUBLE COMPLEX for zgemm
Specifies the scalar *beta*.
When *beta* is equal to zero, then *c* need not be set on input.

c REAL for sgemm
DOUBLE PRECISION for dgemm
COMPLEX for cgemm
DOUBLE COMPLEX for zgemm
Array, DIMENSION (*ldc*, *n*).
Before entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, except when *beta* is equal to zero, in which case *c* need not be set on entry.

ldc INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least max(1, *m*).

Output Parameters

c	Overwritten by the *m*-by-*n* matrix (*alpha**op(*A*)*op(*B*) + *beta***C*).

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gemm interface are the following:

a	Holds the matrix *A* of size (*ma*,*ka*) where *ka* = *k* if *transa*= 'N', *ka* = *m* otherwise, *ma* = *m* if *transa*= 'N', *ma* = *k* otherwise.
b	Holds the matrix *B* of size (*mb*,*kb*) where *kb* = *n* if *transb* = 'N', *kb* = *k* otherwise, *mb* = *k* if *transb* = 'N', *mb* = *n* otherwise.
c	Holds the matrix *C* of size (*m*,*n*).
transa	Must be 'N', 'C', or 'T'. The default value is 'N'.
transb	Must be 'N', 'C', or 'T'. The default value is 'N'.
alpha	The default value is 1.
beta	The default value is 1.

?hemm

Computes a scalar-matrix-matrix product (either one of the matrices is Hermitian) and adds the result to scalar-matrix product.

Syntax

Fortran 77:

```
call chemm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc )
call zhemm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc )
```

Fortran 95:

```
call hemm(a, b, c [,side][,uplo] [,alpha][,beta])
```

Description

The ?hemm routines perform a matrix-matrix operation using Hermitian matrices. The operation is defined as

*C := alpha*A*B + beta*C*

or

*C := alpha*B*A + beta*C*,

where:

alpha and *beta* are scalars,

A is an Hermitian matrix,

B and *C* are *m*-by-*n* matrices.

Input Parameters

side CHARACTER*1. Specifies whether the Hermitian matrix *A* appears on the left or right in the operation as follows:

side value	Operation To Be Performed
L or l	*C := alpha*A*B + beta*C*
R or r	*C := alpha*B*A + beta*C*

uplo CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian matrix *A* is to be referenced as follows:

uplo **value**	**Part of Matrix *A* To Be Referenced**
U or u	Only the upper triangular part of the Hermitian matrix is to be referenced.
L or l	Only the lower triangular part of the Hermitian matrix is to be referenced.

m INTEGER. Specifies the number of rows of the matrix *C*. The value of *m* must be at least zero.

n INTEGER. Specifies the number of columns of the matrix *C*. The value of *n* must be at least zero.

alpha COMPLEX for chemm
DOUBLE COMPLEX for zhemm
Specifies the scalar *alpha*.

a COMPLEX for chemm
DOUBLE COMPLEX for zhemm
Array, DIMENSION (*lda*,*ka*), where *ka* is *m* when *side* = 'L' or 'l' and is *n* otherwise. Before entry with *side* = 'L' or 'l', the *m*-by-*m* part of the array *a* must contain the Hermitian matrix, such that when *uplo* = 'U' or 'u', the leading *m*-by-*m* upper triangular part of the array *a* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *a* is not referenced, and when *uplo* = 'L' or 'l', the leading *m*-by-*m* lower triangular part of the array *a* must contain the lower triangular part of the Hermitian matrix, and the strictly upper triangular part of *a* is not referenced.
Before entry with *side* = 'R' or 'r', the *n*-by-*n* part of the array *a* must contain the Hermitian matrix, such that when *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *a* is not referenced, and when *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the

Hermitian matrix, and the strictly upper triangular part of *a* is not referenced. The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.

lda — INTEGER. Specifies the first dimension of *a* as declared in the calling (sub) program. When *side* = 'L' or 'l' then *lda* must be at least max(1, *m*), otherwise *lda* must be at least max(1,*n*).

b — COMPLEX for chemm
DOUBLE COMPLEX for zhemm
Array, DIMENSION (*ldb*,*n*).
Before entry, the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb — INTEGER. Specifies the first dimension of *b* as declared in the calling (sub)program. The value of *ldb* must be at least max(1, *m*).

beta — COMPLEX for chemm
DOUBLE COMPLEX for zhemm
Specifies the scalar *beta*.
When *beta* is supplied as zero, then *c* need not be set on input.

c — COMPLEX for chemm
DOUBLE COMPLEX for zhemm
Array, DIMENSION (*c*, *n*). Before entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, except when *beta* is zero, in which case *c* need not be set on entry.

ldc — INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least max(1, *m*).

Output Parameters

c — Overwritten by the *m*-by-*n* updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hemm interface are the following:

a	Holds the matrix *A* of size (*k*,*k*) where *k* = *m* if *side* = 'L', *k* = *n* otherwise.
b	Holds the matrix *B* of size (*m*,*n*).
c	Holds the matrix *C* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
uplo	Must be 'U' or 'L'. The default value is 'U'.
alpha	The default value is 1.
beta	The default value is 1.

?herk

Performs a rank-n update of a Hermitian matrix.

Syntax

Fortran 77:

```
call cherk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc )
call zherk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc )
```

Fortran 95:

```
call herk(a, c [,uplo] [, trans] [,alpha][,beta])
```

Description

The ?herk routines perform a matrix-matrix operation using Hermitian matrices. The operation is defined as

C := *alpha***A**conjg(*A*') + *beta***C*,

or

C := *alpha**conjg(*A*')**A* + *beta***C*,

where:

alpha and *beta* are real scalars,

C is an *n*-by-*n* Hermitian matrix,

A is an *n*-by-*k* matrix in the first case and a *k*-by-*n* matrix in the second case.

Input Parameters

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the array *c* is to be referenced as follows:

***uplo* value**	**Part of Array *c* To Be Referenced**
U or u	Only the upper triangular part of *c* is to be referenced.
L or l	Only the lower triangular part of *c* is to be referenced.

trans

CHARACTER*1. Specifies the operation to be performed as follows:

***trans* value**	**Operation to be Performed**
N or n	*C*:= *alpha***A**conjg(*A*')+*beta***C*
C or c	*C*:= *alpha**conjg(*A*')**A*+*beta***C*

n

INTEGER. Specifies the order of the matrix *C*. The value of *n* must be at least zero.

k

INTEGER. With *trans* = 'N' or 'n', *k* specifies the number of columns of the matrix *A*, and with *trans* = 'C' or 'c', *k* specifies the number of rows of the matrix *A*.
The value of *k* must be at least zero.

alpha

REAL for cherk
DOUBLE PRECISION for zherk
Specifies the scalar *alpha*.

a

COMPLEX for cherk
DOUBLE COMPLEX for zherk
Array, DIMENSION (*lda*, *ka*), where *ka* is *k* when *trans* = 'N' or 'n', and is *n* otherwise. Before entry with *trans* = 'N' or 'n', the leading *n*-by-*k* part of the array *a* must contain the matrix *a*, otherwise the leading *k*-by-*n* part of the array *a* must contain the matrix *A*.

lda	INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. When *trans* = 'N' or 'n', then *lda* must be at least max(1, *n*), otherwise *lda* must be at least max(1, *k*).
beta	REAL for cherk DOUBLE PRECISION for zherk Specifies the scalar *beta*.
c	COMPLEX for cherk DOUBLE COMPLEX for zherk Array, DIMENSION (*ldc*,*n*). Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *c* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *c* is not referenced. Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *c* must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of *c* is not referenced. The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.
ldc	INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least max(1, *n*).

Output Parameters

c	With *uplo* = 'U' or 'u', the upper triangular part of the array *c* is overwritten by the upper triangular part of the updated matrix. With *uplo* = 'L' or 'l', the lower triangular part of the array *c* is overwritten by the lower triangular part of the updated matrix. The imaginary parts of the diagonal elements are set to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine herk interface are the following:

a	Holds the matrix *A* of size (*ma*,*ka*) where *ka* = *k* if *transa*= 'N', *ka* = *n* otherwise, *ma* = *n* if *transa*= 'N', *ma* = *k* otherwise.
c	Holds the matrix *C* of size (*n*,*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N' or 'C'. The default value is 'N'.
alpha	The default value is 1.
beta	The default value is 1.

?her2k

Performs a rank-2k update of a Hermitian matrix.

Syntax

Fortran 77:

```
call cher2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc )
call zher2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc )
```

Fortran 95:

```
call her2k(a, b, c [,uplo][,trans] [,alpha][,beta])
```

Description

The ?her2k routines perform a rank-2k matrix-matrix operation using Hermitian matrices. The operation is defined as

C := *alpha***A**conjg(*B*') + conjg(*alpha*)**B**conjg(*A*') + *beta***C*,

or

C := *alpha* *conjg(*B*')*A* + conjg(*alpha*) *conjg(*A*')*B* + *beta*C*,

where:

alpha is a scalar and *beta* is a real scalar,

C is an *n*-by-*n* Hermitian matrix,

A and *B* are *n*-by-*k* matrices in the first case and *k*-by-*n* matrices in the second case.

Input Parameters

uplo — CHARACTER*1. Specifies whether the upper or lower triangular part of the array *c* is to be referenced as follows:

***uplo* value**	**Part of Array *c* To Be Referenced**
U or u	Only the upper triangular part of *c* is to be referenced.
L or l	Only the lower triangular part of *c* is to be referenced.

trans — CHARACTER*1. Specifies the operation to be performed as follows:

***trans* value**	**Operation to be Performed**
N or n	*C*:=*alpha**A**conjg(*B*') + *alpha**B**conjg(*A*') + *beta**C*
C or c	*C*:=*alpha**conjg(*A*')*B* + *alpha**conjg(*B*')*A* + *beta**C*

n — INTEGER. Specifies the order of the matrix *C*. The value of *n* must be at least zero.

k — INTEGER. With *trans* = 'N' or 'n', *k* specifies the number of columns of the matrix *A*, and with *trans* = 'C' or 'c', *k* specifies the number of rows of the matrix *A*.
The value of *k* must be at least equal to zero.

alpha — COMPLEX for cher2k
DOUBLE COMPLEX for zher2k
Specifies the scalar *alpha*.

a — COMPLEX for cher2k
DOUBLE COMPLEX for zher2k

Array, DIMENSION (*lda*, *ka*), where *ka* is *k* when *trans* = 'N' or 'n', and is *n* otherwise. Before entry with *trans* = 'N' or 'n', the leading *n*-by-*k* part of the array *a* must contain the matrix *A*, otherwise the leading *k*-by-*n* part of the array *a* must contain the matrix *A*.

lda — INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. When *trans* = 'N' or 'n', then *lda* must be at least max(1, *n*), otherwise *lda* must be at least max(1, *k*).

beta — REAL for cher2k
DOUBLE PRECISION for zher2k
Specifies the scalar *beta*.

b — COMPLEX for cher2k
DOUBLE COMPLEX for zher2k
Array, DIMENSION (*ldb*, *kb*), where *kb* is *k* when *trans* = 'N' or 'n', and is *n* otherwise. Before entry with *trans* = 'N' or 'n', the leading *n*-by-*k* part of the array *b* must contain the matrix *B*, otherwise the leading *k*-by-*n* part of the array *b* must contain the matrix *B*.

ldb — INTEGER. Specifies the first dimension of *b* as declared in the calling (sub)program. When *trans* = 'N' or 'n', then *ldb* must be at least max(1, *n*), otherwise *ldb* must be at least max(1, *k*).

c — COMPLEX for cher2k
DOUBLE COMPLEX for zher2k
Array, DIMENSION (*ldc*,*n*).
Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *c* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *c* is not referenced.
Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *c* must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of *c* is not referenced.
The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.

ldc	INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least max(1, *n*).

Output Parameters

c	With *uplo* = 'U' or 'u', the upper triangular part of the array *c* is overwritten by the upper triangular part of the updated matrix. With *uplo* = 'L' or 'l', the lower triangular part of the array *c* is overwritten by the lower triangular part of the updated matrix. The imaginary parts of the diagonal elements are set to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine her2k interface are the following:

a	Holds the matrix *A* of size (*ma*,*ka*) where *ka* = *k* if *trans* = 'N', *ka* = *n* otherwise, *ma* = *n* if *trans* = 'N', *ma* = *k* otherwise.
b	Holds the matrix *B* of size (*mb*,*kb*) where *kb* = *k* if *trans* = 'N', *kb* = *n* otherwise, *mb* = *n* if *trans* = 'N', *mb* = *k* otherwise.
c	Holds the matrix *C* of size (*n*,*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N' or 'C'. The default value is 'N'.
alpha	The default value is 1.
beta	The default value is 1.

?symm

Performs a scalar-matrix-matrix product(one matrix operand is symmetric) and adds the result to a scalar-matrix product.

Syntax

Fortran 77:

```
call ssymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc )
call dsymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc )
call csymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc )
call zsymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc )
```

Fortran 95:

```
call symm(a, b, c [,side][,uplo] [,alpha][,beta])
```

Description

The ?symm routines perform a matrix-matrix operation using symmetric matrices. The operation is defined as

*C := alpha*A*B + beta*C*,

or

*C := alpha*B*A + beta*C*,

where:

alpha and *beta* are scalars,

A is a symmetric matrix,

B and *C* are *m*-by-*n* matrices.

Input Parameters

side CHARACTER*1. Specifies whether the symmetric matrix *A* appears on the left or right in the operation as follows:

side value	Operation to be Performed
L or l	*Ac := alpha*A*B + beta*C*

	R or r	*c* := *alpha***B***A* + *beta***C*

uplo CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix *A* is to be referenced as follows:

uplo **value**	**Part of Array *a* To Be Referenced**
U or u	Only the upper triangular part of the symmetric matrix is to be referenced.
L or l	Only the lower triangular part of the symmetric matrix is to be referenced.

m INTEGER. Specifies the number of rows of the matrix *C*. The value of *m* must be at least zero.

n INTEGER. Specifies the number of columns of the matrix *C*. The value of *n* must be at least zero.

alpha
REAL for ssymm
DOUBLE PRECISION for dsymm
COMPLEX for csymm
DOUBLE COMPLEX for zsymm
Specifies the scalar *alpha*.

a
REAL for ssymm
DOUBLE PRECISION for dsymm
COMPLEX for csymm
DOUBLE COMPLEX for zsymm
Array, DIMENSION (*lda*, *ka*), where *ka* is *m* when *side* = 'L' or 'l' and is *n* otherwise.
Before entry with *side* = 'L' or 'l', the *m*-by-*m* part of the array *a* must contain the symmetric matrix, such that when *uplo* = 'U' or 'u', the leading *m*-by-*m* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *a* is not referenced, and when *uplo* = 'L' or 'l', the leading *m*-by-*m* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *a* is not referenced.

Before entry with *side* = 'R' or 'r', the *n*-by-*n* part of the array *a* must contain the symmetric matrix, such that when *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *a* is not referenced, and when *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *a* is not referenced.

lda INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. When *side* = 'L' or 'l' then *lda* must be at least max(1, *m*), otherwise *lda* must be at least max(1, *n*).

b REAL for ssymm
DOUBLE PRECISION for dsymm
COMPLEX for csymm
DOUBLE COMPLEX for zsymm
Array, DIMENSION (*ldb*, *n*). Before entry, the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb INTEGER. Specifies the first dimension of *b* as declared in the calling (sub)program. The value of *ldb* must be at least max(1, *m*).

beta REAL for ssymm
DOUBLE PRECISION for dsymm
COMPLEX for csymm
DOUBLE COMPLEX for zsymm
Specifies the scalar *beta*.
When *beta* is set to zero, then *c* need not be set on input.

c REAL for ssymm
DOUBLE PRECISION for dsymm
COMPLEX for csymm
DOUBLE COMPLEX for zsymm
Array, DIMENSION (*ldc*,*n*). Before entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, except when beta is zero, in which case *c* need not be set on entry.

ldc	INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least max(1, *m*).

Output Parameters

c	Overwritten by the *m*-by-*n* updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine symm interface are the following:

a	Holds the matrix *A* of size (*k*,*k*) where *k* = *m* if *side* = 'L', *k* = *n* otherwise.
b	Holds the matrix *B* of size (*m*,*n*).
c	Holds the matrix *C* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
uplo	Must be 'U' or 'L'. The default value is 'U'.
alpha	The default value is 1.
beta	The default value is 1.

?syrk

Performs a rank-n update of a symmetric matrix.

Syntax

Fortran 77:

```
call ssyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc )
call dsyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc )
call csyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc )
call zsyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc )
```

Fortran 95:

```
call syrk(a, c [,uplo] [, trans] [,alpha][,beta])
```

Description

The ?syrk routines perform a matrix-matrix operation using symmetric matrices. The operation is defined as

C := *alpha***A***A*' + *beta***C*,

or

C := *alpha***A*'**A* + *beta***C*,

where:

alpha and *beta* are scalars,

C is an *n*-by-*n* symmetric matrix,

A is an *n*-by-*k* matrix in the first case and a *k*-by-*n* matrix in the second case.

Input Parameters

uplo — CHARACTER*1. Specifies whether the upper or lower triangular part of the array *c* is to be referenced as follows:

uplo value	Part of Array *c* To Be Referenced
U or u	Only the upper triangular part of *c* is to be referenced.
L or l	Only the lower triangular part of *c* is to be referenced.

trans — CHARACTER*1. Specifies the operation to be performed as follows:

trans value	Operation to be Performed
N or n	*C* := *alpha***A***A*' + *beta***C*
T or t	*C* := *alpha***A*'**A* + *beta***C*
C or c	*C* := *alpha***A*'**A* + *beta***C*

n — INTEGER. Specifies the order of the matrix *C*. The value of *n* must be at least zero.

k	INTEGER. On entry with *trans* = 'N' or 'n', *k* specifies the number of columns of the matrix *a*, and on entry with *trans* = 'T' or 't' or 'C' or 'c', *k* specifies the number of rows of the matrix *a*. The value of *k* must be at least zero.
alpha	REAL for ssyrk DOUBLE PRECISION for dsyrk COMPLEX for csyrk DOUBLE COMPLEX for zsyrk Specifies the scalar *alpha*.
a	REAL for ssyrk DOUBLE PRECISION for dsyrk COMPLEX for csyrk DOUBLE COMPLEX for zsyrk Array, DIMENSION (*lda*,*ka*), where *ka* is *k* when *trans* = 'N' or 'n', and is *n* otherwise. Before entry with *trans* = 'N' or 'n', the leading *n*-by-*k* part of the array *a* must contain the matrix *A*, otherwise the leading *k*-by-*n* part of the array *a* must contain the matrix *A*.
lda	INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. When *trans* = 'N' or 'n', then *lda* must be at least max(1,*n*), otherwise *lda* must be at least max(1, *k*).
beta	REAL for ssyrk DOUBLE PRECISION for dsyrk COMPLEX for csyrk DOUBLE COMPLEX for zsyrk Specifies the scalar *beta*.
c	REAL for ssyrk DOUBLE PRECISION for dsyrk COMPLEX for csyrk DOUBLE COMPLEX for zsyrk Array, DIMENSION (*ldc*,*n*). Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *c* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *c* is not referenced.

Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *c* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *c* is not referenced.

ldc — INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least max(1, *n*).

Output Parameters

c — With *uplo* = 'U' or 'u', the upper triangular part of the array *c* is overwritten by the upper triangular part of the updated matrix.
With *uplo* = 'L' or 'l', the lower triangular part of the array *c* is overwritten by the lower triangular part of the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine syrk interface are the following:

a — Holds the matrix *A* of size (*ma*,*ka*) where
ka = *k* if *transa*= 'N',
ka = *n* otherwise,
ma = *n* if *transa*= 'N',
ma = *k* otherwise.

c — Holds the matrix *C* of size (*n*,*n*).

uplo — Must be 'U' or 'L'. The default value is 'U'.

trans — Must be 'N', 'C', or 'T'.
The default value is 'N'.

alpha — The default value is 1.

beta — The default value is 1.

?syr2k

Performs a rank-2k update of a symmetric matrix.

Syntax

Fortran 77:

```
call ssyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc )
call dsyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc )
call csyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc )
call zsyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc )
```

Fortran 95:

```
call syr2k(a, b, c [,uplo][,trans] [,alpha][,beta])
```

Description

The ?syr2k routines perform a rank-2k matrix-matrix operation using symmetric matrices. The operation is defined as

C := *alpha***A***B*' + *alpha***B***A*' + *beta***C*,

or

C := *alpha***a*'**B* + *alpha***B*'**A* + *beta***C*,

where:

alpha and *beta* are scalars,

C is an *n*-by-*n* symmetric matrix,

A and *B* are *n*-by-*k* matrices in the first case, and *k*-by-*n* matrices in the second case.

Input Parameters

uplo — CHARACTER*1. Specifies whether the upper or lower triangular part of the array *c* is to be referenced as follows:

uplo value	Part of Array *c* To Be Referenced
U or u	Only the upper triangular part of *c* is to be referenced.

	L or l — Only the lower triangular part of *c* is to be referenced.
trans	CHARACTER*1. Specifies the operation to be performed as follows:

***trans* value**	**Operation to be Performed**
N or n	*C* := *alpha***A***B*'+*alpha***B***A*'+*beta***C*
T or t	*C* := *alpha***A*'**B* +*alpha***B*'**A* +*beta***C*
C or c	*C* := *alpha***A*'**B* +*alpha***B*'**A* +*beta***C*

n	INTEGER. Specifies the order of the matrix *C*. The value of *n* must be at least zero.
k	INTEGER. On entry with *trans* = 'N' or 'n', *k* specifies the number of columns of the matrices *A* and *B*, and on entry with *trans* = 'T' or 't' or 'C' or 'c', *k* specifies the number of rows of the matrices *A* and *B*. The value of *k* must be at least zero.
alpha	REAL for ssyr2k DOUBLE PRECISION for dsyr2k COMPLEX for csyr2k DOUBLE COMPLEX for zsyr2k Specifies the scalar *alpha*.
a	REAL for ssyr2k DOUBLE PRECISION for dsyr2k COMPLEX for csyr2k DOUBLE COMPLEX for zsyr2k Array, DIMENSION (*lda*,*ka*), where *ka* is *k* when *trans* = 'N' or 'n', and is *n* otherwise. Before entry with *trans* = 'N' or 'n', the leading *n*-by-*k* part of the array *a* must contain the matrix *A*, otherwise the leading *k*-by-*n* part of the array *a* must contain the matrix *A*.
lda	INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. When *trans* = 'N' or 'n', then *lda* must be at least max(1, *n*), otherwise *lda* must be at least max(1, *k*).

b	REAL for ssyr2k DOUBLE PRECISION for dsyr2k COMPLEX for csyr2k DOUBLE COMPLEX for zsyr2k Array, DIMENSION (*ldb, kb*) where *kb* is *k* when *trans* = 'N' or 'n' and is 'n' otherwise. Before entry with *trans* = 'N' or 'n', the leading *n*-by-*k* part of the array *b* must contain the matrix *B*, otherwise the leading *k*-by-*n* part of the array *b* must contain the matrix *B*.
ldb	INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. When *trans* = 'N' or 'n', then *ldb* must be at least max(1, *n*), otherwise *ldb* must be at least max(1, *k*).
beta	REAL for ssyr2k DOUBLE PRECISION for dsyr2k COMPLEX for csyr2k DOUBLE COMPLEX for zsyr2k Specifies the scalar *beta*.
c	REAL for ssyr2k DOUBLE PRECISION for dsyr2k COMPLEX for csyr2k DOUBLE COMPLEX for zsyr2k Array, DIMENSION (*ldc*, *n*). Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *c* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *c* is not referenced. Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *c* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *c* is not referenced.
ldc	INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least max(1, *n*).

Output Parameters

c	With *uplo* = 'U' or 'u', the upper triangular part of the array *c* is overwritten by the upper triangular part of the updated matrix. With *uplo* = 'L' or 'l', the lower triangular part of the array *c* is overwritten by the lower triangular part of the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine syr2k interface are the following:

a	Holds the matrix *A* of size (*ma*,*ka*) where *ka* = *k* if *trans* = 'N', *ka* = *n* otherwise, *ma* = *n* if *trans* = 'N', *ma* = *k* otherwise.
b	Holds the matrix *B* of size (*mb*,*kb*) where *kb* = *k* if *trans* = 'N', *kb* = *n* otherwise, *mb* = *n* if *trans* = 'N', *mb* = *k* otherwise.
c	Holds the matrix *C* of size (*n*,*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
alpha	The default value is 1.
beta	The default value is 1.

?trmm

Computes a scalar-matrix-matrix product (one matrix operand is triangular).

Syntax

Fortran 77:

```
call strmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
call dtrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
call ctrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
call ztrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
```

Fortran 95:

```
call trmm(a, b [,side] [, uplo] [,transa][,diag] [,alpha])
```

Description

The ?trmm routines perform a matrix-matrix operation using triangular matrices. The operation is defined as

B := *alpha**op(*A*)**B*

or

B := *alpha***B**op(*A*)

where:

alpha is a scalar,

B is an *m*-by-*n* matrix,

A is a unit, or non-unit, upper or lower triangular matrix

op(*A*) is one of op(*A*) = *A*, or op(A) = *A*', or op(*A*) = conjg(*A*').

Input Parameters

side — CHARACTER*1. Specifies whether op(*A*) multiplies *B* from the left or right in the operation as follows:

***side* value**	**Operation To Be Performed**
L or l	*B* := *alpha**op(*A*)**B*

	R or r *B* := *alpha***B**op(*A*)
uplo	CHARACTER*1. Specifies whether the matrix *A* is an upper or lower triangular matrix as follows:

***uplo* value**	**Matrix *A***
U or u	Matrix *A* is an upper triangular matrix.
L or l	Matrix A is a lower triangular matrix.

transa	CHARACTER*1. Specifies the form of op(*A*) to be used in the matrix multiplication as follows:

***transa* value**	**Form of op(*A*)**
N or n	op(*A*) = *A*
T or t	op(*A*) = *A*'
C or c	op(*A*) = conjg(*A*')

diag	CHARACTER*1. Specifies whether or not *A* is unit triangular as follows:

***diag* value**	**Matrix *A***
U or u	Matrix *A* is assumed to be unit triangular.
N or n	Matrix *A* is not assumed to be unit triangular.

m	INTEGER. Specifies the number of rows of *B*. The value of *m* must be at least zero.
n	INTEGER. Specifies the number of columns of *B*. The value of *n* must be at least zero.
alpha	REAL for strmm DOUBLE PRECISION for dtrmm COMPLEX for ctrmm DOUBLE COMPLEX for ztrmm Specifies the scalar *alpha*. When *alpha* is zero, then *a* is not referenced and *b* need not be set before entry.
a	REAL for strmm DOUBLE PRECISION for dtrmm COMPLEX for ctrmm

DOUBLE COMPLEX for ztrmm
Array, DIMENSION (*lda*,*k*), where *k* is *m* when *side* = 'L' or 'l' and is *n* when *side* = 'R' or 'r'. Before entry with *uplo* = 'U' or 'u', the leading *k* by *k* upper triangular part of the array *a* must contain the upper triangular matrix and the strictly lower triangular part of *a* is not referenced. Before entry with *uplo* = 'L' or 'l', the leading *k* by *k* lower triangular part of the array *a* must contain the lower triangular matrix and the strictly upper triangular part of *a* is not referenced.
When *diag* = 'U' or 'u', the diagonal elements of *a* are not referenced either, but are assumed to be unity.

lda — INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. When *side* = 'L' or 'l', then *lda* must be at least max(1, *m*), when *side* = 'R' or 'r', then *lda* must be at least max(1, *n*).

b — REAL for strmm
DOUBLE PRECISION for dtrmm
COMPLEX for ctrmm
DOUBLE COMPLEX for ztrmm
Array, DIMENSION (*ldb*,*n*).
Before entry, the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb — INTEGER. Specifies the first dimension of *b* as declared in the calling (sub)program. The value of *ldb* must be at least max(1, *m*).

Output Parameters

b — Overwritten by the transformed matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine trmm interface are the following:

a — Holds the matrix *A* of size (*k*,*k*) where

	k = *m* if *side* = 'L', *k* = *n* otherwise.
b	Holds the matrix *B* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
uplo	Must be 'U' or 'L'. The default value is 'U'.
transa	Must be 'N', 'C', or 'T'. The default value is 'N'.
diag	Must be 'N' or 'U'. The default value is 'N'.
alpha	The default value is 1.

?trsm

Solves a matrix equation (one matrix operand is triangular).

Syntax

Fortran 77:

```
call strsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
call dtrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
call ctrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
call ztrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb )
```

Fortran 95:

```
call trsm(a, b [,side] [, uplo] [,transa][,diag] [,alpha])
```

Description

The ?trsm routines solve one of the following matrix equations:

op(*A*)**X* = *alpha***B*,

or

*X**op(*A*) = *alpha***B*,

where:

alpha is a scalar,

X and *B* are *m*-by-*n* matrices,

A is a unit, or non-unit, upper or lower triangular matrix

op(*A*) is one of op(*A*) = *A*, or op(*A*) = *A*', or op(*A*) = conjg(*A*').

The matrix *B* is overwritten by the solution matrix *X*.

Input Parameters

side CHARACTER*1. Specifies whether op(*A*) appears on the left or right of *X* for the operation to be performed as follows:

***side* value**	**Operation To Be Performed**
L or l	op(*A*)**X* = *alpha***B*
R or r	*X*op(*A*) = *alpha***B*

uplo CHARACTER*1. Specifies whether the matrix *A* is an upper or lower triangular matrix as follows:

***uplo* value**	**Matrix *A***
U or u	Matrix *A* is an upper triangular matrix.
L or l	Matrix *A* is a lower triangular matrix.

transa CHARACTER*1. Specifies the form of op(*a*) to be used in the matrix multiplication as follows:

***transa* value**	**Form of op(*a*)**
N or n	op(*A*) = *A*
T or t	op(*A*) = *A*'
C or c	op(*A*) = conjg(*A*')

diag CHARACTER*1. Specifies whether or not the matrix *A* is unit triangular as follows:

***diag* value**	**Matrix *A***
U or u	Matrix *A* is assumed to be unit triangular.
N or n	Matrix *A* is not assumed to be unit triangular.

m INTEGER. Specifies the number of rows of *B*. The value of *m* must be at least zero.

n INTEGER. Specifies the number of columns of *B*. The value of *n* must be at least zero.

alpha REAL for strsm
DOUBLE PRECISION for dtrsm
COMPLEX for ctrsm
DOUBLE COMPLEX for ztrsm
Specifies the scalar *alpha*.
When *alpha* is zero, then *a* is not referenced and *b* need not be set before entry.

a REAL for strsm
DOUBLE PRECISION for dtrsm
COMPLEX for ctrsm
DOUBLE COMPLEX for ztrsm
Array, DIMENSION (*lda*, *k*), where *k* is *m* when *side* = 'L' or 'l' and is *n* when *side* = 'R' or 'r'. Before entry with *uplo* = 'U' or 'u', the leading *k* by *k* upper triangular part of the array *a* must contain the upper triangular matrix and the strictly lower triangular part of *a* is not referenced. Before entry with *uplo* = 'L' or 'l', the leading *k* by *k* lower triangular part of the array *a* must contain the lower triangular matrix and the strictly upper triangular part of *a* is not referenced.
When *diag* = 'U' or 'u', the diagonal elements of *a* are not referenced either, but are assumed to be unity.

lda INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. When *side* = 'L' or 'l', then *lda* must be at least max(1, *m*), when *side* = 'R' or 'r', then *lda* must be at least max(1, *n*).

b REAL for strsm
DOUBLE PRECISION for dtrsm
COMPLEX for ctrsm
DOUBLE COMPLEX for ztrsm
Array, DIMENSION (*ldb*, *n*). Before entry, the leading *m*-by-*n* part of the array *b* must contain the right-hand side matrix *B*.

ldb	INTEGER. Specifies the first dimension of *b* as declared in the calling (sub)program. The value of *ldb* must be at least max(1, +*m*).

Output Parameters

b	Overwritten by the solution matrix *X*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine trsm interface are the following:

a	Holds the matrix *A* of size (*k*,*k*) where *k* = *m* if *side* = 'L', *k* = *n* otherwise.
b	Holds the matrix *B* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
uplo	Must be 'U' or 'L'. The default value is 'U'.
transa	Must be 'N', 'C', or 'T'. The default value is 'N'.
diag	Must be 'N' or 'U'. The default value is 'N'.
alpha	The default value is 1.

Sparse BLAS Level 1 Routines and Functions

This section describes Sparse BLAS Level 1, an extension of BLAS Level 1 included in the Intel® Math Kernel Library beginning with the Intel MKL release 2.1. Sparse BLAS Level 1 is a group of routines and functions that perform a number of common vector operations on sparse vectors stored in compressed form.

Sparse vectors are those in which the majority of elements are zeros. Sparse BLAS routines and functions are specially implemented to take advantage of vector sparsity. This allows you to achieve large savings in computer time and memory. If *nz* is the number of non-zero vector elements, the computer time taken by Sparse BLAS operations will be *O*(*nz*).

Vector Arguments

Compressed sparse vectors. Let a be a vector stored in an array, and assume that the only non-zero elements of a are the following:

$a(k_1), a(k_2), a(k_3) \ldots a(k_{nz}),$

where nz is the total number of non-zero elements in a.

In Sparse BLAS, this vector can be represented in compressed form by two FORTRAN arrays, x (values) and $indx$ (indices). Each array has nz elements:

$x(1)=a(k_1), x(2)=a(k_2), \ldots x(nz)= a(k_{nz}),$

$indx(1)=k_1, indx(2)=k_2, \ldots indx(nz)= k_{nz}.$

Thus, a sparse vector is fully determined by the triple (nz, x, $indx$). If you pass a negative or zero value of nz to Sparse BLAS, the subroutines do not modify any arrays or variables.

Full-storage vectors. Sparse BLAS routines can also use a vector argument fully stored in a single FORTRAN array (a full-storage vector). If y is a full-storage vector, its elements must be stored contiguously: the first element in $y(1)$, the second in $y(2)$, and so on. This corresponds to an increment $incy = 1$ in BLAS Level 1. No increment value for full-storage vectors is passed as an argument to Sparse BLAS routines or functions.

Naming Conventions

Similar to BLAS, the names of Sparse BLAS subprograms have prefixes that determine the data type involved: s and d for single- and double-precision real; c and z for single- and double-precision complex respectively.

If a Sparse BLAS routine is an extension of a "dense" one, the subprogram name is formed by appending the suffix i (standing for indexed) to the name of the corresponding "dense" subprogram. For example, the Sparse BLAS routine `saxpyi` corresponds to the BLAS routine `saxpy`, and the Sparse BLAS function `cdotci` corresponds to the BLAS function `cdotc`.

Routines and Data Types

Routines and data types supported in the Intel MKL implementation of Sparse BLAS are listed in Table 2-4 .

Table 2-4 Sparse BLAS Routines and Their Data Types

Routine/Function	Data Types	Description
?axpyi	s, d, c, z	Scalar-vector product plus vector (routines)
?doti	s, d	Dot product (functions)
?dotci	c, z	Complex dot product conjugated (functions)
?dotui	c, z	Complex dot product unconjugated (functions)
?gthr	s, d, c, z	Gathering a full-storage sparse vector into compressed form *nz*, *x*, *indx* (routines)
?gthrz	s, d, c, z	Gathering a full-storage sparse vector into compressed form and assigning zeros to gathered elements in the full-storage vector (routines)
?roti	s, d	Givens rotation (routines)
?sctr	s, d, c, z	Scattering a vector from compressed form to full-storage form (routines)

BLAS Level 1 Routines That Can Work With Sparse Vectors

The following BLAS Level 1 routines will give correct results when you pass to them a compressed-form array *x*(with the increment *incx*=1):

?asum	sum of absolute values of vector elements
?copy	copying a vector
?nrm2	Euclidean norm of a vector
?scal	scaling a vector
i?amax	index of the element with the largest absolute value for real flavors, or the largest sum |Re*x*(*i*)|+|Im*x*(*i*)| for complex flavors.
i?amin	index of the element with the smallest absolute value for real flavors, or the smallest sum |Re*x*(*i*)|+|Im*x*(*i*)| for complex flavors.

The result *i* returned by i?amax and i?amin should be interpreted as index in the compressed-form array, so that the largest (smallest) value is *x*(*i*); the corresponding index in full-storage array is *indx*(*i*).

You can also call ?roti to compute the parameters of Givens rotation and then pass these parameters to the Sparse BLAS routines ?roti.

?axpyi

Adds a scalar multiple of compressed sparse vector to a full-storage vector.

Syntax

Fortran 77:

```
call saxpyi(nz, a, x, indx, y)
call daxpyi(nz, a, x, indx, y)
call caxpyi(nz, a, x, indx, y)
call zaxpyi(nz, a, x, indx, y)
```

Fortran 95:

```
call axpyi(x, indx, y [, a])
```

Description

The ?axpyi routines perform a vector-vector operation defined as

y := *a***x* + *y*

where:

a is a scalar,

x is a sparse vector stored in compressed form,

y is a vector in full storage form.

The ?axpyi routines reference or modify only the elements of *y* whose indices are listed in the array *indx*.

The values in *indx* must be distinct.

Input Parameters

nz	INTEGER. The number of elements in *x* and *indx*.
a	REAL for saxpyi DOUBLE PRECISION for daxpyi

	COMPLEX for caxpyi DOUBLE COMPLEX for zaxpyi Specifies the scalar *a*.
x	REAL for saxpyi DOUBLE PRECISION for daxpyi COMPLEX for caxpyi DOUBLE COMPLEX for zaxpyi Array, DIMENSION at least *nz*.
indx	INTEGER. Specifies the indices for the elements of *x*. Array, DIMENSION at least *nz*.
y	REAL for saxpyi DOUBLE PRECISION for daxpyi COMPLEX for caxpyi DOUBLE COMPLEX for zaxpyi Array, DIMENSION at least max(*indx(i)*).

Output Parameters

y	Contains the updated vector *y*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine axpyi interface are the following:

x	Holds the vector of length (*nz*).
indx	Holds the vector of length (*nz*).
y	Holds the vector of length (*nz*).
a	The default value is 1.

?doti

Computes the dot product of a compressed sparse real vector by a full-storage real vector.

Syntax

Fortran 77:

```
res = sdoti(nz, x, indx, y )
res = ddoti(nz, x, indx, y )
```

Fortran 95:

```
res = doti(x, indx, y)
```

Description

The ?doti functions return the dot product of *x* and *y* defined as

```
x(1)*y(indx(1)) + x(2)*y(indx(2)) +...+ x(nz)*y(indx(nz))
```

where the triple (*nz*, *x*, *indx*) defines a sparse real vector stored in compressed form, and *y* is a real vector in full storage form. The functions reference only the elements of *y* whose indices are listed in the array *indx*. The values in *indx* must be distinct.

Input Parameters

nz	INTEGER. The number of elements in *x* and *indx* .
x	REAL for sdoti DOUBLE PRECISION for ddoti Array, DIMENSION at least *nz*.
indx	INTEGER. Specifies the indices for the elements of *x*. Array, DIMENSION at least *nz*.
y	REAL for sdoti DOUBLE PRECISION for ddoti Array, DIMENSION at least max(*indx(i)*).

Output Parameters

res	REAL for sdoti DOUBLE PRECISION for ddoti

Contains the dot product of *x* and *y*, if *nz* is positive. Otherwise, *res* contains 0.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine doti interface are the following:

x	Holds the vector of length (*nz*).
indx	Holds the vector of length (*nz*).
y	Holds the vector of length (*nz*).

?dotci

Computes the conjugated dot product of a compressed sparse complex vector with a full-storage complex vector.

Syntax

Fortran 77:

```
res = cdotci(nz, x, indx, y )
res = zdotci(nz, x, indx, y )
```

Fortran 95:

```
res = dotci(x, indx, y)
```

Description

The ?dotci functions return the dot product of *x* and *y* defined as

conjg(*x*(1))**y*(*indx*(1)) + ... + conjg(*x*(*nz*))**y*(*indx*(*nz*))

where the triple (*nz*, *x*, *indx*) defines a sparse complex vector stored in compressed form, and *y* is a real vector in full storage form. The functions reference only the elements of *y* whose indices are listed in the array *indx*. The values in *indx* must be distinct.

Input Parameters

nz	INTEGER. The number of elements in *x* and *indx* .
x	COMPLEX for cdotci DOUBLE COMPLEX for zdotci Array, DIMENSION at least *nz*.
indx	INTEGER. Specifies the indices for the elements of *x*. Array, DIMENSION at least *nz*.
y	COMPLEX for cdotci DOUBLE COMPLEX for zdotci Array, DIMENSION at least max(*indx(i)*).

Output Parameters

res	COMPLEX for cdotci DOUBLE COMPLEX for zdotci Contains the conjugated dot product of *x* and *y*, if *nz* is positive. Otherwise, *res* contains 0.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine dotci interface are the following:

x	Holds the vector of length (*nz*).
indx	Holds the vector of length (*nz*).
y	Holds the vector of length (*nz*).

?dotui

Computes the dot product of a compressed sparse complex vector by a full-storage complex vector.

Syntax

Fortran 77:

```
res = cdotui(nz, x, indx, y )
res = zdotui(nz, x, indx, y )
```

Fortran 95:

```
res = dotui(x, indx, y)
```

Description

The ?dotui functions return the dot product of *x* and *y* defined as

x(1)**y*(*indx*(1)) + *x*(2)**y*(*indx*(2)) +...+ *x*(*nz*)**y*(*indx*(*nz*))

where the triple (*nz*, *x*, *indx*) defines a sparse complex vector stored in compressed form, and *y* is a real vector in full storage form. The functions reference only the elements of *y* whose indices are listed in the array *indx*. The values in *indx* must be distinct.

Input Parameters

nz	INTEGER. The number of elements in *x* and *indx* .
x	COMPLEX for cdotui DOUBLE COMPLEX for zdotui Array, DIMENSION at least *nz*.
indx	INTEGER. Specifies the indices for the elements of *x*. Array, DIMENSION at least *nz*.
y	COMPLEX for cdotui DOUBLE COMPLEX for zdotui Array, DIMENSION at least max(*indx(i)*).

Output Parameters

res	COMPLEX for cdotui DOUBLE COMPLEX for zdotui

Contains the dot product of *x* and *y*, if *nz* is positive. Otherwise, *res* contains 0.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine dotui interface are the following:

x	Holds the vector of length (*nz*).
indx	Holds the vector of length (*nz*).
y	Holds the vector of length (*nz*).

?gthr

Gathers a full-storage sparse vector's elements into compressed form.

Syntax

Fortran 77:

```
call sgthr(nz, y, x, indx )
call dgthr(nz, y, x, indx )
call cgthr(nz, y, x, indx )
call zgthr(nz, y, x, indx )
```

Fortran 95:

```
res = gthr(x, indx, y)
```

Description

The ?gthr routines gather the specified elements of a full-storage sparse vector *y* into compressed form(*nz*, *x*, *indx*). The routines reference only the elements of *y* whose indices are listed in the array *indx*:

x(*i*) = *y*(*indx*(*i*)), for *i*=1,2,... +*nz*.

Input Parameters

nz	INTEGER. The number of elements of *y* to be gathered.
indx	INTEGER. Specifies indices of elements to be gathered. Array, DIMENSION at least *nz*.
y	REAL for sgthr DOUBLE PRECISION for dgthr COMPLEX for cgthr DOUBLE COMPLEX for zgthr Array, DIMENSION at least max(*indx(i)*).

Output Parameters

x	REAL for sgthr DOUBLE PRECISION for dgthr COMPLEX for cgthr DOUBLE COMPLEX for zgthr Array, DIMENSION at least *nz*. Contains the vector converted to the compressed form.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gthr interface are the following:

x	Holds the vector of length (*nz*).
indx	Holds the vector of length (*nz*).
y	Holds the vector of length (*nz*).

?gthrz

Gathers a sparse vector's elements into compressed form, replacing them by zeros.

Syntax

Fortran 77:

```
call sgthrz(nz, y, x, indx )
call dgthrz(nz, y, x, indx )
call cgthrz(nz, y, x, indx )
call zgthrz(nz, y, x, indx )
```

Fortran 95:

```
res = gthrz(x, indx, y)
```

Description

The ?gthrz routines gather the elements with indices specified by the array *indx* from a full-storage vector *y* into compressed form (*nz*, *x*, *indx*) and overwrite the gathered elements of *y* by zeros. Other elements of *y* are not referenced or modified (see also ?gthr).

Input Parameters

nz	INTEGER. The number of elements of *y* to be gathered.
indx	INTEGER. Specifies indices of elements to be gathered. Array, DIMENSION at least *nz*.
y	REAL for sgthrz DOUBLE PRECISION for dgthrz COMPLEX for cgthrz DOUBLE COMPLEX for zgthrz Array, DIMENSION at least max(*indx(i)*).

Output Parameters

x	REAL for sgthrz DOUBLE PRECISION for d gthrz COMPLEX for cgthrz

	DOUBLE COMPLEX for zgthrz Array, DIMENSION at least *nz*. Contains the vector converted to the compressed form.
y	The updated vector *y*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gthrz interface are the following:

x	Holds the vector of length (*nz*).
indx	Holds the vector of length (*nz*).
y	Holds the vector of length (*nz*).

?roti

Applies Givens rotation to sparse vectors one of which is in compressed form.

Syntax

Fortran 77:

```
call sroti(nz, x, indx, y, c, s)
call droti(nz, x, indx, y, c, s)
```

Fortran 95:

```
call roti(x, indx, y [, c] [,s])
```

Description

The ?roti routines apply the Givens rotation to elements of two real vectors, *x* (in compressed form *nz*, *x*, *indx*) and *y* (in full storage form):

```
x(i) = c*x(i) + s*y(indx(i))
y(indx(i)) = c*y(indx(i))- s*x(i)
```

The routines reference only the elements of *y* whose indices are listed in the array *indx*. The values in *indx* must be distinct.

Input Parameters

nz	INTEGER. The number of elements in *x* and *indx*.
x	REAL for sroti DOUBLE PRECISION for droti Array, DIMENSION at least *nz*.
indx	INTEGER. Specifies the indices for the elements of *x*. Array, DIMENSION at least *nz*.
y	REAL for sroti DOUBLE PRECISION for droti Array, DIMENSION at least max(*indx(i)*).
c	A scalar: REAL for sroti DOUBLE PRECISION for droti.
s	A scalar: REAL for sroti DOUBLE PRECISION for droti.

Output Parameters

x and *y*	The updated arrays.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine roti interface are the following:

x	Holds the vector of length (*nz*).
indx	Holds the vector of length (*nz*).
y	Holds the vector of length (*nz*).
c	The default value is 1.
s	The default value is 1.

?sctr

Converts compressed sparse vectors into full storage form.

Syntax

Fortran 77:

```
call ssctr(nz, x, indx, y )
call dsctr(nz, x, indx, y )
call csctr(nz, x, indx, y )
call zsctr(nz, x, indx, y )
```

Fortran 95:

```
call sctr(x, indx, y)
```

Description

The ?sctr routines scatter the elements of the compressed sparse vector (*nz*, *x*, *indx*) to a full-storage vector *y*. The routines modify only the elements of *y* whose indices are listed in the array *indx*:

y(*indx*(*i*) = *x*(*i*), for *i*=1,2,... +*nz*.

Input Parameters

nz	INTEGER. The number of elements of *x* to be scattered.
indx	INTEGER. Specifies indices of elements to be scattered. Array, DIMENSION at least *nz*.
x	REAL for ssctr DOUBLE PRECISION for dsctr COMPLEX for csctr DOUBLE COMPLEX for zsctr Array, DIMENSION at least *nz*. Contains the vector to be converted to full-storage form.

Output Parameters

y	REAL for ssctr

DOUBLE PRECISION for dsctr
COMPLEX for csctr
DOUBLE COMPLEX for zsctr
Array, DIMENSION at least max(*indx(i)*).
Contains the vector *y* with updated elements.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine sctr interface are the following:

x	Holds the vector of length (*nz*).
indx	Holds the vector of length (*nz*).
y	Holds the vector of length (*nz*).

Sparse BLAS Level 2 and Level 3

This section describes Sparse BLAS Level 2 and Level 3 included in the Intel® Math Kernel Library. Sparse BLAS Level 2 is a group of routines and functions that perform operations on a sparse matrix and dense vectors. Sparse BLAS Level 3 is a group of routines and functions that perform operations on a sparse matrix and a dense matrices.

Sparse matrix is a matrix in which the majority of elements are zeros. Intel MKL sparse BLAS routines and functions are specially implemented to take advantage of matrix sparsity. This allows to achieve large savings in computer time and memory. The sparse BLAS routines can be considered as building blocks for "Iterative Sparse Solvers based on Reverse Communication Interface (RCI ISS)" in Chapter 8 of the manual.

Naming Conventions in Sparse BLAS Level 2 and Level 3

Each Sparse BLAS routine has a six- or eight-characters base name preceding with the prefix mkl_ (mkl_cspblas_ for routines with simplified interface and zero-based indexing). The routines with standard interfaces have six-characters base names, the routines with simplified interfaces have eight-characters base names in accordance with the templates:

```
mkl<character code> <data> <operation>( )
```

or

```
mkl<character code> <data> <mtype> <operation>( )
```

```
mkl_cspblas_<character code> <data> <mtype> <operation>( )
```

The <character code> is a character that indicates the data type:

s	real, single precision
c	complex, single precision
d	real, double precision
z	complex, double precision

NOTE. Current version of the Intel MKL Sparse BLAS supports only real data with double precision.

The <data> field indicates the data structure of the sparse matrix (see section Sparse Matrix Data Structures):

coo	coordinate format
csr	compressed sparse row format and its variations
csc	compressed sparse column format and its variations
dia	diagonal format
sky	skyline storage format
bsr	block sparse row format and its variations

The <operation> field indicates the type of operation.

mv	matrix-vector product (Level 2)
mm	matrix-matrix product (Level 3)
sm	solving a single triangular system (Level 2)
sm	solving triangular systems with multiple right-hand sides (Level 3)

An optional field <mtype> indicates a matrix type and used in the routines with simplified interfaces:

ge	sparse representation of a general matrix
sy	sparse representation of the upper or lower triangle of a symmetric matrix
tr	sparse representation of a triangular matrix

Sparse Matrix Data Structures

In the current version of Intel MKL sparse BLAS Level 2 and Level 3 the following point entry [Duff86] sparse matrix data structures are supported:

- compressed sparse row format(CSR) and its variation;
- compressed sparse column format(CSC);
- coordinate format;
- diagonal format;
- skyline storage format.
- block sparse row format(BSR) and its variations.

For more information on matrix storage schemes, see "Sparse Storage Formats for Sparse BLAS Levels 2-3" in Appendix A.

Routines and Supported Operations

This section describes two main types of routines and supported operations. The following notations are used here:

A - is a sparse matrix;

B and *C* - are dense matrices;

D - is a diagonal scaling matrix;

x and *y* - are dense vectors;

alpha and *beta* - are scalars;

op(*A*) is one of the possible operations:

op(*A*) = *A*;

op(*A*) = *A*' - transpose of *A*;

op(*A*) = conj(*A*') - conjugated transpose of *A*.

Complete list of all routines is given in the Table 2-9 .

Routines with Standard Interface

Intel MKL Sparse BlAS routines support the following operations:

Level 2.

- computing a sparse matrix-dense vector product:
  `y := alpha*op(A)*x + beta*y`
- solving a single triangular system:
  `y := alpha*inv(op(A))*x`

Level 3.

- computing a sparse matrix-dense matrix product:
  `C := alpha*op(A)*B + beta*C`
- solving a sparse triangular system with multiple right-hand sides:
  `C := alpha*inv(op(A))*B`

These routines have native interface that differs from the interface used in the NIST Sparse BLAS library [Rem05]. Detailed consideration of these differences can be found in the section Interface Consideration.

Routines with Simplified Interface

Some software packages and libraries (PARDISO package used in the Intel MKL, Sparskit 2 [Saad94], Compaq Extended Math Library (CXML)[CXML01]) use different (early) variation of the CSR format and support only level 2 operations with simplified interfaces. Intel MKL provides a set of level 2 routines with similar simplified interfaces. Each of these routines operates on a matrix of the fixed type. The following operations are supported:

`y :=op(A)*x` (general and symmetric matrices)

`y := inv(op(A))*x` (triangular matrices)

Matrix type is indicated by the field <*mtype*> in the routine name (see section Naming Conventions in Sparse BLAS Level 2 and Level 3).

The detail consideration of interfaces for these routines is given in the Interface Consideration section.

These routines can operate only with four sparse data storage formats, specifically:

CSR format in variation accepted in `PARDISO` and `CXML`;

DIA format accepted in `CXML`;

COO format.

BSR format.

Note that routines in both groups described above use the same computational kernel routines that work with certain internal data structures.

Interface Consideration

Differences Between Intel MKL and NIST Interfaces

The Intel MKL Sparse BLAS Level 3 routines have the following standard interfaces:

`mkl_xyyymm(transa, m, n, k, alpha, matdescra, arg(A), b, ldb, beta, c, ldc)`, for matrix-matrix product;

`mkl_xyyysm(transa, m, n, alpha, matdescra, arg(A), b, ldb, c, ldc)`, for triangular solvers with multiple right-hand sides.

Here `x` denotes data type, and *yyy* - sparse matrix data structure (storage format).

The analogous NIST Sparse BLAS (NSB) library routines have the following interfaces:

`xyyymm(transa, m, n, k, alpha, descra, arg(A), b, ldb, beta, c, ldc, work, lwork)`, for matrix-matrix product;

`xyyysm(transa, m, n, unitd, dv, alpha, descra, arg(A), b, ldb, beta, c, ldc, work, lwork)`, for triangular solvers with multiple right-hand sides.

Some similar arguments are used in both libraries. The argument *transa* indicates how to operate with the matrix and is slightly different in the NSB library (see Table 2-5). The arguments *m* and *k* are the number of rows and column in the matrix *A*, respectively, *n* is the number of columns in the matrix *C*. The arguments *alpha* and *beta* are scalar *alpha* and *beta* respectively. (*beta* is not used in the Intel MKL triangular solvers.) The arguments *b* and *c* are rectangular arrays with the first dimension *ldb* and *ldc*, respectively. The symbol `arg(A)` denotes the list of arguments that describe the sparse representation of *A*.

Table 2-5 Parameter *transa*

	MKL interface	NSB interface	Operation
data type	CHARACTER*1	INTEGER	
value	N or n	0	`op(A) = A`

MKL interface	NSB interface	Operation
T or t	1	op(*A*) = *A*'
C or c	2	op(*A*) = *A*'

The argument *matdescra* describes the relevant characteristics of the matrix *A*.

It corresponds to the argument *descra* from NSB library (see Table 2-6 for more details).

Table 2-6 Possible Values of the Parameter *matdescra* (*descra*)

	MKL interface		NSB interface	Matrix characteristics
	one-based indexing	zero-based indexing		
data type	CHARACTER	Char	INTEGER	
1st element	*matdescra*(1)	*Matdescra*[0]	*descra*(1)	matrix structure
value	G	G	0	general
	S	S	1	symmetric (*A* = *A*')
	H	H	2	Hermitian (*A*=conjg(*A*'))
	T	T	3	triangular
	A	A	4	skew(anti)-symmetric (*A*=-*A*')
	D	D	5	diagonal
2nd element	*matdescra*(2)	*Matdescra*[1]	*descra*(2)	upper/lower triangular indicator
value	L	L	1	lower
	U	U	2	upper
3rd element	*matdescra*(3)	*Matdescra*[2]	*descra*(3)	main diagonal type

	MKL interface		NSB interface	Matrix characteristics
value	N	N	0	non-unit
	U	U	1	unit
4th element	*matdescra*(4)	*Matdescra*[3]		type of indexing
value	F			one-based indexing
		C		zero-based indexing

Note that *matdescra* has some specifics in the Intel MKL routines. In particular, for routines that perform matrix-matrix and matrix-vector multiplication, they are as follows:

for general matrices (*matdescra*(1)='G', values of *matdescra*(2) and *matdescra*(3) are ignored;

for skew-symmetrical matrices (*matdescra*(1)='A', values of *matdescra*(3) are ignored;

for diagonal matrices (*matdescra*(1)='D', values of *matdescra*(2) are ignored;

If *matdescra*(1) is not set to 'G' or 'T', and *matdescra*(2) and *matdescra*(3) are not defined, then the following default values are assigned: *matdescra*(2)='L' and *matdescra*(3)='N';

matdescra(1)='G' is not supported for the routines operating with the skyline storage format.

For triangular solvers if *matdescra*(1)='D', then *matdescra*(2) is ignored.

For triangular solvers Intel MKL supports only *matdescra*(1)=T,D;

For both multiplication routines and triangular solvers when *matdescra*(3)='U', and the sparse matrix is not in the skyline format, then non-zero diagonal elements can be stored in the sparse representation even if they are non-unit; when the sparse matrix is in the skyline format, the diagonal elements must be stored in the sparse representation even if they are zero.

The current version of NSB library supports only *descra*(1) for matrix-matrix multiplication; *descra*(2), *descra*(3) are supported for triangular solvers only if *descra*(1)=3.

The argument *work* is a work array, and *lwork* is its dimension.

These arguments are not used in the Intel MKL.

The arguments *unitd* and *dv* are used only in NSB triangular solvers. First of them indicates whether or not the diagonal matrix *D* is unitary. If *unitd*=1, *D* is the identity matrix. The linear array *dv* contains the diagonal scaling matrix *D* if the argument *unitd* = 2 (the rows of *A* are scaled) or *unitd* = 3 (the columns of *A* are scaled).

Simplified Interfaces

The Intel MKL Sparse BLAS Level 2 routines with simplified interfaces have the following interfaces (x denotes data type, and *yyy* - sparse matrix storage format):

one-based indexing

mkl_xyyygemv(*transa*, *m*, arg(*A*), *x*, *y*), matrix-vector product for general sparse matrices;

mkl_xyyysymv(*uplo*, *transa*, *m*, arg(*A*), *x*, *y*), matrix-vector product for symmetrical sparse matrix;

mkl_xyyytrsv(*uplo*, *transa*, *diag*, *m*, arg(*A*), *x*, *y*) solution of the systems of equations with a sparse triangular matrix.

zero-based indexing

mkl_cspblas_xyyygemv(*transa*, *m*, arg(*A*), *x*, *y*), matrix-vector product for general sparse matrices;

mkl_cspblas_xyyysymv(*uplo*, *transa*, *m*, arg(*A*), *x*, *y*), matrix-vector product for symmetrical sparse matrix;

mkl_cspblas_xyyytrsv(*uplo*, *transa*, *diag*, *m*, arg(*A*), *x*, *y*) solution of the systems of equations with a sparse triangular matrix.

The argument *transa* indicates how to operate with the matrix (see Table 2-5). The argument *uplo* specifies whether an upper or low triangle of the sparse matrix will be considered. The argument *diag* specifies whether *A* is a unit triangular or not. The arguments *m* is the number of rows in the matrix *A*. The arg(*A*) denotes the list of arguments that describe the sparse representation of *A*.

The array *x* contains the input vector, and the array *y* contains the result of the performed operation.

Note that all routines for matrix-vector multiplication are able to extract triangles and/or a main diagonal from a sparse representation of the matrix *A*.

Operations with Partial Matrices

One of the distinctive feature of the Intel MKL Sparse BLAS routines is a possibility to perform operations only on certain parts (triangles and main diagonal) of the input sparse matrix specifying the parameter *matdescra*. Assume that the sparse matrix *A* can be decomposed as

A = L + D + U

where *L* is the strict lower triangle of *A*, *U* is the strict upper triangle of *A*, *D* is the main diagonal.

Table 2-7 shows correspondence between the output matrix for matrix-matrix multiplication routines and values of *matdescra* for real sparse matrix *A*. Analogous correspondence exists for matrix-vector multiplication routines.

Table 2-7 Correspondence Between Output Matrix and Values of *matdescra* (Routines for Matrix-Matrix Multiplication)

matdescra(1)	matdescra(2)	matdescra(3)	Output Matrix
G	ignored	ignored	*alpha**op(*A*)**B* + *beta***C*
S or H	L	N	*alpha**op(*L+D+L'*)**B* + *beta***C*
S or H	L	U	*alpha**op(*L+I+L'*)**B* + *beta***C*
S or H	U	N	*alpha**op(*U'+D+U*)**B* + *beta***C*
S or H	U	U	*alpha**op(*U'+I+U*)**B* + *beta***C*
T	L	U	*alpha**op(*L+I*)**B* + *beta***C*
T	L	N	*alpha**op(*L+D*)**B* + *beta***C*
T	U	U	*alpha**op(*U+I*)**B* + *beta***C*
T	U	N	*alpha**op(*U+D*)**B* + *beta***C*
A	L	ignored	*alpha**op(*L-L'*)**B* + *beta***C*
A	U	ignored	*alpha**op(*U-U'*)**B* + *beta***C*
D	ignored	N	*alpha***D***B* + *beta***C*
D	ignored	U	*alpha***B* + *beta***C*

Table 2-8 shows correspondence between the output matrix for triangular solvers and values of *matdescra* for real sparse matrix *A*.

Table 2-8 Correspondence Between Output Matrix and Values of *matdescra* (Triangular Solvers)

matdescra(1)	matdescra(2)	matdescra(3)	Output Matrix
T	L	N	*alpha**inv(op(*L*+*L*))**B*
T	L	U	*alpha**inv(op(*L*+*L*))**B*
T	U	N	*alpha**inv(op(*U*+*U*))**B*
T	U	U	*alpha**inv(op(*U*+*U*))**B*
D	ignored	N	*alpha**inv(*D*)**B*
D	ignored	U	*alpha***B*

Restrictions for Triangular Solver Routines

There are important restrictions for all Intel MKL triangular solvers, specifically:

Column indices for the compressed sparse row format must be sorted in increasing order for each row;

Row indices for the compressed sparse column format must be sorted in increasing order for each column;

For the diagonal format, elements of the array containing the diagonal numbers of the non-zero diagonals of a sparse matrix must be sorted in increasing order.

Sparse BLAS Level 2 and Level 3 Routines.

Table 2-9 lists the sparse BLAS Level 2 and Level 3 routines described in more detail later in this section.

Table 2-9 Sparse BLAS Level 2 and Level 3 Routines

Routine/Function	Description
Level 2	
mkl_dcsrmv	Computes matrix - vector product of a sparse matrix stored in the CSR format.
mkl_dcsrgemv	Computes matrix - vector product of a sparse general matrix stored in the CSR format (PARDISO variation)
mkl_dcsrsymv	Computes matrix - vector product of a sparse symmetrical matrix stored in the CSR format (PARDISO variation)
mkl_cspblas_dcsrsymv	Computes matrix - vector product of a sparse symmetrical matrix stored in the CSR format (PARDISO variation) with zero-based indexing
mkl_dcscmv	Computes matrix - vector product for a sparse matrix in CSC format.
mkl_dcoomv	Computes matrix - vector product for a sparse matrix in the coordinate format.
mkl_dcoogemv	Computes matrix - vector product of a sparse general matrix stored in the coordinate format.
mkl_dcoosymv	Computes matrix - vector product of a sparse symmetrical matrix stored in the coordinate format.
mkl_ddiamv	Computes matrix - vector product of a sparse matrix stored in the diagonal format.
mkl_ddiagemv	Computes matrix - vector product of a sparse general matrix stored in the diagonal format.
mkl_ddiasymv	Computes matrix - vector product of a sparse symmetrical matrix stored in the diagonal format.
mkl_dskymv	Computes matrix - vector product for a sparse matrix in the skyline storage format.
mkl_dbsrmv	Computes matrix - vector product of a sparse matrix stored in the BSR format.

Routine/Function	Description
mkl_dbsrsymv	Computes matrix - vector product of a sparse symmetrical matrix stored in the BSR format (PARDISO variation).
mkl_cspblas_dbsrsymv	Computes matrix - vector product of a sparse symmetrical matrix stored in the BSR format (PARDISO variation) with zero-based indexing.
mkl_dcsrsv	Solves a system of linear equations for a sparse matrix in the CSR format.
mkl_dcsrtrsv	Triangular solvers with simplified interface for a sparse matrix in the CSR format (PARDISO variation).
mkl_dcscsv	Solves a system of linear equations for a sparse matrix in the compressed sparse column format.
mkl_dcoosv	Solves a system of linear equations for a sparse matrix in the coordinate format.
mkl_dcootrsv	Triangular solvers with simplified interface for a sparse matrix in the coordinate format.
mkl_ddiasv	Solves a system of linear equations for a sparse matrix in the diagonal format.
mkl_ddiatrsv	Triangular solvers with simplified interface for a sparse matrix in the diagonal format.
mkl_dskysv	Solves a system of linear equations for a sparse matrix in the skyline format.
Level 3	
mkl_dcsrmm	Computes matrix - matrix product of a sparse matrix stored in the compressed sparse row format
mkl_dcscmm	Computes matrix - matrix product of a sparse matrix stored in the compressed sparse column format
mkl_dcoomm	Computes matrix - matrix product of a sparse matrix stored in the coordinate format.

Routine/Function	Description
mkl_ddiamm	Computes matrix - matrix product of a sparse matrix stored in the diagonal format.
mkl_dskymm	Computes matrix - matrix product of a sparse matrix stored in the skyline storage format.
mkl_dcsrsm	Solves a system of linear matrix equations for a sparse matrix in the CSR format.
mkl_dcscsm	Solves a system of linear matrix equations for a sparse matrix in the CSC format.
mkl_dcoosm	Solves a system of linear matrix equations for a sparse matrix in the coordinate format.
mkl_ddiasm	Solves a system of linear matrix equations for a sparse matrix in the diagonal format.
mkl_dskysm	Solves a system of linear matrix equations for a sparse matrix stored in the skyline storage format.

mkl_dcsrmv

Computes matrix - vector product of a sparse matrix stored in the CSR format.

Syntax

Fortran:

```
call mkl_dcsrmv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
```

C:

```
mkl_dcsrmv(&transa, &m, &k, &alpha, matdescra, val, indx, pntrb, pntre, x, &beta, y);
```

Description

The mkl_dcsrmv routine performs a matrix-vector operation defined as

*y := alpha*A*x + beta*y*

or

*y := alpha*A'*x + beta*y*,

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix in the CSR format, *A'* is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', the matrix-vector product is computed as *y := alpha*A*x + beta*y* If *transa* = 'T' or 't' or 'C' or 'c', the matrix-vector product is computed as *y := alpha*A'*x + beta*y*,
m	INTEGER. Number of rows of the matrix *A*.
k	INTEGER. Number of columns of the matrix *A*.
alpha	REAL*8. Specifies the scalar *alpha*.
matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
val	REAL*8. Array containing non-zero elements of the matrix *A*. Its length is *pntre*(m – *pntrb*(1). Refer to *values* array description in CSR Format for more details.
indx	INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.Its length is equal to length of the *val* array.

	Refer to *columns* array description in CSR Format for more details.
pntrb	INTEGER. Array of length *m*, contains row indices, such that *pntrb*(*i*) – *pntrb*(1)+1 is the first index of row *i* in the arrays *val* and *indx*. Refer to *pointerb* array description in CSR Format for more details.
pntre	INTEGER. Array of length *m*, contains row indices, such that *pntre*(*i*) – *pntrb*(1) is the last index of row *i* in the arrays *val* and *indx*. Refer to *pointerE* array description in CSR Format for more details.
x	REAL*8. Array, DIMENSION at least *k* if *transa* = 'N' or 'n' and at least *m* otherwise. Before entry, the array *x* must contain the vector *x*.
beta	REAL*8. Specifies the scalar *beta*.
y	REAL*8. Array, DIMENSION at least *m* if *transa* = 'N' or 'n' and at least *k* otherwise. Before entry, the array *y* must contain the vector *y*.

Output Parameters

y	Overwritten by the updated vector *y*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dcsrmv(transa, m, k, alpha, matdescra, val, indx,
 pntrb, pntre, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
```

```
INTEGER        m, k
INTEGER        indx(*), pntrb(m), pntre(m)
REAL*8         alpha, beta
REAL*8         val(*), x(*), y(*)
```

C:

```
void mkl_dcsrmv(char *transa, int *m, int *k, double *alpha, char *matdescra,
 double *val, int *indx, int *pntrb, int *pntre, double *x, double *beta,
double *y);
```

mkl_dcsrgemv

Computes matrix - vector product of a sparse general matrix stored in the CSR format (PARDISO variation).

Syntax

Fortran:

```
call mkl_dcsrgemv(transa, m, a, ia, ja, x, y)
```

C:

```
mkl_dcsrgemv(&transa, &m, a, ia, ja, x, y);
```

Description

The `mkl_dcsrgemv` routine performs a matrix-vector operation defined as

`y := A*x`

or

`y := A'*x`,

where:

`x` and `y` are vectors,

`A` is an `m`-by-`m` sparse square matrix in the CSR format (PARDISO variation), `A'` is the transpose of `A`.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', the matrix-vector product is computed as *y* := *A*x* If *transa* = 'T' or 't' or 'C' or 'c', the matrix-vector product is computed as *y* := *A'*x*,
m	INTEGER. Number of rows of the matrix *A*.
a	REAL*8. Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.
ia	INTEGER. Array of length *m* + 1, containing indices of elements in the array *a*, such that *ia*(*i*) is the index in the array *a* of the first non-zero element from the row *i*. The value of the last element *ia*(*m* + 1)−1 is equal to the number of non-zeros plus one. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.
ja	REAL*8. Array containing the column indices for each non-zero element of the matrix *A*. Its length is equal to the length of the array *a*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.
x	REAL*8. Array, DIMENSION is *m*. Before entry, the array *x* must contain the vector *x*.

Output Parameters

y	REAL*8. Array, DIMENSION at least *m*. On exit, the array *y* must contain the vector *y*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dcsrgemv(transa, m, a, ia, ja, x, y)
   CHARACTER*1  transa
   INTEGER      m
   INTEGER      ia(*), ja(*)
   REAL*8       a(*), x(*), y(*)
```

C:

```
void mkl_dcsrgemv(char *transa, int *m, double *a, int *ia, int *ja, double
 *x, double *y);
```

mkl_dcsrsymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the CSR format (PARDISO variation).

Syntax

Fortran:

```
call mkl_dcsrsymv(uplo, m, a, ia, ja, x, y)
```

C:

```
mkl_dcsrsymv(&uplo, &m, a, ia, ja, x, y);
```

Description

The `mkl_dcsrsymv` routine performs a matrix-vector operation defined as

y := *A*x*

or

y := *A'*x*,

where:

x and *y* are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the CSR format (PARDISO variation), *A'* is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo	CHARACTER*1. Specifies whether the upper or low triangle of the matrix *A* is considered. If *uplo* = 'U' or 'u', the upper triangle of the matrix *A* is used. If *uplo* = 'L' or 'l', the low triangle of the matrix *A* is used.
m	INTEGER. Number of rows of the matrix *A*.
a	REAL*8. Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.
ia	INTEGER. Array of length *m* + 1, containing indices of elements in the array *a*, such that *ia(i)* is the index in the array *a* of the first non-zero element from the row *i*. The value of the last element *ia(m* + 1)–1 is equal to the number of non-zeros plus one. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.
ja	REAL*8. Array containing the column indices for each non-zero element of the matrix *A*. Its length is equal to the length of the array *a*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.
x	REAL*8. Array, DIMENSION is *m*. Before entry, the array *x* must contain the vector *x*.

Output Parameters

y	REAL*8. Array, DIMENSION at least *m*. On exit, the array *y* must contain the vector *y*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dcsrsymv(uplo, m, a, ia, ja, x, y)
   CHARACTER*1  uplo
   INTEGER      m
   INTEGER      ia(*), ja(*)
   REAL*8       a(*), x(*), y(*)
```

C:

```
void mkl_dcsrsymv(char *uplo, int *m, double *a, int *ia, int *ja, double
*x, double *y);
```

mkl_cspblas_dcsrsymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the CSR format (PARDISO variation) with zero-based indexing.

Syntax

Fortran:

```
call mkl_cspblas_dcsrsymv(uplo, m, a, ia, ja, x, y)
```

C:

```
mkl_cspblas_dcsrsymv(&uplo, &m, a, ia, ja, x, y);
```

Description

The mkl_cspblas_dcsrsymv routine performs a matrix-vector operation defined as

*y := A*x*

or

y := *A*'**x*,

where:

x and *y* are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the CSR format (PARDISO variation) with zero-based indexing, *A*' is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo	CHARACTER*1. Specifies whether the upper or low triangle of the matrix *A* is considered. If *uplo* = 'U' or 'u', the upper triangle of the matrix *A* is used. If *uplo* = 'L' or 'l', the low triangle of the matrix *A* is used.
m	INTEGER. Number of rows of the matrix *A*.
a	REAL*8. Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.
ia	INTEGER. Array of length *m* + 1, containing indices of elements in the array *a*, such that *ia*(*i*) is the index in the array *a* of the first non-zero element from the row *i*. The value of the last element *ia*(*m* + 1) is equal to the number of non-zeros. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.
ja	REAL*8. Array containing the column indices for each non-zero element of the matrix *A*. Its length is equal to the length of the array *a*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.
x	REAL*8.

Array, DIMENSION is *m*.
Before entry, the array *x* must contain the vector *x*.

Output Parameters

y REAL*8.
Array, DIMENSION at least *m*.
On exit, the array *y* must contain the vector *y*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_cspblas_dcsrsymv(uplo, m, a, ia, ja, x, y)
   CHARACTER*1  uplo
   INTEGER      m
   INTEGER      ia(*), ja(*)
   REAL*8       a(*), x(*), y(*)
```

C:

```
void mkl_cspblas_dcsrsymv(char *uplo, int *m, double *a, int *ia, int *ja,
double *x, double *y);
```

mkl_dcscmv

Computes matrix - vector product for a sparse matrix in the compressed sparse column format.

Syntax

Fortran:

```
call mkl_dcscmv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x,
beta, y)
```

C:

```
mkl_dcscmv(&transa, &m, &k, &alpha, matdescra, val, indx, pntrb, pntre, x,
&beta, y);
```

Description

The mkl_dcscmv routine performs a matrix-vector operation defined as

*y := alpha*A*x + beta*y*

or

*y := alpha*A'*x + beta*y*,

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix in compressed sparse column format, *A'* is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', the matrix-vector product is computed as *y := alpha*A*x + beta*y* If *transa* = 'T' or 't' or 'C' or 'c', the matrix-vector product is computed as *y := alpha*A'*x + beta*y*,
m	INTEGER. Number of rows of the matrix *A*.
k	INTEGER. Number of columns of the matrix *A*.
alpha	REAL*8. Specifies the scalar *alpha*.
matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
val	REAL*8. Array containing non-zero elements of the matrix *A*. Its length is *pntre*(*k*) – *pntrb*(1). Refer to *values* array description in CSC Format for more details.
indx	INTEGER. Array containing the row indices for each non-zero element of the matrix *A*.Its length is equal to length of the *val* array.

	Refer to *rows* array description in CSC Format for more details.
pntrb	INTEGER. Array of length *k*, contains row indices, such that *pntrb*(*i*) – *pntrb*(1)+1 is the starting index of column *i* in the arrays *val* and *indx*. Refer to *pointerb* array description in CSC Format for more details.
pntre	INTEGER. Array of length *k*, contains row indices, such that *pntre*(*i*) – *pntrb*(1) is the last index of column *i* in the arrays *val* and *indx*. Refer to *pointerE* array description in CSC Format for more details.
x	REAL*8. Array, DIMENSION at least *k* if *transa* = 'N' or 'n' and at least *m* otherwise. Before entry, the array *x* must contain the vector *x*.
beta	REAL*8. Specifies the scalar *beta*.
y	REAL*8. Array, DIMENSION at least *m* if *transa* = 'N' or 'n' and at least *k* otherwise. Before entry, the array *y* must contain the vector *y*.

Output Parameters

y	Overwritten by the updated vector *y*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dcscmv(transa, m, k, alpha, matdescra, val, indx, pntrb,
pntre, x, beta, y)

  CHARACTER*1   transa

  CHARACTER     matdescra(*)
```

```
INTEGER        m, k, ldb, ldc
INTEGER        indx(*), pntrb(m), pntre(m)
REAL*8         alpha, beta
REAL*8         val(*), x(*), y(*)
```

C:

```
void mkl_dcscmv(char *transa, int *m, int *k, double *alpha, char *matdescra,
 double *val, int *indx, int *pntrb, int *pntre, double *x, double *beta,
double *y);
```

mkl_dcoomv

Computes matrix - vector product for a sparse matrix in the coordinate format.

Syntax

Fortran:

```
call mkl_dcoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x,
beta, y)
```

C:

```
mkl_dcoomv(&transa, &m, &k, &alpha, matdescra, val, rowind, colind, &nnz, x,
&beta, y);
```

Description

The `mkl_dcoomv` routine performs a matrix-vector operation defined as

*y := alpha*A*x + beta*y*

or

*y := alpha*A'*x + beta*y*,

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix in compressed coordinate format, *A'* is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', the matrix-vector product is computed as *y* := *alpha*A*x* + *beta*y* If *transa* = 'T' or 't' or 'C' or 'c', the matrix-vector product is computed as *y* := *alpha*A'*x* + *beta*y*,
m	INTEGER. Number of rows of the matrix *A*.
k	INTEGER. Number of columns of the matrix *A*.
alpha	REAL*8. Specifies the scalar *alpha*.
matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
val	REAL*8. Array of length *nnz*, contains non-zero elements of the matrix *A* in the arbitrary order. Refer to *values* array description in Coordinate Format for more details.
rowind	INTEGER. Array of length *nnz*, contains the row indices for each non-zero element of the matrix *A*. Refer to *rows* array description in Coordinate Format for more details.
colind	INTEGER. Array of length *nnz*, contains the column indices for each non-zero element of the matrix *A*. Refer to *columns* array description in Coordinate Format for more details.
nnz	INTEGER. Specifies the number of non-zero element of the matrix *A*. Refer to *nnz* description in Coordinate Format for more details.
x	REAL*8. Array, DIMENSION at least *k* if *transa* = 'N' or 'n' and at least *m* otherwise. Before entry, the array *x* must contain the vector *x*.
beta	REAL*8. Specifies the scalar *beta*.

y	REAL*8. Array, DIMENSION at least *m* if *transa* = 'N' or 'n' and at least *k* otherwise. Before entry, the array *y* must contain the vector *y*.

Output Parameters

y	Overwritten by the updated vector *y*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dcoomv(transa, m, k, alpha, matdescra, val, rowind, colind,
nnz, x, beta, y)

  CHARACTER*1   transa

  CHARACTER     matdescra(*)


  INTEGER       m, k, nnz

  INTEGER       rowind(*), colind(*)

  REAL*8        alpha, beta

  REAL*8        val(*), x(*), y(*)
```

C:

```
void mkl_dcoomv(char *transa, int *m, int *k, double *alpha, char *matdescra,
 double  *val, int *rowind, int *colind, int *nnz, double *x, double *beta,
 double *y);
```

mkl_dcoogemv

Computes matrix - vector product of a sparse general matrix stored in the coordinate format.

Syntax

Fortran:

```
call mkl_dcoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

C:

```
mkl_dcoogemv(&transa, &m, val, rowind, colind, &nnz, x, y);
```

Description

The `mkl_dcoogemv` routine performs a matrix-vector operation defined as

y := *A***x*

or

y := *A*'**x*,

where:

x and *y* are vectors,

A is an *m*-by-*m* sparse square matrix in the coordinate format, *A*' is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', the matrix-vector product is computed as *y* := *A***x* If *transa* = 'T' or 't' or 'C' or 'c', the matrix-vector product is computed as *y* := *A*'**x*,
m	INTEGER. Number of rows of the matrix *A*.
val	REAL*8. Array of length *nnz*, contains non-zero elements of the matrix *A* in the arbitrary order. Refer to *values* array description in Coordinate Format for more details.
rowind	INTEGER. Array of length *nnz*, contains the row indices for each non-zero element of the matrix *A*. Refer to *rows* array description in Coordinate Format for more details.
colind	INTEGER. Array of length *nnz*, contains the column indices for each non-zero element of the matrix *A*. Refer to *columns* array description in Coordinate Format for more details.

nnz	INTEGER. Specifies the number of non-zero element of the matrix *A*. Refer to *nnz* description in Coordinate Format for more details.
x	REAL*8. Array, DIMENSION is *m*. Before entry, the array *x* must contain the vector *x*.

Output Parameters

y	REAL*8. Array, DIMENSION at least *m*. On exit, the array *y* must contain the vector *y*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dcoogemv(transa, m, val, rowind, colind, nnz, x, y)

  CHARACTER*1   transa

  INTEGER       m, nnz

  INTEGER       rowind(*), colind(*)

  REAL*8        val(*), x(*), y(*)
```

C:

```
void mkl_dcoogemv(char *transa, int *m, double  *val, int *rowind, int
*colind, int *nnz, double *x, double *y);
```

mkl_dcoosymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the coordinate format.

Syntax

Fortran:

```
call mkl_dcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

C:

```
mkl_dcoosymv(&uplo, &m, val, rowind, colind, &nnz, x, y);
```

Description

The `mkl_dcoosymv` routine performs a matrix-vector operation defined as

y := *A***x*

or

y := *A*'**x*,

where:

x and *y* are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the coordinate format, *A*' is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo	CHARACTER*1. Specifies whether the upper or low triangle of the matrix *A* is considered. If *uplo* = 'U' or 'u', the upper triangle of the matrix *A* is used. If *uplo* = 'L' or 'l', the low triangle of the matrix *A* is used.
m	INTEGER. Number of rows of the matrix *A*.
val	REAL*8. Array of length *nnz*, contains non-zero elements of the matrix *A* in the arbitrary order. Refer to *values* array description in Coordinate Format for more details.
rowind	INTEGER. Array of length *nnz*, contains the row indices for each non-zero element of the matrix *A*. Refer to *rows* array description in Coordinate Format for more details.

colind	INTEGER. Array of length *nnz*, contains the column indices for each non-zero element of the matrix *A*. Refer to *columns* array description in Coordinate Format for more details.
nnz	INTEGER. Specifies the number of non-zero element of the matrix *A*. Refer to *nnz* description in Coordinate Format for more details.
x	REAL*8. Array, DIMENSION is *m*. Before entry, the array *x* must contain the vector *x*.

Output Parameters

y	REAL*8. Array, DIMENSION at least *m*. On exit, the array *y* must contain the vector *y*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1   uplo
  INTEGER       m, nnz
  INTEGER       rowind(*), colind(*)
  REAL*8        val(*), x(*), y(*)
```

C:

```
void mkl_dcoosymv(char *uplo, int *m, double  *val, int *rowind, int *colind,
 int *nnz, double *x, double *y);
```

mkl_ddiamv

Computes matrix - vector product for a sparse matrix in the diagonal format.

Syntax

Fortran:

```
call mkl_ddiamv(transa, m, k, alpha, matdescra, val, lval, idiag, ndiag, x, beta, y)
```

C:

```
mkl_ddiamv(&transa, &m, &k, &alpha, matdescra, val, &lval, idiag, &ndiag, x, &beta, y);
```

Description

The `mkl_ddiamv` routine performs a matrix-vector operation defined as

`y := alpha*A*x + beta*y`

or

`y := alpha*A'*x + beta*y`,

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix stored in the diagonal format, *A'* is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', the matrix-vector product is computed as *y* := *alpha*A*x* + *beta*y*, If *transa* = 'T' or 't' or 'C' or 'c', the matrix-vector product is computed as *y* := *alpha*A'*x* + *beta*y*.
m	INTEGER. Number of rows of the matrix *A*.

k	INTEGER. Number of columns of the matrix *A*.
alpha	REAL*8. Specifies the scalar *alpha*.
matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
val	REAL*8. Two-dimensional array of size *lval* by *ndiag*, contains non-zero diagonals of the matrix *A*. Refer to *values* array description in Diagonal Storage Scheme for more details.
lval	INTEGER. Leading dimension of *val*, *lval* ≥min(*m*, *k*). Refer to *lval* description in Diagonal Storage Scheme for more details.
idiag	INTEGER. Array of length *ndiag*, contains the distances between main diagonal and each non-zero diagonals in the matrix *A*. Refer to *distance* array description in Diagonal Storage Scheme for more details.
ndiag	INTEGER. Specifies the number of non-zero diagonals of the matrix *A*.
x	REAL*8. Array, DIMENSION at least *k* if *transa* = 'N' or 'n', and at least *m* otherwise. Before entry, the array *x* must contain the vector *x*.
beta	REAL*8. Specifies the scalar *beta*.
y	REAL*8. Array, DIMENSION at least *m* if *transa* = 'N' or 'n', and at least *k* otherwise. Before entry, the array *y* must contain the vector *y*.

Output Parameters

y	Overwritten by the updated vector *y*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_ddiamv(transa, m, k, alpha, matdescra, val, lval, idiag,
ndiag, x, beta, y)

  CHARACTER*1    transa

  CHARACTER      matdescra(*)


  INTEGER        m, k, lval, ndiag

  INTEGER       idiag(*)

  REAL*8         alpha, beta

  REAL*8         val(lval,*), x(*), y(*)
```

C:

```
void mkl_ddiamv(char *transa, int *m, int *k, double *alpha, char *matdescra,
 double  *val, int *lval, int *idiag, int *ndiag, double *x, double *beta,
double *y);
```

mkl_ddiagemv

Computes matrix - vector product of a sparse general matrix stored in the diagonal format.

Syntax

Fortran:

```
call mkl_ddiagemv(transa, m, val, lval, idiag, ndiag, x, y)
```

C:

```
mkl_ddiagemv(&transa, &m, val, &lval, idiag, &ndiag, x, y);
```

Description

The `mkl_ddiagemv` routine performs a matrix-vector operation defined as

y := *A*x*

or

y := *A*'**x*,

where:

x and *y* are vectors,

A is an *m*-by-*m* sparse square matrix in the diagonal storage format, *A*' is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', the matrix-vector product is computed as *y* := *A***x* If *transa* = 'T' or 't' or 'C' or 'c', the matrix-vector product is computed as *y* := *A*'**x*,
m	INTEGER. Number of rows of the matrix *A*.
val	REAL*8. Two-dimensional array of size *lval* by *ndiag*, contains non-zero diagonals of the matrix *A*. Refer to *values* array description in Diagonal Storage Scheme for more details.
lval	INTEGER. Leading dimension of *val*, *lval*≥min(*m*, *k*). Refer to *lval* description in Diagonal Storage Scheme for more details.
idiag	INTEGER. Array of length *ndiag*, contains the distances between main diagonal and each non-zero diagonals in the matrix *A*. Refer to *distance* array description in Diagonal Storage Scheme for more details.
ndiag	INTEGER. Specifies the number of non-zero diagonals of the matrix *A*.
x	REAL*8. Array, DIMENSION is *m*. Before entry, the array *x* must contain the vector *x*.

Output Parameters

y	REAL*8. Array, DIMENSION at least *m*. On exit, the array *y* must contain the vector *y*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_ddiagemv(transa, m, val, lval, idiag, ndiag, x, y)
  CHARACTER*1   transa
  INTEGER       m, lval, ndiag
  INTEGER      idiag(*)
  REAL*8        val(lval,*), x(*), y(*)
```

C:

```
void mkl_ddiagemv(char *transa, int *m, double  *val, int *lval, int *idiag,
 int *ndiag, double *x, double *y);
```

mkl_ddiasymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the diagonal format.

Syntax

Fortran:

```
call mkl_ddiasymv(uplo, m, val, lval, idiag, ndiag, x, y)
```

C:

```
mkl_ddiasymv(&uplo, &m, val, &lval, idiag, &ndiag, x, y);
```

Description

The mkl_ddiasymv routine performs a matrix-vector operation defined as

*y := A*x*

or

y := *A*'**x*,

where:

x and *y* are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix, *A*' is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo	CHARACTER*1. Specifies whether the upper or low triangle of the matrix *A* is considered. If *uplo* = 'U' or 'u', the upper triangle of the matrix *A* is used. If *uplo* = 'L' or 'l', the low triangle of the matrix *A* is used.
m	INTEGER. Number of rows of the matrix *A*.
val	REAL*8. Two-dimensional array of size *lval* by *ndiag*, contains non-zero diagonals of the matrix *A*. Refer to *values* array description in Diagonal Storage Scheme for more details.
lval	INTEGER. Leading dimension of *val*, *val*, *lval* ≥min(*m*, *k*). Refer to *lval* description in Diagonal Storage Scheme for more details.
idiag	INTEGER. Array of length *ndiag*, contains the distances between main diagonal and each non-zero diagonals in the matrix *A*. Refer to *distance* array description in Diagonal Storage Scheme for more details.
ndiag	INTEGER. Specifies the number of non-zero diagonals of the matrix *A*.
x	REAL*8. Array, DIMENSION is *m*. Before entry, the array *x* must contain the vector *x*.

Output Parameters

y	REAL*8. Array, DIMENSION at least *m*. On exit, the array *y* must contain the vector *y*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_ddiasymv(uplo, m, val, lval, idiag, ndiag, x, y)
  CHARACTER*1   uplo
  INTEGER       m, lval, ndiag
  INTEGER      idiag(*)
  REAL*8        val(lval,*), x(*), y(*)
```

C:

```
void mkl_ddiasymv(char *uplo, int *m, double  *val, int *lval, int *idiag,
int *ndiag, double *x, double *y);
```

mkl_dskymv

Computes matrix - vector product for a sparse matrix in the skyline storage format.

Syntax

Fortran:

```
call mkl_dskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta, y)
```

C:

```
mkl_dskymv(&transa, &m, &k, &alpha, matdescra, val, pntr, x, &beta, y);
```

Description

The mkl_dskymv routine performs a matrix-vector operation defined as

*y := alpha*A*x + beta*y*

or

y := *alpha***A*'**x* + *beta***y*,

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix stored using the skyline storage scheme, *A*' is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', the matrix-vector product is computed as *y* := *alpha***A***x* + *beta***y* If *transa* = 'T' or 't' or 'C' or 'c', the matrix-vector product is computed as *y* := *alpha***A*'**x* + *beta***y*,
m	INTEGER. Number of rows of the matrix *A*.
k	INTEGER. Number of columns of the matrix *A*.
alpha	REAL*8. Specifies the scalar *alpha*.
matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
val	REAL*8. Array containing the set of elements of the matrix *A* in the skyline profile form. If *matdescrsa*(2)= 'L', then *val* contains elements from the low triangle of the matrix *A*. If *matdescrsa*(2)= 'U', then *val* contains elements from the upper triangle of the matrix *A*. Refer to *values* array description in Skyline Storage Scheme for more details.
pntr	INTEGER. Array of length (*m*+*m*) for lower triangle, and (*k*+*k*) for upper triangle.

	It contains the indices specifying in the *val* the positions of the first element in each row (column) of the matrix *A*. Refer to *pointers* array description in Skyline Storage Scheme for more details.
x	REAL*8. Array, DIMENSION at least *k* if *transa* = 'N' or 'n' and at least *m* otherwise. Before entry, the array *x* must contain the vector *x*.
beta	REAL*8. Specifies the scalar *beta*.
y	REAL*8. Array, DIMENSION at least *m* if *transa* = 'N' or 'n' and at least *k* otherwise. Before entry, the array *y* must contain the vector *y*.

Output Parameters

y	Overwritten by the updated vector *y*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta,
y)

  CHARACTER*1   transa

  CHARACTER     matdescra(*)


  INTEGER       m, k

  INTEGER       pntr(*)

  REAL*8        alpha, beta

  REAL*8        val(*), x(*), y(*)
```

C:

```
void mkl_dskymv (char *transa, int *m, int *k, double *alpha, char *matdescra,
 double  *val, int *pntr, double *x, double *beta, double *y);
```

mkl_dbsrmv

Computes matrix - vector product of a sparse matrix stored in the BSR format.

Syntax

Fortran:

```
call mkl_dbsrmv(transa, m, k, lb, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
```

C:

```
mkl_dbsrmv(&transa, &m, &k, &lb, &alpha, matdescra, val, indx, pntrb, pntre, x, &beta, y);
```

Description

The `mkl_dbsrmv` routine performs a matrix-vector operation defined as

`y := alpha*A*x + beta*y`

or

`y := alpha*A'*x + beta*y`,

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* block sparse matrix in the BSR format, *A'* is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', the matrix-vector product is computed as *y* := *alpha*A*x* + *beta*y* If *transa* = 'T' or 't' or 'C' or 'c', the matrix-vector product is computed as *y* := *alpha*A'*x* + *beta*y*,
m	INTEGER. Number of block rows of the matrix *A*.

k	INTEGER. Number of block columns of the matrix *A*.
lb	INTEGER. Size of the block in the matrix *A*.
alpha	REAL*8. Specifies the scalar *alpha*.
matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
val	REAL*8. Array containing elements of non-zero blocks of the matrix *A*. Its length is equal to the number number of non-zero blocks in the matrix *A* multiplied by *lb***lb*. Refer to *values* array description in BSR Format for more details.
indx	INTEGER. Array containing the column indices for each non-zero block in the matrix *A*. Its length is equal to the number of non-zero blocks in the matrix *A*. Refer to *columns* array description in BSR Format for more details.
pntrb	INTEGER. Array of length *m*. For one-based indexing: this array contains row indices, such that *pntrb*(*i*) – *pntrb*(1)+1 is the first index of block row *i* in the array *indx*. For zero-based indexing: this array contains row indices, such that *pntrb*(*i*) – *pntrb*(0) is the first index of block row *i* in the array *indx* Refer to *pointerB* array description in BSR Format for more details.
pntre	INTEGER. Array of length *m*. For one-based intexing this array contains row indices, such that *pntre*(*i*) – *pntrb*(1) is the last index of block row *i* in the array *indx*. For zero-based intexing this array contains row indices, such that *pntre*(*i*) – *pntrb*(0)–1 is the last index of block row *i* in the array *indx*. Refer to *pointerE* array description in BSR Format for more details.
x	REAL*8.

	Array, DIMENSION at least (*k***lb*) if *transa* = 'N' or 'n', and at least (*m***lb*) otherwise. Before entry, the array *x* must contain the vector *x*.
beta	REAL*8. Specifies the scalar *beta*.
y	REAL*8. Array, DIMENSION at least (*m***lb*) if *transa* = 'N' or 'n', and at least (*k***lb*) otherwise. Before entry, the array *y* must contain the vector *y*.

Output Parameters

y	Overwritten by the updated vector *y*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dbsrmv(transa, m, k, lb, alpha, matdescra, val, indx,
 pntrb, pntre, x, beta, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)

  INTEGER       m, k, lb
  INTEGER       indx(*), pntrb(m), pntre(m)
  REAL*8        alpha, beta
  REAL*8        val(*), x(*), y(*)
```

C:

```
void mkl_dbsrmv(char *transa, int *m, int *k, int *lb, double *alpha, char
*matdescra,
 double *val, int *indx, int *pntrb, int *pntre, double *x, double *beta,
double *y);
```

mkl_dbsrsymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the BSR format (PARDISO variation).

Syntax

Fortran:

```
call mkl_dbsrsymv(uplo, m, lb, a, ia, ja, x, y)
```

C:

```
mkl_dbsrsymv(&uplo, &m, &lb, a, ia, ja, x, y);
```

Description

The `mkl_dbsrsymv` routine performs a matrix-vector operation defined as

`y := A*x`

or

`y := A'*x`,

where:

`x` and `y` are vectors,

`A` is an upper or lower triangle of the symmetrical sparse matrix in the BSR format (PARDISO variation), `A'` is the transpose of `A`.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo	`CHARACTER*1`. Specifies whether the upper or low triangle of the matrix `A` is considered. If `uplo = 'U'` or `'u'`, the upper triangle of the matrix `A` is used. If `uplo = 'L'` or `'l'`, the low triangle of the matrix `A` is used.

m	INTEGER. Number of block rows of the matrix *A*.
lb	INTEGER. Size of the block in the matrix *A*.
a	REAL*8. Array containing elements of non-zero blocks of the matrix *A*. Its length is equal to the number of non-zero blocks in the matrix *A* multiplied by *lb*lb*. Refer to *values* array description in BSR Format for more details.
ia	INTEGER. Array of length *(m + 1)*, containing indices of block in the array *a*, such that *ia(i)* is the index in the array *a* of the first non-zero element from the row *i*. The value of the last element *ia(m + 1)* is equal to the number of non-zero blocks plus one. Refer to *rowIndex* array description in BSR Format for more details.
ja	REAL*8. Array containing the column indices for each non-zero block in the matrix *A*. Its length is equal to the number of non-zero blocks of the matrix *A*. Refer to *columns* array description in BSR Format for more details.
x	REAL*8. Array, DIMENSION *(m*lb)*. Before entry, the array *x* must contain the vector *x*.

Output Parameters

y	REAL*8. Array, DIMENSION at least *(m*lb)*. On exit, the array *y* must contain the vector *y*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dbsrsymv(uplo, m, lb, a, ia, ja, x, y)
   CHARACTER*1  uplo
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   REAL*8       a(*), x(*), y(*)
```

C:

```
void mkl_dbsrsymv(char *uplo, int *m, int *lb, double *a, int *ia, int *ja,
 double *x, double *y);
```

mkl_cspblas_dbsrsymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the BSR format (PARDISO variation) with zero-based indexing.

Syntax

Fortran:

```
call mkl_cspblas_dbsrsymv(uplo, m, lb, a, ia, ja, x, y)
```

C:

```
mkl_cspblas_dbsrsymv(&uplo, &m, &lb, a, ia, ja, x, y);
```

Description

The `mkl_cspblas_dbsrsymv` routine performs a matrix-vector operation defined as

`y := A*x`

or

`y := A'*x`,

where:

`x` and `y` are vectors,

`A` is an upper or lower triangle of the symmetrical sparse matrix in the BSR format (PARDISO variation) with zero-based indexing, `A'` is the transpose of `A`.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo	`CHARACTER*1`. Specifies whether the upper or low triangle of the matrix `A` is considered.

If *uplo* = 'U' or 'u', the upper triangle of the matrix *A* is used.
If *uplo* = 'L' or 'l', the low triangle of the matrix *A* is used.

m — INTEGER. Number of block rows of the matrix *A*.

lb — INTEGER. Size of the block in the matrix *A*.

a — REAL*8. Array containing elements of non-zero blocks of the matrix *A*. Its length is equal to the number of non-zero blocks in the matrix *A* multiplied by *lb***lb*. Refer to *values* array description in BSR Format for more details.

ia — INTEGER. Array of length (*m* + 1), containing indices of block in the array *a*, such that *ia*(*i*) is the index in the array *a* of the first non-zero element from the row *i*. The value of the last element *ia*(*m* + 1) is equal to the number of non-zero blocks plus one. Refer to *rowIndex* array description in BSR Format for more details.

ja — REAL*8. Array containing the column indices for each non-zero block in the matrix *A*.
Its length is equal to the number of non-zero blocks of the matrix *A*. Refer to *columns* array description in BSR Format for more details.

x — REAL*8.
Array, DIMENSION (*m***lb*).
Before entry, the array *x* must contain the vector *x*.

Output Parameters

y — REAL*8.
Array, DIMENSION at least (*m***lb*).
On exit, the array *y* must contain the vector *y*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_cspblas_dbsrsymv(uplo, m, lb, a, ia, ja, x, y)
   CHARACTER*1  uplo
   INTEGER      m, lb
   INTEGER      ia(*), ja(*)
   REAL*8       a(*), x(*), y(*)
```

C:

```
void mkl_cspblas_dbsrsymv(char *uplo, int *m, int *lb, double *a, int *ia,
int *ja, double *x, double *y);
```

mkl_dcsrsv

Solves a system of linear equations for a sparse matrix in the CSR format.

Syntax

Fortran:

```
call mkl_dcsrsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

C:

```
mkl_dcsrsv(&transa, &m, &alpha, matdescra, val, indx, pntrb, pntre, x, y);
```

Description

The `mkl_dcsrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the CSR format:

y := *alpha**inv(*A*)**x*

or

y := *alpha**inv(*A*')**x*,

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', *y* := *alpha**inv(*A*)**x* If *transa* = 'T' or 't' or 'C' or 'c', *y* := *alpha**inv(*A'*)**x*,
m	INTEGER. Number of columns of the matrix *A*.
alpha	REAL*8. Specifies the scalar *alpha*.
matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
val	REAL*8. Array containing non-zero elements of the matrix *A*. Its length is *pntre*(*m*) – *pntrb*(1). Refer to *values* array description in CSR Format for more details.
indx	INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.Its length is equal to length of the *val* array. Refer to *columns* array description in CSR Format for more details.
pntrb	INTEGER. Array of length *m*, contains row indices, such that *pntrb*(*i*) – *pntrb*(1)+1 is the starting index of row *i* in the arrays *val* and *indx*. Refer to *pointerb* array description in CSR Format for more details.
pntre	INTEGER. Array of length *m*, contains row indices, such that *pntre*(*i*) – *pntrb*(1) is the last index of row *i* in the arrays *val* and *indx*. Refer to *pointerE* array description in CSR Format for more details.
x	REAL*8. Array, DIMENSION at least *m*.

	Before entry, the array *x* must contain the vector *x*. The elements are accessed with unit increment.
y	REAL*8. Array, DIMENSION at least *m*. Before entry, the array *y* must contain the vector *y*. The elements are accessed with unit increment.

Output Parameters

y	Contains solution vector *x*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dcsrsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre,
 x, y)

  CHARACTER*1   transa

  CHARACTER     matdescra(*)


  INTEGER       m

  INTEGER       indx(*), pntrb(m), pntre(m)

  REAL*8        alpha

  REAL*8        val(*)

  REAL*8        x(*), y(*)
```

C:

```
void mkl_dcsrsv(char *transa, int *m, double *alpha, char *matdescra, double
 *val, int *indx, int *pntrb, int *pntre, double *x, double *y);
```

mkl_dcsrtrsv

Triangular solvers with simplified interface for a sparse matrix in the CSR format (PARDISO variation).

Syntax

Fortran:

```
call mkl_dcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

C:

```
mkl_dcsrtrsv(&uplo, &transa, &diag, &m, a, ia, ja, x, y);
```

Description

The `mkl_dcsrtrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the CSR format accepted in PARDISO:

`A*y = x`

or

`A'*y = x`,

where:

`x` and `y` are vectors,

`A` is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, `A'` is the transpose of `A`.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo	`CHARACTER*1`. Specifies whether the upper or low triangle of the matrix `A` is considered. If `uplo = 'U'` or `'u'`, the upper triangle of the matrix `A` is used. If `uplo = 'L'` or `'l'`, the low triangle of the matrix `A` is used.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', *A*y* = *x* If *transa* = 'T' or 't' or 'C' or 'c', *A'*y* = *x*,
diag	CHARACTER*1. Specifies whether *A* is a unit triangular or not. If *diag* = 'U' or 'u', *A* is assumed to be a unit triangular. If *diag* = 'N' or 'n', *A* is not assumed to be a unit triangular.
m	INTEGER. Number of rows of the matrix *A*.
a	REAL*8. Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in Sparse Matrix Storage Formats for more details.
ia	INTEGER. Array of length *m* + 1, containing indices of elements in the array *a*, such that *ia*(*i*) is the index in the array *a* of the first non-zero element from the row *i*. The value of the last element *ia*(*m* + 1)–1 is equal to the number of non-zeros plus one. Refer to *rowIndex* array description in Sparse Matrix Storage Formats for more details.
ja	REAL*8. Array containing the column indices for each non-zero element of the matrix *A*. Its length is equal to the length of the array *a*. Refer to *columns* array description in Sparse Matrix Storage Formats for more details.
x	REAL*8. Array, DIMENSION is *m*. Before entry, the array *x* must contain the vector *x*.

Output Parameters

y	REAL*8. Array, DIMENSION at least *m*. Contains the vector *y*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
   CHARACTER*1  uplo, transa, diag
   INTEGER      m
   INTEGER      ia(*), ja(*)
   REAL*8       a(*), x(*), y(*)
```

C:

```
void mkl_dcsrtrsv(char *uplo, char *transa, char *diag, int *m, double *a,
int *ia, int *ja, double *x, double *y);
```

mkl_dcscsv

Solves a system of linear equations for a sparse matrix in the CSC format.

Syntax

Fortran:

```
call mkl_dcscsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

C:

```
mkl_dcscsv(&transa, &m, &alpha, matdescra, val, indx, pntrb, pntre, x, y);
```

Description

The `mkl_dcsrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the CSC format:

y := *alpha**inv(*A*)**x*

or

y := *alpha**inv(*A*')* *x*,

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', *y* := *alpha**inv(*A*)**x* If *transa*= 'T' or 't' or 'C' or 'c', *y* := *alpha**inv(*A'*)* *x*,
m	INTEGER. Number of columns of the matrix *A*.
alpha	REAL*8. Specifies the scalar *alpha*.
matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
val	REAL*8. Array containing non-zero elements of the matrix *A*. Its length is *pntre*(*m*) – *pntrb*(1). Refer to *values* array description in CSC Format for more details.
indx	INTEGER. Array containing the column indices for each non-zero element of the matrix *A*. Its length is equal to length of the *val* array. Refer to *columns* array description in CSC Format for more details.
pntrb	INTEGER. Array of length *m*, contains row indices, such that *pntrb*(*i*) – *pntrb*(1)+1 is the starting index of row *i* in the arrays *val* and *indx*. Refer to *pointerb* array description in CSC Format for more details.
pntre	INTEGER. Array of length *m*, contains row indices, such that *pntre*(*i*) – *pntrb*(1) is the last index of row *i* in the arrays *val* and *indx*. Refer to *pointerE* array description in CSC Format for more details.
x	REAL*8. Array, DIMENSION at least *m*.

	Before entry, the array *x* must contain the vector *x*. The elements are accessed with unit increment.
y	REAL*8. Array, DIMENSION at least *m*. Before entry, the array *y* must contain the vector *y*. The elements are accessed with unit increment.

Output Parameters

y	Contains the solution vector *x*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dcscsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre,
 x, y)

  CHARACTER*1   transa

  CHARACTER     matdescra(*)


  INTEGER       m

  INTEGER       indx(*), pntrb(m), pntre(m)

  REAL*8        alpha

  REAL*8        val(*)

  REAL*8        x(*), y(*)
```

C:

```
void mkl_dcscsv(char *transa, int *m, double *alpha, char *matdescra, double
 *val, int *indx, int *pntrb, int *pntre, double *x, double *y);
```

mkl_dcoosv

Solves a system of linear equations for a sparse matrix in the coordinate format.

Syntax

Fortran:

```
call mkl_dcoosv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, y)
```

C:

```
mkl_dcoosv(&transa, &m, &k, &alpha, matdescra, val, rowind, colind, &nnz, x, y);
```

Description

The `mkl_dcoosv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the coordinate format:

`y := alpha*inv(A)*x`

or

`y := alpha*inv(A')*x`,

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', *y* := *alpha*inv(*A*)**x* If *transa* = 'T' or 't' or 'C' or 'c', *y* := *alpha*inv(*A'*)* *x*,
m	INTEGER. Number of rows of the matrix *A*.
alpha	REAL*8. Specifies the scalar *alpha*.

matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
val	REAL*8. Array of length *nnz*, contains non-zero elements of the matrix *A* in the arbitrary order. Refer to *values* array description in Coordinate Format for more details.
rowind	INTEGER. Array of length *nnz*, contains the row indices for each non-zero element of the matrix *A*. Refer to *rows* array description in Coordinate Format for more details.
colind	INTEGER. Array of length *nnz*, contains the column indices for each non-zero element of the matrix *A*. Refer to *columns* array description in Coordinate Format for more details.
nnz	INTEGER. Specifies the number of non-zero element of the matrix *A*. Refer to *nnz* description in Coordinate Format for more details.
x	REAL*8. Array, DIMENSION at least *m*. Before entry, the array *x* must contain the vector *x*. The elements are accessed with unit increment.
y	REAL*8. Array, DIMENSION at least *m*. Before entry, the array *y* must contain the vector *y*. The elements are accessed with unit increment.

Output Parameters

y	Contains solution vector *x*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dcoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz,
 x, y)
  CHARACTER*1    transa
  CHARACTER      matdescra(*)

  INTEGER        m, nnz
  INTEGER        rowind(*), colind(*)
  REAL*8         alpha
  REAL*8         val(*)
  REAL*8         x(*), y(*)
```

C:

```
void mkl_dcoosv(char *transa, int *m, double *alpha, char *matdescra, double
 *val, int *rowind, int *colind, int *nnz, double *x, double *y);
```

mkl_dcootrsv

Triangular solvers with simplified interface for a sparse matrix in the coordinate format.

Syntax

Fortran:

```
call mkl_dcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
```

C:

```
mkl_dcootrsv(&uplo, &transa, &diag, &m, val, rowind, colind, &nnz, x, y);
```

Description

The `mkl_dcootrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the coordinate format:

$A*y = x$

or

*A'*y = x*,

where:

x and *y* are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo	CHARACTER*1. Specifies whether the upper or low triangle of the matrix *A* is considered. If *uplo* = 'U' or 'u', the upper triangle of the matrix *A* is used. If *uplo* = 'L' or 'l', the low triangle of the matrix *A* is used.
transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', *A*y = x* If *transa* = 'T' or 't' or 'C' or 'c', *A'*y = x*,
diag	CHARACTER*1. Specifies whether or not *A* is a unit triangular or not. If *diag* = 'U' or 'u', *A* is assumed to be a unit triangular. If *diag* = 'N' or 'n', *A* is not assumed to be a unit triangular.
m	INTEGER. Number of rows of the matrix *A*.
val	REAL*8. Array of length *nnz*, contains non-zero elements of the matrix *A* in the arbitrary order. Refer to *values* array description in Coordinate Format for more details.
rowind	INTEGER. Array of length *nnz*, contains the row indices for each non-zero element of the matrix *A*. Refer to *rows* array description in Coordinate Format for more details.

colind	INTEGER. Array of length *nnz*, contains the column indices for each non-zero element of the matrix *A*. Refer to *columns* array description in Coordinate Format for more details.
nnz	INTEGER. Specifies the number of non-zero element of the matrix *A*. Refer to *nnz* description in Coordinate Format for more details.
x	REAL*8. Array, DIMENSION is *m*. Before entry, the array *x* must contain the vector *x*.

Output Parameters

y	REAL*8. Array, DIMENSION at least *m*. Contains the vector *y*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x,
 y)

  CHARACTER*1   uplo, transa, diag

  INTEGER       m, nnz

  INTEGER       rowind(*), colind(*)

  REAL*8        val(*), x(*), y(*)
```

C:

```
void mkl_dcootrsv(char *uplo, char *transa, char *diag, int *m, double *alpha,
 char *matdescra, double  *val, int *rowind, int *colind, int *nnz, double
*x, double *y);
```

mkl_ddiasv

Solves a system of linear equations for a sparse matrix in the diagonal format.

Syntax

Fortran:

```
call mkl_ddiasv(transa, m, alpha, matdescra, val, lval, idiag, ndiag, x, y)
```

C:

```
mkl_ddiasv(&transa, &m, &alpha, matdescra, val, &lval, idiag, &ndiag, x, y);
```

Description

The `mkl_ddiasv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the diagonal format:

y := *alpha**inv(*A*)**x*

or

y := *alpha**inv(*A*')* *x*,

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', *y* := *alpha**inv(*A*)**x* If *transa* = 'T' or 't' or 'C' or 'c', *y* := *alpha**inv(*A'*)**x*,
m	INTEGER. Number of rows of the matrix *A*.
alpha	REAL*8. Specifies the scalar *alpha*.

matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
val	REAL*8. Two-dimensional array of size *lval* by *ndiag*, contains non-zero diagonals of the matrix *A*. Refer to *values* array description in Diagonal Storage Scheme for more details.
lval	INTEGER. Leading dimension of *val*, *lval*≥*m*. Refer to *lval* description in Diagonal Storage Scheme for more details.
idiag	INTEGER. Array of length *ndiag*, contains the distances between main diagonal and each non-zero diagonals in the matrix *A*. Refer to *distance* array description in Diagonal Storage Scheme for more details.
ndiag	INTEGER. Specifies the number of non-zero diagonals of the matrix *A*.
x	REAL*8. Array, DIMENSION at least *m*. Before entry, the array *x* must contain the vector *x*. The elements are accessed with unit increment.
y	REAL*8. Array, DIMENSION at least *m*. Before entry, the array *y* must contain the vector *y*. The elements are accessed with unit increment.

Output Parameters

y	Contains solution vector *x*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_ddiasv(transa, m, alpha, matdescra, val, lval, idiag, ndiag,
 x, y)
  CHARACTER*1    transa
  CHARACTER      matdescra(*)

  INTEGER        m, lval, ndiag
  INTEGER        indiag(*)
  REAL*8         alpha
  REAL*8         val(lval,*), x(*), y(*)
```

C:

```
void mkl_ddiasv(char *transa, int *m, double *alpha, char *matdescra, double
 *val, int *lval, int *idiag, int *ndiag, double *x, double *y);
```

mkl_ddiatrsv

Triangular solvers with simplified interface for a sparse matrix in the diagonal format.

Syntax

Fortran:

```
call mkl_ddiatrsv(uplo, transa, diag, m, val, lval, idiag, ndiag, x, y)
```

C:

```
mkl_ddiatrsv(&uplo, &transa, &diag, &m, val, &lval, idiag, &ndiag, x, y);
```

Description

The `mkl_ddiatrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the diagonal:

$A*y = x$

or

*A'*y = x*,

where:

x and *y* are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

uplo	CHARACTER*1. Specifies whether the upper or low triangle of the matrix *A* is considered. If *uplo* = 'U' or 'u', the upper triangle of the matrix *A* is used. If *uplo* = 'L' or 'l', the low triangle of the matrix *A* is used.
transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', then *A*y = x* If *transa* = 'T' or 't' or 'C' or 'c', then *A'*y = x*,
diag	CHARACTER*1. Specifies whether *A* is a unit triangular or not. If *diag* = 'U' or 'u', *A* is assumed to be a unit triangular. If *diag* = 'N' or 'n', *A* is not assumed to be a unit triangular.
m	INTEGER. Number of rows of the matrix *A*.
val	REAL*8. Two-dimensional array of size *lval* by *ndiag*, contains non-zero diagonals of the matrix *A*. Refer to *values* array description in Diagonal Storage Schemefor more details.
lval	INTEGER. Leading dimension of *val*, *lval*≥*m*. Refer to *lval* description in Diagonal Storage Scheme for more details.
idiag	INTEGER. Array of length *ndiag*, contains the distances between main diagonal and each non-zero diagonals in the matrix *A*.

	Refer to *distance* array description in Diagonal Storage Scheme for more details.
ndiag	INTEGER. Specifies the number of non-zero diagonals of the matrix *A*.
x	REAL*8. Array, DIMENSION is *m*. Before entry, the array *x* must contain the vector *x*.

Output Parameters

y	REAL*8. Array, DIMENSION at least *m*. Contains the vector *y*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_ddiatrsv(uplo, transa, diag, m, val, lval, idiag, ndiag, x,
y)

  CHARACTER*1   uplo, transa, diag

  INTEGER       m, lval, ndiag

  INTEGER       indiag(*)

  REAL*8        val(lval,*), x(*), y(*)
```

C:

```
void mkl_ddiatrsv(char *uplo, char *transa, char *diag, int *m, double  *val,
 int *lval, int *idiag, int *ndiag, double *x, double *y);
```

mkl_dskysv

Solves a system of linear equations for a sparse matrix in the skyline format.

Syntax

Fortran:

```
call mkl_dskysv(transa, m, alpha, matdescra, val, pntr, x, y)
```

C:

```
mkl_dskysv(&transa, &m, &alpha, matdescra, val, pntr, x, y);
```

Description

The `mkl_dskysv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the skyline storage format:

y := *alpha**inv(*A*)**x*

or

y := *alpha**inv(*A*')**x*,

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', then *y* := *alpha**inv(*A*)**x* If *transa* = 'T' or 't' or 'C' or 'c', then *y* := *alpha**inv(*A*')* *x*,
m	INTEGER. Number of rows of the matrix *A*.
alpha	REAL*8. Specifies the scalar *alpha*.

matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
val	REAL*8. Array containing the set of elements of the matrix *A* in the skyline profile form. If *matdescrsa*(2)= 'L', then *val* contains elements from the low triangle of the matrix *A*. If *matdescrsa*(2)= 'U', then *val* contains elements from the upper triangle of the matrix *A*. Refer to *values* array description in Skyline Storage Scheme for more details.
pntr	INTEGER. Array of length (*m*+*m*) for lower triangle, and (*k*+*k*) for upper triangle. It contains the indices specifying in the *val* the positions of the first element in each row (column) of the matrix *A*. Refer to *pointers* array description in Skyline Storage Scheme for more details.
x	REAL*8. Array, DIMENSION at least *m*. Before entry, the array *x* must contain the vector *x*. The elements are accessed with unit increment.
y	REAL*8. Array, DIMENSION at least *m*. Before entry, the array *y* must contain the vector *y*. The elements are accessed with unit increment.

Output Parameters

y	Contains solution vector *x*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dskysv(transa, m, alpha, matdescra, val, pntr, x, y)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
```

```
INTEGER        m
INTEGER        pntr(*)
REAL*8         alpha
REAL*8         val(*), x(*), y(*)
```

C:

```
void mkl_dskysv(char *transa, int *m, double *alpha, char *matdescra, double
 *val, int *pntr, double *x, double *y);
```

mkl_dcsrmm

Computes matrix - matrix product of a sparse matrix stored in the CSR format.

Syntax

Fortran:

```
call mkl_dcsrmm(transa, m, n, k, alpha, matdescra, val, indx, pntrb, pntre,
b, ldb, beta, c, ldc)
```

C:

```
mkl_dcsrmm(&transa, &m, &n, &k, &alpha, matdescra, val, indx, pntrb, pntre,
b, &ldb, &beta, c, &ldc);
```

Description

The `mkl_dcsrmm` routine performs a matrix-matrix operation defined as

`C := alpha*A*B + beta*C`

or

`C := alpha*A'*B + beta*C`,

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in compressed sparse row format, *A'* is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', the matrix-matrix product is computed as *C* := *alpha*A*B* + *beta*C* If *transa* = 'T' or 't' or 'C' or 'c', the matrix-vector product is computed as *C* := *alpha*A'*B* + *beta*C*,
m	INTEGER. Number of rows of the matrix *A*.
n	INTEGER. Number of columns of the matrix *C*.
k	INTEGER. Number of columns of the matrix *A*.
alpha	REAL*8. Specifies the scalar *alpha*.
matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
val	REAL*8. Array containing non-zero elements of the matrix *A*. Its length is *pntre*(*m*) – *pntrb*(1). Refer to *values* array description in CSR Format for more details.
indx	INTEGER. Array containing the column indices for each non-zero element of the matrix *A*. Its length is equal to length of the *val* array. Refer to *columns* array description in CSR Format for more details.
pntrb	INTEGER. Array of length *m*, contains row indices, such that *pntrb*(*i*) – *pntrb*(1)+1 is the starting index of row *i* in the arrays *val* and *indx*. Refer to *pointerb* array description in CSR Format for more details.
pntre	INTEGER. Array of length *m*, contains row indices, such that *pntre*(*i*) – *pntrb*(1) is the last index of row *i* in the arrays *val* and *indx*. Refer to *pointerE* array description in CSR Format for more details.
b	REAL*8. Array, DIMENSION (*ldb*, *n*).

	Before entry with *transa*= 'N' or 'n', the leading *k*-by-*n* part of the array *b* must contain the matrix *B*, otherwise the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.
ldb	INTEGER. Specifies the first dimension of *b* as declared in the calling (sub)program.
beta	REAL*8. Specifies the scalar *beta*.
c	REAL*8. Array, DIMENSION (*ldc*, *n*). Before entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, otherwise the leading *k*-by-*n* part of the array *c* must contain the matrix *C*.
ldc	INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program.

Output Parameters

c	Overwritten by the matrix (*alpha***A***B* + *beta** *C*) or (*alpha***A*'**B* + *beta***C*).

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dcsrmm(transa, m, n, k, alpha, matdescra, val, indx, pntrb,
pntre, b, ldb, beta, c, ldc)

  CHARACTER*1   transa
  CHARACTER     matdescra(*)

  INTEGER       m, n, k, ldb, ldc
  INTEGER       indx(*), pntrb(m), pntre(m)
  REAL*8        alpha, beta
  REAL*8        val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_dcsrmm(char *transa, int *m, int *n, int *k, double *alpha, char
*matdescra, double  *val, int *indx, int *pntrb, int *pntre, double *b, int
 *ldb, double *beta, double *c, int *ldc,);
```

mkl_dcscmm

Computes matrix-matrix product of a sparse matrix stored in the CSC format.

Syntax

Fortran:

```
call mkl_dcscmm(transa, m, n, k, alpha, matdescra, val, indx, pntrb, pntre,
b, ldb, beta, c, ldc)
```

C:

```
mkl_dcscmm(&transa, &m, &n, &k, &alpha, matdescra, val, indx, pntrb, pntre,
b, &ldb, &beta, c, &ldc);
```

Description

The `mkl_dcscmm` routine performs a matrix-matrix operation defined as

`C := alpha*A*B + beta*C`

or

`C := alpha*A'*B + beta*C`,

where:

`alpha` and `beta` are scalars,

`B` and `C` are dense matrices, `A` is an `m`-by-`k` sparse matrix in compressed sparse column format, `A'` is the transpose of `A`.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

`transa`	`CHARACTER*1`. Specifies the operation to be performed.

	If *transa* = 'N' or 'n', the matrix-matrix product is computed as *C* := *alpha***A** *B* + *beta***C* If *transa* = 'T' or 't' or 'C' or 'c', the matrix-vector product is computed as *C* := *alpha***A*'**B* + *beta***C*,
m	INTEGER. Number of rows of the matrix *A*.
n	INTEGER. Number of columns of the matrix *C*.
k	INTEGER. Number of columns of the matrix *A*.
alpha	REAL*8. Specifies the scalar *alpha*.
matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
val	REAL*8. Array containing non-zero elements of the matrix *A*. Its length is *pntre*(*k*) – *pntrb*(1). Refer to *values* array description in CSC Format for more details.
indx	INTEGER. Array containing the row indices for each non-zero element of the matrix *A*.Its length is equal to length of the *val* array. Refer to *rows* array description in CSC Format for more details.
pntrb	INTEGER. Array of length *k*, contains row indices, such that *pntrb*(*i*) – *pntrb*(1)+1 is the starting index of column *i* in the arrays *val* and *indx*. Refer to *pointerb* array description in CSC Format for more details.
pntre	INTEGER. Array of length *k*, contains row indices, such that *pntre*(*i*) – *pntrb*(1) is the last index of column *i* in the arrays *val* and *indx*. Refer to *pointerE* array description in CSC Format for more details.
b	REAL*8. Array, DIMENSION (*ldb*, *n*). Before entry with *transa* = 'N' or 'n', the leading *k*-by-*n* part of the array *b* must contain the matrix *B*, otherwise the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.
ldb	INTEGER. Specifies the first dimension of *b* as declared in the calling (sub)program.

beta	REAL*8. Specifies the scalar *beta*.
c	REAL*8. Array, DIMENSION (*ldc*, *n*). Before entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, otherwise the leading *k*-by-*n* part of the array *c* must contain the matrix *C*.
ldc	INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program.

Output Parameters

c	Overwritten by the matrix (*alpha*A*B + beta* C*) or (*alpha*A'*B + beta*C*).

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dcscmm(transa, m, n, k, alpha, matdescra, val, indx, pntrb,
pntre, b, ldb, beta, c, ldc)

  CHARACTER*1   transa

  CHARACTER     matdescra(*)


  INTEGER       m, n, k, ldb, ldc

  INTEGER       indx(*), pntrb(k), pntre(k)

  REAL*8        alpha, beta

  REAL*8        val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_dcscmm(char *transa, int *m, int *n, int *k, double *alpha, char
*matdescra, double  *val, int *indx, int *pntrb, int *pntre, double *b, int
 *ldb, double *beta, double *c, int *ldc);
```

mkl_dcoomm

Computes matrix-matrix product of a sparse matrix stored in the coordinate format.

Syntax

Fortran:

```
call mkl_dcoomm(transa, m, n, k, alpha, matdescra, val, rowind, colind, nnz, b, ldb, beta, c, ldc)
```

C:

```
mkl_dcoomm(&transa, &m, &n, &k, &alpha, matdescra, val, rowind, colind, &nnz, b, &ldb, &beta, c, &ldc);
```

Description

The `mkl_dcoomm` routine performs a matrix-matrix operation defined as

`C := alpha*A*B + beta*C`

or

`C := alpha*A'*B + beta*C`,

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in the coordinate format, *A'* is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', the matrix-matrix product is computed as *C* := *alpha*A*B* + *beta*C* If *transa* = 'T' or 't' or 'C' or 'c', the matrix-vector product is computed as *C* := *alpha*A'*B* + *beta*C*,
m	INTEGER. Number of rows of the matrix *A*.

n	INTEGER. Number of columns of the matrix *C*.
k	INTEGER. Number of columns of the matrix *A*.
alpha	REAL*8. Specifies the scalar *alpha*.
matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
val	REAL*8. Array of length *nnz*, contains non-zero elements of the matrix *A* in the arbitrary order. Refer to *values* array description in Coordinate Format for more details.
rowind	INTEGER. Array of length *nnz*, contains the row indices for each non-zero element of the matrix *A*. Refer to *rows* array description in Coordinate Format for more details.
colind	INTEGER. Array of length *nnz*, contains the column indices for each non-zero element of the matrix *A*. Refer to *columns* array description in Coordinate Format for more details.
nnz	INTEGER. Specifies the number of non-zero element of the matrix *A*. Refer to *nnz* description in Coordinate Format for more details.
b	REAL*8. Array, DIMENSION (*ldb*, *n*). Before entry with *transa* = 'N' or 'n', the leading *k*-by-*n* part of the array *b* must contain the matrix *B*, otherwise the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.
ldb	INTEGER. Specifies the first dimension of *b* as declared in the calling (sub)program.
beta	REAL*8. Specifies the scalar *beta*.
c	REAL*8. Array, DIMENSION (*ldc*, *n*). Before entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, otherwise the leading *k*-by-*n* part of the array *c* must contain the matrix *C*.

ldc	INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program.

Output Parameters

c	Overwritten by the matrix (*alpha*A*B + beta*C*) or (*alpha*A'*B + beta*C*).

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dcoomm(transa, m, n, k, alpha, matdescra, val, rowind, colind,
 nnz, b, ldb, beta, c, ldc)

  CHARACTER*1    transa

  CHARACTER      matdescra(*)


  INTEGER        m, n, k, ldb, ldc, nnz

  INTEGER        rowind(*), colind(*)

  REAL*8         alpha, beta

  REAL*8         val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_dcoomm(char *transa, int *m, int *n, int *k, double *alpha, char
*matdescra, double  *val, int *rowind, int *colind, int *nnz, double *b, int
 *ldb, double *beta, double *c, int *ldc);
```

mkl_ddiamm

Computes matrix-matrix product of a sparse matrix stored in the diagonal format.

Syntax

Fortran:

```
call mkl_ddiamm(transa, m, n, k, alpha, matdescra, val, lval, idiag, ndiag,
b, ldb, beta, c, ldc)
```

C:

```
mkl_ddiamm(&transa, &m, &n, &k, &alpha, matdescra, val, &lval, idiag, &ndiag,
b, &ldb, &beta, c, &ldc);
```

Description

The `mkl_ddiamm` routine performs a matrix-matrix operation defined as

`C := alpha*A*B + beta*C`

or

`C := alpha*A'*B + beta*C`,

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in the diagonal format, *A'* is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', the matrix-matrix product is computed as *C* := *alpha*A*B* + *beta*C*, If *transa* = 'T' or 't' or 'C' or 'c', the matrix-vector product is computed as *C* := *alpha*A'*B* + *beta*C*.
m	INTEGER. Number of rows of the matrix *A*.
n	INTEGER. Number of columns of the matrix *C*.
k	INTEGER. Number of columns of the matrix *A*.
alpha	REAL*8. Specifies the scalar *alpha*.
matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .

val	REAL*8. Two-dimensional array of size *lval* by *ndiag*, contains non-zero diagonals of the matrix *A*. Refer to *values* array description in Diagonal Storage Scheme for more details.
lval	INTEGER. Leading dimension of *val*, *lval*≥min(*m*, *k*). Refer to *lval* description in Diagonal Storage Scheme for more details.
idiag	INTEGER. Array of length *ndiag*, contains the distances between main diagonal and each non-zero diagonals in the matrix *A*. Refer to *distance* array description in Diagonal Storage Scheme for more details.
ndiag	INTEGER. Specifies the number of non-zero diagonals of the matrix *A*.
b	REAL*8. Array, DIMENSION (*ldb*, *n*). Before entry with *transa* = 'N' or 'n', the leading *k*-by-*n* part of the array *b* must contain the matrix *B*, otherwise the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.
ldb	INTEGER. Specifies the first dimension of *b* as declared in the calling (sub)program.
beta	REAL*8. Specifies the scalar *beta*.
c	REAL*8. Array, DIMENSION (*ldc*, *n*). Before entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, otherwise the leading *k*-by-*n* part of the array *c* must contain the matrix *C*.
ldc	INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program.

Output Parameters

c	Overwritten by the matrix (*alpha***A***B* + *beta***C*) or (*alpha***A*'**B* + *beta***C*).

Interfaces

Fortran 77:

```
SUBROUTINE mkl_ddiamm(transa, m, n, k, alpha, matdescra, val, lval, idiag,
ndiag, b, ldb, beta, c, ldc)

  CHARACTER*1    transa

  CHARACTER      matdescra(*)


  INTEGER        m, n, k, ldb, ldc, lval, ndiag

  INTEGER       idiag(*)

  REAL*8         alpha, beta

  REAL*8         val(lval,*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_ddiamm(char *transa, int *m, int *n, int *k, double *alpha, char
*matdescra, double *val, int *lval, int *idiag, int *ndiag, double *b, int
*ldb, double *beta, double *c, int *ldc);
```

mkl_dskymm

Computes matrix-matrix product of a sparse matrix stored using the skyline storage scheme.

Syntax

Fortran:

```
call mkl_dskymm(transa, m, n, k, alpha, matdescra, val, pntr, b, ldb, beta,
c, ldc)
```

C:

```
mkl_dskymm(&transa, &m, &n, &k, &alpha, matdescra, val, pntr, b, &ldb, &beta,
c, &ldc);
```

Description

The `mkl_dskymm` routine performs a matrix-matrix operation defined as

```
C := alpha*A*B + beta*C
```

or

C := *alpha***A*'**B* + *beta***C*,

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in the skyline storage format, *A*' is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', the matrix-matrix product is computed as *C* := *alpha***A***B* + *beta***C*, If *transa* = 'T' or 't' or 'C' or 'c', the matrix-vector product is computed as *C* := *alpha***A*'**B* + *beta***C*,
m	INTEGER. Number of rows of the matrix *A*.
n	INTEGER. Number of columns of the matrix *C*.
k	INTEGER. Number of columns of the matrix *A*.
alpha	REAL*8. Specifies the scalar *alpha*.
matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
val	REAL*8. Array containing the set of elements of the matrix *A* in the skyline profile form. If *matdescrsa*(2)= 'L', then *val* contains elements from the low triangle of the matrix *A*. If *matdescrsa*(2)= 'U', then *val* contains elements from the upper triangle of the matrix *A*. Refer to *values* array description in Skyline Storage Schemefor more details.
pntr	INTEGER. Array of length (*m*+*m*) for lower triangle, and (*k*+*k*) for upper triangle.

	It contains the indices specifying in the *val* the positions of the first element in each row (column) of the matrix *A*. Refer to *pointers* array description in Skyline Storage Scheme for more details.
b	REAL*8. Array, DIMENSION (*ldb*, *n*). Before entry with *transa* = 'N' or 'n', the leading *k*-by-*n* part of the array *b* must contain the matrix *B*, otherwise the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.
ldb	INTEGER. Specifies the first dimension of *b* as declared in the calling (sub)program.
beta	REAL*8. Specifies the scalar *beta*.
c	REAL*8. Array, DIMENSION (*ldc*, *n*). Before entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, otherwise the leading *k*-by-*n* part of the array *c* must contain the matrix *C*.
ldc	INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program.

Output Parameters

c	Overwritten by the matrix (*alpha*A*B + beta*C*) or (*alpha*A'*B + beta*C*).

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dskymm(transa, m, n, k, alpha, matdescra, val, pntr, b, ldb,
 beta, c, ldc)

  CHARACTER*1   transa

  CHARACTER     matdescra(*)
```

```
INTEGER        m, n, k, ldb, ldc
INTEGER        pntr(*)
REAL*8         alpha, beta
REAL*8         val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_dskymm(char *transa, int *m, int *n, int *k, double *alpha, char
*matdescra, double *val, int *pntr, double *b, int *ldb, double *beta, double
 *c, int *ldc);
```

mkl_dcsrsm

Solves a system of linear matrix equations for a sparse matrix in the CSR format.

Syntax

Fortran:

```
call mkl_dcsrsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b,
ldb, c, ldc)
```

C:

```
mkl_dcsrsm(&transa, &m, &n, &alpha, matdescra, val, indx, pntrb, pntre, b,
&ldb, c, &ldc);
```

Description

The `mkl_dcsrsm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the CSR format:

`C := alpha*inv(A)*B`

or

`C := alpha*inv(A')*B`,

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', the matrix-matrix product is computed as *C* := *alpha**inv(*A*)**B* If *transa* = 'T' or 't' or 'C' or 'c', the matrix-vector product is computed as *C* := *alpha**inv(*A*')**B*,
m	INTEGER. Number of columns of the matrix *A*.
n	INTEGER. Number of columns of the matrix *C*.
alpha	REAL*8. Specifies the scalar *alpha*.
matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
val	REAL*8. Array containing non-zero elements of the matrix *A*. Its length is *pntre*(*m*) – *pntrb*(1). Refer to *values* array description in CSR Format for more details.
indx	INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.Its length is equal to length of the *val* array. Refer to *columns* array description in CSR Format for more details.
pntrb	INTEGER. Array of length *m*, contains row indices, such that *pntrb*(*i*) – *pntrb*(1)+1 is the starting index of row *i* in the arrays *val* and *indx*. Refer to *pointerb* array description in CSR Format for more details.
pntre	INTEGER. Array of length *m*, contains row indices, such that *pntre*(*i*) – *pntrb*(1) is the last index of row *i* in the arrays *val* and *indx*. Refer to *pointerE* array description in CSR Format for more details.
b	REAL*8. Array, DIMENSION (*ldb*, *n*).

	Before entry the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.
ldb	INTEGER. Specifies the first dimension of *b* as declared in the calling (sub)program.
ldc	INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program.

Output Parameters

c	REAL*8. Array, DIMENSION (*ldc*, *n*). The leading *m*-by-*n* part of the array *c* contains the output matrix *C*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dcsrsm(transa, m, n, alpha, matdescra, val, indx, pntrb,
pntre, b, ldb, c, ldc)

  CHARACTER*1   transa

  CHARACTER     matdescra(*)


  INTEGER       m, n, ldb, ldc

  INTEGER       indx(*), pntrb(m), pntre(m)

  REAL*8        alpha

  REAL*8        val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_dcsrsm(char *transa, int *m, int *n, double *alpha, char *matdescra,
 double *val, int *indx, int *pntrb, int *pntre, double *b, int *ldb, double
 *c, int *ldc);
```

mkl_dcscsm

Solves a system of linear matrix equations for a sparse matrix in the CSC format.

Syntax

Fortran:

```
call mkl_dcscsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c, ldc)
```

C:

```
mkl_dcscsm(&transa, &m, &n, &alpha, matdescra, val, indx, pntrb, pntre, b, &ldb, c, &ldc);
```

Description

The `mkl_dcscsm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the CSC format:

`C := alpha*inv(A)*B`

or

`C := alpha*inv(A')*B`,

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', the matrix-matrix product is computed as *C* := *alpha**inv(*A*)**B* If *transa* = 'T' or 't' or 'C' or 'c', the matrix-vector product is computed as *C* := *alpha**inv(*A'*)**B*,
m	INTEGER. Number of columns of the matrix *A*.

n	INTEGER. Number of columns of the matrix *C*.
alpha	REAL*8. Specifies the scalar *alpha*.
matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
val	REAL*8. Array containing non-zero elements of the matrix *A*. Its length is *pntre*(*m*) – *pntrb*(1). Refer to *values* array description in CSC Format for more details.
indx	INTEGER. Array containing the row indices for each non-zero element of the matrix *A*. Its length is equal to length of the *val* array. Refer to *rows* array description in CSC Format for more details.
pntrb	INTEGER. Array of length *k*, contains row indices, such that *pntrb*(*i*) – *pntrb*(1)+1 is the starting index of column *i* in the arrays *val* and *indx*. Refer to *pointerb* array description in CSC Format for more details.
pntre	INTEGER. Array of length *k*, contains row indices, such that *pntre*(*i*) – *pntrb*(1) is the last index of column *i* in the arrays *val* and *indx*. Refer to *pointerE* array description in CSC Format for more details.
b	REAL*8. Array, DIMENSION (*ldb*, *n*). Before entry the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.
ldb	INTEGER. Specifies the first dimension of *b* as declared in the calling (sub)program.
ldc	INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program.

Output Parameters

c	REAL*8. Array, DIMENSION (*ldc*, *n*). The leading *m*-by-*n* part of the array *c* contains the output matrix *C*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dcscsm(transa, m, n, alpha, matdescra, val, indx, pntrb,
pntre, b, ldb, c, ldc)

  CHARACTER*1    transa

  CHARACTER      matdescra(*)


  INTEGER        m, n, ldb, ldc

  INTEGER        indx(*), pntrb(m), pntre(m)

  REAL*8         alpha

  REAL*8         val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_dcscsm(char *transa, int *m, int *n, double *alpha, char *matdescra,
 double *val, int *indx, int *pntrb, int *pntre, double *b, int *ldb, double
 *c, int *ldc);
```

mkl_dcoosm

Solves a system of linear matrix equations for a sparse matrix in the coordinate format.

Syntax

Fortran:

```
call mkl_dcoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b,
ldb, c, ldc)
```

C:

```
mkl_dcoosm(&transa, &m, &n, &alpha, matdescra, val, rowind, colind, &nnz, b,
&ldb, c, &ldc);
```

Description

The mkl_dcoosm routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the coordinate format:

C := *alpha**inv(*A*)**B*

or

C := *alpha**inv(*A*')**B*,

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A*' is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', the matrix-matrix product is computed as *C* := *alpha**inv(*A*)**B* If *transa* = 'T' or 't' or 'C' or 'c', the matrix-vector product is computed as *C* := *alpha**inv(*A*')**B*,
m	INTEGER. Number of rows of the matrix *A*.
n	INTEGER. Number of columns of the matrix *C*.
alpha	REAL*8. Specifies the scalar *alpha*.
matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
val	REAL*8. Array of length *nnz*, contains non-zero elements of the matrix *A* in the arbitrary order. Refer to *values* array description in Coordinate Format for more details.
rowind	INTEGER. Array of length *nnz*, contains the row indices for each non-zero element of the matrix *A*. Refer to *rows* array description in Coordinate Format for more details.

colind	INTEGER. Array of length *nnz*, contains the column indices for each non-zero element of the matrix *A*. Refer to *columns* array description in Coordinate Format for more details.
nnz	INTEGER. Specifies the number of non-zero element of the matrix *A*. Refer to *nnz* description in Coordinate Format for more details.
b	REAL*8. Array, DIMENSION (*ldb*, *n*). Before entry the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.
ldb	INTEGER. Specifies the first dimension of *b* as declared in the calling (sub)program.
ldc	INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program.

Output Parameters

c	REAL*8. Array, DIMENSION (*ldc*, *n*). The leading *m*-by-*n* part of the array *c* contains the output matrix *C*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dcoosm(transa, m, n, alpha, matdescra, val, rowind, colind,
nnz, b, ldb, c, ldc)

  CHARACTER*1    transa

  CHARACTER      matdescra(*)


  INTEGER        m, n, ldb, ldc, nnz

  INTEGER        rowind(*), colind(*)

  REAL*8         alpha

  REAL*8         val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_dcoosm(char *transa, int *m, int *n, double *alpha, char *matdescra,
 double *val, int *rowind, int *colind, int *nnz, double *b, int *ldb, double
 *c, int *ldc);
```

mkl_ddiasm

Solves a system of linear matrix equations for a sparse matrix in the diagonal format.

Syntax

Fortran:

```
call mkl_ddiasm(transa, m, n, alpha, matdescra, val, lval, idiag, ndiag, b,
ldb, c, ldc)
```

C:

```
mkl_ddiasm(&transa, &m, &n, &alpha, matdescra, val, &lval, idiag, &ndiag, b,
&ldb, c, &ldc);
```

Description

The `mkl_ddiasm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the diagonal format:

`C := alpha*inv(A)*B`

or

`C := alpha*inv(A')*B`,

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed.

	If *transa* = 'N' or 'n', the matrix-matrix product is computed as *C* := *alpha**inv(*A*)**B*, If *transa* = 'T' or 't' or 'C' or 'c', the matrix-vector product is computed as *C* := *alpha**inv(*A*')**B*.
m	INTEGER. Number of rows of the matrix *A*.
n	INTEGER. Number of columns of the matrix *C*.
alpha	REAL*8. Specifies the scalar *alpha*.
matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
val	REAL*8. Two-dimensional array of size *lval* by *ndiag*, contains non-zero diagonals of the matrix *A*. Refer to *values* array description in Diagonal Storage Scheme for more details.
lval	INTEGER. Leading dimension of *val*, *lval*≥*m*. Refer to *lval* description in Diagonal Storage Scheme for more details.
idiag	INTEGER. Array of length *ndiag*, contains the distances between main diagonal and each non-zero diagonals in the matrix *A*. Refer to *distance* array description in Diagonal Storage Scheme for more details.
ndiag	INTEGER. Specifies the number of non-zero diagonals of the matrix *A*.
b	REAL*8. Array, DIMENSION (*ldb*, *n*). Before entry the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.
ldb	INTEGER. Specifies the first dimension of *b* as declared in the calling (sub)program.
ldc	INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program.

Output Parameters

c	REAL*8. Array, DIMENSION (*ldc*, *n*).

The leading *m*-by-*n* part of the array *c* contains the matrix *C*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_ddiasm(transa, m, n, alpha, matdescra, val, lval, idiag,
ndiag, b, ldb, c, ldc)

  CHARACTER*1    transa

  CHARACTER      matdescra(*)


  INTEGER        m, n, ldb, ldc, lval, ndiag

  INTEGER       idiag(*)

  REAL*8         alpha

  REAL*8         val(lval,*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_ddiasm(char *transa, int *m, int *n, double *alpha, char *matdescra,
 double *val, int *lval, int *idiag, int *ndiag, double *b, int *ldb, double
 *c, int *ldc);
```

mkl_dskysm

Solves a system of linear matrix equations for a sparse matrix stored using the skyline storage scheme.

Syntax

Fortran:

```
call mkl_dskysm(transa, m, n, alpha, matdescra, val, pntr, b, ldb, c, ldc)
```

C:

```
mkl_dskysm(&transa, &m, &n, &alpha, matdescra, val, pntr, b, &ldb, c, &ldc);
```

Description

The mkl_dskysm routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the skyline storage format:

C := *alpha**inv(*A*)**B*

or

C := *alpha**inv(*A*')**B*,

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A*' is the transpose of *A*.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the Fortran 77 standard types. Data types specific to the different interfaces are described in the section **"Interfaces"** below.

transa	CHARACTER*1. Specifies the operation to be performed. If *transa* = 'N' or 'n', the matrix-matrix product is computed as *C* := *alpha**inv(*A*)**B*, If *transa* = 'T' or 't' or 'C' or 'c', the matrix-vector product is computed as *C* := *alpha**inv(*A*')**B*,
m	INTEGER. Number of rows of the matrix *A*.
n	INTEGER. Number of columns of the matrix *C*.
alpha	REAL*8. Specifies the scalar *alpha*.
matdescra	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in the Table 2-6 .
val	REAL*8. Array containing the set of elements of the matrix *A* in the skyline profile form. If *matdescrsa*(2)= 'L', then *val* contains elements from the low triangle of the matrix *A*. If *matdescrsa*(2)= 'U', then *val* contains elements from the upper triangle of the matrix *A*. Refer to *values* array description in Skyline Storage Scheme for more details.

pntr	INTEGER. Array of length (*m*+*m*). It contains the indices specifying in the *val* the positions of the first non-zero element of each *i*-row (column) of the matrix *A* such that *pointers*(*i*)- *pointers*(1)+1. Refer to *pointers* array description in Skyline Storage Scheme for more details.
b	REAL*8. Array, DIMENSION (*ldb*, *n*). Before entry the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.
ldb	INTEGER. Specifies the first dimension of *b* as declared in the calling (sub)program.
ldc	INTEGER. Specifies the first dimension of *c* as declared in the calling (sub)program.

Output Parameters

c	REAL*8. Array, DIMENSION (*ldc*, *n*). The leading *m*-by-*n* part of the array *c* contains the matrix *C*.

Interfaces

Fortran 77:

```
SUBROUTINE mkl_dskysm(transa, m, n, alpha, matdescra, val, pntr, b, ldb, c,
 ldc)
  CHARACTER*1    transa
  CHARACTER      matdescra(*)
  INTEGER        m, n, ldb, ldc
  INTEGER        pntr(*)
  REAL*8         alpha
  REAL*8         val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_dskysm(char *transa, int *m, int *n, double *alpha, char *matdescra,
 double *val, int *pntr, double *b, int *ldb, double *c, int *ldc,);
```

LAPACK Routines: Linear Equations

3

This chapter describes the Intel® Math Kernel Library implementation of routines from the LAPACK package that are used for solving systems of linear equations and performing a number of related computational tasks. The library includes LAPACK routines for both real and complex data. Routines are supported for systems of equations with the following types of matrices:

- general
- banded
- symmetric or Hermitian positive-definite (both full and packed storage)
- symmetric or Hermitian positive-definite banded
- symmetric or Hermitian indefinite (both full and packed storage)
- symmetric or Hermitian indefinite banded
- triangular (both full and packed storage)
- triangular banded
- tridiagonal.

For each of the above matrix types, the library includes routines for performing the following computations:

- factoring the matrix (except for triangular matrices)
- equilibrating the matrix
- solving a system of linear equations
- estimating the condition number of a matrix
- refining the solution of linear equations and computing its error bounds
- inverting the matrix.

To solve a particular problem, you can call two or more computational routines or call a corresponding driver routine that combines several tasks in one call, such as `?gesv` for factoring and solving. For example, to solve a system of linear equations with a general matrix, call `?getrf` (*LU* factorization) and then `?getrs` (computing the solution). Then, call `?gerfs` to refine the solution and get the error bounds. Alternatively, use the driver routine `?gesvx` that performs all these tasks in one call.

WARNING. LAPACK routines expect that input matrices do not contain `INF` or `NaN` values. When input data is inappropriate for LAPACK, problems may arise, including possible hangs.

Starting from release 8.0, Intel MKL along with Fortran-77 interface to LAPACK computational and driver routines also supports Fortran-95 interface which uses simplified routine calls with shorter argument lists. The syntax section of the routine description gives the calling sequence for Fortran-95 interface immediately after Fortran-77 calls.

Routine Naming Conventions

To call each routine introduced in this chapter from the Fortran-77 program, you can use the LAPACK name.

LAPACK names are listed in Table 3-1 and Table 3-2 , and have the structure `?yyzzz` or `?yyzz`, which is described below.

The initial symbol `?` indicates the data type:

`s`	real, single precision
`c`	complex, single precision
`d`	real, double precision
`z`	complex, double precision

The second and third letters `yy` indicate the matrix type and storage scheme:

`ge`	general
`gb`	general band
`gt`	general tridiagonal
`po`	symmetric or Hermitian positive-definite
`pp`	symmetric or Hermitian positive-definite (packed storage)
`pb`	symmetric or Hermitian positive-definite band
`pt`	symmetric or Hermitian positive-definite tridiagonal
`sy`	symmetric indefinite
`sp`	symmetric indefinite (packed storage)
`he`	Hermitian indefinite
`hp`	Hermitian indefinite (packed storage)
`tr`	triangular
`tp`	triangular (packed storage)
`tb`	triangular band

For computational routines, the last three letters `zzz` indicate the computation performed:

trf	form a triangular matrix factorization
trs	solve the linear system with a factored matrix
con	estimate the matrix condition number
rfs	refine the solution and compute error bounds
tri	compute the inverse matrix using the factorization
equ	equilibrate a matrix.

For example, the sgetrf routine performs the triangular factorization of general real matrices in single precision; the corresponding routine for complex matrices is cgetrf.

For driver routines, the names can end with –sv (meaning a simple driver), or with –svx (meaning an expert driver).

Names of the LAPACK computational and driver routines for Fortran-95 interface in Intel MKL are the same as Fortran-77 names but without the first letter that indicates the data type. For example, the name of the routine that performs triangular factorization of general real matrices in Fortran-95 interface is getrf. Different data types are handled through defining a specific internal parameter that refers to a module block with named constants for single and double precision.

Fortran-95 Interface Conventions

Fortran-95 interface to LAPACK is implemented through wrappers that call respective Fortran-77 routines. This interface uses such features of Fortran-95 as assumed-shape arrays and optional arguments to provide simplified calls to LAPACK routines with fewer arguments.

The main conventions for Fortran-95 interface are as follows:

- The names of arguments used in Fortran-95 call are typically the same as for the respective generic (Fortran-77) interface. However, to reduce the number of argument names used in the library, Fortran-95 interface uses the following identity settings of formal argument names:

Generic Argument Name	Fortran-95 Argument Name
ap	*a*
ab	*a*
afb	*af*

Generic Argument Name	Fortran-95 Argument Name
afp	*af*
bp	*b*
bb	*b*
selctg	*select*

Note that these name changes of formal arguments have no impact on program semantics and follow the unification conventions.

- Input arguments such as array dimensions are not required in Fortran-95 and are skipped from the calling sequence. Array dimensions are reconstructed from the user data that must exactly follow the required array shape.

  Another type of generic arguments that are skipped in Fortran-95 interface are arguments that represent workspace arrays (such as *work*, *rwork*, and so on). The only exception are cases when workspace arrays return significant information on output.

  An argument can also be skipped if its value is completely defined by the presence or absence of another argument in the calling sequence, and the restored value is the only meaningful value for the skipped argument.
- Some generic arguments are declared as optional in Fortran-95 interface and may or may not be present in the calling sequence. An argument can be declared optional if it satisfies one of the following conditions:
  - If the argument value is completely defined by the presence or absence of another argument in the calling sequence, it can be declared as optional. The difference from the skipped argument in this case is that the optional argument can have some meaningful values that are distinct from the value reconstructed by default. For example, if some argument (like *jobz*) can take only two values and one of these values directly implies the use of another argument, then the value of *jobz* can be uniquely reconstructed from the actual presence or absence of this second argument, and *jobz* can be omitted.
  - If an input argument can take only a few possible values, it can be declared as optional. The default value of such argument is typically set as the first value in the list and all exceptions to this rule are explicitly stated in the routine description.
  - If an input argument has a natural default value, it can be declared as optional. The default value of such optional argument is set to its natural default value.
- Argument *info* is declared as optional in Fortran-95 interface. If it is present in the calling sequence, the value assigned to *info* is interpreted as follows:

  - If this value is more than -1000, its meaning is the same as in Fortran-77 routine.
  - If this value is equal to -1000, it means that there is not enough work memory.
  - If this value is equal to -1001, incompatible arguments are present in the calling sequence.

- Optional arguments are given in square brackets in Fortran-95 call syntax.

"Fortran-95 Notes" subsection at the end of the topic describing each routine details concrete rules for reconstructing the values of omitted optional parameters.

MKL Fortran-95 Interfaces for LAPACK Routines vs. Netlib Implementation

The following list presents general digressions of Intel MKL LAPACK-95 implementation from the Netlib analog:

- Intel® MKL Fortran-95 interfaces are provided for pure procedures.
- Names of interfaces do not contain `LA_` prefix.
- An optional array argument always has the `target` attribute.
- Functionality of MKL LAPACK-95 wrapper is close to FORTRAN-77 original implementation in `getrf`, `gbtrf`, and `potrf` interfaces.
- If *jobz* argument value specifies presence or absence of *z* argument, then *z* is always declared as optional and *jobz* is restored depending on whether *z* is present or not. It is not always so in the Netlib version (see "Modified Netlib Interfaces" in Appendix E).
- To avoid double error checking, processing of *info* parameter is limited to checking of allocated memory and disarranging of optional parameters.
- If an argument that is present in the list of arguments completely defines another argument, the latter is always declared as optional.

You can transform an application that uses the Netlib LAPACK interfaces to ensure its work with Intel MKL interfaces by meeting two conditions:

a. The application is correct, that is, unambiguous, compiler-independent, and contains no errors.

b. Each routine name denotes only one specific routine. If any routine name in the application coincides with a name of the original Netlib routine (for example, after removing `LA_` prefix) but denotes a routine different from that Netlib original routine, this name should be modified through context replacement.

You should transform your application in the following five cases (see Appendix E for specific differences of individual interfaces):

1. When using Netlib routines that differ from the Intel MKL routines only by the `LA_` prefix or in the array attribute target. The only transformation required in this case is context name replacement. See "Interfaces Identical to Netlib" in Appendix E for details.

2. When using Netlib routines that differ from the Intel MKL routines by the `LA_` prefix, the `target` array attribute, and the names of formal arguments. In the case of positional passing of arguments, no additional transformation except context name replacement is required. In the case of the key passing of arguments, in addition to the context name replacement the names of mismatching keys should also be modified. See "Interfaces with Replaced Argument Names" in Appendix E for details.
3. When using Netlib routines that differ from the Intel MKL routines by the `LA_` prefix, the `target` array attribute, sequence of the arguments, arguments missing in MKL but present in Netlib and, vice versa, present in MKL but missing in Netlib. Remove the differences in sequence and range of the arguments in process of all the transformations specified in items 2 and 3. See "Modified Netlib Interfaces" in Appendix E for details.
4. When using `getrf`, `gbtrf`, and `potrf` interfaces, that is, new functionality implemented in MKL but unavailable in Netlib source. To overcome the differences, build the desired functionality explicitly with MKL means or create a new subroutine with the new functionality, using specific MKL interfaces corresponding to LAPACK-77 routines. You can call the latter routines directly but using new MKL interfaces is preferable. See "Interfaces Absent From Netlib" and "Interfaces of New Functionality"in Appendix E for details.

   Note that if the transformed application calls `getrf`, `gbtrf` or `potrf` without controlling arguments *rcond* and *norm*, just context replacement is enough in modifying the calls into MKL interfaces, as described in point 1 above. Netlib functionality is preserved in such cases.
5. When using Netlib auxiliary routines. In this case, call a corresponding subroutine directly, using MKL LAPACK-77 interfaces.

You can transform your application as follows:

1. Make sure conditions a. and b. are met.
2. Select Netlib LAPACK-95 calls. For each call do the following:
   - Select the case of digression and do the required transformations.
   - Revise results to eliminate unneeded code or data, which may appear after several identical calls.
3. Make sure the transformations are correct and complete.

Matrix Storage Schemes

LAPACK routines use the following matrix storage schemes:

- `Full storage`: a matrix *A* is stored in a two-dimensional array *a*, with the matrix element a_{ij} stored in the array element *a*(*i*,*j*).

- Packed storage scheme allows you to store symmetric, Hermitian, or triangular matrices more compactly: the upper or lower triangle of the matrix is packed by columns in a one-dimensional array.
- Band storage: an m-by-n band matrix with kl sub-diagonals and ku superdiagonals is stored compactly in a two-dimensional array ab with $kl+ku+1$ rows and n columns. Columns of the matrix are stored in the corresponding columns of the array, and diagonals of the matrix are stored in rows of the array.

In Chapters 4 and 5 , arrays that hold matrices in packed storage have names ending in p; arrays with matrices in band storage have names ending in b.

For more information on matrix storage schemes, see "Matrix Arguments" in Appendix B.

Mathematical Notation

Descriptions of LAPACK routines use the following notation:

$Ax = b$	A system of linear equations with an n-by-n matrix $A = \{a_{ij}\}$, a right-hand side vector $b = \{b_i\}$, and an unknown vector $x = \{x_i\}$.
$AX = B$	A set of systems with a common matrix A and multiple right-hand sides. The columns of B are individual right-hand sides, and the columns of X are the corresponding solutions.
$\|x\|$	the vector with elements $\|x_i\|$ (absolute values of x_i).
$\|A\|$	the matrix with elements $\|a_{ij}\|$ (absolute values of a_{ij}).
$\|\|x\|\|_\infty = \max_i \|x_i\|$	The infinity-norm of the vector x.
$\|\|A\|\|_\infty = \max_i \Sigma_j \|a_{ij}\|$	The infinity-norm of the matrix A.
$\|\|A\|\|_1 = \max_j \Sigma_i \|a_{ij}\|$	The one-norm of the matrix A. $\|\|A\|\|_1 = \|\|A^T\|\|_\infty = \|\|A^H\|\|_\infty$
$\kappa(A) = \|\|A\|\| \ \|\|A^{-1}\|\|$	The condition number of the matrix A.

Error Analysis

In practice, most computations are performed with rounding errors. Besides, you often need to solve a system $Ax = b$, where the data (the elements of A and b) are not known exactly. Therefore, it is important to understand how the data errors and rounding errors can affect the solution x.

Data perturbations. If x is the exact solution of $Ax = b$, and $x + \delta x$ is the exact solution of a perturbed problem $(A + \delta A)x = (b + \delta b)$, then

$$\frac{\|\delta x\|}{\|x\|} \leq \kappa(A)\left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|}\right),$$

where

$$\kappa(A) = \|A\|\,\|A^{-1}\|.$$

In other words, relative errors in A or b may be amplified in the solution vector x by a factor $\kappa(A) = ||A||\ ||A^{-1}||$ called the condition number of A.

Rounding errors have the same effect as relative perturbations $c(n)\varepsilon$ in the original data. Here ε is the machine precision, and $c(n)$ is a modest function of the matrix order n. The corresponding solution error is

$||\delta x||/||x|| \leq c(n)\kappa(A)\varepsilon$. (The value of $c(n)$ is seldom greater than $10n$.)

Thus, if your matrix A is ill-conditioned (that is, its condition number $\kappa(A)$ is very large), then the error in the solution x is also large; you may even encounter a complete loss of precision. LAPACK provides routines that allow you to estimate $\kappa(A)$ (see Routines for Estimating the Condition Number) and also give you a more precise estimate for the actual solution error (see Refining the Solution and Estimating Its Error).

Computational Routines

Table 3-1 lists the LAPACK computational routines (Fortran-77 and Fortran-95 interfaces) for factorizing, equilibrating, and inverting real matrices, estimating their condition numbers, solving systems of equations with real matrices, refining the solution, and estimating its error. Table 3-2 lists similar routines for complex matrices. Respective routine names in Fortran-95 interface are without the first symbol (see Routine Naming Conventions).

Table 3-1 Computational Routines for Systems of Equations with Real Matrices

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general	?getrf	?geequ	?getrs	?gecon	?gerfs	?getri
general band	?gbtrf	?gbequ	?gbtrs	?gbcon	?gbrfs	
general tridiagonal	?gttrf		?gttrs	?gtcon	?gtrfs	
symmetric positive-definite	?potrf	?poequ	?potrs	?pocon	?porfs	?potri
symmetric positive-definite, packed storage	?pptrf	?ppequ	?pptrs	?ppcon	?pprfs	?pptri
symmetric positive-definite, band	?pbtrf	?pbequ	?pbtrs	?pbcon	?pbrfs	
symmetric positive-definite, tridiagonal	?pttrf		?pttrs	?ptcon	?ptrfs	
symmetric indefinite	?sytrf		?sytrs	?sycon	?syrfs	?sytri
symmetric indefinite, packed storage	?sptrf		?sptrs	?spcon	?sprfs	?sptri
triangular			?trtrs	?trcon	?trrfs	?trtri
triangular, packed storage			?tptrs	?tpcon	?tprfs	?tptri
triangular band			?tbtrs	?tbcon	?tbrfs	

In the table above, ? denotes s (single precision) or d (double precision) for Fortran-77 interface.

Table 3-2 Computational Routines for Systems of Equations with Complex Matrices

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general	?getrf	?geequ	?getrs	?gecon	?gerfs	?getri
general band	?gbtrf	?gbequ	?gbtrs	?gbcon	?gbrfs	
general tridiagonal	?gttrf		?gttrs	?gtcon	?gtrfs	
Hermitian positive-definite	?potrf	?poequ	?potrs	?pocon	?porfs	?potri
Hermitian positive-definite, packed storage	?pptrf	?ppequ	?pptrs	?ppcon	?pprfs	?pptri
Hermitian positive-definite, band	?pbtrf	?pbequ	?pbtrs	?pbcon	?pbrfs	
Hermitian positive-definite, tridiagonal	?pttrf		?pttrs	?ptcon	?ptrfs	
Hermitian indefinite	?hetrf		?hetrs	?hecon	?herfs	?hetri
symmetric indefinite	?sytrf		?sytrs	?sycon	?syrfs	?sytri
Hermitian indefinite, packed storage	?hptrf		?hptrs	?hpcon	?hprfs	?hptri
symmetric indefinite, packed storage	?sptrf		?sptrs	?spcon	?sprfs	?sptri
triangular			?trtrs	?trcon	?trrfs	?trtri
triangular, packed storage			?tptrs	?tpcon	?tprfs	?tptri

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
triangular band			?tbtrs	?tbcon	?tbrfs	

In the table above, ? stands for c (single precision complex) or z (double precision complex) for Fortran-77 interface.

Routines for Matrix Factorization

This section describes the LAPACK routines for matrix factorization. The following factorizations are supported:

- *LU* factorization
- Cholesky factorization of real symmetric positive-definite matrices
- Cholesky factorization of Hermitian positive-definite matrices
- Bunch-Kaufman factorization of real and complex symmetric matrices
- Bunch-Kaufman factorization of Hermitian matrices.

You can compute:

- the *LU* factorization using full and band storage of matrices
- the Cholesky factorization using full, packed, and band storage
- the Bunch-Kaufman factorization using full and packed storage.

?getrf

Computes the LU factorization of a general m-by-n matrix.

Syntax

Fortran 77:

```
call sgetrf(  m, n, a, lda, ipiv, info )
call dgetrf( m, n, a, lda, ipiv, info )
call cgetrf( m, n, a, lda, ipiv, info )
call zgetrf( m, n, a, lda, ipiv, info )
```

Fortran 95:

```
call getrf( a [,ipiv] [,info] )
```

Description

The routine computes the *LU* factorization of a general *m*-by-*n* matrix *A* as

*A = P*L*U*,

where *P* is a permutation matrix, *L* is lower triangular with unit diagonal elements (lower trapezoidal if *m* > *n*) and *U* is upper triangular (upper trapezoidal if *m* < *n*). Usually *A* is square (*m* = *n*), and both *L* and *U* are triangular. The routine uses partial pivoting, with row interchanges.

Input Parameters

m	INTEGER. The number of rows in the matrix *A* (*m* ≥ 0).
n	INTEGER. The number of columns in *A*; *n* ≥ 0.
a	REAL for sgetrf DOUBLE PRECISION for dgetrf COMPLEX for cgetrf DOUBLE COMPLEX for zgetrf. Array, DIMENSION (*lda*,*). Contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*).
lda	INTEGER. The first dimension of array *a*.

Output Parameters

a	Overwritten by *L* and *U*. The unit diagonal elements of *L* are not stored.
ipiv	INTEGER. Array, DIMENSION at least max(1,min(*m*, *n*)). The pivot indices: row *i* was interchanged with row *ipiv*(*i*).
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, u_{ii} is 0. The factorization has been completed, but *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `getrf` interface are as follows:

a	Holds the matrix A of size (m, n).
ipiv	Holds the vector of length $\min(m, n)$.

Application Notes

The computed L and U are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(\min(m,n))\varepsilon P|L||U|$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

The approximate number of floating-point operations for real flavors is

$(2/3)n^3$	If $m = n$,
$(1/3)n^2(3m-n)$	If $m > n$,
$(1/3)m^2(3n-m)$	If $m < n$.

The number of operations for complex flavors is four times greater.

After calling this routine with $m = n$, you can call the following:

?getrs	to solve $A*x = B$ or $A^TX = B$ or $A^HX = B$
?gecon	to estimate the condition number of A
?getri	to compute the inverse of A.

?gbtrf

Computes the LU factorization of a general m-by-n band matrix.

Syntax

Fortran 77:

```
call sgbtrf(  m, n, kl, ku, ab, ldab, ipiv, info )
call dgbtrf( m, n, kl, ku, ab, ldab, ipiv, info  )
call cgbtrf( m, n, kl, ku, ab, ldab, ipiv, info )
call zgbtrf( m, n, kl, ku, ab, ldab, ipiv, info  )
```

Fortran 95:

```
call gbtrf(  a [,kl] [,m] [,ipiv] [,info] )
```

Description

The routine forms the *LU* factorization of a general *m*-by-*n* band matrix *A* with *kl* non-zero subdiagonals and *ku* non-zero superdiagonals. Usually *A* is square (*m* = *n*), and then

*A = P*L*U*,

where *P* is a permutation matrix; *L* is lower triangular with unit diagonal elements and at most *kl* non-zero elements in each column; *U* is an upper triangular band matrix with *kl* + *ku* superdiagonals. The routine uses partial pivoting, with row interchanges (which creates the additional *kl* superdiagonals in *U*).

Input Parameters

m	INTEGER. The number of rows in matrix *A* (*m* ≥ 0).
n	INTEGER. The number of columns in matrix *A*; *n* ≥ 0.
kl	INTEGER. The number of subdiagonals within the band of *A*; *kl* ≥ 0.
ku	INTEGER. The number of superdiagonals within the band of *A*; *ku* ≥ 0.
ab	REAL for sgbtrf DOUBLE PRECISION for dgbtrf COMPLEX for cgbtrf DOUBLE COMPLEX for zgbtrf.

Array, DIMENSION (*ldab*,*). The array *ab* contains the matrix *A* in band storage (see Matrix Storage Schemes). The second dimension of *ab* must be at least max(1, *n*).

ldab INTEGER. The first dimension of the array *ab*. (*ldab* ≥ 2**kl* + *ku* + 1)

Output Parameters

ab Overwritten by *L* and *U*. The diagonal and *kl* + *ku* superdiagonals of *U* are stored in the first 1 + *kl* + *ku* rows of *ab*. The multipliers used to form *L* are stored in the next *kl* rows.

ipiv INTEGER.
Array, DIMENSION at least max(1,min(*m*, *n*)). The pivot indices: row *i* was interchanged with row *ipiv*(*i*).

info INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, u_{ii} is 0. The factorization has been completed, but *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gbtrf interface are as follows:

a Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (2**kl*+*ku*+1,*n*).

ipiv Holds the vector of length min(*m*,*n*).

kl If omitted, assumed *kl* = *ku*.

ku Restored as *ku* = *lda*-2**kl*-1.

m If omitted, assumed *m* = *n*.

Application Notes

The computed *L* and *U* are the exact factors of a perturbed matrix *A* + *E*, where

$|E| \leq c(kl+ku+1)\ \varepsilon\ P|L||U|$

$c(k)$ is a modest linear function of k, and ε is the machine precision.

The total number of floating-point operations for real flavors varies between approximately $2n(ku+1)kl$ and $2n(kl+ku+1)kl$. The number of operations for complex flavors is four times greater. All these estimates assume that kl and ku are much less than $\min(m,n)$.

After calling this routine with $m = n$, you can call the following routines:

gbtrs	to solve $A*X = B$ or $A^T*X = B$ or $A^H*X = B$
gbcon	to estimate the condition number of A.

?gttrf

Computes the LU factorization of a tridiagonal matrix.

Syntax

Fortran 77:

```
call sgttrf( n, dl, d, du, du2, ipiv, info )
call dgttrf( n, dl, d, du, du2, ipiv, info )
call cgttrf( n, dl, d, du, du2, ipiv, info )
call zgttrf( n, dl, d, du, du2, ipiv, info )
```

Fortran 95:

```
call gttrf( dl, d, du, du2 [, ipiv] [,info] )
```

Description

The routine computes the LU factorization of a real or complex tridiagonal matrix A in the form $A = P*L*U$,

where P is a permutation matrix; L is lower bidiagonal with unit diagonal elements; and U is an upper triangular matrix with nonzeroes in only the main diagonal and first two superdiagonals. The routine uses elimination with partial pivoting and row interchanges.

Input Parameters

n	INTEGER. The order of the matrix A; $n \geq 0$.
dl, d, du	REAL for sgttrf

DOUBLE PRECISION for dgttrf
COMPLEX for cgttrf
DOUBLE COMPLEX for zgttrf.
Arrays containing elements of *A*.
The array *dl* of dimension (*n* – 1) contains the subdiagonal elements of *A*.
The array *d* of dimension *n* contains the diagonal elements of *A*.
The array *du* of dimension (*n* – 1) contains the superdiagonal elements of *A*.

Output Parameters

dl — Overwritten by the (*n*–1) multipliers that define the matrix *L* from the *LU* factorization of *A*.

d — Overwritten by the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*.

du — Overwritten by the (*n*–1) elements of the first superdiagonal of *U*.

du2 — REAL for sgttrf
DOUBLE PRECISION for dgttrf
COMPLEX for cgttrf
DOUBLE COMPLEX for zgttrf.
Array, dimension (*n*–2). On exit, *du2* contains (*n*–2) elements of the second superdiagonal of *U*.

ipiv — INTEGER.
Array, dimension (*n*). The pivot indices: row *i* was interchanged with row *ipiv*(*i*).

info — INTEGER. If *info* = 0, the execution is successful.
If *info* = –*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, u_{ii} is 0. The factorization has been completed, but *U* is exactly singular. Division by zero will occur if you use the factor *U* for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gttrf interface are as follows:

dl	Holds the vector of length (*n*-1).
d	Holds the vectror of length (*n*).
du	Holds the vector of length (*n*-1).
du2	Holds the vector of length (*n*-2).
ipiv	Holds the vector of length (*n*).

Application Notes

?gbtrs	to solve *A*X = B* or $A^T*X = B$ or $A^H*X = B$
?gbcon	to estimate the condition number of *A*.

?potrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix.

Syntax

Fortran 77:

```
call spotrf(  uplo, n, a, lda, info )
call dpotrf(  uplo, n, a, lda, info )
call cpotrf(  uplo, n, a, lda, info )
call zpotrf(  uplo, n, a, lda, info )
```

Fortran 95:

```
call potrf(  a [, uplo] [,info]  )
```

Description

This routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix *A*:

$A = U^H*U$	if *uplo*='U'
$A = L*L^H$	if *uplo*='L',

where *L* is a lower triangular matrix and *U* is upper triangular.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of *A* is stored and how *A* is factored: If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix *A*, and *A* is factored as U^H*U. If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix *A*; *A* is factored as $L*L^H$.
n	INTEGER. The order of matrix *A*; $n \geq 0$.
a	REAL for spotrf DOUBLE PRECISION for dpotrf COMPLEX for cpotrf DOUBLE COMPLEX for zpotrf. Array, DIMENSION (*lda*,*). The array *a* contains either the upper or the lower triangular part of the matrix *A* (see *uplo*). The second dimension of *a* must be at least max(1, *n*).
lda	INTEGER. The first dimension of *a*.

Output Parameters

a	The upper or lower triangular part of *a* is overwritten by the Cholesky factor *U* or *L*, as specified by *uplo*.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix *A*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine potrf interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If *uplo* = 'U', the computed factor *U* is the exact factor of a perturbed matrix *A* + *E*, where

$$|E| \le c(n)\varepsilon |U^H||U|, |e_{ij}| \le c(n)\varepsilon\sqrt{a_{ii}a_{jj}}$$

c(*n*) is a modest linear function of *n*, and ε is the machine precision.

A similar estimate holds for *uplo* = 'L'.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

?potrs	to solve *A*X* = *B*
?pocon	to estimate the condition number of *A*
?potri	to compute the inverse of *A*.

?pptrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix using packed storage.

Syntax

Fortran 77:

```
call spptrf( uplo, n, ap, info )
call dpptrf( uplo, n, ap, info )
call cpptrf( uplo, n, ap, info )
call zpptrf( uplo, n, ap, info )
```

Fortran 95:

```
call pptrf( a [, uplo] [,info] )
```

Description

This routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite packed matrix *A*:

A = U^H**U* if *uplo*='U'

A = *L**L^H if *uplo*='L',

where *L* is a lower triangular matrix and *U* is upper triangular.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of *A* is packed in the array *ap*, and how *A* is factored:
If *uplo* = 'U', the array *ap* stores the upper triangular part of the matrix *A*, and *A* is factored as U^H**U*.
If *uplo* = 'L', the array *ap* stores the lower triangular part of the matrix *A*; *A* is factored as *L**L^H.

n INTEGER. The order of matrix *A*; $n \geq 0$.

ap REAL for spptrf
DOUBLE PRECISION for dpptrf
COMPLEX for cpptrf
DOUBLE COMPLEX for zpptrf.
Array, DIMENSION at least max(1, *n*(*n*+1)/2). The array *ap* contains either the upper or the lower triangular part of the matrix *A* (as specified by *uplo*) in packed storage (see Matrix Storage Schemes).

Output Parameters

ap The upper or lower triangular part of *A* in packed storage is overwritten by the Cholesky factor *U* or *L*, as specified by *uplo*.

info INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix *A*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `pptrf` interface are as follows:

a	Stands for argument *ap* in Fortran 77 interface. Holds the array *A* of size `(n*(n+1)/2)`.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

If *uplo* = `'U'`, the computed factor *U* is the exact factor of a perturbed matrix *A* + *E*, where

$$|E| \le c(n)\varepsilon |U^H||U|, |e_{ij}| \le c(n)\varepsilon\sqrt{a_{ii}a_{jj}}$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

A similar estimate holds for *uplo* = `'L'`.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors and $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

`?pptrs`	to solve *A***X* = *B*
`?ppcon`	to estimate the condition number of *A*
`?pptri`	to compute the inverse of *A*.

?pbtrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite band matrix.

Syntax

Fortran 77:

```
call spbtrf( uplo, n, kd, ab, ldab, info )
call dpbtrf( uplo, n, kd, ab, ldab, info )
call cpbtrf( uplo, n, kd, ab, ldab, info )
call zpbtrf( uplo, n, kd, ab, ldab, info )
```

Fortran 95:

```
call pbtrf( a [, uplo] [,info] )
```

Description

This routine forms the Cholesky factorization of a symmetric positive- definite or, for complex data, Hermitian positive-definite band matrix A:

$A = U^T*U$ for real data, $A = U^H*U$ for complex data	if *uplo*='U'
$A = L*L^T$ for real data, $A = L*L^H$ for complex data	if *uplo*='L',

where L is a lower triangular matrix and U is upper triangular.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored in the array *ab*, and how A is factored: If *uplo* = 'U', the upper triangle of A is stored. If *uplo* = 'L', the lower triangle of A is stored.
n	INTEGER. The order of matrix A; $n \geq 0$.

kd	INTEGER. The number of superdiagonals or subdiagonals in the matrix *A*(*kd* ≥ 0).
ab	REAL for spbtrf DOUBLE PRECISION for dpbtrf COMPLEX for cpbtrf DOUBLE COMPLEX for zpbtrf. Array, DIMENSION (,*). The array *ap* contains either the upper or the lower triangular part of the matrix *A* (as specified by *uplo*) in band storage (see Matrix Storage Schemes). The second dimension of *ab* must be at least max(1, *n*).
ldab	INTEGER. The first dimension of the array *ab*. (*ldab* ≥ *kd* + 1)

Output Parameters

ap	The upper or lower triangular part of *A* (in band storage) is overwritten by the Cholesky factor *U* or *L*, as specified by *uplo*.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix *A*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine pbtrf interface are as follows:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*kd*+1,*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If *uplo* = 'U', the computed factor *U* is the exact factor of a perturbed matrix *A* + *E*, where

$$|E| \leq c(kd+1)\varepsilon |U^H||U|, |e_{ij}| \leq c(kd+1)\varepsilon \sqrt{a_{ii}a_{jj}}$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

A similar estimate holds for `uplo = 'L'`.

The total number of floating-point operations for real flavors is approximately $n(kd+1)^2$. The number of operations for complex flavors is 4 times greater. All these estimates assume that kd is much less than n.

After calling this routine, you can call the following routines:

?pbtrs	to solve `A*X = B`
?pbcon	to estimate the condition number of `A`.

?pttrf

Computes the factorization of a symmetric (Hermitian) positive-definite tridiagonal matrix.

Syntax

Fortran 77:

```
call spttrf( n, d, e, info )
call dpttrf( n, d, e, info )
call cpttrf( n, d, e, info )
call zpttrf( n, d, e, info )
```

Fortran 95:

```
call pttrf( d, e [,info] )
```

Description

This routine forms the factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite tridiagonal matrix `A`:

`A = L*D*L'`,

where `D` is diagonal and `L` is unit lower bidiagonal. The factorization may also be regarded as having the form `A = U'*D*U`, where `D` is unit upper bidiagonal.

Input Parameters

n	INTEGER. The order of the matrix *A*; *n* ≥ 0.
d	REAL for sptrf, cpttrf DOUBLE PRECISION for dpttrf, zpttrf. Array, dimension (*n*). Contains the diagonal elements of *A*.
e	REAL for sptrf DOUBLE PRECISION for dpttrf COMPLEX for cpttrf DOUBLE COMPLEX for zpttrf. Array, dimension (*n* -1). Contains the subdiagonal elements of *A*.

Output Parameters

d	Overwritten by the *n* diagonal elements of the diagonal matrix *D* from the *L*D*L'* factorization of *A*.
e	Overwritten by the (*n* – 1) off-diagonal elements of the unit bidiagonal factor *L* or *U* from the factorization of *A*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = –*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite; if *i* < *n*, the factorization could not be completed, while if *i* = *n*, the factorization was completed, but *d*(*n*) ≤ 0.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine pttrf interface are as follows:

d	Holds the vector of length (*n*).
e	Holds the vector of length (*n*-1).

?sytrf

Computes the Bunch-Kaufman factorization of a symmetric matrix.

Syntax

Fortran 77:

```
call ssytrf( uplo, n, a, lda, ipiv, work, lwork, info )
call dsytrf( uplo, n, a, lda, ipiv, work, lwork, info )
call csytrf( uplo, n, a, lda, ipiv, work, lwork, info )
call zsytrf( uplo, n, a, lda, ipiv, work, lwork, info )
```

Fortran 95:

```
call sytrf( a [, uplo] [,ipiv]  [, info] )
```

Description

This routine forms the Bunch-Kaufman factorization of a symmetric matrix:

if *uplo*='U', *A* = $P*U*D*U^T*P^T$

if *uplo*='L', *A* = $P*L*D*L^T*P^T$,

where *A* is the input matrix, *P* is a permutation matrix, *U* and *L* are upper and lower triangular matrices with unit diagonal, and *D* is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. *U* and *L* have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of *D*.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of *A* is stored and how *A* is factored: If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix *A*, and *A* is factored as $P*U*D*U^T*P^T$. If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix *A*, and *A* is factored as $P*L*D*L^T*P^T$.
n	INTEGER. The order of matrix *A*; $n \geq 0$.

a	REAL for ssytrf DOUBLE PRECISION for dsytrf COMPLEX for csytrf DOUBLE COMPLEX for zsytrf. Array, DIMENSION (*lda*,*). The array *a* contains either the upper or the lower triangular part of the matrix *A* (see *uplo*). The second dimension of *a* must be at least max(1, *n*).
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).
work	Same type as *a*. A workspace array, dimension at least max(1,*lwork*).
lwork	INTEGER. The size of the *work* array (*lwork* ≥ *n*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	The upper or lower triangular part of *a* is overwritten by details of the block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L*).
work(1)	If *info*=0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
ipiv	INTEGER. Array, DIMENSION at least max(1, *n*). Contains details of the interchanges and the block structure of *D*. If *ipiv*(*i*) = *k* >0, then d_{ii} is a 1-by-1 block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column. If *uplo* = 'U' and *ipiv*(*i*) =*ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and (*i*–1)-th row and column of *A* was interchanged with the *m*-th row and column. If *uplo* = 'L' and *ipiv*(*i*) =*ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = –*i*, the *i*–th parameter had an illegal value.

If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular. Division by 0 will occur if you use *D* for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `sytrf` interface are as follows:

a	holds the matrix *A* of size (*n*, *n*)
ipiv	holds the vector of length (*n*)
uplo	must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

For better performance, try using *lwork* = *n*blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the `blocked algorithm`.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The 2-by-2 unit diagonal blocks and the unit diagonal elements of *U* and *L* are not stored. The remaining elements of *U* and *L* are stored in the corresponding columns of the array *a*, but additional row interchanges are required to recover *U* or *L* explicitly (which is seldom necessary).

If *ipiv*(*i*) = *i* for all *i* =1...*n*, then all off-diagonal elements of *U* (*L*) are stored explicitly in the corresponding elements of the array *a*.

If *uplo* = 'U', the computed factors *U* and *D* are the exact factors of a perturbed matrix *A* + *E*, where

$|E| \leq c(n)\varepsilon\ P|U||D||U^T|P^T$

$c(n)$ is a modest linear function of n, and ε is the machine precision. A similar estimate holds for the computed *L* and *D* when *uplo* = 'L'.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

?sytrs	to solve *A*X = B*
?sycon	to estimate the condition number of *A*
?sytri	to compute the inverse of *A*.

?hetrf

Computes the Bunch-Kaufman factorization of a complex Hermitian matrix.

Syntax

Fortran 77:

```
call chetrf( uplo, n, a, lda, ipiv, work, lwork, info )
call zhetrf( uplo, n, a, lda, ipiv, work, lwork, info )
```

Fortran 95:

```
call hetrf( a [, uplo] [,ipiv] [,info] )
```

Description

This routine forms the Bunch-Kaufman factorization of a Hermitian matrix:

if *uplo*='U', $A = P*U*D*U^H*P^T$

if *uplo*='L', $A = P*L*D*L^H*P^T$,

where A is the input matrix, P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored and how A is factored: If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix A, and A is factored as $P*U*D*U^H*P^T$. If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix A, and A is factored as $P*L*D*L^H*P^T$.
n	INTEGER. The order of matrix A; $n \geq 0$.
a, *work*	COMPLEX for chetrf DOUBLE COMPLEX for zhetrf. Arrays, DIMENSION *a*(*lda*,*), *work*(*). The array *a* contains the upper or the lower triangular part of the matrix A (see *uplo*). The second dimension of *a* must be at least max(1, *n*). *work*(*) is a workspace array of dimension at least max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).
lwork	INTEGER. The size of the *work* array ($lwork \geq n$). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	The upper or lower triangular part of *a* is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
ipiv	INTEGER.

	Array, DIMENSION at least max(1, *n*). Contains details of the interchanges and the block structure of *D*. If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column. If *uplo* = 'U' and *ipiv*(*i*) = *ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and the (*i*–1)-th row and column of *A* was interchanged with the *m*-th row and column. If *uplo* = 'L' and *ipiv*(*i*) = *ipiv*(*i*+1) = –*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and the (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = –*i*, the *i*-th parameter had an illegal value. If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular. Division by 0 will occur if you use *D* for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hetrf interface are as follows:

a	holds the matrix *A* of size (*n*, *n*)
ipiv	holds the vector of length (*n*)
uplo	must be 'U' or 'L'. The default value is 'U'.

Application Notes

This routine is suitable for Hermitian matrices that are not known to be positive-definite. If *A* is in fact positive-definite, the routine does not perform interchanges, and no 2-by-2 diagonal blocks occur in *D*.

For better performance, try using *lwork* = *n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The 2-by-2 unit diagonal blocks and the unit diagonal elements of *U* and *L* are not stored. The remaining elements of *U* and *L* are stored in the corresponding columns of the array *a*, but additional row interchanges are required to recover *U* or *L* explicitly (which is seldom necessary).

If *ipiv*(*i*) = *i* for all *i* =1...*n*, then all off-diagonal elements of *U* (*L*) are stored explicitly in the corresponding elements of the array *a*.

If *uplo* = 'U', the computed factors *U* and *D* are the exact factors of a perturbed matrix *A* + *E*, where

$|E| \leq c(n)\varepsilon P|U||D||U^T|P^T$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

A similar estimate holds for the computed *L* and *D* when *uplo* = 'L'.

The total number of floating-point operations is approximately $(4/3)n^3$.

After calling this routine, you can call the following routines:

?hetrs	to solve *A***X* = *B*
?hecon	to estimate the condition number of *A*
?hetri	to compute the inverse of *A*.

?sptrf

Computes the Bunch-Kaufman factorization of a symmetric matrix using packed storage.

Syntax

Fortran 77:

```
call ssptrf( uplo, n, ap, ipiv, info )
call dsptrf( uplo, n, ap, ipiv, info )
call csptrf( uplo, n, ap, ipiv, info )
call zsptrf( uplo, n, ap, ipiv, info )
```

Fortran 95:

```
call sptrf( a [,uplo] [,ipiv]  [,info] )
```

Description

This routine forms the Bunch-Kaufman factorization of a symmetric matrix A using packed storage:

if $uplo$='U', $A = P*U*D*U^T*P^T$

if $uplo$='L', $A = P*L*D*L^T*P^T$,

where P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is packed in the array ap and how A is factored: If $uplo$ = 'U', the array ap stores the upper triangular part of the matrix A, and A is factored as $P*U*D*U^T*P^T$. If $uplo$ = 'L', the array ap stores the lower triangular part of the matrix A, and A is factored as $P*L*D*L^T*P^T$.
n	INTEGER. The order of matrix A; $n \geq 0$.

ap REAL for ssptrf
DOUBLE PRECISION for dsptrf
COMPLEX for csptrf
DOUBLE COMPLEX for zsptrf.
Array, DIMENSION at least max(1, *n*(*n*+1)/2). The array *ap* contains the upper or the lower triangular part of the matrix *A* (as specified by *uplo*) in packed storage (see Matrix Storage Schemes).

Output Parameters

ap The upper or lower triangle of *A* (as specified by *uplo*) is overwritten by details of the block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L*).

ipiv INTEGER.
Array, DIMENSION at least max(1, *n*). Contains details of the interchanges and the block structure of *D*. If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.
If *uplo* = 'U' and *ipiv*(*i*) = *ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and the (*i*–1)-th row and column of *A* was interchanged with the *m*-th row and column.
If *uplo* = 'L' and *ipiv*(*i*) = *ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and the (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.

info INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular. Division by 0 will occur if you use *D* for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine sptrf interface are as follows:

a stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).

ipiv	holds the vector of length (*n*).
uplo	must be 'U' or 'L'. The default value is 'U'.

Application Notes

The 2-by-2 unit diagonal blocks and the unit diagonal elements of *U* and *L* are not stored. The remaining elements of *U* and *L* overwrite elements of the corresponding columns of the matrix *A*, but additional row interchanges are required to recover *U* or *L* explicitly (which is seldom necessary).

If *ipiv*(*i*) = *i* for all *i* = 1...*n*, then all off-diagonal elements of *U* (*L*) are stored explicitly in packed form.

If *uplo* = 'U', the computed factors *U* and *D* are the exact factors of a perturbed matrix *A* + *E*, where

$$|E| \leq c(n)\varepsilon\ P|U||D||U^T|P^T$$

$c(n)$ is a modest linear function of n, and ε is the machine precision. A similar estimate holds for the computed *L* and *D* when *uplo* = 'L'.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

?sptrs	to solve *A*X* = *B*
?spcon	to estimate the condition number of *A*
?sptri	to compute the inverse of *A*.

?hptrf

Computes the Bunch-Kaufman factorization of a complex Hermitian matrix using packed storage.

Syntax

Fortran 77:

```
call chptrf( uplo, n, ap, ipiv, info )
call zhptrf( uplo, n, ap, ipiv, info )
```

Fortran 95:

```
call hptrf( a [,uplo] [,ipiv] [,info] )
```

Description

This routine forms the Bunch-Kaufman factorization of a Hermitian matrix using packed storage:

if *uplo*='U', $A = P*U*D*U^H*P^T$

if *uplo*='L', $A = P*L*D*L^H*P^T$,

where A is the input matrix, P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D.

Input Parameters

uplo — CHARACTER*1. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of A is packed and how A is factored:
If *uplo* = 'U', the array *ap* stores the upper triangular part of the matrix A, and A is factored as $P*U*D*U^H*P^T$.
If *uplo* = 'L', the array *ap* stores the lower triangular part of the matrix A, and A is factored as $P*L*D*L^H*P^T$.

n — INTEGER. The order of matrix A; $n \geq 0$.

ap — COMPLEX for chptrf
DOUBLE COMPLEX for zhptrf.
Array, DIMENSION at least max(1, $n(n+1)/2$). The array *ap* contains the upper or the lower triangular part of the matrix A (as specified by *uplo*) in packed storage (see Matrix Storage Schemes).

Output Parameters

ap — The upper or lower triangle of A (as specified by *uplo*) is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).

ipiv — INTEGER.

	Array, DIMENSION at least max(1, *n*). Contains details of the interchanges and the block structure of *D*. If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column. If *uplo* = 'U' and *ipiv*(*i*) = *ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and the (*i*–1)-th row and column of *A* was interchanged with the *m*–th row and column. If *uplo* = 'L' and *ipiv*(*i*) = *ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and the (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = –*i*, the *i*-th parameter had an illegal value. If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular. Division by 0 will occur if you use *D* for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hptrf interface are as follows:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).
ipiv	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The 2-by-2 unit diagonal blocks and the unit diagonal elements of *U* and *L* are not stored. The remaining elements of *U* and *L* are stored in the corresponding columns of the array *a*, but additional row interchanges are required to recover *U* or *L* explicitly (which is seldom necessary).

If *ipiv*(*i*) = *i* for all *i* = 1...*n*, then all off-diagonal elements of *U* (*L*) are stored explicitly in the corresponding elements of the array *a*.

If *uplo* = 'U', the computed factors *U* and *D* are the exact factors of a perturbed matrix *A* + *E*, where

$|E| \leq c(n)\varepsilon P|U||D||U^T|P^T$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

A similar estimate holds for the computed L and D when *uplo* = 'L'.

The total number of floating-point operations is approximately $(4/3)n^3$.

After calling this routine, you can call the following routines:

?hptrs	to solve *A*X = B*
?hpcon	to estimate the condition number of *A*
?hptri	to compute the inverse of *A*.

Routines for Solving Systems of Linear Equations

This section describes the LAPACK routines for solving systems of linear equations. Before calling most of these routines, you need to factorize the matrix of your system of equations (see Routines for Matrix Factorization in this chapter). However, the factorization is not necessary if your system of equations has a triangular matrix.

?getrs

Solves a system of linear equations with an LU-factored square matrix, with multiple right-hand sides.

Syntax

Fortran 77:

```
call sgetrs( trans, n, nrhs, a, lda, ipiv, b, ldb, info )
call dgetrs( trans, n, nrhs, a, lda, ipiv, b, ldb, info )
call cgetrs( trans, n, nrhs, a, lda, ipiv, b, ldb, info )
call zgetrs( trans, n, nrhs, a, lda, ipiv, b, ldb, info )
```

Fortran 95:

```
call getrs( a, ipiv, b [, trans] [,info] )
```

Description

This routine solves for X the following systems of linear equations:

*A*X = B* if *trans*='N',

$A^T*X = B$	if *trans*='T',
$A^H*X = B$	if *trans*='C' (for complex matrices only).

Before calling this routine, you must call ?getrf to compute the *LU* factorization of *A*.

Input Parameters

trans	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If *trans* = 'N', then *A*X* = *B* is solved for *X*. If *trans* = 'T', then $A^T*X = B$ is solved for *X*. If *trans* = 'C', then $A^H*X = B$ is solved for *X*.
n	INTEGER. The order of *A*; the number of rows in *B*(*n* ≥ 0).
nrhs	INTEGER. The number of right-hand sides; *nrhs* ≥ 0.
a, *b*	REAL for sgetrs DOUBLE PRECISION for dgetrs COMPLEX for cgetrs DOUBLE COMPLEX for zgetrs. Arrays: *a*(*lda*,*), *b*(*ldb*,*). The array *a* contains *LU* factorization of matrix *A* resulting from the call of ?getrf . The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *a* must be at least max(1,*n*), the second dimension of *b* at least max(1,*nrhs*).
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).
ipiv	INTEGER. Array, DIMENSION at least max(1, *n*). The *ipiv* array, as returned by ?getrf.

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `getrs` interface are as follows:

a	Holds the matrix A of size (n, n).
b	Holds the matrix B of size (n, $nrhs$).
ipiv	Holds the vector of length (n).
trans	Must be `'N'`, `'C'`, or `'T'`. The default value is `'N'`.

Application Notes

For each right-hand side b, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$|E| \leq c(n)\varepsilon P|L||U|$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n)\, cond\,(A, x)\varepsilon$$

where $\text{cond}(A,x) = \| |A^{-1}||A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A,x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call `?gecon`.

To refine the solution and estimate the error, call `?gerfs`.

?gbtrs

Solves a system of linear equations with an LU-factored band matrix, with multiple right-hand sides.

Syntax

Fortran 77:

```
call sgbtrs( trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call dgbtrs( trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call cgbtrs( trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call zgbtrs( trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
```

Fortran 95:

```
call gbtrs( a, b, ipiv,  [, kl] [, trans] [, info] )
```

Description

This routine solves for X the following systems of linear equations:

$A*X = B$	if *trans*='N',
$A^T*X = B$	if *trans*='T',
$A^H*X = B$	if *trans*='C' (for complex matrices only).

Here A is an LU-factored general band matrix of order n with kl non-zero subdiagonals and ku nonzero superdiagonals. Before calling this routine, call ?gbtrf to compute the LU factorization of A.

Input Parameters

trans	CHARACTER*1. Must be 'N' or 'T' or 'C'.
n	INTEGER. The order of A; the number of rows in B; $n \geq 0$.
kl	INTEGER. The number of subdiagonals within the band of A; $kl \geq 0$.
ku	INTEGER. The number of superdiagonals within the band of A; $ku \geq 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
ab, *b*	REAL for sgbtrs

DOUBLE PRECISION for dgbtrs
COMPLEX for cgbtrs
DOUBLE COMPLEX for zgbtrs.
Arrays: *ab*(*ldab*,*), *b*(*ldb*,*).
The array *ab* contains the matrix *A* in band storage (see Matrix Storage Schemes).
The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.
The second dimension of *ab* must be at least max(1, *n*), and the second dimension of *b* at least max(1,*nrhs*).

ldab — INTEGER. The leading dimension of the array *ab*; *ldab* ≥ 2**kl* + *ku* +1.

ldb — INTEGER. The leading dimension of *b*; *ldb* ≥ max(1, *n*).

ipiv — INTEGER. Array, DIMENSION at least max(1, *n*). The *ipiv* array, as returned by ?gbtrf.

Output Parameters

b — Overwritten by the solution matrix *X*.

info — INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gbtrs interface are as follows:

a — Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (2**kl*+*ku*+1,*n*).

b — Holds the matrix *B* of size (*n*, *nrhs*).

ipiv — Holds the vector of length min(*m*, *n*).

kl — If omitted, assumed *kl* = *ku*.

ku — Restored as *lda*-2**kl*-1.

trans — Must be 'N', 'C', or 'T'. The default value is 'N'.

Application Notes

For each right-hand side b, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$|E| \leq c(kl + ku + 1)\varepsilon \ P|L||U|$

$c(k)$ is a modest linear function of k, and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(kl + ku + 1)\, cond\,(A, x)\varepsilon$$

where $cond(A, x) = \| |A^{-1}||A|\,|x|\,\|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $cond(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector is $2n(ku + 2kl)$ for real flavors. The number of operations for complex flavors is 4 times greater. All these estimates assume that kl and ku are much less than $\min(m,n)$.

To estimate the condition number $\kappa_\infty(A)$, call ?gbcon.

To refine the solution and estimate the error, call ?gbrfs.

?gttrs

Solves a system of linear equations with a tridiagonal matrix using the LU factorization computed by ?gttrf.

Syntax

Fortran 77:

```
call sgttrs( trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info )
call dgttrs( trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info )
call cgttrs( trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info )
call zgttrs( trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info )
```

Fortran 95:

```
call gttrs( dl, d, du, du2, b, ipiv [, trans] [,info] )
```

Description

This routine solves for X the following systems of linear equations with multiple right hand sides:

$A*X = B$	if *trans*='N',
$A^T*X = B$	if *trans*='T',
$A^H*X = B$	if *trans*='C' (for complex matrices only).

Before calling this routine, you must call ?gttrf to compute the LU factorization of A.

Input Parameters

trans	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If *trans* = 'N', then $A*X = B$ is solved for X. If *trans* = 'T', then $A^T*X = B$ is solved for X. If *trans* = 'C', then $A^H*X = B$ is solved for X.
n	INTEGER. The order of A; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides, that is, the number of columns in B; $nrhs \geq 0$.

dl,*d*,*du*,*du2*,*b*	REAL for sgttrs DOUBLE PRECISION for dgttrs COMPLEX for cgttrs DOUBLE COMPLEX for zgttrs. Arrays: *dl*(*n* -1), *d*(*n*), *du*(*n* -1), *du2*(*n* -2), *b*(*ldb*,*nrhs*). The array *dl* contains the (*n* - 1) multipliers that define the matrix *L* from the *LU* factorization of *A*. The array *d* contains the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*. The array *du* contains the (*n* - 1) elements of the first superdiagonal of *U*. The array *du2* contains the (*n* - 2) elements of the second superdiagonal of *U*. The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.
ldb	INTEGER. The leading dimension of *b*; *ldb* ≥ max(1, *n*).
ipiv	INTEGER. Array, DIMENSION (*n*). The *ipiv* array, as returned by ?gttrf.

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gttrs interface are as follows:

dl	Holds the vector of length (*n*-1).
d	Holds the vector of length (*n*).
du	Holds the vector of length (*n*-1).
du2	Holds the vector of length (*n*-2).
b	Holds the matrix *B* of size (*n*, *nrhs*).

ipiv	Holds the vector of length (n).
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.

Application Notes

For each right-hand side b, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$|E| \leq c(n)\varepsilon\ P|L||U|$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(kl + ku + 1)\, cond\,(A, x)\varepsilon$$

where $cond(A, x) = || |A^{-1}||A|\ |x|\ ||_\infty / ||x||_\infty \leq ||A^{-1}||_\infty\ ||A||_\infty = \kappa_\infty(A)$.

Note that $cond(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is $7n$ (including n divisions) for real flavors and $34n$ (including $2n$ divisions) for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call ?gtcon.

To refine the solution and estimate the error, call ?gtrfs.

?potrs

Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite matrix.

Syntax

Fortran 77:

```
call spotrs( uplo, n, nrhs, a, lda, b, ldb, info )
call dpotrs( uplo, n, nrhs, a, lda, b, ldb, info )
call cpotrs( uplo, n, nrhs, a, lda, b, ldb, info )
call zpotrs( uplo, n, nrhs, a, lda, b, ldb, info )
```

Fortran 95:

```
call potrs( a, b [,uplo] [, info] )
```

Description

This routine solves for X the system of linear equations $A*X = B$ with a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A, given the Cholesky factorization of A:

$A = U^T*U$ for real data, $A = U^H*U$ for complex data	If $uplo$ = 'U'
$A = L*L^T$ for real data, $A = L*L^H$ for complex data	If $uplo$ = 'L',

where L is a lower triangular matrix and U is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix B.

Before calling this routine, you must call ?potrf to compute the Cholesky factorization of A.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If $uplo$ = 'U', the upper triangle of A is stored.

If *uplo* = 'L', the lower triangle of *A* is stored.

n — INTEGER. The order of matrix *A*; *n* ≥ 0.

nrhs — INTEGER. The number of right-hand sides (*nrhs* ≥ 0).

a, *b* — REAL for spotrs
DOUBLE PRECISION for dpotrs
COMPLEX for cpotrs
DOUBLE COMPLEX for zpotrs.
Arrays: *a*(*lda*,*), *b*(*ldb*,*).
The array *a* contains the factor *U* or *L* (see *uplo*).
The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.
The second dimension of *a* must be at least max(1,*n*), the second dimension of *b* at least max(1,*nrhs*).

lda — INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).

ldb — INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

b — Overwritten by the solution matrix *X*.

info — INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine potrs interface are as follows:

a — Holds the matrix *A* of size (*n*, *n*).

b — Holds the matrix *B* of size (*n*, *nrhs*).

uplo — Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If `uplo = 'U'`, the computed solution for each right-hand side b is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$|E| \leq c(n)\varepsilon \ |U^H||U|$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

A similar estimate holds for `uplo = 'L'`. If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n)\, cond\,(A, x)\varepsilon$$

where $\text{cond}(A, x) = \| |A^{-1}||A|\,|x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$. The approximate number of floating-point operations for one right-hand side vector b is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call `?pocon`.

To refine the solution and estimate the error, call `?porfs`.

?pptrs

Solves a system of linear equations with a packed Cholesky-factored symmetric (Hermitian) positive-definite matrix.

Syntax

Fortran 77:

```
call spptrs( uplo, n, nrhs, ap, b, ldb, info )
call dpptrs( uplo, n, nrhs, ap, b, ldb, info )
call cpptrs( uplo, n, nrhs, ap, b, ldb, info )
call zpptrs( uplo, n, nrhs, ap, b, ldb, info )
```

Fortran 95:

```
call pptrs( a, b [,uplo] [,info] )
```

Description

This routine solves for X the system of linear equations $A*X = B$ with a packed symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A, given the Cholesky factorization of A:

$A = U^T*U$ for real data, $A = U^H*U$ for complex data	If $uplo$ = 'U'
$A = L*L^T$ for real data, $A = L*L^H$ for complex data	If $uplo$ = 'L',

where L is a lower triangular matrix and U is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix B.

Before calling this routine, you must call ?pptrf to compute the Cholesky factorization of A.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If $uplo$ = 'U', the upper triangle of A is stored. If $uplo$ = 'L', the lower triangle of A is stored.
n	INTEGER. The order of matrix A; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides ($nrhs \geq 0$).
ap, *b*	REAL for spptrs DOUBLE PRECISION for dpptrs COMPLEX for cpptrs DOUBLE COMPLEX for zpptrs. Arrays: $ap(*)$, $b(ldb,*)$ The dimension of ap must be at least max(1,n(n+1)/2). The array ap contains the factor U or L, as specified by $uplo$, in packed storage (see Matrix Storage Schemes).

	The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1, *nrhs*).
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine pptrs interface are as follows:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).
b	Holds the matrix *B* of size (*n*, *nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If *uplo* = 'U', the computed solution for each right-hand side *b* is the exact solution of a perturbed system of equations (*A* + *E*)*x* = *b*, where

$|E| \leq c(n)\varepsilon \ |U^H||U|$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

A similar estimate holds for *uplo* = 'L'.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n)\, cond\,(A, x)\varepsilon$$

where $\mathrm{cond}(A, x) = \| \, |A^{-1}||A| \, |x| \, \|_\infty / \|x\|_\infty \le \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\mathrm{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call ?ppcon.

To refine the solution and estimate the error, call ?pprfs.

?pbtrs

Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite band matrix.

Syntax

Fortran 77:

```
call spbtrs( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call dpbtrs( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call cpbtrs( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call zpbtrs( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
```

Fortran 95:

```
call pbtrs( a, b [,uplo] [,info] )
```

Description

This routine solves for real data a system of linear equations $A*X = B$ with a symmetric positive-definite or, for complex data, Hermitian positive-definite band matrix A, given the Cholesky factorization of A:

$A = U^T*U$ for real data, $A = U^H*U$ for complex data	If uplo='U'

$A = L*L^T$ for real data, $A = L*L^H$ for complex data	If *uplo*='L',

where L is a lower triangular matrix and U is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix B.

Before calling this routine, you must call ?pbtrf to compute the Cholesky factorization of A in the band storage form.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If *uplo* = 'U', the upper triangular factor is stored in *ab*. If *uplo* = 'L', the lower triangular factor is stored in *ab*.
n	INTEGER. The order of matrix A; $n \geq 0$.
kd	INTEGER. The number of superdiagonals or subdiagonals in the matrix A; $kd \geq 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
ab, *b*	REAL for spbtrs DOUBLE PRECISION for dpbtrs COMPLEX for cpbtrs DOUBLE COMPLEX for zpbtrs. Arrays: *ab*(*ldab*,*), *b*(*ldb*,*). The array *ab* contains the Cholesky factor, as returned by the factorization routine, in band storage form. The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of *ab* must be at least max(1, *n*), and the second dimension of *b* at least max(1,*nrhs*).
ldab	INTEGER. The first dimension of the array *ab*; *ldab* ≥ *kd* +1.
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

b	Overwritten by the solution matrix X.

info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine pbtrs interface are as follows:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*kd*+1, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$|E| \le c(kd + 1)\varepsilon\ P|U^H||U|$ or $|E| \le c(kd + 1)\varepsilon\ P|L^H||L|$

$c(k)$ is a modest linear function of k, and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \le c(kd + 1)\, cond\,(A, x)\varepsilon$$

where $cond(A, x) = \| |A^{-1}||A|\,|x|\,\|_\infty / \|x\|_\infty \le \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $cond(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector is $4n*kd$ for real flavors and $16n*kd$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call ?pbcon.

To refine the solution and estimate the error, call ?pbrfs.

?pttrs

Solves a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal matrix using the factorization computed by ?pttrf.

Syntax

Fortran 77:

```
call spttrs( n, nrhs, d, e, b, ldb, info )
call dpttrs( n, nrhs, d, e, b, ldb, info )
call cpttrs( uplo, n, nrhs, d, e, b, ldb, info )
call zpttrs( uplo, n, nrhs, d, e, b, ldb, info )
```

Fortran 95:

```
call pttrs( d, e, b [,info] )
call pttrs( d, e, b [,uplo]  [,info] )
```

Description

This routine solves for *X* a system of linear equations *A*X = B* with a symmetric (Hermitian) positive-definite tridiagonal matrix *A*. Before calling this routine, call ?pttrf to compute the *L*D*L'* for real data and the *L*D*L'* or *U'*D*U* factorization of *A* for complex data.

Input Parameters

uplo	CHARACTER*1. Used for cpttrs/zpttrs only. Must be 'U' or 'L'. Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix *A* is stored and how *A* is factored: If *uplo* = 'U', the array *e* stores the superdiagonal of *A*, and *A* is factored as *U'*D*U*. If *uplo* = 'L', the array *e* stores the subdiagonal of *A*, and *A* is factored as *L*D*L'*.
n	INTEGER. The order of *A*; *n* ≥ 0.
nrhs	INTEGER. The number of right-hand sides, that is, the number of columns of the matrix *B*; *nrhs* ≥ 0.
d	REAL for spttrs, cpttrs

DOUBLE PRECISION for dpttrs, zpttrs.
Array, dimension (*n*). Contains the diagonal elements of the diagonal matrix *D* from the factorization computed by ?pttrf.

e, *b*
REAL for spttrs
DOUBLE PRECISION for dpttrs
COMPLEX for cpttrs
DOUBLE COMPLEX for zpttrs.
Arrays: *e*(*n* -1), *b*(*ldb*, *nrhs*).
The array *e* contains the (*n* - 1) off-diagonal elements of the unit bidiagonal factor *U* or *L* from the factorization computed by ?pttrf (see *uplo*).
The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

ldb
INTEGER. The leading dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

b
Overwritten by the solution matrix *X*.

info
INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine pttrs interface are as follows:

d
Holds the vector of length (*n*).

e
Holds the vector of length (*n*-1).

b
Holds the matrix *B* of size (*n*, *nrhs*).

uplo
Used in complex flavors only. Must be 'U' or 'L'. The default value is 'U'.

?sytrs

Solves a system of linear equations with a UDU- or LDL-factored symmetric matrix.

Syntax

Fortran 77:

```
call ssytrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call dsytrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call csytrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call zsytrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
```

Fortran 95:

```
call sytrs( a, b, ipiv [,uplo] [,info] )
```

Description

This routine solves for X the system of linear equations $A*X = B$ with a symmetric matrix A, given the Bunch-Kaufman factorization of A:

if $uplo$='U', $A = P*U*D*U^T*P^T$

if $uplo$='L', $A = P*L*D*L^T*P^T$,

where P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B. You must supply to this routine the factor U (or L) and the array $ipiv$ returned by the factorization routine ?sytrf.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If $uplo$ = 'U', the array a stores the upper triangular factor U of the factorization $A = P*U*D*U^T*P^T$. If $uplo$ = 'L', the array a stores the lower triangular factor L of the factorization $A = P*L*D*L^T*P^T$.
n	INTEGER. The order of matrix A; $n \geq 0$.

nrhs	INTEGER. The number of right-hand sides; *nrhs* ≥ 0.
ipiv	INTEGER. Array, DIMENSION at least max(1, *n*). The *ipiv* array, as returned by ?sytrf.
a, *b*	REAL for ssytrs DOUBLE PRECISION for dsytrs COMPLEX for csytrs DOUBLE COMPLEX for zsytrs. Arrays: *a*(*lda*,*), *b*(*ldb*,*). The array *a* contains the factor *U* or *L* (see *uplo*). The array *b* contains the matrix *B* whose columns are the right-hand sides for the system of equations. The second dimension of *a* must be at least max(1,*n*), and the second dimension of *b* at least max(1,*nrhs*).
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info*=0, the execution is successful. If *info* = *-i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine sytrs interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
ipiv	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For each right-hand side b, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$|E| \le c(n)\varepsilon\ P|U||D||U^T|P^T$ or $|E| \le c(n)\varepsilon\ P|L||D||U^T|P^T$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \le c(n)\, cond\,(A, x)\varepsilon$$

where $cond(A, x) = \| |A^{-1}||A|\,|x| \|_\infty / \|x\|_\infty \le \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $cond(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $2n^2$ for real flavors or $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call ?sycon.

To refine the solution and estimate the error, call ?syrfs.

?hetrs

Solves a system of linear equations with a UDU- or LDL-factored Hermitian matrix.

Syntax

Fortran 77:

```
call chetrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call zhetrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
```

Fortran 95:

```
call hetrs( a, b, ipiv [, uplo] [,info] )
```

Description

This routine solves for *X* the system of linear equations *A*X = B* with a Hermitian matrix *A*, given the Bunch-Kaufman factorization of *A*:

if *uplo* = 'U' $A = P*U*D*U^H*P^T$

if *uplo* = 'L' $A = P*L*D*L^H*P^T$,

where *P* is a permutation matrix, *U* and *L* are upper and lower triangular matrices with unit diagonal, and *D* is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix *B*. You must supply to this routine the factor *U* (or *L*) and the array *ipiv* returned by the factorization routine ?hetrf.

Input Parameters

uplo
CHARACTER*1. Must be 'U' or 'L'.
Indicates how the input matrix *A* has been factored:
If *uplo* = 'U', the array *a* stores the upper triangular factor *U* of the factorization *A* = $P*U*D*U^H*P^T$.
If *uplo* = 'L', the array *a* stores the lower triangular factor *L* of the factorization *A* = $P*L*D*L^H*P^T$.

n
INTEGER. The order of matrix *A*; $n \geq 0$.

nrhs
INTEGER. The number of right-hand sides; $nrhs \geq 0$.

ipiv
INTEGER.
Array, DIMENSION at least max(1, *n*).
The *ipiv* array, as returned by ?hetrf.

a, *b*
COMPLEX for chetrs
DOUBLE COMPLEX for zhetrs.
Arrays: *a*(*lda*,*), *b*(*ldb*,*).
The array *a* contains the factor *U* or *L* (see *uplo*).
The array *b* contains the matrix *B* whose columns are the right-hand sides for the system of equations.
The second dimension of *a* must be at least max(1,*n*), the second dimension of *b* at least max(1,*nrhs*).

lda
INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).

ldb
INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `hetrs` interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
ipiv	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For each right-hand side b, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$|E| \le c(n)\varepsilon\ P|U||D||U^H|P^T$ or $|E| \le c(n)\varepsilon\ P|L||D||L^H|P^T$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \le c(n)\, cond\,(A, x)\varepsilon$$

where $cond(A, x) = \| |A^{-1}||A|\,|x|\,\|_\infty / \|x\|_\infty \le \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $cond(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $8n^2$.

To estimate the condition number $\kappa_\infty(A)$, call ?hecon.

To refine the solution and estimate the error, call ?herfs.

?sptrs

Solves a system of linear equations with a UDU- or LDL-factored symmetric matrix using packed storage.

Syntax

Fortran 77:

```
call ssptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call dsptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call csptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zsptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
```

Fortran 95:

```
call sptrs( a, b, ipiv [, uplo] [,info] )
```

Description

This routine solves for X the system of linear equations $A*X = B$ with a symmetric matrix A, given the Bunch-Kaufman factorization of A:

if uplo='U', $A = PUDU^TP^T$

if uplo='L', $A = PLDL^TP^T$,

where P is a permutation matrix, U and L are upper and lower packed triangular matrices with unit diagonal, and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B. You must supply the factor U (or L) and the array ipiv returned by the factorization routine ?sptrf.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored:

If *uplo* = 'U', the array *ap* stores the packed factor *U* of the factorization *A* = *P*U*D*U^T*P^T*. If *uplo* = 'L', the array *ap* stores the packed factor *L* of the factorization *A* = *P*L*D*L^T*P^T*.

n — INTEGER. The order of matrix *A*; *n* ≥ 0.

nrhs — INTEGER. The number of right-hand sides; *nrhs* ≥ 0.

ipiv — INTEGER.
Array, DIMENSION at least max(1, *n*). The *ipiv* array, as returned by ?sptrf.

ap, *b* — REAL for ssptrs
DOUBLE PRECISION for dsptrs
COMPLEX for csptrs
DOUBLE COMPLEX for zsptrs.
Arrays: *ap*(*), *b*(*ldb*,*).
The dimension of *ap* must be at least max(1,*n*(*n*+1)/2). The array *ap* contains the factor *U* or *L*, as specified by *uplo*, in packed storage (see Matrix Storage Schemes).
The array *b* contains the matrix *B* whose columns are the right-hand sides for the system of equations. The second dimension of *b* must be at least max(1, *nrhs*).

ldb — INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

b — Overwritten by the solution matrix *X*.

info — INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine sptrs interface are as follows:

a — Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).

b — Holds the matrix *B* of size (*n*, *nrhs*).

ipiv	Holds the vector of length (n).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

For each right-hand side b, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$|E| \le c(n)\varepsilon\ P|U||D||U^T|P^T$ or $|E| \le c(n)\varepsilon\ P|L||D||L^T|P^T$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \le c(n)\ cond\ (A, x)\varepsilon$$

where $\mathrm{cond}(A, x) = \| |A^{-1}||A|\ |x|\ \|_\infty / \|x\|_\infty \le \|A^{-1}\|_\infty\ \|A\|_\infty = \kappa_\infty(A)$.

Note that $\mathrm{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $2n^2$ for real flavors or $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call `?spcon`.

To refine the solution and estimate the error, call `?sprfs`.

?hptrs

Solves a system of linear equations with a UDU- or LDL-factored Hermitian matrix using packed storage.

Syntax

Fortran 77:

```
call chptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zhptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
```

Fortran 95:

```
call hptrs( a, b, ipiv [,uplo] [,info] )
```

Description

This routine solves for X the system of linear equations $A*X = B$ with a Hermitian matrix A, given the Bunch-Kaufman factorization of A:

if $uplo$='U', $A = P*U*D*U^H*P^T$

if $uplo$='L', $A = P*L*D*L^H*P^T$,

where P is a permutation matrix, U and L are upper and lower packed triangular matrices with unit diagonal, and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B.

You must supply to this routine the arrays ap (containing U or L)and $ipiv$ in the form returned by the factorization routine ?hptrf.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If $uplo$ = 'U', the array ap stores the packed factor U of the factorization $A = P*U*D*U^H*P^T$. If $uplo$ = 'L', the array ap stores the packed factor L of the factorization $A = P*L*D*L^H*P^T$.
n	INTEGER. The order of matrix A; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \geq 0$.

ipiv	INTEGER. Array, DIMENSION at least max(1, *n*). The *ipiv* array, as returned by ?hptrf.
ap, *b*	COMPLEX for chptrs DOUBLE COMPLEX for zhptrs. Arrays: *ap*(*), *b*(*ldb*,*). The dimension of *ap* must be at least max(1,*n*(*n*+1)/2). The array *ap* contains the factor *U* or *L*, as specified by *uplo*, in packed storage (see Matrix Storage Schemes). The array *b* contains the matrix *B* whose columns are the right-hand sides for the system of equations. The second dimension of *b* must be at least max(1, *nrhs*).
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hptrs interface are as follows:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).
b	Holds the matrix *B* of size (*n*, *nrhs*).
ipiv	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$|E| \le c(n)\varepsilon\ P|U||D||U^H|P^T$ or $|E| \le c(n)\varepsilon\ P|L||D||L^H|P^T$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n)\, cond\,(A, x)\varepsilon$$

where $cond(A,x) = \| |A^{-1}||A|\,|x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $cond(A,x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call ?hpcon.

To refine the solution and estimate the error, call ?hprfs.

?trtrs

Solves a system of linear equations with a triangular matrix, with multiple right-hand sides.

Syntax

Fortran 77:

```
call strtrs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, info )
call dtrtrs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, info )
call ctrtrs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, info )
call ztrtrs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, info )
```

Fortran 95:

```
call trtrs( a, b [,uplo] [, trans] [,diag] [,info] )
```

Description

This routine solves for X the following systems of linear equations with a triangular matrix A, with multiple right-hand sides stored in B:

$A*X = B$	if *trans*='N',
$A^T*X = B$	if *trans*='T',
$A^H*X = B$	if *trans*='C' (for complex matrices only).

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If *uplo* = 'U', then A is upper triangular. If *uplo* = 'L', then A is lower triangular.
trans	CHARACTER*1. Must be 'N' or 'T' or 'C'. If *trans* = 'N', then $A*X = B$ is solved for X. If *trans* = 'T', then $A^T*X = B$ is solved for X. If *trans* = 'C', then $A^H*X = B$ is solved for X.
diag	CHARACTER*1. Must be 'N' or 'U'. If *diag* = 'N', then A is not a unit triangular matrix. If *diag* = 'U', then A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array *a*.
n	INTEGER. The order of A; the number of rows in B; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
a, *b*	REAL for strtrs DOUBLE PRECISION for dtrtrs COMPLEX for ctrtrs DOUBLE COMPLEX for ztrtrs. Arrays: *a*(*lda*,*), *b*(*ldb*,*). The array *a* contains the matrix A. The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of *a* must be at least max(1,*n*), the second dimension of *b* at least max(1,*nrhs*).
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).

ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `trtrs` interface are as follows:

a	Stands for argument *ap* in Fortan 77 interface. Holds the matrix *A* of size (*n**(*n*+1)/2).
b	Holds the matrix *B* of size (*n*, *nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
diag	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations (*A* + *E*)*x* = *b*, where

$|E| \le c(n)\varepsilon\ |A|$

$c(n)$ is a modest linear function of n, and ε is the machine precision. If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \le c(n)\ cond\ (A, x)\varepsilon \text{ provided } c(n)\ cond\ (A, x)\varepsilon < 1$$

where cond(*A*, *x*) = $|| |A^{-1}||A|\ |x|\ ||_\infty / ||x||_\infty \le ||A^{-1}||_\infty\ ||A||_\infty = \kappa_\infty(A)$.

Note that cond(A, x) can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is n^2 for real flavors and $4n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call ?trcon.

To estimate the error in the solution, call ?trrfs.

?tptrs

Solves a system of linear equations with a packed triangular matrix, with multiple right-hand sides.

Syntax

Fortran 77:

```
call stptrs( uplo, trans, diag, n, nrhs, ap, b, ldb, info )
call dtptrs( uplo, trans, diag, n, nrhs, ap, b, ldb, info )
call ctptrs( uplo, trans, diag, n, nrhs, ap, b, ldb, info )
call ztptrs( uplo, trans, diag, n, nrhs, ap, b, ldb, info )
```

Fortran 95:

```
call tptrs( a, b [,uplo] [, trans] [,diag] [,info] )
```

Description

This routine solves for X the following systems of linear equations with a packed triangular matrix A, with multiple right-hand sides stored in B:

$A*X = B$	if *trans*='N',
$A^T*X = B$	if *trans*='T',
$A^H*X = B$	if *trans*='C' (for complex matrices only).

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular:

	If *uplo* = 'U', then *A* is upper triangular. If *uplo* = 'L', then *A* is lower triangular.
trans	CHARACTER*1. Must be 'N' or 'T' or 'C'. If *trans* = 'N', then *A*X* = *B* is solved for *X*. If *trans* = 'T', then $A^T*X = B$ is solved for *X*. If *trans* = 'C', then $A^H*X = B$ is solved for *X*.
diag	CHARACTER*1. Must be 'N' or 'U'. If *diag* = 'N', then *A* is not a unit triangular matrix. If *diag* = 'U', then *A* is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array *ap*.
n	INTEGER. The order of *A*; the number of rows in *B*; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
ap, *b*	REAL for stptrs DOUBLE PRECISION for dtptrs COMPLEX for ctptrs DOUBLE COMPLEX for ztptrs. Arrays: *ap*(*), *b*(*ldb*,*). The dimension of *ap* must be at least max(1,*n*(*n*+1)/2). The array *ap* contains the matrix *A* in packed storage (see Matrix Storage Schemes). The array *b* contains the matrix *B* whose columns are the right-hand sides for the system of equations. The second dimension of *b* must be at least max(1, *nrhs*).
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine tptrs interface are as follows:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size *(n*(n+1)/2)*.
b	Holds the matrix *B* of size (*n*, *nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
diag	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations *(A + E)x = b*, where

$|E| \leq c(n)\varepsilon\ |A|$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n)\, cond\,(A, x)\varepsilon \text{ provided } c(n)\, cond\,(A, x)\varepsilon < 1$$

where $cond(A, x) = \| |A^{-1}||A|\,|x|\,\|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $cond(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is n^2 for real flavors and $4n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call ?tpcon.

To estimate the error in the solution, call ?tprfs.

?tbtrs

Solves a system of linear equations with a band triangular matrix, with multiple right-hand sides.

Syntax

Fortran 77:

```
call stbtrs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info )
call dtbtrs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info )
call ctbtrs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info )
call ztbtrs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info )
```

Fortran 95:

```
call tbtrs( a, b [,uplo] [, trans] [,diag] [,info] )
```

Description

This routine solves for X the following systems of linear equations with a band triangular matrix A, with multiple right-hand sides stored in B:

$A*X = B$	if *trans*='N',
$A^T*X = B$	if *trans*='T',
$A^H*X = B$	if *trans*='C' (for complex matrices only).

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If *uplo* = 'U', then A is upper triangular. If *uplo* = 'L', then A is lower triangular.
trans	CHARACTER*1. Must be 'N' or 'T' or 'C'. If *trans* = 'N', then $A*X = B$ is solved for X. If *trans* = 'T', then $A^T*X = B$ is solved for X. If *trans* = 'C', then $A^H*X = B$ is solved for X.
diag	CHARACTER*1. Must be 'N' or 'U'. If *diag* = 'N', then A is not a unit triangular matrix.

	If *diag* = 'U', then *A* is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array *ab*.
n	INTEGER. The order of *A*; the number of rows in *B*; *n* ≥ 0.
kd	INTEGER. The number of superdiagonals or subdiagonals in the matrix *A*; *kd* ≥ 0.
nrhs	INTEGER. The number of right-hand sides; *nrhs* ≥ 0.
ab, *b*	REAL for stbtrs DOUBLE PRECISION for dtbtrs COMPLEX for ctbtrs DOUBLE COMPLEX for ztbtrs. Arrays: *ab*(*ldab*,*), *b*(*ldb*,*). The array *ab* contains the matrix *A* in band storage form. The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *ab* must be at least max(1, *n*), the second dimension of *b* at least max(1,*nrhs*).
ldab	INTEGER. The first dimension of *ab*; *ldab* ≥ *kd* + 1.
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine tbtrs interface are as follows:

a	Stands for the argument *ab* in Fortan 77 interface. Holds the array *A* of size (*kd*+1,*n*)
b	Holds the matrix *B* of size (*n*, *nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
diag	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

For each right-hand side b, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$|E| \leq c(n)\varepsilon|A|$

$c(n)$ is a modest linear function of n, and ε is the machine precision. If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n)\, cond\,(A, x)\varepsilon \text{ provided } c(n)\, cond\,(A, x)\varepsilon < 1$$

where $cond(A, x) = \| |A^{-1}||A|\,|x|\,\|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $cond(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is $2n*kd$ for real flavors and $8n*kd$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call ?tbcon.

To estimate the error in the solution, call ?tbrfs.

Routines for Estimating the Condition Number

This section describes the LAPACK routines for estimating the condition number of a matrix. The condition number is used for analyzing the errors in the solution of a system of linear equations (see Error Analysis). Since the condition number may be arbitrarily large when the matrix is nearly singular, the routines actually compute the reciprocal condition number.

?gecon

Estimates the reciprocal of the condition number of a general matrix in the 1-norm or the infinity-norm.

Syntax

Fortran 77:

```
call sgecon( norm, n, a, lda, anorm, rcond, work, iwork, info )
call dgecon( norm, n, a, lda, anorm, rcond, work, iwork, info )
call cgecon( norm, n, a, lda, anorm, rcond, work, rwork, info )
call zgecon( norm, n, a, lda, anorm, rcond, work, rwork, info )
```

Fortran 95:

```
call gecon( a, anorm, rcond [,norm] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a general matrix A in the 1-norm or infinity-norm:

$$\kappa_1(A) = ||A||_1 ||A^{-1}||_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$

$$\kappa_\infty(A) = ||A||_\infty ||A^{-1}||_\infty = \kappa_1(A^T) = \kappa_1(A^H) .$$

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \sum_i |a_{ij}|$ or $||A||_\infty = \max_i \sum_j |a_{ij}|$)
- call ?getrf to compute the LU factorization of A.

Input Parameters

norm — CHARACTER*1. Must be '1' or 'O' or 'I'.
If *norm* = '1' or 'O', then the routine estimates the condition number of matrix A in 1-norm.
If *norm* = 'I', then the routine estimates the condition number of matrix A in infinity-norm.

n	INTEGER. The order of the matrix *A*; $n \geq 0$.
a, *work*	REAL for sgecon DOUBLE PRECISION for dgecon COMPLEX for cgecon DOUBLE COMPLEX for zgecon. Arrays: *a*(*lda*,*), *work*(*). The array *a* contains the *LU*-factored matrix *A*, as returned by ?getrf. The second dimension of *a* must be at least max(1,*n*). The array *work* is a workspace for the routine. The dimension of *work* must be at least max(1, 4*n) for real flavors and max(1, 2*n) for complex flavors.
anorm	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The norm of the original matrix *A* (see Description).
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).
iwork	INTEGER. Workspace array, DIMENSION at least max(1, *n*).
rwork	REAL for cgecon DOUBLE PRECISION for zgecon. Workspace array, DIMENSION at least max(1, 2*n).

Output Parameters

rcond	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets *rcond* = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
info	INTEGER. If *info*=0, the execution is successful. If *info* = –*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gecon interface are as follows:

a	Holds the matrix A of size (n, n).
norm	Must be '1', 'O', or 'I'. The default value is '1'.

Application Notes

The computed *rcond* is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$ or $A^H*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2*n^2$ floating-point operations for real flavors and $8*n^2$ for complex flavors.

?gbcon

Estimates the reciprocal of the condition number of a band matrix in the 1-norm or the infinity-norm.

Syntax

Fortran 77:

```
call sgbcon( norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, iwork, info
)
call dgbcon( norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, iwork, info
)
call cgbcon( norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, rwork, info
)
call zgbcon( norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, rwork, info
)
```

Fortran 95:

```
call gbcon( a, ipiv, anorm, rcond [,kl] [,norm] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a general band matrix A in the 1-norm or infinity-norm:

$$\kappa_1(A) = ||A||_1||A^{-1}||_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$

$$\kappa_\infty(A) = ||A||_\infty||A^{-1}||_\infty = \kappa_1(A^T) = \kappa_1(A^H) \ .$$

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?gbtrf to compute the *LU* factorization of *A*.

Input Parameters

norm	CHARACTER*1. Must be '1' or 'O' or 'I'. If *norm* = '1' or 'O', then the routine estimates the condition number of matrix *A* in 1-norm. If *norm* = 'I', then the routine estimates the condition number of matrix *A* in infinity-norm.
n	INTEGER. The order of the matrix *A*; $n \geq 0$.
kl	INTEGER. The number of subdiagonals within the band of *A*; $kl \geq 0$.
ku	INTEGER. The number of superdiagonals within the band of *A*; $ku \geq 0$.
ldab	INTEGER. The first dimension of the array *ab*. ($ldab \geq 2*kl + ku + 1$).
ipiv	INTEGER. Array, DIMENSION at least max(1, *n*). The *ipiv* array, as returned by ?gbtrf.
ab, *work*	REAL for sgbcon DOUBLE PRECISION for dgbcon COMPLEX for cgbcon DOUBLE COMPLEX for zgbcon. Arrays: *ab*(*ldab*,*), *work*(*). The array *ab* contains the factored band matrix *A*, as returned by ?gbtrf. The second dimension of *ab* must be at least max(1,*n*). The array *work* is a workspace for the routine. The dimension of *work* must be at least max(1, 3*n) for real flavors and max(1, 2*n) for complex flavors.
anorm	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The norm of the original matrix *A* (see Description).
iwork	INTEGER. Workspace array, DIMENSION at least max(1, *n*).
rwork	REAL for cgbcon DOUBLE PRECISION for zgbcon. Workspace array, DIMENSION at least max(1, 2*n).

Output Parameters

rcond	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gbcon interface are as follows:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (2**kl*+*ku*+1,*n*).
ipiv	Holds the vector of length (*n*).
norm	Must be '1', 'O', or 'I'. The default value is '1'.
kl	If omitted, assumed *kl* = *ku*.
ku	Restored as *ku* = *lda*-2**kl*-1.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations *A***x* = *b* or A^H**x* = *b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately 2*n*(*ku* + 2*kl*) floating-point operations for real flavors and 8*n*(*ku* + 2*kl*) for complex flavors.

?gtcon

Estimates the reciprocal of the condition number of a tridiagonal matrix using the factorization computed by ?gttrf.

Syntax

Fortran 77:

```
call sgtcon( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, iwork, info )
call dgtcon( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, iwork, info )
call cgtcon( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, info )
call zgtcon( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, info )
```

Fortran 95:

```
call gtcon( dl, d, du, du2, ipiv, anorm, rcond [,norm] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a real or complex tridiagonal matrix A in the 1-norm or infinity-norm:

$\kappa_1(A) = ||A||_1||A^{-1}||_1$

$\kappa_\infty(A) = ||A||_\infty||A^{-1}||_\infty$

An estimate is obtained for $||A^{-1}||$, and the reciprocal of the condition number is computed as $rcond = 1 / (||A|| \; ||A^{-1}||)$.

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?gttrf to compute the LU factorization of A.

Input Parameters

norm CHARACTER*1. Must be '1' or 'O' or 'I'.

If *norm* = '1' or 'O', then the routine estimates the condition number of matrix *A* in 1-norm.
If *norm* = 'I', then the routine estimates the condition number of matrix *A* in infinity-norm.

n — INTEGER. The order of the matrix *A*; $n \geq 0$.

dl,*d*,*du*,*du2* — REAL for sgtcon
DOUBLE PRECISION for dgtcon
COMPLEX for cgtcon
DOUBLE COMPLEX for zgtcon.
Arrays: *dl*(*n* -1), *d*(*n*), *du*(*n* -1), *du2*(*n* -2).
The array *dl* contains the (*n* – 1) multipliers that define the matrix *L* from the *LU* factorization of *A* as computed by ?gttrf.
The array *d* contains the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*.
The array *du* contains the (*n* - 1) elements of the first superdiagonal of *U*.
The array *du2* contains the (*n* - 2) elements of the second superdiagonal of *U*.

ipiv — INTEGER.
Array, DIMENSION (*n*). The array of pivot indices, as returned by ?gttrf.

anorm — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
The norm of the original matrix *A* (see Description).

work — REAL for sgtcon
DOUBLE PRECISION for dgtcon
COMPLEX for cgtcon
DOUBLE COMPLEX for zgtcon.
Workspace array, DIMENSION (2**n*).

iwork — INTEGER. Workspace array, DIMENSION (*n*). Used for real flavors only.

Output Parameters

rcond — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.

	An estimate of the reciprocal of the condition number. The routine sets *rcond*=0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gtcon interface are as follows:

dl	Holds the vector of length (*n*-1).
d	Holds the vector of length (*n*).
du	Holds the vector of length (*n*-1).
du2	Holds the vector of length (*n*-2).
ipiv	Holds the vector of length (*n*).
norm	Must be '1', 'O', or 'I'. The default value is '1'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations *A***x* = *b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?pocon

Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite matrix.

Syntax

Fortran 77:

```
call spocon( uplo, n, a, lda, anorm, rcond, work, iwork, info )
call dpocon( uplo, n, a, lda, anorm, rcond, work, iwork, info )
call cpocon( uplo, n, a, lda, anorm, rcond, work, rwork, info )
call zpocon( uplo, n, a, lda, anorm, rcond, work, rwork, info )
```

Fortran 95:

```
call pocon( a, anorm, rcond [,uplo] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite matrix A:

$\kappa_1(A) = ||A||_1 \ ||A^{-1}||_1$ (since A is symmetric or Hermitian, $\kappa_\infty(A) = \kappa_1(A)$).

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?potrf to compute the Cholesky factorization of A.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If *uplo* = 'U', the upper triangle of A is stored. If *uplo* = 'L', the lower triangle of A is stored.
n	INTEGER. The order of the matrix A; $n \geq 0$.
a, *work*	REAL for spocon DOUBLE PRECISION for dpocon COMPLEX for cpocon

DOUBLE COMPLEX for zpocon.
Arrays: *a*(*lda*,*), *work*(*).
The array *a* contains the factored matrix *A*, as returned by ?potrf. The second dimension of *a* must be at least max(1,*n*).
The array *work* is a workspace for the routine. The dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.

lda — INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).

anorm — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
The norm of the original matrix *A* (see Description).

iwork — INTEGER. Workspace array, DIMENSION at least max(1, *n*).

rwork — REAL for cpocon
DOUBLE PRECISION for zpocon.
Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

rcond — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info — INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine pocon interface are as follows:

a — Holds the matrix *A* of size (*n*, *n*).

uplo — Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations *A*x = b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?ppcon

Estimates the reciprocal of the condition number of a packed symmetric (Hermitian) positive-definite matrix.

Syntax

Fortran 77:

```
call sppcon( uplo, n, ap, anorm, rcond, work, iwork, info )
call dppcon( uplo, n, ap, anorm, rcond, work, iwork, info )
call cppcon( uplo, n, ap, anorm, rcond, work, rwork, info )
call zppcon( uplo, n, ap, anorm, rcond, work, rwork, info )
```

Fortran 95:

```
call ppcon( a, anorm, rcond [,uplo] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a packed symmetric (Hermitian) positive-definite matrix *A*:

$\kappa_1(A) = ||A||_1 \ ||A^{-1}||_1$ (since *A* is symmetric or Hermitian, $\kappa_\infty(A) = \kappa_1(A)$).

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?pptrf to compute the Cholesky factorization of *A*.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix *A* has been factored: If *uplo* = 'U', the upper triangle of *A* is stored. If *uplo* = 'L', the lower triangle of *A* is stored.
n	INTEGER. The order of the matrix *A*; $n \geq 0$.
ap, *work*	REAL for sppcon DOUBLE PRECISION for dppcon COMPLEX for cppcon DOUBLE COMPLEX for zppcon. Arrays: *ap*(*), *work*(*). The array *ap* contains the packed factored matrix *A*, as returned by ?pptrf. The dimension of *ap* must be at least max(1,*n*(*n*+1)/2). The array *work* is a workspace for the routine. The dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.
anorm	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. The norm of the original matrix *A* (see Description).
iwork	INTEGER. Workspace array, DIMENSION at least max(1, *n*).
rwork	REAL for cppcon DOUBLE PRECISION for zppcon. Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

rcond	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `ppcon` interface are as follows:

a	Stands for the argument *ap* in Fortan 77 interface. Holds the array *A* of size `(n*(n+1)/2)`.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations *A*x = b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?pbcon

Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite band matrix.

Syntax

Fortran 77:

```
call spbcon( uplo, n, kd, ab, ldab, anorm, rcond, work, iwork, info )
call dpbcon( uplo, n, kd, ab, ldab, anorm, rcond, work, iwork, info )
call cpbcon( uplo, n, kd, ab, ldab, anorm, rcond, work, rwork, info )
call zpbcon( uplo, n, kd, ab, ldab, anorm, rcond, work, rwork, info )
```

Fortran 95:

```
call pbcon( a, anorm, rcond [,uplo] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite band matrix A:

$\kappa_1(A) = ||A||_1 \ ||A^{-1}||_1$ (since A is symmetric or Hermitian, $\kappa_\infty(A) = \kappa_1(A)$).

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?pbtrf to compute the Cholesky factorization of A.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If *uplo* = 'U', the upper triangular factor is stored in *ab*. If *uplo* = 'L', the lower triangular factor is stored in *ab*.
n	INTEGER. The order of the matrix A; $n \geq 0$.
kd	INTEGER. The number of superdiagonals or subdiagonals in the matrix A; $kd \geq 0$.
ldab	INTEGER. The first dimension of the array *ab*. (*ldab* ≥ *kd* +1).
ab, *work*	REAL for spbcon DOUBLE PRECISION for dpbcon COMPLEX for cpbcon DOUBLE COMPLEX for zpbcon. Arrays: *ab*(*ldab*,*), *work*(*). The array *ab* contains the factored matrix A in band form, as returned by ?pbtrf. The second dimension of *ab* must be at least max(1, *n*). The array *work* is a workspace for the routine. The dimension of *work* must be at least max(1, 3*n) for real flavors and max(1, 2*n) for complex flavors.
anorm	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The norm of the original matrix A (see Description).
iwork	INTEGER. Workspace array, DIMENSION at least max(1, *n*).
rwork	REAL for cpbcon

DOUBLE PRECISION for zpbcon.
Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

rcond — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info — INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine pbcon interface are as follows:

a — Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*kd*+1,*n*).

uplo — Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations *A***x* = *b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately 4**n*(*kd* + 1) floating-point operations for real flavors and 16**n*(*kd* + 1) for complex flavors.

?ptcon

Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite tridiagonal matrix.

Syntax

Fortran 77:

```
call sptcon( n, d, e, anorm, rcond, work, info )
call dptcon( n, d, e, anorm, rcond, work, info )
call cptcon( n, d, e, anorm, rcond, work, info )
call zptcon( n, d, e, anorm, rcond, work, info )
```

Fortran 95:

```
call ptcon( d, e, anorm, rcond [,info] )
```

Description

This routine computes the reciprocal of the condition number (in the 1-norm) of a real symmetric or complex Hermitian positive-definite tridiagonal matrix using the factorization $A = L*D*L^T$ for real flavors and $A = L*D*L^H$ for complex flavors or $A = U^T*D*U$ for real flavors and $A = U^H*D*U$ for complex flavors computed by ?pttrf :

$\kappa_1(A) = ||A||_1 \; ||A^{-1}||_1$ (since A is symmetric or Hermitian, $\kappa_\infty(A) = \kappa_1(A)$).

The norm $||A^{-1}||$ is computed by a direct method, and the reciprocal of the condition number is computed as $rcond = 1 / (||A|| \; ||A^{-1}||)$.

Before calling this routine:

- compute *anorm* as $||A||_1 = \max_j \Sigma_i \; |a_{ij}|$
- call ?pttrf to compute the factorization of A.

Input Parameters

n	INTEGER. The order of the matrix A; $n \geq 0$.
d, *work*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

Arrays, dimension (*n*).
The array *d* contains the *n* diagonal elements of the diagonal matrix *D* from the factorization of *A*, as computed by ?pttrf ;
work is a workspace array.

e — REAL for sptcon
DOUBLE PRECISION for dptcon
COMPLEX for cptcon
DOUBLE COMPLEX for zptcon.
Array, DIMENSION (*n* -1).
Contains off-diagonal elements of the unit bidiagonal factor *U* or *L* from the factorization computed by ?pttrf .

anorm — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
The 1- norm of the original matrix *A* (see Description).

Output Parameters

rcond — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gtcon interface are as follows:

d — Holds the vector of length (*n*).

e — Holds the vector of length (*n*-1).

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations *A*x = b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately 4*n(kd + 1) floating-point operations for real flavors and 16*n(kd + 1) for complex flavors.

?sycon

Estimates the reciprocal of the condition number of a symmetric matrix.

Syntax

Fortran 77:

```
call ssycon( uplo, n, a, lda, ipiv, anorm, rcond, work, iwork, info )
call dsycon( uplo, n, a, lda, ipiv, anorm, rcond, work, iwork, info )
call csycon( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call zsycon( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
```

Fortran 95:

```
call sycon( a, ipiv, anorm, rcond [,uplo] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a symmetric matrix *A*:

$\kappa_1(A) = ||A||_1 \; ||A^{-1}||_1$ (since *A* is symmetric, $\kappa_\infty(A) = \kappa_1(A)$).

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?sytrf to compute the factorization of *A*.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.

Indicates how the input matrix *A* has been factored:
If *uplo* = 'U', the array *a* stores the upper triangular factor *U* of the factorization *A* = *P*U*D*U*^{T}**P*^{T}.
If *uplo* = 'L', the array *a* stores the lower triangular factor *L* of the factorization *A* = *P*L*D*L*^{T}**P*^{T}.

n — INTEGER. The order of matrix *A*; $n \geq 0$.

a, *work* — REAL for ssycon
DOUBLE PRECISION for dsycon
COMPLEX for csycon
DOUBLE COMPLEX for zsycon.
Arrays: *a*(*lda*,*), *work*(*).
The array *a* contains the factored matrix *A*, as returned by ?sytrf. The second dimension of *a* must be at least max(1,*n*).
The array *work* is a workspace for the routine.
The dimension of *work* must be at least max(1, 2**n*).

lda — INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).

ipiv — INTEGER. Array, DIMENSION at least max(1, *n*).
The array *ipiv*, as returned by ?sytrf.

anorm — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
The norm of the original matrix *A* (see Description).

iwork — INTEGER. Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

rcond — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info — INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `sycon` interface are as follows:

a	Holds the matrix A of size (n, n).
ipiv	Holds the vector of length (n).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

The computed *rcond* is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?hecon

Estimates the reciprocal of the condition number of a Hermitian matrix.

Syntax

Fortran 77:

```
call checon( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call zhecon( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
```

Fortran 95:

```
call hecon( a, ipiv, anorm, rcond [,uplo] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a Hermitian matrix A:

$\kappa_1(A) = ||A||_1 \ ||A^{-1}||_1$ (since A is Hermitian, $\kappa_\infty(A) = \kappa_1(A)$).

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?hetrf to compute the factorization of *A*.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix *A* has been factored: If *uplo* = 'U', the array *a* stores the upper triangular factor *U* of the factorization *A* = *P*U*D*U^H*P^T*. If *uplo* = 'L', the array *a* stores the lower triangular factor *L* of the factorization *A* = *P*L*D*L^H*P^T*.
n	INTEGER. The order of matrix *A*; $n \geq 0$.
a, *work*	COMPLEX for checon DOUBLE COMPLEX for zhecon. Arrays: *a*(*lda*,*), *work*(*). The array *a* contains the factored matrix *A*, as returned by ?hetrf. The second dimension of *a* must be at least max(1,*n*). The array *work* is a workspace for the routine. The dimension of *work* must be at least max(1, 2**n*).
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).
ipiv	INTEGER. Array, DIMENSION at least max(1, *n*). The array *ipiv*, as returned by ?hetrf.
anorm	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. The norm of the original matrix *A* (see Description).

Output Parameters

rcond	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `hecon` interface are as follows:

a	Holds the matrix A of size (n, n).
ipiv	Holds the vector of length (n).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

The computed *rcond* is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$; the number is usually 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

?spcon

Estimates the reciprocal of the condition number of a packed symmetric matrix.

Syntax

Fortran 77:

```
call sspcon( uplo, n, ap, ipiv, anorm, rcond, work, iwork, info )
call dspcon( uplo, n, ap, ipiv, anorm, rcond, work, iwork, info )
call cspcon( uplo, n, ap, ipiv, anorm, rcond, work, info )
call zspcon( uplo, n, ap, ipiv, anorm, rcond, work, info )
```

Fortran 95:

```
call spcon( a, ipiv, anorm, rcond [,uplo] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a packed symmetric matrix A:

$\kappa_1(A) = ||A||_1 \; ||A^{-1}||_1$ (since A is symmetric, $\kappa_\infty(A) = \kappa_1(A)$).

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?sptrf to compute the factorization of *A*.

Input Parameters

uplo — CHARACTER*1. Must be 'U' or 'L'.
Indicates how the input matrix *A* has been factored:
If *uplo* = 'U', the array *ap* stores the packed upper triangular factor *U* of the factorization *A* = *P*U*D*U*^{T}**P*^{T}.
If *uplo* = 'L', the array *ap* stores the packed lower triangular factor *L* of the factorization *A* = *P*L*D*L*^{T}**P*^{T}.

n — INTEGER. The order of matrix *A*; *n* ≥ 0.

ap, *work* — REAL for sspcon
DOUBLE PRECISION for dspcon
COMPLEX for cspcon
DOUBLE COMPLEX for zspcon.
Arrays: *ap*(*), *work*(*).
The array *ap* contains the packed factored matrix *A*, as returned by ?sptrf. The dimension of *ap* must be at least max(1,*n*(*n*+1)/2).
The array *work* is a workspace for the routine.
The dimension of *work* must be at least max(1, 2*n).

ipiv — INTEGER. Array, DIMENSION at least max(1, *n*).
The array *ipiv*, as returned by ?sptrf.

anorm — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
The norm of the original matrix *A* (see Description).

iwork — INTEGER. Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

rcond — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.

	An estimate of the reciprocal of the condition number. The routine sets *rcond* = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine spcon interface are as follows:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).
ipiv	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations *A***x* = *b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?hpcon

Estimates the reciprocal of the condition number of a packed Hermitian matrix.

Syntax

Fortran 77:

```
call chpcon( uplo, n, ap, ipiv, anorm, rcond, work, info )
call zhpcon( uplo, n, ap, ipiv, anorm, rcond, work, info )
```

Fortran 95:

```
call hpcon( a, ipiv, anorm, rcond [,uplo] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a Hermitian matrix A:

$\kappa_1(A) = ||A||_1 \ ||A^{-1}||_1$ (since A is Hermitian, $\kappa_\infty(A) = k_1(A)$).

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call ?hptrf to compute the factorization of A.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If *uplo* = 'U', the array *ap* stores the packed upper triangular factor U of the factorization $A = P*U*D*U^T*P^T$. If *uplo* = 'L', the array *ap* stores the packed lower triangular factor L of the factorization $A = P*L*D*L^T*P^T$.
n	INTEGER. The order of matrix A; $n \geq 0$.
ap, *work*	COMPLEX for chpcon DOUBLE COMPLEX for zhpcon. Arrays: *ap*(*), *work*(*). The array *ap* contains the packed factored matrix A, as returned by ?hptrf. The dimension of *ap* must be at least max(1,n(n+1)/2). The array *work* is a workspace for the routine. The dimension of *work* must be at least max(1, 2*n).
ipiv	INTEGER. Array, DIMENSION at least max(1, n). The array *ipiv*, as returned by ?hptrf.
anorm	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. The norm of the original matrix A (see Description).

Output Parameters

rcond	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hbcon interface are as follows:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).
ipiv	Holds the vector of length (*n*).

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations *A***x* = *b*; the number is usually 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

?trcon

Estimates the reciprocal of the condition number of a triangular matrix.

Syntax

Fortran 77:

```
call strcon( norm, uplo, diag, n, a, lda, rcond, work, iwork, info )
call dtrcon( norm, uplo, diag, n, a, lda, rcond, work, iwork, info )
call ctrcon( norm, uplo, diag, n, a, lda, rcond, work, rwork, info )
call ztrcon( norm, uplo, diag, n, a, lda, rcond, work, rwork, info )
```

Fortran 95:

```
call trcon( a, rcond [,uplo] [,diag] [,norm] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a triangular matrix A in either the 1-norm or infinity-norm:

$\kappa_1(A) = ||A||_1 \, ||A^{-1}||_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$

$\kappa_\infty(A) = ||A||_\infty \, ||A^{-1}||_\infty = k_1(A^T) = \kappa_1(A^H)$.

Input Parameters

norm — CHARACTER*1. Must be '1' or 'O' or 'I'.
If *norm* = '1' or 'O', then the routine estimates the condition number of matrix A in 1-norm.
If *norm* = 'I', then the routine estimates the condition number of matrix A in infinity-norm.

uplo — CHARACTER*1. Must be 'U' or 'L'.
Indicates whether A is upper or lower triangular:
If *uplo* = 'U', the array *a* stores the upper triangle of A, other array elements are not referenced.
If *uplo* = 'L', the array *a* stores the lower triangle of A, other array elements are not referenced.

diag — CHARACTER*1. Must be 'N' or 'U'.

If *diag* = 'N', then *A* is not a unit triangular matrix.
If *diag* = 'U', then *A* is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array *a*.

n — INTEGER. The order of the matrix *A*; $n \geq 0$.

a, *work* — REAL for strcon
DOUBLE PRECISION for dtrcon
COMPLEX for ctrcon
DOUBLE COMPLEX for ztrcon.
Arrays: *a*(*lda*,*), *work*(*).
The array *a* contains the matrix *A*. The second dimension of *a* must be at least max(1,*n*).
The array *work* is a workspace for the routine. The dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.

lda — INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).

iwork — INTEGER. Workspace array, DIMENSION at least max(1, *n*).

rwork — REAL for ctrcon
DOUBLE PRECISION for ztrcon.
Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

rcond — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info — INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `trcon` interface are as follows:

a	Holds the matrix A of size (n, n).
norm	Must be '1', 'O', or 'I'. The default value is '1'.
uplo	Must be 'U' or 'L'. The default value is 'U'.
diag	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The computed *rcond* is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors and $4n^2$ operations for complex flavors.

?tpcon

Estimates the reciprocal of the condition number of a packed triangular matrix.

Syntax

Fortran 77:

```
call stpcon( norm, uplo, diag, n, ap, rcond, work, iwork, info )
call dtpcon( norm, uplo, diag, n, ap, rcond, work, iwork, info )
call ctpcon( norm, uplo, diag, n, ap, rcond, work, rwork, info )
call ztpcon( norm, uplo, diag, n, ap, rcond, work, rwork, info )
```

Fortran 95:

```
call tpcon( a, rcond [,uplo] [,diag] [,norm] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a packed triangular matrix A in either the 1-norm or infinity-norm:

$\kappa_1(A) = ||A||_1 \, ||A^{-1}||_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$

$\kappa_\infty(A) = ||A||_\infty \, ||A^{-1}||_\infty = \kappa_1(A^T) = \kappa_1(A^H)$.

Input Parameters

norm	CHARACTER*1. Must be '1' or 'O' or 'I'. If *norm* = '1' or 'O', then the routine estimates the condition number of matrix *A* in 1-norm. If *norm* = 'I', then the routine estimates the condition number of matrix *A* in infinity-norm.
uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether *A* is upper or lower triangular: If *uplo* = 'U', the array *ap* stores the upper triangle of *A* in packed form. If *uplo* = 'L', the array *ap* stores the lower triangle of *A* in packed form.
diag	CHARACTER*1. Must be 'N' or 'U'. If *diag* = 'N', then *A* is not a unit triangular matrix. If *diag* = 'U', then *A* is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array *ap*.
n	INTEGER. The order of the matrix *A*; $n \geq 0$.
ap, *work*	REAL for stpcon DOUBLE PRECISION for dtpcon COMPLEX for ctpcon DOUBLE COMPLEX for ztpcon. Arrays: *ap*(*), *work*(*). The array *ap* contains the packed matrix *A*. The dimension of *ap* must be at least max(1,*n*(*n*+1)/2). The array *work* is a workspace for the routine. The dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.
iwork	INTEGER. Workspace array, DIMENSION at least max(1, *n*).
rwork	REAL for ctpcon DOUBLE PRECISION for ztpcon. Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

rcond	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors.

	An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine tpcon interface are as follows:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).
norm	Must be '1', 'O', or 'I'. The default value is '1'.
uplo	Must be 'U' or 'L'. The default value is 'U'.
diag	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations *A***x* = *b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors and $4n^2$ operations for complex flavors.

?tbcon

Estimates the reciprocal of the condition number of a triangular band matrix.

Syntax

Fortran 77:

```
call stbcon( norm, uplo, diag, n, kd, ab, ldab, rcond, work, iwork, info )
call dtbcon( norm, uplo, diag, n, kd, ab, ldab, rcond, work, iwork, info )
call ctbcon( norm, uplo, diag, n, kd, ab, ldab, rcond, work, rwork, info )
call ztbcon( norm, uplo, diag, n, kd, ab, ldab, rcond, work, rwork, info )
```

Fortran 95:

```
call tbcon( a, rcond [,uplo] [,diag] [,norm] [,info] )
```

Description

This routine estimates the reciprocal of the condition number of a triangular band matrix A in either the 1-norm or infinity-norm:

$\kappa_1(A) = ||A||_1 \, ||A^{-1}||_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$

$\kappa_\infty(A) = ||A||_\infty \, ||A^{-1}||_\infty = \kappa_1(A^T) = \kappa_1(A^H)$.

Input Parameters

norm	CHARACTER*1. Must be '1' or 'O' or 'I'. If *norm* = '1' or 'O', then the routine estimates the condition number of matrix A in 1-norm. If *norm* = 'I', then the routine estimates the condition number of matrix A in infinity-norm.
uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If *uplo* = 'U', the array *ap* stores the upper triangle of A in packed form. If *uplo* = 'L', the array *ap* stores the lower triangle of A in packed form.
diag	CHARACTER*1. Must be 'N' or 'U'.

If *diag* = 'N', then *A* is not a unit triangular matrix.
If *diag* = 'U', then *A* is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array *ab*.

n — INTEGER. The order of the matrix *A*; $n \geq 0$.

kd — INTEGER. The number of superdiagonals or subdiagonals in the matrix *A*; $kd \geq 0$.

ab, *work* — REAL for stbcon
DOUBLE PRECISION for dtbcon
COMPLEX for ctbcon
DOUBLE COMPLEX for ztbcon.
Arrays: *ab*(*ldab*,*), *work*(*).
The array *ab* contains the band matrix *A*. The second dimension of *ab* must be at least max(1,*n*). The array *work* is a workspace for the routine.
The dimension of *work* must be at least max(1, 3*n) for real flavors and max(1, 2*n) for complex flavors.

ldab — INTEGER. The first dimension of the array *ab*. (*ldab* ≥ *kd* +1).

iwork — INTEGER. Workspace array, DIMENSION at least max(1, *n*).

rwork — REAL for ctbcon
DOUBLE PRECISION for ztbcon.
Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

rcond — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal of the condition number. The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info — INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `tbcon` interface are as follows:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*kd*+1,*n*).
norm	Must be '1', 'O', or 'I'. The default value is '1'.
uplo	Must be 'U' or 'L'. The default value is 'U'.
diag	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations *A***x* = *b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately 2**n*(*kd* + 1) floating-point operations for real flavors and 8**n*(*kd* + 1) operations for complex flavors.

Refining the Solution and Estimating Its Error

This section describes the LAPACK routines for refining the computed solution of a system of linear equations and estimating the solution error. You can call these routines after factorizing the matrix of the system of equations and computing the solution (see Routines for Matrix Factorization and Routines for Solving Systems of Linear Equations).

?gerfs

Refines the solution of a system of linear equations with a general matrix and estimates its error.

Syntax

Fortran 77:

```
call sgerfs( trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
berr, work, iwork, info )
call dgerfs( trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
berr, work, iwork, info )
call cgerfs( trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
berr, work, rwork, info )
call zgerfs( trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
berr, work, rwork, info )
```

Fortran 95:

```
call gerfs( a, af, ipiv, b, x [,trans] [,ferr] [,berr] [,info] )
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $A*X = B$ or $A^T*X = B$ or $A^H*X = B$ with a general matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the component-wise backward error β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta|a_{ij}|$, $|\delta b_i|/|b_i| \le \beta |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the component-wise forward error in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?getrf
- call the solver routine ?getrs.

Input Parameters

trans	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If *trans* = 'N', the system has the form *A*X* = *B*. If *trans* = 'T', the system has the form $A^T*X = B$. If *trans* = 'C', the system has the form $A^H*X = B$.
n	INTEGER. The order of the matrix *A*; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
a,*af*,*b*,*x*,*work*	REAL for sgerfs DOUBLE PRECISION for dgerfs COMPLEX for cgerfs DOUBLE COMPLEX for zgerfs. Arrays: *a*(*lda*,*) contains the original matrix *A*, as supplied to ?getrf. *af*(*ldaf*,*) contains the factored matrix *A*, as returned by ?getrf. *b*(*ldb*,*) contains the right-hand side matrix *B*. *x*(*ldx*,*) contains the solution matrix *X*. *work*(*) is a workspace array. The second dimension of *a* and *af* must be at least max(1, *n*); the second dimension of *b* and *x* must be at least max(1, *nrhs*); the dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).
ldaf	INTEGER. The first dimension of *af*; *ldaf* ≥ max(1, *n*).
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).
ldx	INTEGER. The first dimension of *x*; *ldx* ≥ max(1, *n*).
ipiv	INTEGER. Array, DIMENSION at least max(1, *n*). The *ipiv* array, as returned by ?getrf.
iwork	INTEGER. Workspace array, DIMENSION at least max(1, *n*).
rwork	REAL for cgerfs DOUBLE PRECISION for zgerfs.

Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

x	The refined solution matrix *X*.
ferr, *berr*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1, *nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gerfs interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
af	Holds the matrix *AF* of size (*n*, *n*).
ipiv	Holds the vector of length (*n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward

error involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?gbrfs

Refines the solution of a system of linear equations with a general band matrix and estimates its error.

Syntax

Fortran 77:

```
call sgbrfs( trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, b, ldb, x,
ldx, ferr, berr, work, iwork, info )

call dgbrfs( trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, b, ldb, x,
ldx, ferr, berr, work, iwork, info )

call cgbrfs( trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, b, ldb, x,
ldx, ferr, berr, work, rwork, info )

call zgbrfs( trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, b, ldb, x,
ldx, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call gbrfs( a, af, ipiv, b, x [,kl] [,trans] [,ferr] [,berr] [,info] )
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $A*X = B$ or $A^T*X = B$ or $A^H*X = B$ with a band matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the component-wise backward error β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta|a_{ij}|$, $|\delta b_i|/|b_i| \le \beta|b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the component-wise forward error in the computed solution $||x - x_e||_\infty / ||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?gbtrf

- call the solver routine ?gbtrs.

Input Parameters

trans	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If *trans* = 'N', the system has the form *A*X* = *B*. If *trans* = 'T', the system has the form A^T*X = *B*. If *trans* = 'C', the system has the form A^H*X = *B*.
n	INTEGER. The order of the matrix *A*; *n* ≥ 0.
kl	INTEGER. The number of sub-diagonals within the band of *A*; *kl* ≥ 0.
ku	INTEGER. The number of super-diagonals within the band of *A*; *ku* ≥ 0.
nrhs	INTEGER. The number of right-hand sides; *nrhs* ≥ 0.
ab,*afb*,*b*,*x*,*work*	REAL for sgbrfs DOUBLE PRECISION for dgbrfs COMPLEX for cgbrfs DOUBLE COMPLEX for zgbrfs. Arrays: *ab*(*ldab*,*) contains the original band matrix *A*, as supplied to ?gbtrf, but stored in rows from 1 to *kl* + *ku* + 1. *afb*(*ldafb*,*) contains the factored band matrix *A*, as returned by ?gbtrf. *b*(*ldb*,*) contains the right-hand side matrix *B*. *x*(*ldx*,*) contains the solution matrix *X*. *work*(*) is a workspace array. The second dimension of *ab* and *afb* must be at least max(1, *n*); the second dimension of *b* and *x* must be at least max(1, *nrhs*); the dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.
ldab	INTEGER. The first dimension of *ab*.
ldafb	INTEGER. The first dimension of *afb* .
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).
ldx	INTEGER. The first dimension of *x*; *ldx* ≥ max(1, *n*).
ipiv	INTEGER.

	Array, DIMENSION at least max(1, *n*). The *ipiv* array, as returned by ?gbtrf.
iwork	INTEGER. Workspace array, DIMENSION at least max(1, *n*).
rwork	REAL for cgbrfs DOUBLE PRECISION for zgbrfs. Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

x	The refined solution matrix *X*.
ferr, *berr*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1, *nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	INTEGER. If *info* =0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gbrfs interface are as follows:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*kl*+*ku*+1, *n*).
af	Stands for argument *afb* in Fortan 77 interface. Holds the array *AF* of size (2**kl***ku*+1, *n*).
ipiv	Holds the vector of length (*n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
kl	If omitted, assumed *kl* = *ku*.
ku	Restored as *ku* = *lda*-*kl*-1.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n(kl + ku)$ floating-point operations (for real flavors) or $16n(kl + ku)$ operations (for complex flavors). In addition, each step of iterative refinement involves $2n(4kl + 3ku)$ operations (for real flavors) or $8n(4kl + 3ku)$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?gtrfs

Refines the solution of a system of linear equations with a tridiagonal matrix and estimates its error.

Syntax

Fortran 77:

```
call sgtrfs( trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x,
ldx, ferr, berr, work, iwork, info )
```

```
call dgtrfs( trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x,
ldx, ferr, berr, work, iwork, info )
```

```
call cgtrfs( trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x,
ldx, ferr, berr, work, rwork, info )
```

```
call zgtrfs( trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x,
ldx, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call gtrfs( dl, d, du, dlf, df, duf, du2, ipiv, b, x [,trans] [,ferr] [,berr]
[,info] )
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $A*X = B$ or $A^T*X = B$ or $A^H*X = B$ with a tridiagonal matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the component-wise backward error β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta|a_{ij}|$, $|\delta b_i|/|b_i| \le \beta|b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the component-wise forward error in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?gttrf
- call the solver routine ?gttrs.

Input Parameters

trans	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If *trans* = 'N', the system has the form $A*X = B$. If *trans* = 'T', the system has the form $A^T*X = B$. If *trans* = 'C', the system has the form $A^H*X = B$.
n	INTEGER. The order of the matrix A; $n \ge 0$.
nrhs	INTEGER. The number of right-hand sides, that is, the number of columns of the matrix B; $nrhs \ge 0$.
dl,*d*,*du*,*dlf*, *df*,*duf*,*du2*,	REAL for sgtrfs DOUBLE PRECISION for dgtrfs COMPLEX for cgtrfs DOUBLE COMPLEX for zgtrfs.
b,*x*,*work*	Arrays: *dl*, dimension (n -1), contains the subdiagonal elements of A. *d*, dimension (n), contains the diagonal elements of A. *du*, dimension (n -1), contains the superdiagonal elements of A. *dlf*, dimension (n -1), contains the (n - 1) multipliers that define the matrix L from the LU factorization of A as computed by ?gttrf.

	df, dimension (*n*), contains the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*. *duf*, dimension (*n* -1), contains the (*n* - 1) elements of the first superdiagonal of *U*. *du2*, dimension (*n* -2), contains the (*n* - 2) elements of the second superdiagonal of *U*. *b*(*ldb*,*nrhs*) contains the right-hand side matrix *B*. *x*(*ldx*,*nrhs*) contains the solution matrix *X*, as computed by ?gttrs. *work*(*) is a workspace array; the dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).
ldx	INTEGER. The first dimension of *x*; *ldx* ≥ max(1, *n*).
ipiv	INTEGER. Array, DIMENSION at least max(1, *n*). The *ipiv* array, as returned by ?gttrf.
iwork	INTEGER. Workspace array, DIMENSION (*n*). Used for real flavors only.
rwork	REAL for cgtrfs DOUBLE PRECISION for zgtrfs. Workspace array, DIMENSION (*n*). Used for complex flavors only.

Output Parameters

x	The refined solution matrix *X*.
ferr, *berr*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1,*nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gtrfs interface are as follows:

dl	Holds the vector of length (*n*-1).
d	Holds the vector of length (*n*).
du	Holds the vector of length (*n*-1).
dlf	Holds the vector of length (*n*-1).
df	Holds the vector of length (*n*).
duf	Holds the vector of length (*n*-1).
du2	Holds the vector of length (*n*-2).
ipiv	Holds the vector of length (*n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.

?porfs

Refines the solution of a system of linear equations with a symmetric (Hermitian) positive-definite matrix and estimates its error.

Syntax

Fortran 77:

```
call sporfs( uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr,
work, iwork, info )

call dporfs( uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr,
work, iwork, info )

call cporfs( uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr,
work, rwork, info )

call zporfs( uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr,
work, rwork, info )
```

Fortran 95:

```
call porfs( a, af, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations *A*X = B* with a symmetric (Hermitian) positive definite matrix *A*, with multiple right-hand sides. For each computed solution vector *x*, the routine computes the component-wise backward error β. This error is the smallest relative perturbation in elements of *A* and *b* such that *x* is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta|a_{ij}|$, $|\delta b_i|/|b_i| \le \beta|b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the component-wise forward error in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?potrf
- call the solver routine ?potrs.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', the upper triangle of *A* is stored. If *uplo* = 'L', the lower triangle of *A* is stored.
n	INTEGER. The order of the matrix *A*; *n* ≥ 0.
nrhs	INTEGER. The number of right-hand sides; *nrhs* ≥ 0.
a,*af*,*b*,*x*,*work*	REAL for sporfs DOUBLE PRECISION for dporfs COMPLEX for cporfs DOUBLE COMPLEX for zporfs. Arrays: *a*(*lda*,*) contains the original matrix *A*, as supplied to ?potrf. *af*(*ldaf*,*) contains the factored matrix *A*, as returned by ?potrf. *b*(*ldb*,*) contains the right-hand side matrix *B*. *x*(*ldx*,*) contains the solution matrix *X*. *work*(*) is a workspace array. The second dimension of *a* and *af* must be at least max(1, *n*); the second dimension of *b* and *x* must be at least max(1, *nrhs*); the dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.

lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).
ldaf	INTEGER. The first dimension of *af*; *ldaf* ≥ max(1, *n*).
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).
ldx	INTEGER. The first dimension of *x*; *ldx* ≥ max(1, *n*).
iwork	INTEGER. Workspace array, DIMENSION at least max(1, *n*).
rwork	REAL for cporfs DOUBLE PRECISION for zporfs. Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

x	The refined solution matrix *X*.
ferr, *berr*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1, *nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine porfs interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
af	Holds the matrix *AF* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations *A*x = b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?pprfs

Refines the solution of a system of linear equations with a packed symmetric (Hermitian) positive-definite matrix and estimates its error.

Syntax

Fortran 77:

```
call spprfs( uplo, n, nrhs, ap, afp, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call dpprfs( uplo, n, nrhs, ap, afp, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call cpprfs( uplo, n, nrhs, ap, afp, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call zpprfs( uplo, n, nrhs, ap, afp, b, ldb, x, ldx, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call pprfs( a, af, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations *A*X = B* with a packed symmetric (Hermitian)positive definite matrix *A*, with multiple right-hand sides. For each computed solution vector *x*, the routine computes the component-wise backward error β. This error is the smallest relative perturbation in elements of *A* and *b* such that *x* is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta|a_{ij}|$, $|\delta b_i|/|b_i| \le \beta|b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the component-wise forward error in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?pptrf
- call the solver routine ?pptrs.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix *A* has been factored: If *uplo* = 'U', the upper triangle of *A* is stored. If *uplo* = 'L', the lower triangle of *A* is stored.
n	INTEGER. The order of the matrix *A*; $n \ge 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \ge 0$.
ap,*afp*,*b*,*x*,*work*	REAL for spprfs DOUBLE PRECISION for dpprfs COMPLEX for cpprfs DOUBLE COMPLEX for zpprfs. Arrays: *ap*(*) contains the original packed matrix *A*, as supplied to ?pptrf. *afp*(*) contains the factored packed matrix *A*, as returned by ?pptrf. *b*(*ldb*,*) contains the right-hand side matrix *B*. *x*(*ldx*,*) contains the solution matrix *X*. *work*(*) is a workspace array.

The dimension of arrays *ap* and *afp* must be at least max(1,*n*(*n*+1)/2); the second dimension of *b* and *x* must be at least max(1, *nrhs*); the dimension of *work* must be at least max(1, 3*n) for real flavors and max(1, 2*n) for complex flavors.

ldb INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

ldx INTEGER. The first dimension of *x*; *ldx* ≥ max(1, *n*).

iwork INTEGER. Workspace array, DIMENSION at least max(1, *n*).

rwork REAL for cpprfs
DOUBLE PRECISION for zpprfs.
Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

x The refined solution matrix *X*.

ferr, *berr* REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least max(1, *nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.

info INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine pprfs interface are as follows:

a Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).

af Stands for argument *apf* in Fortan 77 interface. Holds the array *AF* of size (*n**(*n*+1)/2).

b Holds the matrix *B* of size (*n*, *nrhs*).

x Holds the matrix *X* of size (*n*, *nrhs*).

ferr Holds the vector of length (*nrhs*).

berr Holds the vector of length (*nrhs*).

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A^*x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?pbrfs

Refines the solution of a system of linear equations with a band symmetric (Hermitian) positive-definite matrix and estimates its error.

Syntax

Fortran 77:

```
call spbrfs( uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx, ferr,
berr, work, iwork, info )

call dpbrfs( uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx, ferr,
berr, work, iwork, info )

call cpbrfs( uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx, ferr,
berr, work, rwork, info )

call zpbrfs( uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx, ferr,
berr, work, rwork, info )
```

Fortran 95:

```
call pbrfs( a, af, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations *A*X = B* with a symmetric (Hermitian) positive definite band matrix *A*, with multiple right-hand sides. For each computed solution vector *x*, the routine computes the component-wise backward error β. This error is the smallest relative perturbation in elements of *A* and *b* such that *x* is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta|a_{ij}|$, $|\delta b_i|/|b_i| \le \beta|b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the component-wise forward error in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?pbtrf
- call the solver routine ?pbtrs.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix *A* has been factored: If *uplo* = 'U', the upper triangle of *A* is stored. If *uplo* = 'L', the lower triangle of *A* is stored.
n	INTEGER. The order of the matrix *A*; $n \ge 0$.
kd	INTEGER. The number of superdiagonals or subdiagonals in the matrix *A*; $kd \ge 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \ge 0$.
ab,*afb*,*b*,*x*,*work*	REAL for spbrfs DOUBLE PRECISION for dpbrfs COMPLEX for cpbrfs DOUBLE COMPLEX for zpbrfs. Arrays: *ab*(*ldab*,*) contains the original band matrix *A*, as supplied to ?pbtrf. *afb*(*ldafb*,*) contains the factored band matrix *A*, as returned by ?pbtrf. *b*(*ldb*,*) contains the right-hand side matrix *B*. *x*(*ldx*,*) contains the solution matrix *X*. *work*(*) is a workspace array.

	The second dimension of *ab* and *afb* must be at least max(1, *n*); the second dimension of *b* and *x* must be at least max(1, *nrhs*); the dimension of *work* must be at least max(1, 3*n) for real flavors and max(1, 2*n) for complex flavors.
ldab	INTEGER. The first dimension of *ab*; *ldab* ≥ *kd* + 1.
ldafb	INTEGER. The first dimension of *afb*; *ldafb* ≥ *kd* + 1.
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).
ldx	INTEGER. The first dimension of *x*; *ldx* ≥ max(1, *n*).
iwork	INTEGER. Workspace array, DIMENSION at least max(1, *n*).
rwork	REAL for cpbrfs DOUBLE PRECISION for zpbrfs. Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

x	The refined solution matrix *X*.
ferr, *berr*	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1, *nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine pbrfs interface are as follows:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*kd*+1, *n*).
af	Stands for argument *afb* in Fortan 77 interface. Holds the array *AF* of size (*kd*+1, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).

ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $8n*kd$ floating-point operations (for real flavors) or $32n*kd$ operations (for complex flavors). In addition, each step of iterative refinement involves $12n*kd$ operations (for real flavors) or $48n*kd$ operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $4n*kd$ floating-point operations for real flavors or $16n*kd$ for complex flavors.

?ptrfs

Refines the solution of a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal matrix and estimates its error.

Syntax

Fortran 77:

```
call sptrfs( n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr, work, info )
call dptrfs( n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr, work, info )
call cptrfs( uplo, n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr, work,
rwork, info )
call cptrfs( uplo, n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr, work,
rwork, info )
```

Fortran 95:

```
call ptrfs( d, df, e, ef, b, x [,ferr] [,berr] [,info] )
call ptrfs( d, df, e, ef, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations *A*X = B* with a symmetric (Hermitian) positive definite tridiagonal matrix *A*, with multiple right-hand sides. For each computed solution vector *x*, the routine computes the component-wise backward error β. This error is the smallest relative perturbation in elements of *A* and *b* such that *x* is the exact solution of the perturbed system:

$|\delta a_{ij}| / |a_{ij}| \le \beta |a_{ij}|$, $|\delta b_i| / |b_i| \le \beta |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the component-wise forward error in the computed solution $||x - x_e||_\infty / ||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?pttrf
- call the solver routine ?pttrs.

Input Parameters

uplo	CHARACTER*1. Used for complex flavors only. Must be 'U' or 'L'. Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix *A* is stored and how *A* is factored: If *uplo* = 'U', the array *e* stores the superdiagonal of *A*, and *A* is factored as U^H*D*U. If *uplo* = 'L', the array *e* stores the subdiagonal of *A*, and *A* is factored as $L*D*L^H$.
n	INTEGER. The order of the matrix *A*; $n \ge 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \ge 0$.
d, *df*, *rwork*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors Arrays: *d*(*n*), *df*(*n*), *rwork*(*n*). The array *d* contains the *n* diagonal elements of the tridiagonal matrix *A*. The array *df* contains the *n* diagonal elements of the diagonal matrix *D* from the factorization of *A* as computed by ?pttrf. The array *rwork* is a workspace array used for complex flavors only.
e,*ef*,*b*,*x*,*work*	REAL for sptrfs DOUBLE PRECISION for dptrfs

COMPLEX for cptrfs
DOUBLE COMPLEX for zptrfs.
Arrays: e(n -1), ef(n -1), b(ldb,nrhs), x(ldx,nrhs), work(*).
The array e contains the (n - 1) off-diagonal elements of the tridiagonal matrix A (see uplo).
The array ef contains the (n - 1) off-diagonal elements of the unit bidiagonal factor U or L from the factorization computed by ?pttrf (see uplo).
The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.
The array x contains the solution matrix X as computed by ?pttrs.
The array work is a workspace array. The dimension of work must be at least 2*n for real flavors, and at least n for complex flavors.

ldb — INTEGER. The leading dimension of b; ldb ≥ max(1, n).

ldx — INTEGER. The leading dimension of x; ldx ≥ max(1, n).

Output Parameters

x — The refined solution matrix X.

ferr, *berr* — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least max(1, nrhs). Contain the component-wise forward and backward errors, respectively, for each solution vector.

info — INTEGER.
If info = 0, the execution is successful.
If info = -i, the i-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ptrfs interface are as follows:

d — Holds the vector of length (n).

df — Holds the vector of length (n).

e — Holds the vector of length (n-1).

ef	Holds the vector of length (*n*-1).
b	Holds the matrix *B* of size (*n*,*nrhs*).
x	Holds the matrix *X* of size (*n*,*nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Used in complex flavors only. Must be 'U' or 'L'. The default value is 'U'.

?syrfs

Refines the solution of a system of linear equations with a symmetric matrix and estimates its error.

Syntax

Fortran 77:

```
call ssyrfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
berr, work, iwork, info )
call dsyrfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
berr, work, iwork, info )
call csyrfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
berr, work, rwork, info )
call zsyrfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
berr, work, rwork, info )
```

Fortran 95:

```
call syrfs( a, af, ipiv, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations *A*X = B* with a symmetric full-storage matrix *A*, with multiple right-hand sides. For each computed solution vector *x*, the routine computes the component-wise backward error β. This error is the smallest relative perturbation in elements of *A* and *b* such that *x* is the exact solution of the perturbed system:

$|\delta a_{ij}| / |a_{ij}| \le \beta |a_{ij}|$, $|\delta b_i| / |b_i| \le \beta |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the component-wise forward error in the computed solution $||x - x_e||_\infty / ||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?sytrf
- call the solver routine ?sytrs.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', the upper triangle of *A* is stored. If *uplo* = 'L', the lower triangle of *A* is stored.
n	INTEGER. The order of the matrix *A*; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
a,*af*,*b*,*x*,*work*	REAL for ssyrfs DOUBLE PRECISION for dsyrfs COMPLEX for csyrfs DOUBLE COMPLEX for zsyrfs. Arrays: *a*(*lda*,*) contains the original matrix *A*, as supplied to ?sytrf. *af*(*ldaf*,*) contains the factored matrix *A*, as returned by ?sytrf. *b*(*ldb*,*) contains the right-hand side matrix *B*. *x*(*ldx*,*) contains the solution matrix *X*. *work*(*) is a workspace array. The second dimension of *a* and *af* must be at least max(1, *n*); the second dimension of *b* and *x* must be at least max(1, *nrhs*); the dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).
ldaf	INTEGER. The first dimension of *af*; *ldaf* ≥ max(1, *n*).
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).
ldx	INTEGER. The first dimension of *x*; *ldx* ≥ max(1, *n*).
ipiv	INTEGER.

	Array, DIMENSION at least max(1, *n*). The *ipiv* array, as returned by ?sytrf.
iwork	INTEGER. Workspace array, DIMENSION at least max(1, *n*).
rwork	REAL for csyrfs DOUBLE PRECISION for zsyrfs. Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

x	The refined solution matrix *X*.
ferr, *berr*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1, *nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine syrfs interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
af	Holds the matrix *AF* of size (*n*, *n*).
ipiv	Holds the vector of length (*n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?herfs

Refines the solution of a system of linear equations with a complex Hermitian matrix and estimates its error.

Syntax

Fortran 77:

```
call cherfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
berr, work, rwork, info )
call zherfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr,
berr, work, rwork, info )
```

Fortran 95:

```
call herfs( a, af, ipiv, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $A*X = B$ with a complex Hermitian full-storage matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the component-wise backward error β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta|a_{ij}|$, $|\delta b_i|/|b_i| \le \beta|b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the component-wise forward error in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?hetrf

- call the solver routine ?hetrs.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', the upper triangle of *A* is stored. If *uplo* = 'L', the lower triangle of *A* is stored.
n	INTEGER. The order of the matrix *A*; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
a,*af*,*b*,*x*,*work*	COMPLEX for cherfs DOUBLE COMPLEX for zherfs. Arrays: *a*(*lda*,*) contains the original matrix *A*, as supplied to ?hetrf. *af*(*ldaf*,*) contains the factored matrix *A*, as returned by ?hetrf. *b*(*ldb*,*) contains the right-hand side matrix *B*. *x*(*ldx*,*) contains the solution matrix *X*. *work*(*) is a workspace array. The second dimension of *a* and *af* must be at least max(1, *n*); the second dimension of *b* and *x* must be at least max(1, *nrhs*); the dimension of *work* must be at least max(1, 2*n).
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).
ldaf	INTEGER. The first dimension of *af*; *ldaf* ≥ max(1, *n*).
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).
ldx	INTEGER. The first dimension of *x*; *ldx* ≥ max(1, *n*).
ipiv	INTEGER. Array, DIMENSION at least max(1, *n*). The *ipiv* array, as returned by ?hetrf.
rwork	REAL for cherfs DOUBLE PRECISION for zherfs. Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

x	The refined solution matrix *X*.
ferr, *berr*	REAL for cherfs

DOUBLE PRECISION for zherfs.
Arrays, DIMENSION at least max(1, *nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.

info — INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine herfs interface are as follows:

a — Holds the matrix *A* of size (*n*, *n*).

af — Holds the matrix *AF* of size (*n*, *n*).

ipiv — Holds the vector of length (*n*).

b — Holds the matrix *B* of size (*n*, *nrhs*).

x — Holds the matrix *X* of size (*n*, *nrhs*).

ferr — Holds the vector of length (*nrhs*).

berr — Holds the vector of length (*nrhs*).

uplo — Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $16n^2$ operations. In addition, each step of iterative refinement involves $24n^2$ operations; the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations *A***x* = *b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

The real counterpart of this routine is ?ssyrfs/?dsyrfs

?sprfs

Refines the solution of a system of linear equations with a packed symmetric matrix and estimates the solution error.

Syntax

Fortran 77:

```
call ssprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call dsprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call csprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call zsprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call sprfs( a, af, ipiv, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations `A*X = B` with a packed symmetric matrix `A`, with multiple right-hand sides. For each computed solution vector `x`, the routine computes the `component-wise backward error` β. This error is the smallest relative perturbation in elements of `A` and `b` such that `x` is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta|a_{ij}|$, $|\delta b_i|/|b_i| \le \beta|b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the `component-wise forward error` in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine `?sptrf`
- call the solver routine `?sptrs`.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', the upper triangle of *A* is stored. If *uplo* = 'L', the lower triangle of *A* is stored.
n	INTEGER. The order of the matrix *A*; *n* ≥ 0.
nrhs	INTEGER. The number of right-hand sides; *nrhs* ≥ 0.
ap,*afp*,*b*,*x*,*work*	REAL for ssprfs DOUBLE PRECISION for dsprfs COMPLEX for csprfs DOUBLE COMPLEX for zsprfs. Arrays: *ap*(*) contains the original packed matrix *A*, as supplied to ?sptrf. *afp*(*) contains the factored packed matrix *A*, as returned by ?sptrf. *b*(*ldb*,*) contains the right-hand side matrix *B*. *x*(*ldx*,*) contains the solution matrix *X*. *work*(*) is a workspace array. The dimension of arrays *ap* and *afp* must be at least max(1, *n*(*n*+1)/2); the second dimension of *b* and *x* must be at least max(1, *nrhs*); the dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).
ldx	INTEGER. The first dimension of *x*; *ldx* ≥ max(1, *n*).
ipiv	INTEGER. Array, DIMENSION at least max(1, *n*). The *ipiv* array, as returned by ?sptrf.
iwork	INTEGER. Workspace array, DIMENSION at least max(1, *n*).
rwork	REAL for csprfs DOUBLE PRECISION for zsprfs. Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

x	The refined solution matrix *X*.
ferr, *berr*	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors.

	Arrays, DIMENSION at least max(1, *nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine sprfs interface are as follows:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).
af	Stands for argument *afp* in Fortan 77 interface. Holds the array *AF* of size (*n**(*n*+1)/2).
ipiv	Holds the vector of length (*n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations *A***x* = *b*; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?hprfs

Refines the solution of a system of linear equations with a packed complex Hermitian matrix and estimates the solution error.

Syntax

Fortran 77:

```
call chprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call zhprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call hprfs( a, af, ipiv, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Description

This routine performs an iterative refinement of the solution to a system of linear equations $A*X = B$ with a packed complex Hermitian matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the component-wise backward error β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta|a_{ij}|$, $|\delta b_i|/|b_i| \le \beta|b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the component-wise forward error in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine ?hptrf
- call the solver routine ?hptrs.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', the upper triangle of A is stored. If *uplo* = 'L', the lower triangle of A is stored.

n	INTEGER. The order of the matrix *A*; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
ap,*afp*,*b*,*x*,*work*	COMPLEX for chprfs DOUBLE COMPLEX for zhprfs. Arrays: *ap*(*) contains the original packed matrix *A*, as supplied to ?hptrf. *afp*(*) contains the factored packed matrix *A*, as returned by ?hptrf. *b*(*ldb*,*) contains the right-hand side matrix *B*. *x*(*ldx*,*) contains the solution matrix *X*. *work*(*) is a workspace array. The dimension of arrays *ap* and *afp* must be at least max(1,*n*(*n*+1)/2); the second dimension of *b* and *x* must be at least max(1,*nrhs*); the dimension of *work* must be at least max(1, 2*n).
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).
ldx	INTEGER. The first dimension of *x*; *ldx* ≥ max(1, *n*).
ipiv	INTEGER. Array, DIMENSION at least max(1, *n*). The *ipiv* array, as returned by ?hptrf.
rwork	REAL for chprfs DOUBLE PRECISION for zhprfs. Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

x	The refined solution matrix *X*.
ferr, *berr*	REAL for chprfs. DOUBLE PRECISION for zhprfs. Arrays, DIMENSION at least max(1,*nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `hprfs` interface are as follows:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).
af	Stands for argument *afp* in Fortan 77 interface. Holds the array *AF* of size (*n**(*n*+1)/2).
ipiv	Holds the vector of length (*n*).
b	Holds the matrix *B* of size (*n*,*nrhs*).
x	Holds the matrix *X* of size (*n*,*nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $16n^2$ operations. In addition, each step of iterative refinement involves $24n^2$ operations; the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations *A***x* = *b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

The real counterpart of this routine is `?ssprfs`/`?dsprfs`.

?trrfs

Estimates the error in the solution of a system of linear equations with a triangular matrix.

Syntax

Fortran 77:

```
call strrfs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr, berr,
work, iwork, info )
call dtrrfs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr, berr,
work, iwork, info )
call ctrrfs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr, berr,
work, rwork, info )
call ztrrfs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr, berr,
work, rwork, info )
```

Fortran 95:

```
call trrfs( a, b, x [,uplo] [,trans] [,diag] [,ferr] [,berr] [,info] )
```

Description

This routine estimates the errors in the solution to a system of linear equations $A*X = B$ or $A^T*X = B$ or $A^H*X = B$ with a triangular matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the component-wise backward error β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta|a_{ij}|$, $|\delta b_i|/|b_i| \le \beta|b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

The routine also estimates the component-wise forward error in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine, call the solver routine ?trtrs.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular:

	If *uplo* = 'U', then *A* is upper triangular. If *uplo* = 'L', then *A* is lower triangular.
trans	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If *trans* = 'N', the system has the form *A*X = B*. If *trans* = 'T', the system has the form $A^T*X = B$. If *trans* = 'C', the system has the form $A^H*X = B$.
diag	CHARACTER*1. Must be 'N' or 'U'. If *diag* = 'N', then *A* is not a unit triangular matrix. If *diag* = 'U', then *A* is unit triangular: diagonal elements of *A* are assumed to be 1 and not referenced in the array *a*.
n	INTEGER. The order of the matrix *A*; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
a, *b*, *x*, *work*	REAL for strrfs DOUBLE PRECISION for dtrrfs COMPLEX for ctrrfs DOUBLE COMPLEX for ztrrfs. Arrays: *a*(*lda*,*) contains the upper or lower triangular matrix *A*, as specified by *uplo*. *b*(*ldb*,*) contains the right-hand side matrix *B*. *x*(*ldx*,*) contains the solution matrix *X*. *work*(*) is a workspace array. The second dimension of *a* must be at least max(1,*n*); the second dimension of *b* and *x* must be at least max(1,*nrhs*); the dimension of *work* must be at least max(1,3*n) for real flavors and max(1,2*n) for complex flavors.
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).
ldx	INTEGER. The first dimension of *x*; *ldx* ≥ max(1, *n*).
iwork	INTEGER. Workspace array, DIMENSION at least max(1, *n*).
rwork	REAL for ctrrfs DOUBLE PRECISION for ztrrfs. Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

ferr, *berr*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1, *nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine trrfs interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
diag	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations *A***x* = *b*; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors or $4n^2$ for complex flavors.

?tprfs

Estimates the error in the solution of a system of linear equations with a packed triangular matrix.

Syntax

Fortran 77:

```
call stprfs( uplo, trans, diag, n, nrhs, ap, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call dtprfs( uplo, trans, diag, n, nrhs, ap, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call ctprfs( uplo, trans, diag, n, nrhs, ap, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call ztprfs( uplo, trans, diag, n, nrhs, ap, b, ldb, x, ldx, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call tprfs( a, b, x [,uplo] [,trans] [,diag] [,ferr] [,berr] [,info] )
```

Description

This routine estimates the errors in the solution to a system of linear equations $A*X = B$ or $A^T*X = B$ or $A^H*X = B$ with a packed triangular matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the component-wise backward error β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta|a_{ij}|$, $|\delta b_i|/|b_i| \le \beta|b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

The routine also estimates the component-wise forward error in the computed solution $||x - x_e||_\infty / ||x||_\infty$ (here x_e is the exact solution).

Before calling this routine, call the solver routine ?tptrs.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular:

	If *uplo* = 'U', then *A* is upper triangular. If *uplo* = 'L', then *A* is lower triangular.
trans	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If *trans* = 'N', the system has the form *A*X* = *B*. If *trans* = 'T', the system has the form $A^T*X = B$. If *trans* = 'C', the system has the form $A^H*X = B$.
diag	CHARACTER*1. Must be 'N' or 'U'. If *diag* = 'N', *A* is not a unit triangular matrix. If *diag* = 'U', *A* is unit triangular: diagonal elements of *A* are assumed to be 1 and not referenced in the array *ap*.
n	INTEGER. The order of the matrix *A*; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
ap, *b*, *x*, *work*	REAL for stprfs DOUBLE PRECISION for dtprfs COMPLEX for ctprfs DOUBLE COMPLEX for ztprfs. Arrays: *ap*(*) contains the upper or lower triangular matrix *A*, as specified by *uplo*. *b*(*ldb*,*) contains the right-hand side matrix *B*. *x*(*ldx*,*) contains the solution matrix *X*. *work*(*) is a workspace array. The dimension of *ap* must be at least max(1,*n*(*n*+1)/2); the second dimension of *b* and *x* must be at least max(1,*nrhs*); the dimension of *work* must be at least max(1,3**n*) for real flavors and max(1,2**n*) for complex flavors.
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).
ldx	INTEGER. The first dimension of *x*; *ldx* ≥ max(1, *n*).
iwork	INTEGER. Workspace array, DIMENSION at least max(1, *n*).
rwork	REAL for ctprfs DOUBLE PRECISION for ztprfs. Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

ferr, *berr*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1, *nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine tprfs interface are as follows:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).
b	Holds the matrix *B* of size (*n*,*nrhs*).
x	Holds the matrix *X* of size (*n*,*nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
diag	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations *A***x* = *b*; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors or $4n^2$ for complex flavors.

?tbrfs

Estimates the error in the solution of a system of linear equations with a triangular band matrix.

Syntax

Fortran 77:

```
call stbrfs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call dtbrfs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call ctbrfs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call ztbrfs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call tbrfs( a, b, x [,uplo] [,trans] [,diag] [,ferr] [,berr] [,info] )
```

Description

This routine estimates the errors in the solution to a system of linear equations $A*X = B$ or $A^T*X = B$ or $A^H*X = B$ with a triangular band matrix A, with multiple right-hand sides. For each computed solution vector x, the routine computes the component-wise backward error β. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}|/|a_{ij}| \le \beta|a_{ij}|$, $|\delta b_i|/|b_i| \le \beta|b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

The routine also estimates the component-wise forward error in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here x_e is the exact solution).

Before calling this routine, call the solver routine ?tbtrs.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular:

If *uplo* = 'U', then *A* is upper triangular.
If *uplo* = 'L', then *A* is lower triangular.

trans — CHARACTER*1. Must be 'N' or 'T' or 'C'.
Indicates the form of the equations:
If *trans* = 'N', the system has the form *A*X* = *B*.
If *trans* = 'T', the system has the form $A^T*X = B$.
If *trans* = 'C', the system has the form $A^H*X = B$.

diag — CHARACTER*1. Must be 'N' or 'U'.
If *diag* = 'N', *A* is not a unit triangular matrix.
If *diag* = 'U', *A* is unit triangular: diagonal elements of *A* are assumed to be 1 and not referenced in the array *ab*.

n — INTEGER. The order of the matrix *A*; $n \geq 0$.

kd — INTEGER. The number of super-diagonals or sub-diagonals in the matrix *A*; $kd \geq 0$.

nrhs — INTEGER. The number of right-hand sides; $nrhs \geq 0$.

ab, *b*, *x*, *work* — REAL for stbrfs
DOUBLE PRECISION for dtbrfs
COMPLEX for ctbrfs
DOUBLE COMPLEX for ztbrfs.
Arrays:
ab(*ldab*,*) contains the upper or lower triangular matrix *A*, as specified by *uplo*, in band storage format.
b(*ldb*,*) contains the right-hand side matrix *B*.
x(*ldx*,*) contains the solution matrix *X*.
work(*) is a workspace array.
The second dimension of *a* must be at least max(1,*n*); the second dimension of *b* and *x* must be at least max(1,*nrhs*). The dimension of *work* must be at least max(1,3**n*) for real flavors and max(1,2**n*) for complex flavors.

ldab — INTEGER. The first dimension of the array *ab*; $ldab \geq kd + 1$.

ldb — INTEGER. The first dimension of *b*; $ldb \geq \max(1, n)$.

ldx — INTEGER. The first dimension of *x*; $ldx \geq \max(1, n)$.

iwork — INTEGER. Workspace array, DIMENSION at least max(1, *n*).

rwork — REAL for ctbrfs

DOUBLE PRECISION for ztbrfs.
Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

ferr, *berr*	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1, *nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine tbrfs interface are as follows:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*kd*+1, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
diag	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations *A***x* = *b*; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately 2*n***kd* floating-point operations for real flavors or 8*n***kd* operations for complex flavors.

Routines for Matrix Inversion

It is seldom necessary to compute an explicit inverse of a matrix. In particular, do not attempt to solve a system of equations $Ax = b$ by first computing A^{-1} and then forming the matrix-vector product $x = A^{-1}b$. Call a solver routine instead (see Routines for Solving Systems of Linear Equations); this is more efficient and more accurate.

However, matrix inversion routines are provided for the rare occasions when an explicit inverse matrix is needed.

?getri

Computes the inverse of an LU-factored general matrix.

Syntax

Fortran 77:

```
call sgetri( n, a, lda, ipiv, work, lwork, info )
call dgetri( n, a, lda, ipiv, work, lwork, info )
call cgetri( n, a, lda, ipiv, work, lwork, info )
call zgetri( n, a, lda, ipiv, work, lwork, info )
```

Fortran 95:

```
call getri( a, ipiv [,info] )
```

Description

This routine computes the inverse `inv(A)` of a general matrix *A*. Before calling this routine, call `?getrf` to factorize *A*.

Input Parameters

n	`INTEGER`. The order of the matrix *A*; $n \geq 0$.

a, *work*	`REAL` for `sgetri`
	`DOUBLE PRECISION` for `dgetri`
	`COMPLEX` for `cgetri`
	`DOUBLE COMPLEX` for `zgetri`.

	Arrays: *a*(*lda*,*), *work*(*). *a*(*lda*,*) contains the factorization of the matrix *A*, as returned by ?getrf: *A* = *P***L***U*. The second dimension of *a* must be at least max(1,*n*). *work*(*) is a workspace array of dimension at least max(1,*lwork*).
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).
ipiv	INTEGER. Array, DIMENSION at least max(1, *n*). The *ipiv* array, as returned by ?getrf.
lwork	INTEGER. The size of the *work* array; *lwork* ≥ *n*. If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See Application Notes below for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the *n*-by-*n* matrix inv(*A*).
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the *i*-th diagonal element of the factor *U* is zero, *U* is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine getri interface are as follows:

a	Holds the matrix *A* of size (*n*,*n*).
ipiv	Holds the vector of length (*n*).

Application Notes

For better performance, try using *lwork* = *n*blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed inverse *X* satisfies the following error bound:

$$|XA - I| \leq c(n)\varepsilon|X|P|L||U|,$$

where $c(n)$ is a modest linear function of n; ε is the machine precision; I denotes the identity matrix; P, L, and U are the factors of the matrix factorization $A = P*L*U$.

The total number of floating-point operations is approximately $(4/3)n^3$ for real flavors and $(16/3)n^3$ for complex flavors.

?potri

Computes the inverse of a symmetric (Hermitian) positive-definite matrix.

Syntax

Fortran 77:

```
call spotri( uplo, n, a, lda, info )
call dpotri( uplo, n, a, lda, info )
call cpotri( uplo, n, a, lda, info )
call zpotri( uplo, n, a, lda, info )
```

Fortran 95:

```
call potri( a [,uplo] [,info] )
```

Description

This routine computes the inverse `inv(A)` of a symmetric positive definite or, for complex flavors, Hermitian positive-definite matrix `A`. Before calling this routine, call `?potrf` to factorize `A`.

Input Parameters

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Indicates whether `A` is upper or lower triangular: If `uplo = 'U'`, then `A` is upper triangular. If `uplo = 'L'`, then `A` is lower triangular.
n	`INTEGER`. The order of the matrix `A`; $n \geq 0$.

a	`REAL` for `spotri`
	`DOUBLE PRECISION` for `dpotri`
	`COMPLEX` for `cpotri`
	`DOUBLE COMPLEX` for `zpotri`.
	Array `a(lda,*)`. Contains the factorization of the matrix `A`, as returned by `?potrf`.
	The second dimension of `a` must be at least `max(1, n)`.

lda	`INTEGER`. The first dimension of `a`; $lda \geq \max(1, n)$.

Output Parameters

a	Overwritten by the n-by-n matrix `inv(A)`.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the *i*-th diagonal element of the Cholesky factor (and therefore the factor itself) is zero, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `potri` interface are as follows:

a	Holds the matrix A of size (n, n).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

The computed inverse X satisfies the following error bounds:

$$||XA - I||_2 \le c(n)\varepsilon\kappa_2(A), \quad ||AX - I||_2 \le c(n)\varepsilon\kappa_2(A),$$

where $c(n)$ is a modest linear function of n, and ε is the machine precision; I denotes the identity matrix.

The 2-norm $||A||_2$ of a matrix A is defined by $||A||_2 = max_{x \cdot x=1}(Ax \cdot Ax)^{1/2}$, and the condition number $\kappa_2(A)$ is defined by $\kappa_2(A) = ||A||_2 \, ||A^{-1}||_2$.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

?pptri

Computes the inverse of a packed symmetric (Hermitian) positive-definite matrix

Syntax

Fortran 77:

```
call spptri( uplo, n, ap, info )
call dpptri( uplo, n, ap, info )
call cpptri( uplo, n, ap, info )
call zpptri( uplo, n, ap, info )
```

Fortran 95:

```
call pptri( a [,uplo] [,info] )
```

Description

This routine computes the inverse `inv(A)` of a symmetric positive definite or, for complex flavors, Hermitian positive-definite matrix `A` in `packed` form. Before calling this routine, call `?pptrf` to factorize `A`.

Input Parameters

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Indicates whether the upper or lower triangular factor is stored in *ap*: If *uplo* = `'U'`, then the upper triangular factor is stored. If *uplo* = `'L'`, then the lower triangular factor is stored.
n	`INTEGER`. The order of the matrix *A*; *n* ≥ 0.

ap	`REAL` for `spptri`
	`DOUBLE PRECISION` for `dpptri`
	`COMPLEX` for `cpptri`
	`DOUBLE COMPLEX` for `zpptri`.
	Array, `DIMENSION` at least max(1, *n*(*n*+1)/2).
	Contains the factorization of the packed matrix *A*, as returned by `?pptrf`.
	The dimension *ap* must be at least max(1,*n*(*n*+1)/2).

Output Parameters

ap	Overwritten by the packed *n*-by-*n* matrix `inv(A)`.
info	`INTEGER`. If *info* = `0`, the execution is successful. If *info* = `-i`, the *i*-th parameter had an illegal value. If *info* = *i*, the *i*-th diagonal element of the Cholesky factor (and therefore the factor itself) is zero, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `pptri` interface are as follows:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size `(n*(n+1)/2)`.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$||XA - I||_2 \le c(n)\varepsilon\kappa_2(A), \quad ||AX - I||_2 \le c(n)\varepsilon\kappa_2(A),$$

where $c(n)$ is a modest linear function of n, and ε is the machine precision; I denotes the identity matrix.

The 2-norm $||A||_2$ of a matrix A is defined by $||A||_2 = max_{x \cdot x=1}(Ax \cdot Ax)^{1/2}$, and the condition number $\kappa_2(A)$ is defined by $\kappa_2(A) = ||A||_2 \, ||A^{-1}||_2$.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

?sytri

Computes the inverse of a symmetric matrix.

Syntax

Fortran 77:

```
call ssytri( uplo, n, a, lda, ipiv, work, info )
call dsytri( uplo, n, a, lda, ipiv, work, info )
call csytri( uplo, n, a, lda, ipiv, work, info )
call zsytri( uplo, n, a, lda, ipiv, work, info )
```

Fortran 95:

```
call sytri( a, ipiv [,uplo] [,info] )
```

Description

This routine computes the inverse `inv(A)` of a symmetric matrix *A*. Before calling this routine, call `?sytrf` to factorize *A*.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates how the input matrix *A* has been factored:
If *uplo* = `'U'`, the array *a* stores the Bunch-Kaufman factorization $A = P*U*D*U^T*P^T$.
If *uplo* = `'L'`, the array *a* stores the Bunch-Kaufman factorization $A = P*L*D*L^T*P^T$.

n — `INTEGER`. The order of the matrix *A*; $n \geq 0$.

a, *work* — `REAL` for `ssytri`
`DOUBLE PRECISION` for `dsytri`
`COMPLEX` for `csytri`
`DOUBLE COMPLEX` for `zsytri`.
Arrays:
a`(`*lda*`,*)` contains the factorization of the matrix *A*, as returned by `?sytrf`.
The second dimension of *a* must be at least `max(1,`*n*`)`.
work`(*)` is a workspace array.

	The dimension of *work* must be at least max(1,2*n).
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).
ipiv	INTEGER. Array, DIMENSION at least max(1, *n*). The *ipiv* array, as returned by ?sytrf.

Output Parameters

a	Overwritten by the *n*-by-*n* matrix inv(*A*).
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the *i*-th diagonal element of *D* is zero, *D* is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine sytri interface are as follows:

a	Holds the matrix *A* of size (*n*,*n*).
ipiv	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|D*U^T*P^T*X*P*U - I| \le c(n)\varepsilon(|D||U^T|P^T|X|P|U| + |D||D^{-1}|)$$

for *uplo* = 'U', and

$$|D*L^T*P^T*X*P*L - I| \le c(n)\varepsilon(|D||L^T|P^T|X|P|L| + |D||D^{-1}|)$$

for *uplo* = 'L'. Here $c(n)$ is a modest linear function of *n*, and ε is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

?hetri

Computes the inverse of a complex Hermitian matrix.

Syntax

Fortran 77:

```
call chetri( uplo, n, a, lda, ipiv, work, info )
call zhetri( uplo, n, a, lda, ipiv, work, info )
```

Fortran 95:

```
call hetri( a, ipiv [,uplo] [,info] )
```

Description

This routine computes the inverse `inv(A)` of a complex Hermitian matrix `A`. Before calling this routine, call `?hetrf` to factorize `A`.

Input Parameters

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Indicates how the input matrix `A` has been factored: If `uplo = 'U'`, the array `a` stores the Bunch-Kaufman factorization $A = P*U*D*U^H*P^T$. If `uplo = 'L'`, the array `a` stores the Bunch-Kaufman factorization $A = P*L*D*L^H*P^T$.
n	`INTEGER`. The order of the matrix `A`; $n \geq 0$.
a, *work*	`COMPLEX` for `chetri` `DOUBLE COMPLEX` for `zhetri`. Arrays: `a(lda,*)` contains the factorization of the matrix `A`, as returned by `?hetrf`. The second dimension of `a` must be at least `max(1,n)`. `work(*)` is a workspace array. The dimension of `work` must be at least `max(1,n)`.
lda	`INTEGER`. The first dimension of `a`; $lda \geq \max(1, n)$.
ipiv	`INTEGER`.

Array, DIMENSION at least max(1, *n*). The *ipiv* array, as returned by ?hetrf.

Output Parameters

a	Overwritten by the *n*-by-*n* matrix inv(*A*).
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the *i*-th diagonal element of *D* is zero, *D* is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hetri interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
ipiv	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|D*U^H*P^T*X*P*U - I| \le c(n)\varepsilon(|D||U^H|P^T|X|P|U| + |D||D^{-1}|)$$

for *uplo* = 'U', and

$$|D*L^H*P^T*X*P*L - I| \le c(n)\varepsilon(|D||L^H|P^T|X|P|L| + |D||D^{-1}|)$$

for *uplo* = 'L'. Here $c(n)$ is a modest linear function of *n*, and ε is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

The real counterpart of this routine is ?sytri.

?sptri

Computes the inverse of a symmetric matrix using packed storage.

Syntax

Fortran 77:

```
call ssptri( uplo, n, ap, ipiv, work, info )
call dsptri( uplo, n, ap, ipiv, work, info )
call csptri( uplo, n, ap, ipiv, work, info )
call zsptri( uplo, n, ap, ipiv, work, info )
```

Fortran 95:

```
call sptri( a, ipiv [,uplo] [,info] )
```

Description

This routine computes the inverse inv(*A*) of a packed symmetric matrix *A*. Before calling this routine, call ?sptrf to factorize *A*.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix *A* has been factored: If *uplo* = 'U', the array *ap* stores the Bunch-Kaufman factorization $A = P*U*D*U^T*P^T$. If *uplo* = 'L', the array *ap* stores the Bunch-Kaufman factorization $A = P*L*D*L^T*P^T$.
n	INTEGER. The order of the matrix *A*; $n \geq 0$.
ap, *work*	REAL for ssptri DOUBLE PRECISION for dsptri COMPLEX for csptri DOUBLE COMPLEX for zsptri. Arrays: *ap*(*) contains the factorization of the matrix *A*, as returned by ?sptrf. The dimension of *ap* must be at least max(1,*n*(*n*+1)/2). *work*(*) is a workspace array.

The dimension of *work* must be at least max(1,*n*).

ipiv INTEGER.
Array, DIMENSION at least max(1, *n*). The *ipiv* array, as returned by ?sptrf.

Output Parameters

ap Overwritten by the *n*-by-*n* matrix inv(*A*) in packed form.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, the *i*-th diagonal element of *D* is zero, *D* is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine sptri interface are as follows:

a Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).

ipiv Holds the vector of length (*n*).

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|D*U^T*P^T*X*P*U - I| \le c(n)\varepsilon(|D||U^T|P^T|X|P|U| + |D||D^{-1}|)$$

for *uplo* = 'U', and

$$|D*L^T*P^T*X*P*L - I| \le c(n)\varepsilon(|D||L^T|P^T|X|P|L| + |D||D^{-1}|)$$

for *uplo* = 'L'. Here $c(n)$ is a modest linear function of n, and ε is the machine precision; I denotes the identity matrix.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

?hptri

Computes the inverse of a complex Hermitian matrix using packed storage.

Syntax

Fortran 77:

```
call chptri( uplo, n, ap, ipiv, work, info )
call zhptri( uplo, n, ap, ipiv, work, info )
```

Fortran 95:

```
call hptri( a, ipiv [,uplo] [,info] )
```

Description

This routine computes the inverse `inv(A)` of a complex Hermitian matrix `A` using packed storage. Before calling this routine, call `?hptrf` to factorize `A`.

Input Parameters

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Indicates how the input matrix `A` has been factored: If `uplo = 'U'`, the array `ap` stores the packed Bunch-Kaufman factorization $A = P*U*D*U^H*P^T$. If `uplo = 'L'`, the array `ap` stores the packed Bunch-Kaufman factorization $A = P*L*D*L^H*P^T$.
n	`INTEGER`. The order of the matrix `A`; $n \geq 0$.
ap, *work*	`COMPLEX` for `chptri` `DOUBLE COMPLEX` for `zhptri`. Arrays: `ap(*)` contains the factorization of the matrix `A`, as returned by `?hptrf`. The dimension of `ap` must be at least `max(1,n(n+1)/2)`. `work(*)` is a workspace array. The dimension of `work` must be at least `max(1,n)`.
ipiv	`INTEGER`. Array, `DIMENSION` at least `max(1, n)`. The `ipiv` array, as returned by `?hptrf`.

Output Parameters

ap	Overwritten by the *n*-by-*n* matrix `inv(A)`.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the *i*-th diagonal element of *D* is zero, *D* is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `hptri` interface are as follows:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).
ipiv	Holds the vector of length (*n*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|D*U^H*P^T*X*P*U - I| \le c(n)\varepsilon(|D||U^H|P^T|X|P|U| + |D||D^{-1}|)$$

for *uplo* = `'U'`, and

$$|D*L^H*P^T*X*PL - I| \le c(n)\varepsilon(|D||L^H|P^T|X|P|L| + |D||D^{-1}|)$$

for *uplo* = `'L'`. Here $c(n)$ is a modest linear function of *n*, and ε is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

The real counterpart of this routine is `?sptri`.

?trtri

Computes the inverse of a triangular matrix.

Syntax

Fortran 77:

```
call strtri( uplo, diag, n, a, lda, info )
call dtrtri( uplo, diag, n, a, lda, info )
call ctrtri( uplo, diag, n, a, lda, info )
call ztrtri( uplo, diag, n, a, lda, info )
```

Fortran 95:

```
call trtri( a [,uplo] [,diag] [,info] )
```

Description

This routine computes the inverse `inv(A)` of a triangular matrix *A*.

Input Parameters

uplo — `CHARACTER*1`. Must be `'U'` or `'L'`.
Indicates whether *A* is upper or lower triangular:
If *uplo* = `'U'`, then *A* is upper triangular.
If *uplo* = `'L'`, then *A* is lower triangular.

diag — `CHARACTER*1`. Must be `'N'` or `'U'`.
If *diag* = `'N'`, then *A* is not a unit triangular matrix.
If *diag* = `'U'`, *A* is unit triangular: diagonal elements of *A* are assumed to be 1 and not referenced in the array *a*.

n — `INTEGER`. The order of the matrix *A*; $n \geq 0$.

a — `REAL` for `strtri`
`DOUBLE PRECISION` for `dtrtri`
`COMPLEX` for `ctrtri`
`DOUBLE COMPLEX` for `ztrtri`.
Array: `DIMENSION (,*)`.
Contains the matrix *A*.
The second dimension of *a* must be at least `max(1,n)`.

lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).

Output Parameters

a	Overwritten by the *n*-by-*n* matrix inv(*A*).
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the *i*-th diagonal element of *A* is zero, *A* is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine trtri interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
diag	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|XA - I| \le c(n)\varepsilon\ |X||A|$$

$$|XA - I| \le c(n)\varepsilon\ |A^{-1}||A||X|,$$

where $c(n)$ is a modest linear function of n; ε is the machine precision; I denotes the identity matrix.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors and $(4/3)n^3$ for complex flavors.

?tptri

Computes the inverse of a triangular matrix using packed storage.

Syntax

Fortran 77:

```
call stptri( uplo, diag, n, ap, info )
call dtptri( uplo, diag, n, ap, info )
call ctptri( uplo, diag, n, ap, info )
call ztptri( uplo, diag, n, ap, info )
```

Fortran 95:

```
call tptri( a [,uplo] [,diag] [,info] )
```

Description

This routine computes the inverse `inv(A)` of a packed triangular matrix `A`.

Input Parameters

uplo	`CHARACTER*1`. Must be `'U'` or `'L'`. Indicates whether `A` is upper or lower triangular: If *uplo* = `'U'`, then `A` is upper triangular. If *uplo* = `'L'`, then `A` is lower triangular.
diag	`CHARACTER*1`. Must be `'N'` or `'U'`. If *diag* = `'N'`, then `A` is not a unit triangular matrix. If *diag* = `'U'`, `A` is unit triangular: diagonal elements of `A` are assumed to be 1 and not referenced in the array *ap*.
n	`INTEGER`. The order of the matrix `A`; $n \geq 0$.

ap	`REAL` for `stptri`
	`DOUBLE PRECISION` for `dtptri`
	`COMPLEX` for `ctptri`
	`DOUBLE COMPLEX` for `ztptri`.
	Array, `DIMENSION` at least `max(1,n(n+1)/2)`.
	Contains the packed triangular matrix A.

Output Parameters

ap	Overwritten by the packed *n*-by-*n* matrix `inv(A)` .
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the *i*-th diagonal element of *A* is zero, *A* is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `tptri` interface are as follows:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
diag	Must be `'N'` or `'U'`. The default value is `'N'`.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|XA - I| \leq c(n)\varepsilon\ |X||A|$$

$$|X - A^{-1}| \leq c(n)\varepsilon\ |A^{-1}||A||X|,$$

where $c(n)$ is a modest linear function of n; ε is the machine precision; I denotes the identity matrix.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors and $(4/3)n^3$ for complex flavors.

Routines for Matrix Equilibration

Routines described in this section are used to compute scaling factors needed to equilibrate a matrix. Note that these routines do not actually scale the matrices.

?geequ

Computes row and column scaling factors intended to equilibrate a matrix and reduce its condition number.

Syntax

Fortran 77:

```
call sgeequ( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call dgeequ( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call cgeequ( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call zgeequ( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
```

Fortran 95:

```
call geequ( a, r, c [,rowcnd] [,colcnd] [,amax] [,info] )
```

Description

This routine computes row and column scalings intended to equilibrate an m-by-n matrix A and reduce its condition number. The output array r returns the row scale factors and the array c the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix B with elements $b_{ij}=r(i)*a_{ij}*c(j)$ have absolute value 1.

See ?laqge auxiliary function that uses scaling factors computed by ?geequ.

Input Parameters

m	INTEGER. The number of rows of the matrix A; $m \geq 0$.
n	INTEGER. The number of columns of the matrix A; $n \geq 0$.
a	REAL for sgeequ DOUBLE PRECISION for dgeequ COMPLEX for cgeequ DOUBLE COMPLEX for zgeequ. Array: DIMENSION (*lda*,*). Contains the m-by-n matrix A whose equilibration factors are to be computed. The second dimension of *a* must be at least max(1,*n*).

lda INTEGER. The leading dimension of *a*; *lda* ≥ max(1, *m*).

Output Parameters

r, *c* REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Arrays: *r*(*m*), *c*(*n*).
If *info* = 0, or *info* > *m*, the array *r* contains the row scale factors of the matrix *A*.
If *info* = 0 , the array *c* contains the column scale factors of the matrix *A*.

rowcnd REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
If *info* = 0 or *info* > *m*, *rowcnd* contains the ratio of the smallest *r*(i) to the largest *r*(i).

colcnd REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
If *info* = 0, *colcnd* contains the ratio of the smallest *c*(i) to the largest *c*(i).

amax REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Absolute value of the largest element of the matrix *A*.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i* and

i ≤ *m*, the *i*-th row of *A* is exactly zero;
i > *m*, the (*i*-*m*)th column of *A* is exactly zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine geequ interface are as follows:

a Holds the matrix *A* of size (*m*, *n*).

r Holds the vector of length (*m*).

c	Holds the vector of length (*n*).

Application Notes

All the components of *r* and *c* are restricted to be between SMLNUM = smallest safe number and BIGNUM= largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of *A* but works well in practice.

If *rowcnd* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *r*.

If *colcnd* ≥ 0.1, it is not worth scaling by *c*.

If *amax* is very close to overflow or very close to underflow, the matrix *A* should be scaled.

?gbequ

Computes row and column scaling factors intended to equilibrate a band matrix and reduce its condition number.

Syntax

Fortran 77:

```
call sgbequ( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
call dgbequ( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
call cgbequ( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
call zgbequ( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
```

Fortran 95:

```
call gbequ( a, r, c [,kl] [,rowcnd] [,colcnd] [,amax] [,info] )
```

Description

This routine computes row and column scalings intended to equilibrate an *m*-by-*n* band matrix *A* and reduce its condition number. The output array *r* returns the row scale factors and the array *c* the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix *B* with elements b_{ij}=r(i)*a_{ij}*c(j) have absolute value 1.

See ?laqgb auxiliary function that uses scaling factors computed by ?gbequ.

Input Parameters

m	INTEGER. The number of rows of the matrix *A*; *m* ≥ 0.
n	INTEGER. The number of columns of the matrix *A*; *n* ≥ 0.
kl	INTEGER. The number of subdiagonals within the band of *A*; *kl* ≥ 0.
ku	INTEGER. The number of superdiagonals within the band of *A*; *ku* ≥ 0.
ab	REAL for sgbequ DOUBLE PRECISION for dgbequ COMPLEX for cgbequ DOUBLE COMPLEX for zgbequ. Array, DIMENSION (*ldab*,*). Contains the original band matrix *A* stored in rows from 1 to *kl* + *ku* + 1. The second dimension of *ab* must be at least max(1,*n*);
ldab	INTEGER. The leading dimension of *ab*; *ldab* ≥ *kl*+*ku*+1.

Output Parameters

r, *c*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays: *r*(*m*), *c*(*n*). If *info* = 0, or *info* > *m*, the array *r* contains the row scale factors of the matrix *A*. If *info* = 0 , the array *c* contains the column scale factors of the matrix *A*.
rowcnd	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If *info* = 0 or *info* > *m*, *rowcnd* contains the ratio of the smallest *r*(i) to the largest *r*(i).
colcnd	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If *info* = 0, *colcnd* contains the ratio of the smallest *c*(i) to the largest *c*(i).
amax	REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

	Absolute value of the largest element of the matrix *A*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i* and *i* ≤ *m*, the *i*-th row of *A* is exactly zero; *i* > *m*, the (*i*-*m*)th column of *A* is exactly zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gbequ interface are as follows:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*kl*+*ku*+1, *n*).
r	Holds the vector of length (*m*).
c	Holds the vector of length (*n*).
kl	If omitted, assumed *kl* = *ku*.
ku	Restored as *ku* = *lda*-*kl*-1.

Application Notes

All the components of *r* and *c* are restricted to be between SMLNUM = smallest safe number and BIGNUM= largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of *A* but works well in practice.

If *rowcnd* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *r*.

If *colcnd* ≥ 0.1, it is not worth scaling by *c*.

If *amax* is very close to overflow or very close to underflow, the matrix *A* should be scaled.

?poequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix and reduce its condition number.

Syntax

Fortran 77:

```
call spoequ( n, a, lda, s, scond, amax, info )
call dpoequ( n, a, lda, s, scond, amax, info )
call cpoequ( n, a, lda, s, scond, amax, info )
call zpoequ( n, a, lda, s, scond, amax, info )
```

Fortran 95:

```
call poequ( a, s [,scond] [,amax] [,info] )
```

Description

This routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive-definite matrix A and reduce its condition number (with respect to the two-norm). The output array s returns scale factors computed as

$$s(i) = \frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled matrix B with elements b_{ij}=s(i)*a_{ij}*s(j) has diagonal elements equal to 1.

This choice of s puts the condition number of B within a factor n of the smallest possible condition number over all possible diagonal scalings.

See ?laqsy auxiliary function that uses scaling factors computed by ?poequ.

Input Parameters

n	INTEGER. The order of the matrix A; $n \geq 0$.

a	REAL for spoequ DOUBLE PRECISION for dpoequ COMPLEX for cpoequ DOUBLE COMPLEX for zpoequ. Array: DIMENSION (*lda*,*). Contains the *n*-by-*n* symmetric or Hermitian positive definite matrix *A* whose scaling factors are to be computed. Only diagonal elements of *A* are referenced. The second dimension of *a* must be at least max(1,*n*).
lda	INTEGER. The leading dimension of *a*; *lda* ≥ max(1, *m*).

Output Parameters

s	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION (*n*). If *info* = 0, the array *s* contains the scale factors for *A*.
scond	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If *info* = 0, *scond* contains the ratio of the smallest *s*(i) to the largest *s*(i).
amax	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Absolute value of the largest element of the matrix *A*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the *i*-th diagonal element of *A* is nonpositive.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine poequ interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
s	Holds the vector of length (*n*).

Application Notes

If *scond* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *s*.

If *amax* is very close to overflow or very close to underflow, the matrix *A* should be scaled.

?ppequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix in packed storage and reduce its condition number.

Syntax

Fortran 77:

```
call sppequ( uplo, n, ap, s, scond, amax, info )
call dppequ( uplo, n, ap, s, scond, amax, info )
call cppequ( uplo, n, ap, s, scond, amax, info )
call zppequ( uplo, n, ap, s, scond, amax, info )
```

Fortran 95:

```
call ppequ( a, s [,scond] [,amax] [,uplo] [,info] )
```

Description

This routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive definite matrix *A* in packed storage and reduce its condition number (with respect to the two-norm). The output array *s* returns scale factors computed as

$$s(i) = \frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled matrix *B* with elements b_{ij}=s(i)*a_{ij}*s(j) has diagonal elements equal to 1.

This choice of *s* puts the condition number of *B* within a factor *n* of the smallest possible condition number over all possible diagonal scalings.

See ?laqsp auxiliary function that uses scaling factors computed by ?ppequ.

Input Parameters

uplo — CHARACTER*1. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of *A* is packed in the array *ap*:
If *uplo* = 'U', the array *ap* stores the upper triangular part of the matrix *A*.
If *uplo* = 'L', the array *ap* stores the lower triangular part of the matrix *A*.

n — INTEGER. The order of matrix *A*; *n* ≥ 0.

ap — REAL for sppequ
DOUBLE PRECISION for dppequ
COMPLEX for cppequ
DOUBLE COMPLEX for zppequ.
Array, DIMENSION at least max(1,*n*(*n*+1)/2). The array *ap* contains the upper or the lower triangular part of the matrix *A* (as specified by *uplo*) in packed storage (see Matrix Storage Schemes).

Output Parameters

s — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Array, DIMENSION (*n*).
If *info* = 0, the array *s* contains the scale factors for *A*.

scond — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
If *info* = 0, *scond* contains the ratio of the smallest *s*(i) to the largest *s*(i).

amax — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Absolute value of the largest element of the matrix *A*.

info — INTEGER.
If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, the *i*-th diagonal element of *A* is nonpositive.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ppequ interface are as follows:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).
s	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If *scond* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *s*.

If *amax* is very close to overflow or very close to underflow, the matrix *A* should be scaled.

?pbequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive-definite band matrix and reduce its condition number.

Syntax

Fortran 77:

```
call spbequ( uplo, n, kd, ab, ldab, s, scond, amax, info )
call dpbequ( uplo, n, kd, ab, ldab, s, scond, amax, info )
call cpbequ( uplo, n, kd, ab, ldab, s, scond, amax, info )
call zpbequ( uplo, n, kd, ab, ldab, s, scond, amax, info )
```

Fortran 95:

```
call pbequ( a, s [,scond] [,amax] [,uplo] [,info] )
```

Description

This routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive definite matrix *A* in packed storage and reduce its condition number (with respect to the two-norm). The output array *s* returns scale factors computed as

$$s(i) = \frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled matrix *B* with elements b_{ij}=*s*(i)**a*$_{ij}$**s*(j) has diagonal elements equal to 1. This choice of *s* puts the condition number of *B* within a factor *n* of the smallest possible condition number over all possible diagonal scalings.

See ?laqsb auxiliary function that uses scaling factors computed by ?pbequ.

Input Parameters

uplo
CHARACTER*1. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of *A* is packed in the array *ab*:
If *uplo* = 'U', the array *ab* stores the upper triangular part of the matrix *A*.
If *uplo* = 'L', the array *ab* stores the lower triangular part of the matrix *A*.

n
INTEGER. The order of matrix *A*; *n* ≥ 0.

kd
INTEGER. The number of superdiagonals or subdiagonals in the matrix *A*; *kd* ≥ 0.

ab
REAL for spbequ
DOUBLE PRECISION for dpbequ
COMPLEX for cpbequ
DOUBLE COMPLEX for zpbequ.
Array, DIMENSION (*ldab*,*).
The array *ap* contains either the upper or the lower triangular part of the matrix *A* (as specified by *uplo*) in band storage (see Matrix Storage Schemes).
The second dimension of *ab* must be at least max(1, *n*).

ldab	INTEGER. The leading dimension of the array *ab*; *ldab* ≥ *kd* +1.

Output Parameters

s	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION (*n*). If *info* = 0, the array *s* contains the scale factors for *A*.
scond	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If *info* = 0, *scond* contains the ratio of the smallest *s*(i) to the largest *s*(i).
amax	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Absolute value of the largest element of the matrix *A*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the *i*-th diagonal element of *A* is nonpositive.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine pbequ interface are as follows:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*kd*+1, *n*).
s	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If *scond* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *s*.

If *amax* is very close to overflow or very close to underflow, the matrix *A* should be scaled.

Driver Routines

Table 3-3 lists the LAPACK driver routines for solving systems of linear equations with real or complex matrices.

Table 3-3 Driver Routines for Solving Systems of Linear Equations

Matrix type, storage scheme	Simple Driver	Expert Driver
general	?gesv	?gesvx
general band	?gbsv	?gbsvx
general tridiagonal	?gtsv	?gtsvx
symmetric/Hermitian positive-definite	?posv	?posvx
symmetric/Hermitian positive-definite, storage	?ppsv	?ppsvx
symmetric/Hermitian positive-definite, band	?pbsv	?pbsvx
symmetric/Hermitian positive-definite, tridiagonal	?ptsv	?ptsvx
symmetric/Hermitian indefinite	?sysv/?hesv	?sysvx/?hesvx
symmetric/Hermitian indefinite, packed storage	?spsv/?hpsv	?spsvx/?hpsvx
complex symmetric	?sysv	?sysvx
complex symmetric, packed storage	?spsv	?spsvx

In this table ? stands for s (single precision real), d (double precision real), c (single precision complex), or z (double precision complex). In the description of ?gesv routine ? stands for combined character codes ds and zc for the mixed precision subroutines.

?gesv

Computes the solution to the system of linear equations with a square matrix A and multiple right-hand sides.

Syntax

Fortran 77:

```
call sgesv( n, nrhs, a, lda, ipiv, b, ldb, info )
call dgesv( n, nrhs, a, lda, ipiv, b, ldb, info )
call cgesv( n, nrhs, a, lda, ipiv, b, ldb, info )
call zgesv( n, nrhs, a, lda, ipiv, b, ldb, info )
call dsgesv( n, nrhs, a, lda, ipiv, b, ldb, x, ldx, work, swork, iter, info
)
call zcgesv( n, nrhs, a, lda, ipiv, b, ldb, x, ldx, work, swork, iter, info
)
```

Fortran 95:

```
call gesv( a, b [,ipiv] [,info] )
```

Description

This routine solves for *X* the system of linear equations *A*X = B*, where *A* is an *n*-by-*n* matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

The *LU* decomposition with partial pivoting and row interchanges is used to factor *A* as *A = P*L*U*, where *P* is a permutation matrix, *L* is unit lower triangular, and *U* is upper triangular. The factored form of *A* is then used to solve the system of equations *A*X = B*.

The dsgesv and zcgesv are mixed precision iterative refinement subroutines for exploiting fast single precision hardware. They first attempt to factorize the matrix in single precision (dsgesv) or single complex precision (zcgesv) and use this factorization within an iterative refinement procedure to produce a solution with double precision (dsgesv) / double complex precision (zcgesv) normwise backward error quality (see below). If the approach fails the method switches to a double precision or double complex precision factorization respectively and solve.

The iterative refinement is not going to be a winning strategy if the ratio single precision performance over double precision performance is too small. A reasonable strategy should take the number of right-hand sides and the size of the matrix into account. This might be done with a call to `ilaenv` in the future. At present, iterative refinement is implemented.

The iterative refinement process is stopped if

```
iter > itermax
```

or for all the `RHS`:

```
RNRM < SQRT(N)*XNRM*ANRM*EPS*BWDMAX,
```

where

- `iter` is the number of the current iteration in the iterativerefinement process
- `RNRM` is the infinity-norm of the residual
- `XNRM` is the infinity-norm of the solution
- `ANRM` is the infinity-operator-norm of the matrix *A*
- `EPS` is the machine epsilon returned by `dlamch` ('Epsilon').

The values `itermax` and `BWDMAX` are fixed to 30 and 1.0D+00 respectively.

Input Parameters

n	`INTEGER`. The number of linear equations, that is, the order of the matrix *A*; $n \geq 0$.
nrhs	`INTEGER`. The number of right-hand sides, that is, the number of columns of the matrix *B*; $nrhs \geq 0$.

a, *b*	`REAL` for `sgesv`
	`DOUBLE PRECISION` for `dgesv` and `dsgesv`
	`COMPLEX` for `cgesv`
	`DOUBLE COMPLEX` for `zgesv` and `zcgesv`.
	Arrays: *a*(*lda*,*), *b*(*ldb*,*).
	The array *a* contains the *n*-by-*n* coefficient matrix *A*.
	The array *b* contains the *n*-by-*nrhs* matrix of right hand side matrix *B*.
	The second dimension of *a* must be at least `max(1, n)`, the second dimension of *b* at least `max(1, nrhs)`.

lda INTEGER. The leading dimension of the array *a*; *lda* ≥ max(1, *n*).

ldb INTEGER. The leading dimension of the array *b*; *ldb* ≥ max(1, *n*).

ldx INTEGER. The leading dimension of the array *x*; *ldx* ≥ max(1, *n*).

work DOUBLE PRECISION for dsgesv
DOUBLE COMPLEX for zcgesv.
Workspace array, DIMENSION (*n***nrhs*). This array is used to hold the residual vectors.

swork SINGLE PRECISION for dsgesv
COMPLEX for zcgesv.
Workspace array, DIMENSION (*n**(*n*+*nrhs*)). This array is used to use the single precision matrix and the right-hand sides or solutions in single precision.

Output Parameters

a Overwritten by the factors *L* and *U* from the factorization of *A = P*L*U*; the unit diagonal elements of *L* are not stored.
If iterative refinement has been successfully used (*info*= 0 and *iter* ≥ 0, see description below), then *A* is unchanged.
If double precision factorization has been used (*info*= 0 and *iter* < 0, see description below), then the array *A* contains the factors *L* and *U* from the factorization *A = P*L*U*; the unit diagonal elements of *L* are not stored.

b Overwritten by the solution matrix *X*.

ipiv INTEGER.
Array, DIMENSION at least max(1, *n*). The pivot indices that define the permutation matrix *P*; row *i* of the matrix was interchanged with row *ipiv*(i). Corresponds to the single precision factorization (if *info*= 0 and *iter* ≥ 0) or the double precision factorization (if *info*= 0 and *iter* < 0).

x DOUBLE PRECISION for dsgesv
DOUBLE COMPLEX for zcgesv.
Array, DIMENSION (*ldx*, *nrhs*). If *info* = 0, contains the *n*-by-*nrhs* solution matrix *X*.

iter INTEGER.

If *iter* < 0: iterative refinement has failed, double precision factorization has been performed

- If *iter* = -1: taking into account machine parameters, *n*, *nrhs*, it is a priori not worth working in single precision
- If *iter* = -2: overflow of an entry when moving from double to single precision
- If *iter* = -3: failure of sgetrf
- If *iter* = -31: stop the iterative refinement after the 30th iteration.

If *iter* > 0: iterative refinement has been sucessfully used. Returns the number of iterations.

info INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, *U*(*i*, *i*) (computed in double precision for mixed precision subroutines) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gesv interface are as follows:

a Holds the matrix *A* of size (*n*, *n*).

b Holds the matrix *B* of size (*n*, *nrhs*).

ipiv Holds the vector of length (*n*).

NOTE. Fortran 95 Interface is so far not available for the mixed precision subroutines dsgesv/zcgesv.

?gesvx

Computes the solution to the system of linear equations with a square matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax

Fortran 77:

```
call sgesvx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, ferr, berr, work, iwork, info )

call dgesvx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, ferr, berr, work, iwork, info )

call cgesvx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, ferr, berr, work, rwork, info )

call zgesvx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call gesvx( a, b, x [,af] [,ipiv] [,fact] [,trans] [,equed] [,r] [,c] [,ferr]
[,berr] [,rcond] [,rpvgrw] [,info] )
```

Description

This routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations *A*X = B*, where *A* is an *n*-by-*n* matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?gesvx performs the following steps:

1. If *fact* = 'E', real scaling factors *r* and *c* are computed to equilibrate the system:

trans = 'N': *diag*(*r*)**A***diag*(*c*)*inv(*diag*(*c*))**X* = *diag*(*r*)**B*

trans = 'T': $(diag(r)*A*diag(c))^{T}$*inv(*diag*(*r*))**X* = *diag*(*c*)**B*

trans = 'C': $(diag(r)*A*diag(c))^{H}$*inv(*diag*(*r*))**X* = *diag*(*c*)**B*

Whether or not the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by *diag(r)*A*diag(c)* and *B* by *diag(r)*B* (if *trans*='N') or *diag(c)*B* (if *trans* = 'T' or 'C').

2. If *fact* = 'N' or 'E', the *LU* decomposition is used to factor the matrix *A* (after equilibration if *fact* = 'E') as *A* = *P*L*U*, where *P* is a permutation matrix, *L* is a unit lower triangular matrix, and *U* is upper triangular.
3. If some $U_{i,i}$= 0, so that *U* is exactly singular, then the routine returns with *info* = *i*. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than machine precision, *info* = *n* + 1 is returned as a warning, but the routine still goes on to solve for *X* and compute error bounds as described below.
4. The system of equations is solved for *X* using the factored form of *A*.
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix *X* is premultiplied by *diag(c)* (if *trans* = 'N') or *diag(r)* (if *trans* = 'T' or 'C') so that it solves the original system before equilibration.

Input Parameters

fact
CHARACTER*1. Must be 'F', 'N', or 'E'.
Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.
If *fact* = 'F': on entry, *af* and *ipiv* contain the factored form of *A*. If *equed* is not 'N', the matrix *A* has been equilibrated with scaling factors given by *r* and *c*.

a, *af*, and *ipiv* are not modified.
If *fact* = 'N', the matrix *A* will be copied to *af* and factored.
If *fact* = 'E', the matrix *A* will be equilibrated if necessary, then copied to *af* and factored.

trans
CHARACTER*1. Must be 'N', 'T', or 'C'.
Specifies the form of the system of equations:
If *trans* = 'N', the system has the form *A*X* = *B* (No transpose).
If *trans* = 'T', the system has the form A^T*X = *B* (Transpose).
If *trans* = 'C', the system has the form A^H*X = *B* (Conjugate transpose).

n INTEGER. The number of linear equations; the order of the matrix *A*; *n* ≥ 0.

nrhs INTEGER. The number of right hand sides; the number of columns of the matrices *B* and *X*; *nrhs* ≥ 0.

a,*af*,*b*,*work* REAL for sgesvx
DOUBLE PRECISION for dgesvx
COMPLEX for cgesvx
DOUBLE COMPLEX for zgesvx.
Arrays: *a*(*lda*,*), *af*(*ldaf*,*), *b*(*ldb*,*), *work*(*).
The array *a* contains the matrix *A*. If *fact* = 'F' and *equed* is not 'N', then *A* must have been equilibrated by the scaling factors in *r* and/or *c*. The second dimension of *a* must be at least max(1,*n*).
The array *af* is an input argument if *fact* = 'F'. It contains the factored form of the matrix *A*, that is, the factors *L* and *U* from the factorization *A* = *P*L*U* as computed by ?getrf. If *equed* is not 'N', then *af* is the factored form of the equilibrated matrix *A*. The second dimension of *af* must be at least max(1,*n*).
The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1, *nrhs*).
work(*) is a workspace array. The dimension of *work* must be at least max(1,4*n) for real flavors, and at least max(1,2*n) for complex flavors.

lda INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).

ldaf INTEGER. The first dimension of *af*; *ldaf* ≥ max(1, *n*).

ldb INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

ipiv INTEGER.
Array, DIMENSION at least max(1, *n*). The array *ipiv* is an input argument if *fact* = 'F'. It contains the pivot indices from the factorization *A* = *P*L*U* as computed by ?getrf; row *i* of the matrix was interchanged with row *ipiv*(*i*).

equed CHARACTER*1. Must be 'N', 'R', 'C', or 'B'.
equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

	If *equed* = 'N', no equilibration was done (always true if *fact* = 'N'). If *equed* = 'R', row equilibration was done and *A* has been premultiplied by *diag*(*r*). If *equed* = 'C', column equilibration was done and *A* has been postmultiplied by *diag*(*c*). If *equed* = 'B', both row and column equilibration was done; *A* has been replaced by *diag*(*r*)**A***diag*(*c*).
r, *c*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays: *r*(*n*), *c*(*n*). The array *r* contains the row scale factors for *A*, and the array *c* contains the column scale factors for *A*. These arrays are input arguments if *fact* = 'F' only; otherwise they are output arguments. If *equed* = 'R' or 'B', *A* is multiplied on the left by *diag*(*r*); if *equed* = 'N' or 'C', *r* is not accessed. If *fact* = 'F' and *equed* = 'R' or 'B', each element of *r* must be positive. If *equed* = 'C' or 'B', *A* is multiplied on the right by *diag*(*c*); if *equed* = 'N' or 'R', *c* is not accessed. If *fact* = 'F' and *equed* = 'C' or 'B', each element of *c* must be positive.
ldx	INTEGER. The first dimension of the output array *x*; *ldx* ≥ max(1, *n*).
iwork	INTEGER. Workspace array, DIMENSION at least max(1, *n*); used in real flavors only.
rwork	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Workspace array, DIMENSION at least max(1, 2**n*); used in complex flavors only.

Output Parameters

x	REAL for sgesvx DOUBLE PRECISION for dgesvx COMPLEX for cgesvx DOUBLE COMPLEX for zgesvx.

Array, DIMENSION (*ldx*,*).
If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the original system of equations. Note that *A* and *B* are modified on exit if *equed* ≠ 'N', and the solution to the equilibrated system is:
diag(*C*)$^{-1}$**X*, if *trans* = 'N' and *equed* = 'C' or 'B'; *diag*(*R*)$^{-1}$**X*, if *trans* = 'T' or 'C' and *equed* = 'R' or 'B'. The second dimension of *x* must be at least max(1,*nrhs*).

a — Array *a* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'. If *equed* ≠ 'N', *A* is scaled on exit as follows:
equed = 'R': *A* = *diag*(*R*)**A*
equed = 'C': *A* = *A***diag*(*c*)
equed = 'B': *A* = *diag*(*R*)**A***diag*(*c*).

af — If *fact* = 'N' or 'E', then *af* is an output argument and on exit returns the factors *L* and *U* from the factorization *A* = *PLU* of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *a* for the form of the equilibrated matrix.

b — Overwritten by *diag*(*r*)**B* if *trans* = 'N' and *equed* = 'R' or 'B'; overwritten by *diag*(*c*)**B* if *trans* = 'T' and *equed* = 'C' or 'B'; not changed if *equed* = 'N'.

r, *c* — These arrays are output arguments if *fact* ≠ 'F'. See the description of *r*, *c* in Input Arguments section.

rcond — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). The routine sets *rcond* = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

ferr, *berr* — REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least max(1, *nrhs*). Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

ipiv	If *fact* = 'N' or 'E', then *ipiv* is an output argument and on exit contains the pivot indices from the factorization *A* = *P*L*U* of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E').
equed	If *fact* ≠ 'F' , then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in Input Arguments section).
work, *rwork*	On exit, *work*(1) for real flavors, or *rwork*(1) for complex flavors, contains the reciprocal pivot growth factor norm(*A*)/norm(*U*). The "max absolute element" norm is used. If *work*(1) for real flavors, or *rwork*(1) for complex flavors is much less than 1, then the stability of the *LU* factorization of the (equilibrated) matrix *A* could be poor. This also means that the solution *x*, condition estimator *rcond*, and forward error bound *ferr* could be unreliable. If factorization fails with 0 < *info* ≤ *n*, then *work*(1) for real flavors, or *rwork*(1) for complex flavors contains the reciprocal pivot growth factor for the leading *info* columns of *A*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, and *i* ≤ *n*, then *U*(*i*, *i*) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned. If *info* = *i*, and *i* = *n*+1, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gesvx interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).

x	Holds the matrix *X* of size (*n*, *nrhs*).
af	Holds the matrix *AF* of size (*n*, *n*).
ipiv	Holds the vector of length (*n*).
r	Holds the vector of length (*n*). Default value for each element is *r*(*i*) = 1.0_WP.
c	Holds the vector of length (*n*). Default value for each element is *c*(*i*) = 1.0_WP.
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
fact	Must be 'N', 'E', or 'F'. The default value is 'N'. If *fact* = 'F', then both arguments *af* and *ipiv* must be present; otherwise, an error is returned.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
equed	Must be 'N', 'B', 'C', or 'R'. The default value is 'N'.
rpvgrw	Real value that contains the reciprocal pivot growth factor norm(*A*)/norm(*U*).

?gbsv

Computes the solution to the system of linear equations with a band matrix A and multiple right-hand sides.

Syntax

Fortran 77:

```
call sgbsv( n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call dgbsv( n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call cgbsv( n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call zgbsv( n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
```

Fortran 95:

```
call gbsv( a, b [,kl] [,ipiv] [,info] )
```

Description

This routine solves for *X* the real or complex system of linear equations *A*X = B*, where *A* is an *n*-by-*n* band matrix with *kl* subdiagonals and *ku* superdiagonals, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

The *LU* decomposition with partial pivoting and row interchanges is used to factor *A* as *A = L*U*, where *L* is a product of permutation and unit lower triangular matrices with *kl* subdiagonals, and *U* is upper triangular with *kl*+*ku* superdiagonals. The factored form of *A* is then used to solve the system of equations *A*X = B*.

Input Parameters

n	INTEGER. The order of *A*. The number of rows in *B*; *n* ≥ 0.
kl	INTEGER. The number of subdiagonals within the band of *A*; *kl* ≥ 0.
ku	INTEGER. The number of superdiagonals within the band of *A*; *ku* ≥ 0.
nrhs	INTEGER. The number of right-hand sides. The number of columns in *B*; *nrhs* ≥ 0.
ab, *b*	REAL for sgbsv DOUBLE PRECISION for dgbsv COMPLEX for cgbsv DOUBLE COMPLEX for zgbsv. Arrays: *ab*(*ldab*,*), *b*(*ldb*,*). The array *ab* contains the matrix *A* in band storage (see Matrix Storage Schemes). The second dimension of *ab* must be at least max(1, *n*). The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).
ldab	INTEGER. The first dimension of the array *ab*. (*ldab* ≥ 2*kl* + *ku* +1)
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

ab	Overwritten by *L* and *U*. The diagonal and *kl* + *ku* superdiagonals of *U* are stored in the first 1 + *kl* + *ku* rows of *ab*. The multipliers used to form *L* are stored in the next *kl* rows.
b	Overwritten by the solution matrix *X*.

ipiv	INTEGER. Array, DIMENSION at least max(1, *n*). The pivot indices: row *i* was interchanged with row *ipiv*(*i*).
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, *U*(*i*, *i*) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gbsv interface are as follows:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (2**kl*+*ku*+1,*n*).
b	Holds the matrix *B* of size (*n*,*nrhs*).
ipiv	Holds the vector of length (*n*).
kl	If omitted, assumed *kl* = *ku*.
ku	Restored as *ku* = *lda*-2**kl*-1.

?gbsvx

Computes the solution to the real or complex system of linear equations with a band matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax

Fortran 77:

```
call sgbsvx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed,
r, c, b, ldb, x, ldx, rcond, ferr, berr, work, iwork, info )

call dgbsvx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed,
r, c, b, ldb, x, ldx, rcond, ferr, berr, work, iwork, info )

call cgbsvx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed,
r, c, b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info )

call zgbsvx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed,
r, c, b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call gbsvx( a, b, x [,kl] [,af] [,ipiv] [,fact] [,trans] [,equed] [,r] [,c]
[,ferr] [,berr] [,rcond] [,rpvgrw] [,info] )
```

Description

This routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations $A*X = B$, $A^T*X = B$, or $A^H*X = B$, where A is a band matrix of order n with kl subdiagonals and ku superdiagonals, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?gbsvx performs the following steps:

1. If *fact* = 'E', real scaling factors r and c are computed to equilibrate the system:

trans = 'N': *diag*(*r*)**A*diag*(*c*) *inv(*diag*(*c*))**X* = *diag*(*r*)**B*

trans = 'T': $(diag(r)*A*diag(c))^T$ *inv(*diag*(*r*))**X* = *diag*(*c*)**B*

trans = 'C': $(diag(r)*A*diag(c))^H$ *inv(*diag*(*r*))**X* = *diag*(*c*)**B*

Whether or not the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by *diag(r)*A*diag(c)* and *B* by *diag(r)*B* (if *trans*='N') or *diag(c)*B* (if *trans* = 'T' or 'C').

2. If *fact* = 'N' or 'E', the *LU* decomposition is used to factor the matrix *A* (after equilibration if *fact* = 'E') as *A* = *L*U*, where *L* is a product of permutation and unit lower triangular matrices with *kl* subdiagonals, and *U* is upper triangular with *kl*+*ku* superdiagonals.
3. If some $U_{i,i}$ = 0, so that *U* is exactly singular, then the routine returns with *info* = *i*. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than machine precision, *info* = *n* + 1 is returned as a warning, but the routine still goes on to solve for *X* and compute error bounds as described below.
4. The system of equations is solved for *X* using the factored form of *A*.
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix *X* is premultiplied by *diag(c)* (if *trans* = 'N') or *diag(r)* (if *trans* = 'T' or 'C') so that it solves the original system before equilibration.

Input Parameters

fact

CHARACTER*1. Must be 'F', 'N', or 'E'.
Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.
If *fact* = 'F': on entry, *afb* and *ipiv* contain the factored form of *A*. If *equed* is not 'N', the matrix *A* has been equilibrated with scaling factors given by *r* and *c*.

ab, *afb*, and *ipiv* are not modified.
If *fact* = 'N', the matrix *A* will be copied to *afb* and factored.
If *fact* = 'E', the matrix *A* will be equilibrated if necessary, then copied to *afb* and factored.

trans

CHARACTER*1. Must be 'N', 'T', or 'C'.
Specifies the form of the system of equations:
If *trans* = 'N', the system has the form *A*X* = *B* (No transpose).
If *trans* = 'T', the system has the form $A^T*X = B$ (Transpose).
If *trans* = 'C', the system has the form $A^H*X = B$ (Conjugate transpose).

n	INTEGER. The number of linear equations, the order of the matrix *A*; *n* ≥ 0.
kl	INTEGER. The number of subdiagonals within the band of *A*; *kl* ≥ 0.
ku	INTEGER. The number of superdiagonals within the band of *A*; *ku* ≥ 0.
nrhs	INTEGER. The number of right hand sides, the number of columns of the matrices *B* and *X*; *nrhs* ≥ 0.
ab,*afb*,*b*,*work*	REAL for sgesvx DOUBLE PRECISION for dgesvx COMPLEX for cgesvx DOUBLE COMPLEX for zgesvx. Arrays: *a*(*lda*,*), *af*(*ldaf*,*), *b*(*ldb*,*), *work*(*). The array *ab* contains the matrix *A* in band storage (see Matrix Storage Schemes). The second dimension of *ab* must be at least max(1, *n*). If *fact* = 'F' and *equed* is not 'N', then *A* must have been equilibrated by the scaling factors in *r* and/or *c*. The array *afb* is an input argument if *fact* = 'F'. The second dimension of *afb* must be at least max(1,*n*). It contains the factored form of the matrix *A*, that is, the factors *L* and *U* from the factorization *A* = *L*U* as computed by ?gbtrf. *U* is stored as an upper triangular band matrix with *kl* + *ku* superdiagonals in the first 1 + *kl* + *ku* rows of *afb*. The multipliers used during the factorization are stored in the next *kl* rows. If *equed* is not 'N', then *afb* is the factored form of the equilibrated matrix *A*. The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1, *nrhs*). *work(*)* is a workspace array. The dimension of *work* must be at least max(1,3*n) for real flavors, and at least max(1,2*n) for complex flavors.
ldab	INTEGER. The first dimension of *ab*; *ldab* ≥ *kl*+*ku*+1.
ldafb	INTEGER. The first dimension of *afb*; *ldafb* ≥ 2**kl*+*ku*+1.
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).
ipiv	INTEGER.

Array, DIMENSION at least max(1, *n*). The array *ipiv* is an input argument if *fact* = 'F'. It contains the pivot indices from the factorization *A* = *L***U* as computed by ?gbtrf; row *i* of the matrix was interchanged with row *ipiv*(*i*).

equed

CHARACTER*1. Must be 'N', 'R', 'C', or 'B'.
equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:
If *equed* = 'N', no equilibration was done (always true if *fact* = 'N').
If *equed* = 'R', row equilibration was done and *A* has been premultiplied by *diag*(*r*).
If *equed* = 'C', column equilibration was done and *A* has been postmultiplied by *diag*(*c*).
if *equed* = 'B', both row and column equilibration was done; *A* has been replaced by *diag*(*r*)**A***diag*(*c*).

r, *c*

REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Arrays: *r*(*n*), *c*(*n*).
The array *r* contains the row scale factors for *A*, and the array *c* contains the column scale factors for *A*. These arrays are input arguments if *fact* = 'F' only; otherwise they are output arguments.
If *equed* = 'R' or 'B', *A* is multiplied on the left by *diag*(*r*); if *equed* = 'N' or 'C', *r* is not accessed.
If *fact* = 'F' and *equed* = 'R' or 'B', each element of *r* must be positive.
If *equed* = 'C' or 'B', *A* is multiplied on the right by *diag*(*c*); if *equed* = 'N' or 'R', *c* is not accessed.
If *fact* = 'F' and *equed* = 'C' or 'B', each element of *c* must be positive.

ldx

INTEGER. The first dimension of the output array *x*; *ldx* ≥ max(1, *n*).

iwork

INTEGER. Workspace array, DIMENSION at least max(1, *n*); used in real flavors only.

rwork

REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Workspace array, DIMENSION at least max(1, *n*); used in complex flavors only.

Output Parameters

x	REAL for sgbsvx DOUBLE PRECISION for dgbsvx COMPLEX for cgbsvx DOUBLE COMPLEX for zgbsvx. Array, DIMENSION (*ldx*,*). If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the original system of equations. Note that *A* and *B* are modified on exit if *equed* ≠ 'N', and the solution to the equilibrated system is: inv(*diag*(*c*))***X*, if *trans* = 'N' and *equed* = 'C' or 'B'; inv(*diag*(*r*))***X*, if *trans* = 'T' or 'C' and *equed* = 'R' or 'B'. The second dimension of *x* must be at least max(1,*nrhs*).
ab	Array *ab* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'. If *equed* ≠ 'N', *A* is scaled on exit as follows: *equed* = 'R': *A* = *diag*(*r*)***A* *equed* = 'C': *A* = *A***diag*(*c*) *equed* = 'B': *A* = *diag*(*r*)***A***diag*(*c*).
afb	If *fact* = 'N' or 'E', then *afb* is an output argument and on exit returns details of the *LU* factorization of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *ab* for the form of the equilibrated matrix.
b	Overwritten by *diag*(*r*)***b* if *trans* = 'N' and *equed* = 'R' or 'B'; overwritten by *diag*(*c*)***b* if *trans* = 'T' and *equed* = 'C' or 'B'; not changed if *equed* = 'N'.
r, *c*	These arrays are output arguments if *fact* ≠ 'F'. See the description of *r*, *c* in Input Arguments section.
rcond	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision (in particular, if *rcond* =0), the matrix is singular to working precision. This condition is indicated by a return code of *info*>0.
ferr, *berr*	REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least max(1, *nrhs*). Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

ipiv — If *fact* = 'N' or 'E', then *ipiv* is an output argument and on exit contains the pivot indices from the factorization *A* = *L*U* of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E').

equed — If *fact* ≠ 'F', then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in Input Arguments section).

work, *rwork* — On exit, *work*(1) for real flavors, or *rwork*(1) for complex flavors, contains the reciprocal pivot growth factor norm(*A*)/norm(*U*). The "max absolute element" norm is used. If *work*(1) for real flavors, or *rwork*(1) for complex flavors is much less than 1, then the stability of the *LU* factorization of the (equilibrated) matrix *A* could be poor. This also means that the solution *x*, condition estimator *rcond*, and forward error bound *ferr* could be unreliable. If factorization fails with 0 < *info* ≤ *n*, then *work*(1) for real flavors, or *rwork*(1) for complex flavors contains the reciprocal pivot growth factor for the leading *info* columns of *A*.

info — INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, and *i* ≤ *n*, then *U*(*i*, *i*) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned. If *info* = *i*, and *i* = *n*+1, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gbsvx interface are as follows:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*kl*+*ku*+1,*n*).
b	Holds the matrix *B* of size (*n*,*nrhs*).
x	Holds the matrix *X* of size (*n*,*nrhs*).
af	Stands for argument *ab* in Fortan 77 interface. Holds the array *AF* of size (2**kl*+*ku*+1,*n*).
ipiv	Holds the vector of length (*n*).
r	Holds the vector of length (*n*). Default value for each element is *r*(*i*) = 1.0_WP.
c	Holds the vector of length (*n*). Default value for each element is *c*(*i*) = 1.0_WP.
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.
equed	Must be 'N', 'B', 'C', or 'R'. The default value is 'N'.
fact	Must be 'N', 'E', or 'F'. The default value is 'N'. If *fact* = 'F', then both arguments *af* and *ipiv* must be present; otherwise, an error is returned.
rpvgrw	Real value that contains the reciprocal pivot growth factor norm(*A*)/norm(*U*).
kl	If omitted, assumed *kl* = *ku*.
ku	Restored as *ku* = *lda*-*kl*-1.

?gtsv

Computes the solution to the system of linear equations with a tridiagonal matrix A and multiple right-hand sides.

Syntax

Fortran 77:

```
call sgtsv( n, nrhs, dl, d, du, b, ldb, info )
call dgtsv( n, nrhs, dl, d, du, b, ldb, info )
call cgtsv( n, nrhs, dl, d, du, b, ldb, info )
call zgtsv( n, nrhs, dl, d, du, b, ldb, info )
```

Fortran 95:

```
call gtsv( dl, d, du, b [,info] )
```

Description

This routine solves for X the system of linear equations $A*X = B$, where A is an n-by-n tridiagonal matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions. The routine uses Gaussian elimination with partial pivoting.

Note that the equation $A^T*X = B$ may be solved by interchanging the order of the arguments *du* and *dl*.

Input Parameters

n	INTEGER. The order of *A*, the number of rows in *B*; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides, the number of columns in *B*; $nrhs \geq 0$.
dl, *d*, *du*, *b*	REAL for sgtsv DOUBLE PRECISION for dgtsv COMPLEX for cgtsv DOUBLE COMPLEX for zgtsv. Arrays: *dl*(*n* - 1), *d*(*n*), *du*(*n* - 1), *b*(*ldb*,*). The array *dl* contains the (*n* - 1) subdiagonal elements of *A*. The array *d* contains the diagonal elements of *A*.

	The array *du* contains the (*n* – 1) superdiagonal elements of *A*. The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1, *nrhs*).
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

dl	Overwritten by the (*n*–2) elements of the second superdiagonal of the upper triangular matrix *U* from the *LU* factorization of A. These elements are stored in *dl*(1), ... , *dl*(*n*–2).
d	Overwritten by the *n* diagonal elements of *U*.
du	Overwritten by the (*n*–1) elements of the first superdiagonal of *U*.
b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = –*i*, the *i*-th parameter had an illegal value. If *info* = *i*, *U*(*i*, *i*) is exactly zero, and the solution has not been computed. The factorization has not been completed unless *i* = *n*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gtsv interface are as follows:

dl	Holds the vector of length (*n*–1).
d	Holds the vector of length (*n*).
dl	Holds the vector of length (*n*–1).
b	Holds the matrix *B* of size (*n*, *nrhs*).

?gtsvx

Computes the solution to the real or complex system of linear equations with a tridiagonal matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax

Fortran 77:

```
call sgtsvx( fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb,
x, ldx, rcond, ferr, berr, work, iwork, info )

call dgtsvx( fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb,
x, ldx, rcond, ferr, berr, work, iwork, info )

call cgtsvx( fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb,
x, ldx, rcond, ferr, berr, work, rwork, info )

call zgtsvx( fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb,
x, ldx, rcond, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call gtsvx( dl, d, du, b, x [,dlf] [,df] [,duf] [,du2] [,ipiv] [,fact] [,trans]
[,ferr] [,berr] [,rcond] [,info] )
```

Description

This routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations *A*X = B*, $A^T*X = B$, or $A^H*X = B$, where *A* is a tridiagonal matrix of order *n*, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?gtsvx performs the following steps:

1. If *fact = 'N'*, the *LU* decomposition is used to factor the matrix *A* as *A = L*U*, where *L* is a product of permutation and unit lower bidiagonal matrices and *U* is an upper triangular matrix with nonzeroes in only the main diagonal and first two superdiagonals.

2. If some $U_{i,i}$ = 0, so that U is exactly singular, then the routine returns with *info* = *i*. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than machine precision, *info* = *n* + 1 is returned as a warning, but the routine still goes on to solve for *X* and compute error bounds as described below.
3. The system of equations is solved for *X* using the factored form of *A*.
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

fact	CHARACTER*1. Must be 'F' or 'N'. Specifies whether or not the factored form of the matrix *A* has been supplied on entry. If *fact* = 'F': on entry, *dlf*, *df*, *duf*, *du2*, and *ipiv* contain the factored form of *A*; arrays *dl*, *d*, *du*, *dlf*, *df*, *duf*, *du2*, and *ipiv* will not be modified. If *fact* = 'N', the matrix *A* will be copied to *dlf*, *df*, and *duf* and factored.
trans	CHARACTER*1. Must be 'N', 'T', or 'C'. Specifies the form of the system of equations: If *trans* = 'N', the system has the form *A***X* = *B* (No transpose). If *trans* = 'T', the system has the form A^T**X* = *B* (Transpose). If *trans* = 'C', the system has the form A^H**X* = *B* (Conjugate transpose).
n	INTEGER. The number of linear equations, the order of the matrix *A*; *n* ≥ 0.
nrhs	INTEGER. The number of right hand sides, the number of columns of the matrices *B* and *X*; *nrhs* ≥ 0.
dl,*d*,*du*,*dlf*,*df*, *duf*,*du2*,*b*, *x*,*work*	REAL for sgtsvx DOUBLE PRECISION for dgtsvx COMPLEX for cgtsvx DOUBLE COMPLEX for zgtsvx. Arrays: *dl*, dimension (*n* -1), contains the subdiagonal elements of *A*. *d*, dimension (*n*), contains the diagonal elements of *A*. *du*, dimension (*n* -1), contains the superdiagonal elements of *A*.

dlf, dimension (*n* -1). If *fact* = 'F' , then *dlf* is an input argument and on entry contains the (*n* -1) multipliers that define the matrix *L* from the *LU* factorization of *A* as computed by ?gttrf.
df, dimension (*n*). If *fact* = 'F' , then *df* is an input argument and on entry contains the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*.
duf, dimension (*n* -1). If *fact* = 'F' , then *duf* is an input argument and on entry contains the (*n* -1) elements of the first superdiagonal of *U*.
du2, dimension (*n* -2). If *fact* = 'F' , then *du2* is an input argument and on entry contains the (*n*-2) elements of the second superdiagonal of *U*.
b(*ldb**) contains the right-hand side matrix *B*. The second dimension of *b* must be at least max(1, *nrhs*).
x(*ldx**) contains the solution matrix *X*. The second dimension of *x* must be at least max(1, *nrhs*).
work(*) is a workspace array;
the dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.

ldb — INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

ldx — INTEGER. The first dimension of *x*; *ldx* ≥ max(1, *n*).

ipiv — INTEGER.
Array, DIMENSION at least max(1, *n*). If *fact* = 'F' , then *ipiv* is an input argument and on entry contains the pivot indices, as returned by ?gttrf.

iwork — INTEGER. Workspace array, DIMENSION (*n*). Used for real flavors only.

rwork — REAL for cgtsvx
DOUBLE PRECISION for zgtsvx.
Workspace array, DIMENSION (*n*). Used for complex flavors only.

Output Parameters

x — REAL for sgtsvx
DOUBLE PRECISION for dgtsvx
COMPLEX for cgtsvx
DOUBLE COMPLEX for zgtsvx.
Array, DIMENSION (*ldx*,*).

	If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X*. The second dimension of *x* must be at least max(1, *nrhs*).
dlf	If *fact* = 'N', then *dlf* is an output argument and on exit contains the (*n*–1) multipliers that define the matrix *L* from the *LU* factorization of A.
df	If *fact* = 'N', then *df* is an output argument and on exit contains the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*.
duf	If *fact* = 'N', then *duf* is an output argument and on exit contains the (*n*–1) elements of the first superdiagonal of *U*.
du2	If *fact* = 'N', then *du2* is an output argument and on exit contains the (*n*–2) elements of the second superdiagonal of *U*.
ipiv	The array *ipiv* is an output argument if *fact* = 'N' and, on exit, contains the pivot indices from the factorization *A* = *L***U*; row *i* of the matrix was interchanged with row *ipiv*(*i*). The value of *ipiv*(*i*) will always be *i* or *i*+1; *ipiv*(*i*)=*i* indicates a row interchange was not required.
rcond	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix *A*. If *rcond* is less than the machine precision (in particular, if *rcond* =0), the matrix is singular to working precision. This condition is indicated by a return code of *info*>0.
ferr, *berr*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1, *nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = –*i*, the *i*-th parameter had an illegal value. If *info* = *i*, and *i* ≤ *n*, then *U*(*i*, *i*) is exactly zero. The factorization has not been completed unless *i* = *n*, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned. If *info* = *i*, and *i* = *n* + 1, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution

and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `gtsvx` interface are as follows:

dl	Holds the vector of length (*n*–1).
d	Holds the vector of length (*n*).
du	Holds the vector of length (*n*–1).
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).
dlf	Holds the vector of length (*n*–1).
df	Holds the vector of length (*n*).
duf	Holds the vector of length (*n*–1).
du2	Holds the vector of length (*n*–2).
ipiv	Holds the vector of length (*n*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
fact	Must be `'N'` or `'F'`. The default value is `'N'`. If *fact* = `'F'`, then the arguments *dlf*, *df*, *duf*, *du2*, and *ipiv* must be present; otherwise, an error is returned.
trans	Must be `'N'`, `'C'`, or `'T'`. The default value is `'N'`.

?posv

Computes the solution to the system of linear equations with a symmetric or Hermitian positive-definite matrix A and multiple right-hand sides.

Syntax

Fortran 77:

```
call sposv( uplo, n, nrhs, a, lda, b, ldb, info )
call dposv( uplo, n, nrhs, a, lda, b, ldb, info )
call cposv( uplo, n, nrhs, a, lda, b, ldb, info )
call zposv( uplo, n, nrhs, a, lda, b, ldb, info )
```

Fortran 95:

```
call posv( a, b [,uplo] [,info] )
```

Description

This routine solves for X the real or complex system of linear equations $A*X = B$, where A is an n-by-n symmetric/Hermitian positive-definite matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The Cholesky decomposition is used to factor A as

$A = U^T*U$ (real flavors) and $A = U^H*U$ (complex flavors), if $uplo$ = 'U'

or $A = L*L^T$ (real flavors) and $A = L*L^H$ (complex flavors), if $uplo$ = 'L',

where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of A is then used to solve the system of equations $A*X = B$.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If $uplo$ = 'U', the upper triangle of A is stored. If $uplo$ = 'L', the lower triangle of A is stored.
n	INTEGER. The order of matrix A; $n \geq 0$.

nrhs	INTEGER. The number of right-hand sides, the number of columns in *B*; *nrhs* ≥ 0.
a, *b*	REAL for sposv DOUBLE PRECISION for dposv COMPLEX for cposv DOUBLE COMPLEX for zposv. Arrays: *a*(*lda*,*), *b*(*ldb*,*). The array *a* contains the upper or the lower triangular part of the matrix *A* (see *uplo*). The second dimension of *a* must be at least max(1, *n*). The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

a	If *info* = 0, the upper or lower triangular part of *a* is overwritten by the Cholesky factor *U* or *L*, as specified by *uplo*.
b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive definite, so the factorization could not be completed, and the solution has not been computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine posv interface are as follows:

a	Holds the matrix *A* of size (*n*,*n*).
b	Holds the matrix *B* of size (*n*,*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

?posvx

Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric or Hermitian positive definite matrix A, and provides error bounds on the solution.

Syntax

Fortran 77:

```
call sposvx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx,
rcond, ferr, berr, work, iwork, info )

call dposvx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx,
rcond, ferr, berr, work, iwork, info )

call cposvx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx,
rcond, ferr, berr, work, rwork, info )

call zposvx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx,
rcond, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call posvx( a, b, x [,uplo] [,af] [,fact] [,equed] [,s] [,ferr] [,berr] [,rcond]
[,info] )
```

Description

This routine uses the Cholesky factorization $A=U^T*U$ (real flavors) / $A=U^H*U$ (complex flavors) or $A=L*L^T$ (real flavors) / $A=L*L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A*X = B$, where A is a n-by-n real symmetric/Hermitian positive definite matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?posvx performs the following steps:

1. If *fact* = 'E', real scaling factors *s* are computed to equilibrate the system:
   *diag(s)*A*diag(s)*inv(diag(s))*X = diag(s)*B*.

   Whether or not the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by *diag(s)*A*diag(s)* and *B* by *diag(s)*B*.

2. If *fact* = 'N' or 'E', the Cholesky decomposition is used to factor the matrix *A* (after equilibration if *fact* = 'E') as

   A = U^T**U* (real), *A* = U^H**U* (complex), if *uplo* = 'U',

   or *A* = *L**L^T (real), *A* = *L**L^H (complex) , if *uplo* = 'L',

   where *U* is an upper triangular matrix and *L* is a lower triangular matrix.

3. If the leading *i*-by-*i* principal minor is not positive-definite, then the routine returns with *info* = *i*. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than machine precision, *info* = *n* + 1 is returned as a warning, but the routine still goes on to solve for *X* and compute error bounds as described below.

4. The system of equations is solved for *X* using the factored form of *A*.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix *X* is premultiplied by *diag*(*s*) so that it solves the original system before equilibration.

Input Parameters

fact — CHARACTER*1. Must be 'F', 'N', or 'E'.
Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.
If *fact* = 'F': on entry, *af* contains the factored form of *A*. If *equed* = 'Y', the matrix *A* has been equilibrated with scaling factors given by *s*.

a and *af* will not be modified.
If *fact* = 'N', the matrix *A* will be copied to *af* and factored.
If *fact* = 'E', the matrix *A* will be equilibrated if necessary, then copied to *af* and factored.

uplo — CHARACTER*1. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of *A* is stored:
If *uplo* = 'U', the upper triangle of *A* is stored.
If *uplo* = 'L', the lower triangle of *A* is stored.

n — INTEGER. The order of matrix *A*; *n* ≥ 0.

nrhs	INTEGER. The number of right-hand sides, the number of columns in *B*; *nrhs* ≥ 0.
a,*af*,*b*,*work*	REAL for sposvx DOUBLE PRECISION for dposvx COMPLEX for cposvx DOUBLE COMPLEX for zposvx. Arrays: *a*(*lda*,*), *af*(*ldaf*,*), *b*(*ldb*,*), *work*(*). The array *a* contains the matrix *A* as specified by *uplo*. If *fact* = 'F' and *equed* = 'Y', then *A* must have been equilibrated by the scaling factors in *s*, and *a* must contain the equilibrated matrix *diag*(*s*)**A***diag*(*s*). The second dimension of *a* must be at least max(1,*n*). The array *af* is an input argument if *fact* = 'F'. It contains the triangular factor *U* or *L* from the Cholesky factorization of *A* in the same storage format as *A*. If *equed* is not 'N', then *af* is the factored form of the equilibrated matrix *diag*(*s*)**A***diag*(*s*). The second dimension of *af* must be at least max(1,*n*). The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1, *nrhs*). *work(*)* is a workspace array. The dimension of *work* must be at least max(1,3**n*) for real flavors, and at least max(1,2**n*) for complex flavors.
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).
ldaf	INTEGER. The first dimension of *af*; *ldaf* ≥ max(1, *n*).
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).
equed	CHARACTER*1. Must be 'N' or 'Y'. *equed* is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done: if *equed* = 'N', no equilibration was done (always true if *fact* = 'N'); if *equed* = 'Y', equilibration was done and *A* has been replaced by *diag*(*s*)**A***diag*(*s*).
s	REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

Array, DIMENSION (*n*). The array *s* contains the scale factors for *A*. This array is an input argument if *fact* = 'F' only; otherwise it is an output argument.
If *equed* = 'N', *s* is not accessed.
If *fact* = 'F' and *equed* = 'Y', each element of *s* must be positive.

ldx — INTEGER. The first dimension of the output array *x*; *ldx* ≥ max(1, *n*).

iwork — INTEGER. Workspace array, DIMENSION at least max(1, *n*); used in real flavors only.

rwork — REAL for cposvx
DOUBLE PRECISION for zposvx.
Workspace array, DIMENSION at least max(1, *n*); used in complex flavors only.

Output Parameters

x — REAL for sposvx
DOUBLE PRECISION for dposvx
COMPLEX for cposvx
DOUBLE COMPLEX for zposvx.
Array, DIMENSION (*ldx*,*).
If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the original system of equations. Note that if *equed* = 'Y', *A* and *B* are modified on exit, and the solution to the equilibrated system is inv(*diag*(*s*))*X. The second dimension of *x* must be at least max(1,*nrhs*).

a — Array *a* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'.
If *fact* = 'E' and *equed* = 'Y', *A* is overwritten by *diag*(*s*)*A*diag*(*s*).

af — If *fact* = 'N' or 'E', then *af* is an output argument and on exit returns the triangular factor *U* or *L* from the Cholesky factorization $A=U^T*U$ or $A=L*L^T$ (real routines), $A=U^H*U$ or $A=L*L^H$ (complex routines) of the original matrix *A* (if *fact* = 'N'), or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *a* for the form of the equilibrated matrix.

b	Overwritten by *diag(s)*B* , if *equed* = 'Y'; not changed if *equed* = 'N'.
s	This array is an output argument if *fact* ≠ 'F'. See the description of *s* in Input Arguments section.
rcond	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision (in particular, if *rcond* =0), the matrix is singular to working precision. This condition is indicated by a return code of *info*>0.
ferr, *berr*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1, *nrhs*). Contain the component-wise forward and relative backward errors, respectively, for each solution vector.
equed	If *fact* ≠ 'F' , then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in Input Arguments section).
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, and *i* ≤ *n*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, so the factorization could not be completed, and the solution and error bounds could not be computed; *rcond* =0 is returned. If *info* = *i*, and *i* = *n* + 1, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine posvx interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).
af	Holds the matrix *AF* of size (*n*, *n*).
s	Holds the vector of length (*n*). Default value for each element is *s*(*i*) = 1.0_WP.
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
fact	Must be 'N', 'E', or 'F'. The default value is 'N'. If *fact* = 'F', then *af* must be present; otherwise, an error is returned.
equed	Must be 'N' or 'Y'. The default value is 'N'.

?ppsv

Computes the solution to the system of linear equations with a symmetric (Hermitian) positive definite packed matrix A and multiple right-hand sides.

Syntax

Fortran 77:

```
call sppsv( uplo, n, nrhs, ap, b, ldb, info )
call dppsv( uplo, n, nrhs, ap, b, ldb, info )
call cppsv( uplo, n, nrhs, ap, b, ldb, info )
call zppsv( uplo, n, nrhs, ap, b, ldb, info )
```

Fortran 95:

```
call ppsv( a, b [,uplo] [,info] )
```

Description

This routine solves for X the real or complex system of linear equations $A*X = B$, where A is an n-by-n real symmetric/Hermitian positive-definite matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The Cholesky decomposition is used to factor A as

$A = U^T*U$ (real flavors) and $A = U^H*U$ (complex flavors), if $uplo$ = 'U'

or $A = L*L^T$ (real flavors) and $A = L*L^H$ (complex flavors), if $uplo$ = 'L',

where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of A is then used to solve the system of equations $A*X = B$.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If $uplo$ = 'U', the upper triangle of A is stored. If $uplo$ = 'L', the lower triangle of A is stored.
n	INTEGER. The order of matrix A; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides, the number of columns in B; $nrhs \geq 0$.
ap, *b*	REAL for sppsv DOUBLE PRECISION for dppsv COMPLEX for cppsv DOUBLE COMPLEX for zppsv. Arrays: $ap(*)$, $b(ldb,*)$. The array ap contains the upper or the lower triangular part of the matrix A (as specified by $uplo$) in packed storage (see Matrix Storage Schemes). The dimension of ap must be at least $\max(1, n(n+1)/2)$. The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.
ldb	INTEGER. The first dimension of b; $ldb \geq \max(1, n)$.

Output Parameters

ap	If *info* = 0, the upper or lower triangular part of *A* in packed storage is overwritten by the Cholesky factor *U* or *L*, as specified by *uplo*.
b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, so the factorization could not be completed, and the solution has not been computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ppsv interface are as follows:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).
b	Holds the matrix *B* of size (*n*,*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

?ppsvx

Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive definite packed matrix A, and provides error bounds on the solution.

Syntax

Fortran 77:

```
call sppsvx( fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx, rcond,
ferr, berr, work, iwork, info )

call dppsvx( fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx, rcond,
ferr, berr, work, iwork, info )

call cppsvx( fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx, rcond,
ferr, berr, work, rwork, info )

call zppsvx( fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx, rcond,
ferr, berr, work, rwork, info )
```

Fortran 95:

```
call ppsvx( a, b, x [,uplo] [,af] [,fact] [,equed] [,s] [,ferr] [,berr] [,rcond]
[,info] )
```

Description

This routine uses the Cholesky factorization $A=U^T*U$ (real flavors) / $A=U^H*U$ (complex flavors) or $A=L*L^T$ (real flavors) / $A=L*L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A*X = B$, where A is a n-by-n symmetric or Hermitian positive-definite matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?ppsvx performs the following steps:

1. If *fact* = 'E', real scaling factors s are computed to equilibrate the system:
   *diag(s)*A*diag(s)*inv(diag(s))*X = diag(s)*B*.

Whether or not the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by *diag(s)*A*diag(s)* and *B* by *diag(s)*B*.

2. If *fact* = 'N' or 'E', the Cholesky decomposition is used to factor the matrix *A* (after equilibration if *fact* = 'E') as

   $A = U^T*U$ (real), $A = U^H*U$ (complex), if *uplo* = 'U',

   or $A = L*L^T$ (real), $A = L*L^H$ (complex) , if *uplo* = 'L',

   where *U* is an upper triangular matrix and *L* is a lower triangular matrix.

3. If the leading *i*-by-*i* principal minor is not positive-definite, then the routine returns with *info* = *i*. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than machine precision, *info* = *n*+1 is returned as a warning, but the routine still goes on to solve for *X* and compute error bounds as described below.

4. The system of equations is solved for *X* using the factored form of *A*.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix *X* is premultiplied by *diag(s)* so that it solves the original system before equilibration.

Input Parameters

fact — CHARACTER*1. Must be 'F', 'N', or 'E'.
Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.
If *fact* = 'F': on entry, *afp* contains the factored form of *A*. If *equed* = 'Y', the matrix *A* has been equilibrated with scaling factors given by *s*.

ap and *afp* will not be modified.
If *fact* = 'N', the matrix *A* will be copied to *afp* and factored.
If *fact* = 'E', the matrix *A* will be equilibrated if necessary, then copied to *afp* and factored.

uplo — CHARACTER*1. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of *A* is stored:
If *uplo* = 'U', the upper triangle of *A* is stored.

	If *uplo* = 'L', the lower triangle of *A* is stored.
n	INTEGER. The order of matrix *A*; *n* ≥ 0.
nrhs	INTEGER. The number of right-hand sides; the number of columns in *B*; *nrhs* ≥ 0.
ap,*afp*,*b*,*work*	REAL for sppsvx DOUBLE PRECISION for dppsvx COMPLEX for cppsvx DOUBLE COMPLEX for zppsvx. Arrays: *ap*(*), *afp*(*), *b*(*ldb*,*), *work*(*). The array *ap* contains the upper or lower triangle of the original symmetric/Hermitian matrix *A* in packed storage (see Matrix Storage Schemes). In case when *fact* = 'F' and *equed* = 'Y', *ap* must contain the equilibrated matrix *diag*(*s*)*A**diag*(*s*). The array *afp* is an input argument if *fact* = 'F' and contains the triangular factor *U* or *L* from the Cholesky factorization of *A* in the same storage format as *A*. If *equed* is not 'N', then *afp* is the factored form of the equilibrated matrix *A*. The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. *work*(*) is a workspace array. The dimension of arrays *ap* and *afp* must be at least max(1, *n*(*n*+1)/2); the second dimension of *b* must be at least max(1,*nrhs*); the dimension of *work* must be at least max(1, 3**n*) for real flavors and max(1, 2**n*) for complex flavors.
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).
equed	CHARACTER*1. Must be 'N' or 'Y'. *equed* is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done: if *equed* = 'N', no equilibration was done (always true if *fact* = 'N'); if *equed* = 'Y', equilibration was done and *A* has been replaced by *diag*(*s*)*A**diag*(*s*).
s	REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

Array, DIMENSION (*n*). The array *s* contains the scale factors for *A*. This array is an input argument if *fact* = 'F' only; otherwise it is an output argument.
If *equed* = 'N', *s* is not accessed.
If *fact* = 'F' and *equed* = 'Y', each element of *s* must be positive.

ldx — INTEGER. The first dimension of the output array *x*; *ldx* ≥ max(1, *n*).

iwork — INTEGER. Workspace array, DIMENSION at least max(1, *n*); used in real flavors only.

rwork — REAL for cppsvx;
DOUBLE PRECISION for zppsvx.
Workspace array, DIMENSION at least max(1, *n*); used in complex flavors only.

Output Parameters

x — REAL for sppsvx
DOUBLE PRECISION for dppsvx
COMPLEX for cppsvx
DOUBLE COMPLEX for zppsvx.
Array, DIMENSION (*ldx*,*).
If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the original system of equations. Note that if *equed* = 'Y', *A* and *B* are modified on exit, and the solution to the equilibrated system is inv(*diag*(*s*))**X*. The second dimension of *x* must be at least max(1,*nrhs*).

ap — Array *ap* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'.
If *fact* = 'E' and *equed* = 'Y', *A* is overwritten by *diag*(*s*)**A***diag*(*s*).

afp — If *fact* = 'N' or 'E', then *afp* is an output argument and on exit returns the triangular factor *U* or *L* from the Cholesky factorization $A=U^T*U$ or $A=L*L^T$ (real routines), $A=U^H*U$ or $A=L*L^H$ (complex routines) of the original matrix *A* (if *fact* = 'N'), or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *ap* for the form of the equilibrated matrix.

b	Overwritten by *diag(s)*B* , if *equed* = 'Y'; not changed if *equed* = 'N'.
s	This array is an output argument if *fact* ≠ 'F'. See the description of *s* in Input Arguments section.
rcond	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.
ferr, *berr*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1, *nrhs*). Contain the component-wise forward and relative backward errors, respectively, for each solution vector.
equed	If *fact* ≠ 'F' , then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in Input Arguments section).
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, and *i* ≤ *n*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, so the factorization could not be completed, and the solution and error bounds could not be computed; *rcond* = 0 is returned. If *info* = *i*, and *i* = *n* + 1, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ppsvx interface are as follows:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).
b	Holds the matrix *B* of size (*n*,*nrhs*).
x	Holds the matrix *X* of size (*n*,*nrhs*).
af	Stands for argument *afp* in Fortan 77 interface. Holds the matrix *AF* of size (*n**(*n*+1)/2).
s	Holds the vector of length (*n*). Default value for each element is *s*(*i*) = 1.0_WP.
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
fact	Must be 'N', 'E', or 'F'. The default value is 'N'. If *fact* = 'F', then *af* must be present; otherwise, an error is returned.
equed	Must be 'N' or 'Y'. The default value is 'N'.

?pbsv

Computes the solution to the system of linear equations with a symmetric or Hermitian positive-definite band matrix A and multiple right-hand sides.

Syntax

Fortran 77:

```
call spbsv( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call dpbsv( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call cpbsv( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call zpbsv( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
```

Fortran 95:

```
call pbsv( a, b [,uplo] [,info] )
```

Description

This routine solves for X the real or complex system of linear equations $A*X = B$, where A is an n-by-n symmetric/Hermitian positive definite band matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The Cholesky decomposition is used to factor A as

$A = U^T*U$ (real flavors) and $A = U^H*U$ (complex flavors), if $uplo$ = 'U'

or $A = L*L^T$ (real flavors) and $A = L*L^H$ (complex flavors), if $uplo$ = 'L',

where U is an upper triangular band matrix and L is a lower triangular band matrix, with the same number of superdiagonals or subdiagonals as A. The factored form of A is then used to solve the system of equations $A*X = B$.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If $uplo$ = 'U', the upper triangle of A is stored. If $uplo$ = 'L', the lower triangle of A is stored.
n	INTEGER. The order of matrix A; $n \geq 0$.
kd	INTEGER. The number of superdiagonals of the matrix A if $uplo$ = 'U', or the number of subdiagonals if $uplo$ = 'L'; $kd \geq 0$.
nrhs	INTEGER. The number of right-hand sides, the number of columns in B; $nrhs \geq 0$.
ab, *b*	REAL for spbsv DOUBLE PRECISION for dpbsv COMPLEX for cpbsv DOUBLE COMPLEX for zpbsv. Arrays: $ab(ldab, *)$, $b(ldb,*)$. The array ab contains the upper or the lower triangular part of the matrix A (as specified by $uplo$) in band storage (see Matrix Storage Schemes). The second dimension of ab must be at least max(1, n). The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least max(1,$nrhs$).
ldab	INTEGER. The first dimension of the array ab; $ldab \geq kd + 1$.

ldb INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

ab The upper or lower triangular part of *A* (in band storage) is overwritten by the Cholesky factor *U* or *L*, as specified by *uplo*, in the same storage format as *A*.

b Overwritten by the solution matrix *X*.

info INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, so the factorization could not be completed, and the solution has not been computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine pbsv interface are as follows:

a Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*kd*+1,*n*).

b Holds the matrix *B* of size (*n*,*nrhs*).

uplo Must be 'U' or 'L'. The default value is 'U'.

?pbsvx

Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive-definite band matrix A, and provides error bounds on the solution.

Syntax

Fortran 77:

```
call spbsvx( fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b, ldb,
x, ldx, rcond, ferr, berr, work, iwork, info )
```

```
call dpbsvx( fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b, ldb,
x, ldx, rcond, ferr, berr, work, iwork, info )
```

```
call cpbsvx( fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b, ldb,
x, ldx, rcond, ferr, berr, work, iwork, info )
```

```
call zpbsvx( fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b, ldb,
x, ldx, rcond, ferr, berr, work, iwork, info )
```

Fortran 95:

```
call pbsvx( a, b, x [,uplo] [,af] [,fact] [,equed] [,s] [,ferr] [,berr] [,rcond]
[,info] )
```

Description

This routine uses the Cholesky factorization $A=U^T*U$ (real flavors) / $A=U^H*U$ (complex flavors) or $A=L*L^T$ (real flavors) / $A=L*L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A*X = B$, where A is a n-by-n symmetric or Hermitian positive definite band matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?pbsvx performs the following steps:

1. If $fact = 'E'$, real scaling factors s are computed to equilibrate the system:
   $diag(s)*A*diag(s)*inv(diag(s))*X = diag(s)*B$.

Whether or not the system will be equilibrated depends on the scaling of the matrix A, but if equilibration is used, A is overwritten by $diag(s)*A*diag(s)$ and B by $diag(s)*B$.

2. If $fact$ = 'N' or 'E', the Cholesky decomposition is used to factor the matrix A (after equilibration if $fact$ = 'E') as

   $A = U^T*U$ (real), $A = U^H*U$ (complex), if $uplo$ = 'U',

   or $A = L*L^T$ (real), $A = L*L^H$ (complex) , if $uplo$ = 'L',

   where U is an upper triangular band matrix and L is a lower triangular band matrix.

3. If the leading i-by-i principal minor is not positive definite, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by $diag(s)$ so that it solves the original system before equilibration.

Input Parameters

fact	CHARACTER*1. Must be 'F', 'N', or 'E'. Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored. If $fact$ = 'F': on entry, afb contains the factored form of A. If $equed$ = 'Y', the matrix A has been equilibrated with scaling factors given by s. ab and afb will not be modified. If $fact$ = 'N', the matrix A will be copied to afb and factored. If $fact$ = 'E', the matrix A will be equilibrated if necessary, then copied to afb and factored.
uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If $uplo$ = 'U', the upper triangle of A is stored.

	If *uplo* = 'L', the lower triangle of *A* is stored.
n	INTEGER. The order of matrix *A*; *n* ≥ 0.
kd	INTEGER. The number of superdiagonals or subdiagonals in the matrix *A*; *kd* ≥ 0.
nrhs	INTEGER. The number of right-hand sides, the number of columns in *B*; *nrhs* ≥ 0.
ab,*afb*,*b*,*work*	REAL for spbsvx DOUBLE PRECISION for dpbsvx COMPLEX for cpbsvx DOUBLE COMPLEX for zpbsvx. Arrays: *ab*(*ldab*,*), *afb*(*ldab*,*), *b*(*ldb*,*), *work*(*). The array *ab* contains the upper or lower triangle of the matrix *A* in band storage (see Matrix Storage Schemes). If *fact* = 'F' and *equed* = 'Y', then *ab* must contain the equilibrated matrix *diag*(*s*)***A***diag*(*s*). The second dimension of *ab* must be at least max(1, *n*). The array *afb* is an input argument if *fact* = 'F'. It contains the triangular factor *U* or *L* from the Cholesky factorization of the band matrix *A* in the same storage format as *A*. If *equed* = 'Y', then *afb* is the factored form of the equilibrated matrix *A*. The second dimension of *afb* must be at least max(1, *n*). The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1, *nrhs*). *work*(*) is a workspace array. The dimension of *work* must be at least max(1,3**n*) for real flavors, and at least max(1,2**n*) for complex flavors.
ldab	INTEGER. The first dimension of *ab*; *ldab* ≥ *kd*+1.
ldafb	INTEGER. The first dimension of *afb*; *ldafb* ≥ *kd*+1.
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).
equed	CHARACTER*1. Must be 'N' or 'Y'. *equed* is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done: if *equed* = 'N', no equilibration was done (always true if *fact* = 'N')

if *equed* = 'Y', equilibration was done and *A* has been replaced by *diag(s)*A*diag(s)*.

s — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Array, DIMENSION (*n*). The array *s* contains the scale factors for *A*. This array is an input argument if *fact* = 'F' only; otherwise it is an output argument.
If *equed* = 'N', *s* is not accessed.
If *fact* = 'F' and *equed* = 'Y', each element of *s* must be positive.

ldx — INTEGER. The first dimension of the output array *x*; *ldx* ≥ max(1, *n*).

iwork — INTEGER. Workspace array, DIMENSION at least max(1, *n*); used in real flavors only.

rwork — REAL for cpbsvx
DOUBLE PRECISION for zpbsvx.
Workspace array, DIMENSION at least max(1, *n*); used in complex flavors only.

Output Parameters

x — REAL for spbsvx
DOUBLE PRECISION for dpbsvx
COMPLEX for cpbsvx
DOUBLE COMPLEX for zpbsvx.
Array, DIMENSION (*ldx*,*).
If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the original system of equations. Note that if *equed* = 'Y', *A* and *B* are modified on exit, and the solution to the equilibrated system is inv(*diag(s)*)**X*. The second dimension of *x* must be at least max(1,*nrhs*).

ab — On exit, if *fact* = 'E' and *equed* = 'Y', *A* is overwritten by *diag(s)*A*diag(s)*.

afb — If *fact* = 'N' or 'E', then *afb* is an output argument and on exit returns the triangular factor *U* or *L* from the Cholesky factorization $A=U^T*U$ or $A=L*L^T$ (real routines), $A=U^H*U$ or $A=L*L^H$ (complex routines)

	of the original matrix *A* (if *fact* = 'N'), or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *ab* for the form of the equilibrated matrix.
b	Overwritten by *diag(s)*B* , if *equed* = 'Y'; not changed if *equed* = 'N'.
s	This array is an output argument if *fact* ≠ 'F'. See the description of *s* in Input Arguments section.
rcond	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.
ferr, *berr*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1, *nrhs*). Contain the component-wise forward and relative backward errors, respectively, for each solution vector.
equed	If *fact* ≠ 'F' , then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in Input Arguments section).
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, and *i* ≤ *n*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive definite, so the factorization could not be completed, and the solution and error bounds could not be computed; *rcond* =0 is returned. If *info* = *i*, and *i* = *n* + 1, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `pbsvx` interface are as follows:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*kd*+1, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).
af	Stands for argument *afb* in Fortan 77 interface. Holds the array *AF* of size (*kd*+1, *n*).
s	Holds the vector of length (*n*). Default value for each element is *s*(*i*) = 1.0_WP.
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
fact	Must be 'N', 'E', or 'F'. The default value is 'N'. If *fact* = 'F', then *af* must be present; otherwise, an error is returned.
equed	Must be 'N' or 'Y'. The default value is 'N'.

?ptsv

Computes the solution to the system of linear equations with a symmetric or Hermitian positive definite tridiagonal matrix A and multiple right-hand sides.

Syntax

Fortran 77:

```
call sptsv( n, nrhs, d, e, b, ldb, info )
call dptsv( n, nrhs, d, e, b, ldb, info )
call cptsv( n, nrhs, d, e, b, ldb, info )
call zptsv( n, nrhs, d, e, b, ldb, info )
```

Fortran 95:

```
call ptsv( d, e, b [,info] )
```

Description

This routine solves for X the real or complex system of linear equations $A*X = B$, where A is an n-by-n symmetric/Hermitian positive-definite tridiagonal matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

A is factored as $A = L*D*L^T$ (real flavors) or $A = L*D*L^H$ (complex flavors), and the factored form of A is then used to solve the system of equations $A*X = B$.

Input Parameters

n	INTEGER. The order of matrix A; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides, the number of columns in B; $nrhs \geq 0$.
d	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, dimension at least max(1, *n*). Contains the diagonal elements of the tridiagonal matrix A.
e, *b*	REAL for sptsv DOUBLE PRECISION for dptsv COMPLEX for cptsv DOUBLE COMPLEX for zptsv. Arrays: *e*(*n* - 1) , *b*(*ldb*,*). The array *e* contains the (*n* - 1) subdiagonal elements of A. The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

d	Overwritten by the n diagonal elements of the diagonal matrix D from the $L*D*L^T$ (real)/ $L*D*L^H$ (complex) factorization of A.
e	Overwritten by the (*n* - 1) subdiagonal elements of the unit bidiagonal factor L from the factorization of A.

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, and the solution has not been computed. The factorization has not been completed unless *i* = *n*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ptsv interface are as follows:

d	Holds the vector of length (*n*).
e	Holds the vector of length (*n*-1).
b	Holds the matrix *B* of size (*n*,*nrhs*).

?ptsvx

Uses factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive definite tridiagonal matrix A, and provides error bounds on the solution.

Syntax

Fortran 77:

```
call sptsvx( fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr, berr,
work, info )
call dptsvx( fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr, berr,
work, info )
call cptsvx( fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr, berr,
work, rwork, info )
call zptsvx( fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr, berr,
work, rwork, info )
```

Fortran 95:

```
call ptsvx( d, e, b, x [,df] [,ef] [,fact] [,ferr] [,berr] [,rcond] [,info] )
```

Description

This routine uses the Cholesky factorization $A = L*D*L^T$ (real)/$A = L*D*L^H$ (complex) to compute the solution to a real or complex system of linear equations $A*X = B$, where A is a n-by-n symmetric or Hermitian positive definite tridiagonal matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?ptsvx` performs the following steps:

1. If $fact = 'N'$, the matrix A is factored as $A = L*D*L^T$ (real flavors)/$A = L*D*L^H$ (complex flavors), where L is a unit lower bidiagonal matrix and D is diagonal. The factorization can also be regarded as having the form $A = U^T*D*U$ (real flavors)/$A = U^H*D*U$ (complex flavors).
2. If the leading i-by-i principal minor is not positive-definite, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A.
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

fact	`CHARACTER*1`. Must be `'F'` or `'N'`. Specifies whether or not the factored form of the matrix A is supplied on entry. If $fact = 'F'$: on entry, df and ef contain the factored form of A. Arrays d, e, df, and ef will not be modified. If $fact = 'N'$, the matrix A will be copied to df and ef, and factored.
n	`INTEGER`. The order of matrix A; $n \geq 0$.
nrhs	`INTEGER`. The number of right-hand sides, the number of columns in B; $nrhs \geq 0$.

d, *df*, *rwork*
REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Arrays: *d*(*n*), *df*(*n*), *rwork*(*n*).
The array *d* contains the *n* diagonal elements of the tridiagonal matrix *A*.
The array *df* is an input argument if *fact* = 'F' and on entry contains the *n* diagonal elements of the diagonal matrix *D* from the $L*D*L^T$ (real)/ $L*D*L^H$ (complex) factorization of *A*.
The array *rwork* is a workspace array used for complex flavors only.

e,*ef*,*b*,*work*
REAL for sptsvx
DOUBLE PRECISION for dptsvx
COMPLEX for cptsvx
DOUBLE COMPLEX for zptsvx.
Arrays: *e*(*n* -1), *ef*(*n* -1), *b*(*ldb**), *work*(*). The array *e* contains the (*n* - 1) subdiagonal elements of the tridiagonal matrix *A*.
The array *ef* is an input argument if *fact* = 'F' and on entry contains the (*n* - 1) subdiagonal elements of the unit bidiagonal factor *L* from the $L*D*L^T$ (real)/ $L*D*L^H$ (complex) factorization of *A*.
The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.
The array *work* is a workspace array. The dimension of *work* must be at least 2**n* for real flavors, and at least *n* for complex flavors.

ldb
INTEGER. The leading dimension of *b*; *ldb* ≥ max(1, *n*).

ldx
INTEGER. The leading dimension of *x*; *ldx* ≥ max(1, *n*).

Output Parameters

x
REAL for sptsvx
DOUBLE PRECISION for dptsvx
COMPLEX for cptsvx
DOUBLE COMPLEX for zptsvx.
Array, DIMENSION (*ldx*,*).
If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the system of equations. The second dimension of *x* must be at least max(1,*nrhs*).

df, *ef*
These arrays are output arguments if *fact* = 'N'. See the description of *df*, *ef* in Input Arguments section.

rcond	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.
ferr, *berr*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1, *nrhs*). Contain the component-wise forward and relative backward errors, respectively, for each solution vector.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, and *i* ≤ *n*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, so the factorization could not be completed, and the solution and error bounds could not be computed; *rcond* =0 is returned. If *info* = *i*, and *i* = *n* + 1, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ptsvx interface are as follows:

d	Holds the vector of length (*n*).
e	Holds the vector of length (*n*-1).
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).
df	Holds the vector of length (*n*).
ef	Holds the vector of length (*n*-1).
ferr	Holds the vector of length (*nrhs*).

berr Holds the vector of length (*nrhs*).

fact Must be 'N' or 'F'. The default value is 'N'. If *fact* = 'F', then both arguments *af* and *ipiv* must be present; otherwise, an error is returned.

?sysv

Computes the solution to the system of linear equations with a real or complex symmetric matrix A and multiple right-hand sides.

Syntax

Fortran 77:

```
call ssysv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call dsysv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call csysv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call zsysv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
```

Fortran 95:

```
call sysv( a, b [,uplo] [,ipiv] [,info] )
```

Description

This routine solves for *X* the real or complex system of linear equations *A*X = B*, where *A* is an *n*-by-*n* symmetric matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

The diagonal pivoting method is used to factor *A* as $A = U*D*U^T$ or $A = L*D*L^T$, where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of *A* is then used to solve the system of equations *A*X = B*.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of *A* is stored:
If *uplo* = 'U', the upper triangle of *A* is stored.

If *uplo* = 'L', the lower triangle of *A* is stored.

n — INTEGER. The order of matrix *A*; *n* ≥ 0.

nrhs — INTEGER. The number of right-hand sides; the number of columns in *B*; *nrhs* ≥ 0.

a, *b*, *work* — REAL for ssysv
DOUBLE PRECISION for dsysv
COMPLEX for csysv
DOUBLE COMPLEX for zsysv.
Arrays: *a*(*lda*,*), *b*(*ldb*,*), *work*(*).
The array *a* contains the upper or the lower triangular part of the symmetric matrix *A* (see *uplo*). The second dimension of *a* must be at least max(1, *n*).
The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).
work is a workspace array, dimension at least max(1,*lwork*).

lda — INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).

ldb — INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

lwork — INTEGER. The size of the *work* array; *lwork* ≥ 1.
If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes below for details and for the suggested value of *lwork*.

Output Parameters

a — If *info* = 0, *a* is overwritten by the block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L*) from the factorization of *A* as computed by ?sytrf.

b — If *info* = 0, *b* is overwritten by the solution matrix *X*.

ipiv — INTEGER.
Array, DIMENSION at least max(1, *n*). Contains details of the interchanges and the block structure of *D*, as determined by ?sytrf.

If *ipiv*(*i*) = *k* >0, then d_{ii} is a 1-by-1 diagonal block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.
If *uplo* = 'U' and *ipiv*(*i*) = *ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and (*i*–1)-th row and column of *A* was interchanged with the *m*–th row and column.
If *uplo* = 'L' and *ipiv*(*i*) = *ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER. If *info* = 0, the execution is successful.
If *info* = –*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular, so the solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine sysv interface are as follows:

a — Holds the matrix *A* of size (*n*, *n*).

b — Holds the matrix *B* of size (*n*, *nrhs*).

ipiv — Holds the vector of length (*n*).

uplo — Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For better performance, try using *lwork* = *n*blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?sysvx

Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a real or complex symmetric matrix A, and provides error bounds on the solution.

Syntax

Fortran 77:

```
call ssysvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx,
rcond, ferr, berr, work, lwork, iwork, info )

call dsysvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx,
rcond, ferr, berr, work, lwork, iwork, info )

call csysvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx,
rcond, ferr, berr, work, lwork, rwork, info )

call zsysvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx,
rcond, ferr, berr, work, lwork, rwork, info )
```

Fortran 95:

```
call sysvx( a, b, x [,uplo] [,af] [,ipiv] [,fact] [,ferr] [,berr] [,rcond]
[,info] )
```

Description

This routine uses the diagonal pivoting factorization to compute the solution to a real or complex system of linear equations *A*X = B*, where *A* is a *n*-by-*n* symmetric matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?sysvx performs the following steps:

1. If *fact* = 'N', the diagonal pivoting method is used to factor the matrix *A*. The form of the factorization is $A = U*D*U^T$ or $A = L*D*L^T$, where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
2. If some $d_{i,i}$= 0, so that *D* is exactly singular, then the routine returns with *info* = *i*. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than machine precision, *info* = *n*+1 is returned as a warning, but the routine still goes on to solve for *X* and compute error bounds as described below.
3. The system of equations is solved for *X* using the factored form of *A*.
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

fact — CHARACTER*1. Must be 'F' or 'N'.
Specifies whether or not the factored form of the matrix *A* has been supplied on entry.
If *fact* = 'F': on entry, *af* and *ipiv* contain the factored form of *A*. Arrays *a*, *af*, and *ipiv* will not be modified.
If *fact* = 'N', the matrix *A* will be copied to *af* and factored.

uplo — CHARACTER*1. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of *A* is stored:
If *uplo* = 'U', the upper triangle of *A* is stored.
If *uplo* = 'L', the lower triangle of *A* is stored.

n — INTEGER. The order of matrix *A*; $n \geq 0$.

nrhs — INTEGER. The number of right-hand sides, the number of columns in *B*; $nrhs \geq 0$.

a,*af*,*b*,*work* — REAL for ssysvx
DOUBLE PRECISION for dsysvx
COMPLEX for csysvx
DOUBLE COMPLEX for zsysvx.
Arrays: *a*(*lda*,*), *af*(*ldaf*,*), *b*(*ldb*,*), *work*(*).
The array *a* contains the upper or the lower triangular part of the symmetric matrix *A* (see *uplo*). The second dimension of *a* must be at least max(1,*n*).

The array *af* is an input argument if *fact* = 'F'. It contains he block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* from the factorization $A = U*D*U^T$ or $A = L*D*L^T$ as computed by ?sytrf. The second dimension of *af* must be at least max(1,*n*).
The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1, *nrhs*).
work(*) is a workspace array, dimension at least max(1,*lwork*).

lda — INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).

ldaf — INTEGER. The first dimension of *af*; *ldaf* ≥ max(1, *n*).

ldb — INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

ipiv — INTEGER.
Array, DIMENSION at least max(1, *n*). The array *ipiv* is an input argument if *fact* = 'F'. It contains details of the interchanges and the block structure of *D*, as determined by ?sytrf.
If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 diagonal block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.
If *uplo* = 'U' and *ipiv*(*i*) = *ipiv*(*i*–1) = –*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and (*i*–1)-th row and column of *A* was interchanged with the *m*-th row and column.
If *uplo* = 'L' and *ipiv*(*i*) = *ipiv*(*i*+1) = –*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.

ldx — INTEGER. The leading dimension of the output array *x*; *ldx* ≥ max(1, *n*).

lwork — INTEGER. The size of the *work* array.
If *lwork* = –1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes below for details and for the suggested value of *lwork*.

iwork — INTEGER. Workspace array, DIMENSION at least max(1, *n*); used in real flavors only.

rwork — REAL for csysvx;
DOUBLE PRECISION for zsysvx.

Workspace array, DIMENSION at least max(1, *n*); used in complex flavors only.

Output Parameters

x
REAL for ssysvx
DOUBLE PRECISION for dsysvx
COMPLEX for csysvx
DOUBLE COMPLEX for zsysvx.
Array, DIMENSION (*ldx*,*).
If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the system of equations. The second dimension of *x* must be at least max(1,*nrhs*).

af, *ipiv*
These arrays are output arguments if *fact* = 'N'.
See the description of *af*, *ipiv* in Input Arguments section.

rcond
REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal condition number of the matrix *A*. If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr, *berr*
REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least max(1, *nrhs*). Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

work(1)
If *info*=0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info
INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, and $i \le n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *i*, and *i* = *n* + 1, then *D* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `sysvx` interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).
af	Holds the matrix *AF* of size (*n*, *n*).
ipiv	Holds the vector of length (*n*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
fact	Must be `'N'` or `'F'`. The default value is `'N'`. If *fact* = `'F'`, then both arguments *af* and *ipiv* must be present; otherwise, an error is returned.

Application Notes

For real flavors, *lwork* must be at least 3**n*, and for complex flavors at least 2**n*. For better performance, try using *lwork* = *n***blocksize*, where *blocksize* is the optimal block size for `?sytrf`.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*`(1)`) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hesv

Computes the solution to the system of linear equations with a Hermitian matrix A and multiple right-hand sides.

Syntax

Fortran 77:

```
call chesv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call zhesv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
```

Fortran 95:

```
call hesv( a, b [,uplo] [,ipiv] [,info] )
```

Description

This routine solves for *X* the complex system of linear equations *A*X = B*, where *A* is an *n*-by-*n* symmetric matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

The diagonal pivoting method is used to factor *A* as $A = U*D*U^H$ or $A = L*D*L^H$, where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of *A* is then used to solve the system of equations *A*X = B*.

Input Parameters

uplo — CHARACTER*1. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of *A* is stored and how *A* is factored:

If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix *A*, and *A* is factored as $U*D*U^H$.
If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix *A*, and *A* is factored as $L*D*L^H$.

n — INTEGER. The order of matrix *A*; $n \geq 0$.

nrhs — INTEGER. The number of right-hand sides, the number of columns in *B*; $nrhs \geq 0$.

a, *b*, *work* — COMPLEX for chesv
DOUBLE COMPLEX for zhesv.
Arrays: *a*(*lda*,*), *b*(*ldb*,*), *work*(*). The array *a* contains the upper or the lower triangular part of the Hermitian matrix *A* (see *uplo*). The second dimension of *a* must be at least max(1, *n*).
The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).
work is a workspace array, dimension at least max(1,*lwork*).

lda — INTEGER. The first dimension of *a*; $lda \geq \max(1, n)$.

ldb — INTEGER. The first dimension of *b*; $ldb \geq \max(1, n)$.

lwork — INTEGER. The size of the *work* array ($lwork \geq 1$).
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes below for details and for the suggested value of *lwork*.

Output Parameters

a — If *info* = 0, *a* is overwritten by the block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L*) from the factorization of *A* as computed by ?hetrf.

b — If *info* = 0, *b* is overwritten by the solution matrix *X*.

ipiv — INTEGER.
Array, DIMENSION at least max(1, *n*). Contains details of the interchanges and the block structure of *D*, as determined by ?hetrf.

If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 diagonal block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.
If *uplo* = 'U' and *ipiv*(*i*) = *ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and (*i*–1)-th row and column of *A* was interchanged with the *m*-th row and column.
If *uplo* = 'L' and *ipiv*(*i*) = *ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)–th row and column of *A* was interchanged with the *m*-th row and column.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER. If *info* = 0, the execution is successful.
If *info* = –*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular, so the solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hesv interface are as follows:

a — Holds the matrix *A* of size (*n*, *n*).

b — Holds the matrix *B* of size (*n*, *nrhs*).

ipiv — Holds the vector of length (*n*).

uplo — Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For better performance, try using *lwork* = *n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hesvx

Uses the diagonal pivoting factorization to compute the solution to the complex system of linear equations with a Hermitian matrix A, and provides error bounds on the solution.

Syntax

Fortran 77:

```
call chesvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx,
rcond, ferr, berr, work, lwork, rwork, info )
```

```
call zhesvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx,
rcond, ferr, berr, work, lwork, rwork, info )
```

Fortran 95:

```
call hesvx( a, b, x [,uplo] [,af] [,ipiv] [,fact] [,ferr] [,berr] [,rcond]
[,info] )
```

Description

This routine uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations *A*X* = *B*, where *A* is an *n*-by-*n* Hermitian matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?hesvx performs the following steps:

1. If *fact* = 'N', the diagonal pivoting method is used to factor the matrix *A*. The form of the factorization is $A = U*D*U^H$ or $A = L*D*L^H$, where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A.
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

fact	CHARACTER*1. Must be 'F' or 'N'. Specifies whether or not the factored form of the matrix A has been supplied on entry. If $fact = 'F'$: on entry, af and $ipiv$ contain the factored form of A. Arrays a, af, and $ipiv$ are not modified. If $fact = 'N'$, the matrix A is copied to af and factored.
uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored and how A is factored: If $uplo = 'U'$, the array a stores the upper triangular part of the Hermitian matrix A, and A is factored as $U*D*U^H$. If $uplo = 'L'$, the array a stores the lower triangular part of the Hermitian matrix A; A is factored as $L*D*L^H$.
n	INTEGER. The order of matrix A; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides, the number of columns in B; $nrhs \geq 0$.
a,*af*,*b*,*work*	COMPLEX for chesvx DOUBLE COMPLEX for zhesvx. Arrays: $a(lda,*)$, $af(ldaf,*)$, $b(ldb,*)$, $work(*)$. The array a contains the upper or the lower triangular part of the Hermitian matrix A (see $uplo$). The second dimension of a must be at least $\max(1,n)$.

The array *af* is an input argument if *fact* = 'F'. It contains he block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* from the factorization $A = U*D*U^H$ or $A = L*D*L^H$ as computed by ?hetrf. The second dimension of *af* must be at least max(1,*n*).
The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1, *nrhs*).
work(*) is a workspace array of dimension at least max(1, *lwork*).

lda — INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).

ldaf — INTEGER. The first dimension of *af*; *ldaf* ≥ max(1, *n*).

ldb — INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

ipiv — INTEGER.
Array, DIMENSION at least max(1, *n*). The array *ipiv* is an input argument if *fact* = 'F'. It contains details of the interchanges and the block structure of *D*, as determined by ?hetrf.
If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 diagonal block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.
If *uplo* = 'U' and *ipiv*(*i*) = *ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and (*i*–1)-th row and column of *A* was interchanged with the *m*-th row and column.
If *uplo* = 'L' and *ipiv*(*i*) = *ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.

ldx — INTEGER. The leading dimension of the output array *x*; *ldx* ≥ max(1, *n*).

lwork — INTEGER. The size of the *work* array.
If *lwork* = –1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes below for details and for the suggested value of *lwork*.

rwork — REAL for chesvx
DOUBLE PRECISION for zhesvx.
Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

x	COMPLEX for chesvx DOUBLE COMPLEX for zhesvx. Array, DIMENSION (*ldx*,*). If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the system of equations. The second dimension of *x* must be at least max(1,*nrhs*).
af, *ipiv*	These arrays are output arguments if *fact* = 'N'. See the description of *af*, *ipiv* in Input Arguments section.
rcond	REAL for chesvx DOUBLE PRECISION for zhesvx. An estimate of the reciprocal condition number of the matrix *A*. If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.
ferr, *berr*	REAL for chesvx DOUBLE PRECISION for zhesvx. Arrays, DIMENSION at least max(1, *nrhs*). Contain the component-wise forward and relative backward errors, respectively, for each solution vector.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, and *i* ≤ *n*, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned. If *info* = *i*, and *i* = *n* + 1, then *D* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `hesvx` interface are as follows:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).
af	Holds the matrix *AF* of size (*n*, *n*).
ipiv	Holds the vector of length (*n*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
fact	Must be `'N'` or `'F'`. The default value is `'N'`. If *fact* = `'F'`, then both arguments *af* and *ipiv* must be present; otherwise, an error is returned.

Application Notes

The value of *lwork* must be at least `2*`*n*. For better performance, try using *lwork* = *n***blocksize*, where *blocksize* is the optimal block size for `?hetrf`.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*`(1)`) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?spsv

Computes the solution to the system of linear equations with a real or complex symmetric matrix A stored in packed format, and multiple right-hand sides.

Syntax

Fortran 77:

```
call sspsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call dspsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call cspsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zspsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
```

Fortran 95:

```
call spsv( a, b [,uplo] [,ipiv] [,info] )
```

Description

This routine solves for X the real or complex system of linear equations $A*X = B$, where A is an n-by-n symmetric matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The diagonal pivoting method is used to factor A as $A = U*D*U^T$ or $A = L*D*L^T$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of A is then used to solve the system of equations $A*X = B$.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If *uplo* = 'U', the upper triangle of A is stored. If *uplo* = 'L', the lower triangle of A is stored.
n	INTEGER. The order of matrix A; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides, the number of columns in B; $nrhs \geq 0$.

ap, *b*	REAL for sspsv DOUBLE PRECISION for dspsv COMPLEX for cspsv DOUBLE COMPLEX for zspsv. Arrays: *ap*(*), *b*(*ldb*,*). The dimension of *ap* must be at least max(1,*n*(*n*+1)/2). The array *ap* contains the factor *U* or *L*, as specified by *uplo*, in packed storage (see Matrix Storage Schemes). The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1,*nrhs*).
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

ap	The block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L*) from the factorization of *A* as computed by ?sptrf, stored as a packed triangular matrix in the same storage format as *A*.
b	If *info* = 0, *b* is overwritten by the solution matrix *X*.
ipiv	INTEGER. Array, DIMENSION at least max(1, *n*). Contains details of the interchanges and the block structure of *D*, as determined by ?sptrf. If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column. If *uplo* = 'U' and *ipiv*(*i*) = *ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and (*i*–1)-th row and column of *A* was interchanged with the *m*-th row and column. If *uplo* = 'L' and *ipiv*(*i*) = *ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = –*i*, the *i*-th parameter had an illegal value. If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular, so the solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `spsv` interface are as follows:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).
b	Holds the matrix *B* of size (*n*,*nrhs*).
ipiv	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

?spsvx

Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a real or complex symmetric matrix A stored in packed format, and provides error bounds on the solution.

Syntax

Fortran 77:

```
call sspsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr,
berr, work, iwork, info )

call dspsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr,
berr, work, iwork, info )

call cspsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr,
berr, work, rwork, info )

call zspsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr,
berr, work, rwork, info )
```

Fortran 95:

```
call spsvx( a, b, x [,uplo] [,af] [,ipiv] [,fact] [,ferr] [,berr] [,rcond]
[,info] )
```

Description

This routine uses the diagonal pivoting factorization to compute the solution to a real or complex system of linear equations *A*X = B*, where *A* is a *n*-by-*n* symmetric matrix stored in packed format, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?spsvx performs the following steps:

1. If *fact = 'N'*, the diagonal pivoting method is used to factor the matrix A. The form of the factorization is $A = U*D*U^T$ or $A = L*D*L^T$, where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
2. If some $d_{i,i} = 0$, so that *D* is exactly singular, then the routine returns with *info = i*. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than machine precision, *info = n*+1 is returned as a warning, but the routine still goes on to solve for *X* and compute error bounds as described below.
3. The system of equations is solved for *X* using the factored form of *A*.
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

fact — CHARACTER*1. Must be 'F' or 'N'.
Specifies whether or not the factored form of the matrix *A* has been supplied on entry.
If *fact = 'F'*: on entry, *afp* and *ipiv* contain the factored form of *A*. Arrays *ap*, *afp*, and *ipiv* are not modified.
If *fact = 'N'*, the matrix *A* is copied to *afp* and factored.

uplo — CHARACTER*1. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of *A* is stored and how *A* is factored:
If *uplo = 'U'*, the array *ap* stores the upper triangular part of the symmetric matrix *A*, and *A* is factored as $U*D*U^T$.
If *uplo = 'L'*, the array *ap* stores the lower triangular part of the symmetric matrix *A*; *A* is factored as $L*D*L^T$.

n INTEGER. The order of matrix *A*; $n \geq 0$.

nrhs INTEGER. The number of right-hand sides, the number of columns in *B*; $nrhs \geq 0$.

ap,*afp*,*b*,*work* REAL for sspsvx
DOUBLE PRECISION for dspsvx
COMPLEX for cspsvx
DOUBLE COMPLEX for zspsvx.
Arrays: *ap*(*), *afp*(*), *b*(*ldb*,*), *work*(*).
The array *ap* contains the upper or lower triangle of the symmetric matrix *A* in packed storage (see Matrix Storage Schemes).
The array *afp* is an input argument if *fact* = 'F'. It contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* from the factorization $A = U*D*U^T$ or $A = L*D*L^T$ as computed by ?sptrf, in the same storage format as *A*.
The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.
work(*) is a workspace array.
The dimension of arrays *ap* and *afp* must be at least max(1, *n*(*n*+1)/2); the second dimension of *b* must be at least max(1,*nrhs*); the dimension of *work* must be at least max(1,3**n*) for real flavors and max(1,2**n*) for complex flavors.

ldb INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

ipiv INTEGER.
Array, DIMENSION at least max(1, *n*). The array *ipiv* is an input argument if *fact* = 'F'. It contains details of the interchanges and the block structure of *D*, as determined by ?sptrf.
If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 diagonal block, and the *i*-th row and column of *A* was interchanged with the *k*–th row and column.
If *uplo* = 'U' and *ipiv*(*i*) =*ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and (*i*–1)-th row and column of *A* was interchanged with the *m*-th row and column.
If *uplo* = 'L' and *ipiv*(*i*) =*ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.

ldx INTEGER. The leading dimension of the output array *x*; *ldx* ≥ max(1, *n*).

iwork INTEGER. Workspace array, DIMENSION at least max(1, *n*); used in real flavors only.

rwork REAL for cspsvx
DOUBLE PRECISION for zspsvx.
Workspace array, DIMENSION at least max(1, *n*); used in complex flavors only.

Output Parameters

x REAL for sspsvx
DOUBLE PRECISION for dspsvx
COMPLEX for cspsvx
DOUBLE COMPLEX for zspsvx.
Array, DIMENSION (*ldx*,*).
If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the system of equations. The second dimension of *x* must be at least max(1,*nrhs*).

afp, *ipiv* These arrays are output arguments if *fact* = 'N'. See the description of *afp*, *ipiv* in Input Arguments section.

rcond REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal condition number of the matrix *A*. If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr, *berr* REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least max(1, *nrhs*). Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

info INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, and $i \le n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *i*, and *i* = *n* + 1, then *D* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine spsvx interface are as follows:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).
b	Holds the matrix *B* of size (*n*, *nrhs*).
x	Holds the matrix *X* of size (*n*, *nrhs*).
af	Stands for argument *afp* in Fortan 77 interface. Holds the array *AF* of size (*n**(*n*+1)/2).
ipiv	Holds the vector of length (*n*).
ferr	Holds the vector of length (*nrhs*).
berr	Holds the vector of length (*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
fact	Must be 'N' or 'F'. The default value is 'N'. If *fact* = 'F', then both arguments *af* and *ipiv* must be present; otherwise, an error is returned.

?hpsv

Computes the solution to the system of linear equations with a Hermitian matrix A stored in packed format, and multiple right-hand sides.

Syntax

Fortran 77:

```
call chpsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zhpsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
```

Fortran 95:

```
call hpsv( a, b [,uplo] [,ipiv] [,info] )
```

Description

This routine solves for X the system of linear equations $A*X = B$, where A is an n-by-n Hermitian matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The diagonal pivoting method is used to factor A as $A = U*D*U^H$ or $A = L*D*L^H$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of A is then used to solve the system of equations $A*X = B$.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If *uplo* = 'U', the upper triangle of A is stored. If *uplo* = 'L', the lower triangle of A is stored.
n	INTEGER. The order of matrix A; $n \geq 0$.
nrhs	INTEGER. The number of right-hand sides; the number of columns in B; $nrhs \geq 0$.
ap, *b*	COMPLEX for chpsv DOUBLE COMPLEX for zhpsv. Arrays: *ap*(*), *b*(*ldb*,*).

The dimension of *ap* must be at least max(1, *n*(*n*+1)/2). The array *ap* contains the factor *U* or *L*, as specified by *uplo*, in packed storage (see Matrix Storage Schemes).
The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least max(1, *nrhs*).

ldb — INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).

Output Parameters

ap — The block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L*) from the factorization of *A* as computed by ?hptrf, stored as a packed triangular matrix in the same storage format as *A*.

b — If *info* = 0, *b* is overwritten by the solution matrix *X*.

ipiv — INTEGER.
Array, DIMENSION at least max(1, *n*). Contains details of the interchanges and the block structure of *D*, as determined by ?hptrf.
If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.
If *uplo* = 'U' and *ipiv*(*i*) = *ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and (*i*–1)-th row and column of *A* was interchanged with the *m*-th row and column.
If *uplo* = 'L' and *ipiv*(*i*) = *ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.

info — INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular, so the solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hpsv interface are as follows:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).
b	Holds the matrix *B* of size (*n*, *nrhs*).
ipiv	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

?hpsvx

Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a Hermitian matrix A stored in packed format, and provides error bounds on the solution.

Syntax

Fortran 77:

```
call chpsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info )
```

```
call zhpsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call hpsvx( a, b, x [,uplo] [,af] [,ipiv] [,fact] [,ferr] [,berr] [,rcond] [,info] )
```

Description

This routine uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations *A*X = B*, where *A* is a *n*-by-*n* Hermitian matrix stored in packed format, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?hpsvx performs the following steps:

1. If *fact* = 'N', the diagonal pivoting method is used to factor the matrix *A*. The form of the factorization is $A = U*D*U^H$ or $A = L*D*L^H$, where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is a Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with *info* = *i*. Otherwise, the factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, *info* = *n*+1 is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A.
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

fact
CHARACTER*1. Must be 'F' or 'N'.
Specifies whether or not the factored form of the matrix A has been supplied on entry.
If *fact* = 'F': on entry, *afp* and *ipiv* contain the factored form of A. Arrays *ap*, *afp*, and *ipiv* are not modified.
If *fact* = 'N', the matrix A is copied to *afp* and factored.

uplo
CHARACTER*1. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of A is stored and how A is factored:
If *uplo* = 'U', the array *ap* stores the upper triangular part of the Hermitian matrix A, and A is factored as $U*D*U^H$.
If *uplo* = 'L', the array *ap* stores the lower triangular part of the Hermitian matrix A, and A is factored as $L*D*L^H$.

n
INTEGER. The order of matrix A; $n \geq 0$.

nrhs
INTEGER. The number of right-hand sides, the number of columns in B; $nrhs \geq 0$.

ap,*afp*,*b*,*work*
COMPLEX for chpsvx
DOUBLE COMPLEX for zhpsvx.
Arrays: *ap*(*), *afp*(*), *b*(*ldb*,*), *work*(*).
The array *ap* contains the upper or lower triangle of the Hermitian matrix A in packed storage (see Matrix Storage Schemes).
The array *afp* is an input argument if *fact* = 'F'. It contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization $A = U*D*U^H$ or $A = L*D*L^H$ as computed by ?hptrf, in the same storage format as A.

	The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. *work*(*) is a workspace array. The dimension of arrays *ap* and *afp* must be at least max(1,*n*(*n*+1)/2); the second dimension of *b* must be at least max(1,*nrhs*); the dimension of *work* must be at least max(1,2**n*).
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ max(1, *n*).
ipiv	INTEGER. Array, DIMENSION at least max(1, *n*). The array *ipiv* is an input argument if *fact* = 'F'. It contains details of the interchanges and the block structure of *D*, as determined by ?hptrf. If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 diagonal block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column. If *uplo* = 'U' and *ipiv*(*i*) = *ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*–1, and (*i*–1)-th row and column of *A* was interchanged with the *m*-th row and column. If *uplo* = 'L' and *ipiv*(*i*) = *ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of *A* was interchanged with the *m*–th row and column.
ldx	INTEGER. The leading dimension of the output array *x*; *ldx* ≥ max(1, *n*).
rwork	REAL for chpsvx DOUBLE PRECISION for zhpsvx. Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

x	COMPLEX for chpsvx DOUBLE COMPLEX for zhpsvx. Array, DIMENSION (*ldx*,*). If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the system of equations. The second dimension of *x* must be at least max(1,*nrhs*).
afp, *ipiv*	These arrays are output arguments if *fact* = 'N'. See the description of *afp*, *ipiv* in Input Arguments section.
rcond	REAL for chpsvx

DOUBLE PRECISION for zhpsvx.
An estimate of the reciprocal condition number of the matrix *A*. If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr, *berr*
REAL for chpsvx
DOUBLE PRECISION for zhpsvx.
Arrays, DIMENSION at least max(1, *nrhs*). Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

info
INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, and $i \leq n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.
If *info* = *i*, and *i* = *n* + 1, then *D* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hpsvx interface are as follows:

a
Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).

b
Holds the matrix *B* of size (*n*, *nrhs*).

x
Holds the matrix *X* of size (*n*, *nrhs*).

af
Stands for argument *ap* in Fortan 77 interface. Holds the array *AF* of size (*n**(*n*+1)/2).

ipiv
Holds the vector of length (*n*).

ferr
Holds the vector of length (*nrhs*).

berr	Holds the vector of length (*nrhs*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
fact	Must be 'N' or 'F'. The default value is 'N'. If *fact* = 'F', then both arguments *af* and *ipiv* must be present; otherwise, an error is returned.

LAPACK Routines: Least Squares and Eigenvalue Problems

4

This chapter describes the Intel® Math Kernel Library implementation of routines from the LAPACK package that are used for solving linear least-squares problems, eigenvalue and singular value problems, as well as performing a number of related computational tasks.

Sections in this chapter include descriptions of LAPACK computational routines and driver routines. For full reference on LAPACK routines and related information see [LUG].

Least-Squares Problems. A typical `least-squares problem` is as follows: given a matrix A and a vector b, find the vector x that minimizes the sum of squares $\sigma_i((Ax)_i - b_i)^2$ or, equivalently, find the vector x that minimizes the 2-norm $||Ax - b||_2$.

In the most usual case, A is an m-by-n matrix with $m < n$ and $\mathrm{rank}(A) = n$. This problem is also referred to as finding the `least-squares solution` to an `overdetermined` system of linear equations (here we have more equations than unknowns). To solve this problem, you can use the QR factorization of the matrix A (see QR Factorization).

If $m < n$ and $\mathrm{rank}(A) = m$, there exist an infinite number of solutions x which exactly satisfy $Ax = b$, and thus minimize the norm $||Ax - b||_2$. In this case it is often useful to find the unique solution that minimizes $||x||_2$. This problem is referred to as finding the `minimum-norm solution` to an `underdetermined` system of linear equations (here we have more unknowns than equations). To solve this problem, you can use the LQ factorization of the matrix A (see LQ Factorization).

In the general case you may have a `rank-deficient least-squares problem`, with $\mathrm{rank}(A) < \min(m, n)$: find the `minimum-norm least-squares solution` that minimizes both $||x||_2$ and $||Ax - b||^2$. In this case (or when the rank of A is in doubt) you can use the QR factorization with pivoting or `singular value decomposition` (see Singular Value Decomposition).

Eigenvalue Problems. The eigenvalue problems (from German *eigen* "own") are stated as follows: given a matrix A, find the `eigenvalues` λ and the corresponding `eigenvectors` z that satisfy the equation

$Az = \lambda z$ (right eigenvectors z)

or the equation

$z^H A = \lambda z^H$ (left eigenvectors z).

If A is a real symmetric or complex Hermitian matrix, the above two equations are equivalent, and the problem is called a `symmetric` eigenvalue problem. Routines for solving this type of problems are described in the section Symmetric Eigenvalue Problems .

Routines for solving eigenvalue problems with nonsymmetric or non-Hermitian matrices are described in the section Nonsymmetric Eigenvalue Problems .

The library also includes routines that handle generalized symmetric-definite eigenvalue problems: find the eigenvalues λ and the corresponding eigenvectors x that satisfy one of the following equations:

$Az = \lambda Bz$, $ABz = \lambda z$, or $BAz = \lambda z$,

where A is symmetric or Hermitian, and B is symmetric positive-definite or Hermitian positive-definite. Routines for reducing these problems to standard symmetric eigenvalue problems are described in the section Generalized Symmetric-Definite Eigenvalue Problems .

To solve a particular problem, you usually call several computational routines. Sometimes you need to combine the routines of this chapter with other LAPACK routines described in Chapter 3 as well as with BLAS routines described in Chapter 2 .

For example, to solve a set of least-squares problems minimizing $||Ax - b||^2$ for all columns b of a given matrix B (where A and B are real matrices), you can call ?geqrf to form the factorization $A = QR$, then call ?ormqr to compute $C = Q^H B$ and finally call the BLAS routine ?trsm to solve for X the system of equations $RX = C$.

Another way is to call an appropriate driver routine that performs several tasks in one call. For example, to solve the least-squares problem the driver routine ?gels can be used.

WARNING. LAPACK routines expect that input matrices do not contain INF or NaN values. When input data is inappropriate for LAPACK, problems may arise, including possible hangs.

Starting from release 8.0, Intel MKL along with Fortran-77 interface to LAPACK computational and driver routines supports also Fortran-95 interface which uses simplified routine calls with shorter argument lists. The calling sequence for Fortran-95 interface is given in the syntax section of the routine description immediately after Fortran-77 calls.

Routine Naming Conventions

For each routine in this chapter, when calling it from the Fortran-77 program you can use the LAPACK name.

LAPACK names have the structure xyyzzz, which is described below.

The initial letter x indicates the data type:

s	real, single precision
c	complex, single precision
d	real, double precision

z	complex, double precision

The second and third letters yy indicate the matrix type and storage scheme:

bd	bidiagonal matrix
ge	general matrix
gb	general band matrix
hs	upper Hessenberg matrix
or	(real) orthogonal matrix
op	(real) orthogonal matrix (packed storage)
un	(complex) unitary matrix
up	(complex) unitary matrix (packed storage)
pt	symmetric or Hermitian positive-definite tridiagonal matrix
sy	symmetric matrix
sp	symmetric matrix (packed storage)
sb	(real) symmetric band matrix
st	(real) symmetric tridiagonal matrix
he	Hermitian matrix
hp	Hermitian matrix (packed storage)
hb	(complex) Hermitian band matrix
tr	triangular or quasi-triangular matrix.

The last three letters zzz indicate the computation performed, for example:

qrf	form the *QR* factorization
lqf	form the *LQ* factorization.

Thus, the routine sgeqrf forms the *QR* factorization of general real matrices in single precision; the corresponding routine for complex matrices is cgeqrf.

Names of the LAPACK computational and driver routines for Fortran-95 interface in Intel MKL are the same as Fortran-77 names but without the first letter that indicates the data type. For example, the name of the routine that forms the *QR* factorization of general real matrices in Fortran-95 interface is geqrf. Handling of different data types is done through defining a specific internal parameter referring to a module block with named constants for single and double precision.

For details on the design of Fortran-95 interface for LAPACK computational and driver routines in Intel MKL and for the general information on how the optional arguments are reconstructed, see Fortran-95 Interface Conventions in chapter 3 .

Matrix Storage Schemes

LAPACK routines use the following matrix storage schemes:

- `Full storage`: a matrix A is stored in a two-dimensional array a, with the matrix element a_{ij} stored in the array element $a(i,j)$.
- `Packed storage` scheme allows you to store symmetric, Hermitian, or triangular matrices more compactly: the upper or lower triangle of the matrix is packed by columns in a one-dimensional array.
- `Band storage`: an m-by-n band matrix with kl sub-diagonals and ku super-diagonals is stored compactly in a two-dimensional array ab with $kl+ku+1$ rows and n columns. Columns of the matrix are stored in the corresponding columns of the array, and `diagonals` of the matrix are stored in rows of the array.

In Chapters 3 and 4 , arrays that hold matrices in packed storage have names ending in p; arrays with matrices in band storage have names ending in b. For more information on matrix storage schemes, see "Matrix Arguments" in Appendix B .

Mathematical Notation

In addition to the mathematical notation used in previous chapters, descriptions of routines in this chapter use the following notation:

λ_i	`Eigenvalues` of the matrix A (for the definition of eigenvalues, see Eigenvalue Problems).
σ_i	`Singular values` of the matrix A. They are equal to square roots of the eigenvalues of $A^H A$. (For more information, see Singular Value Decomposition).
$\|\|x\|\|_2$	The `2-norm` of the vector x: $\|\|x\|\|_2 = (\Sigma_i \|x_i\|^2)^{1/2} = \|\|x\|\|_E$.
$\|\|A\|\|_2$	The `2-norm` (or `spectral norm`) of the matrix A. $\|\|A\|\|_2 = \max_i \sigma_i$, $\|\|A\|\|_2^2 = \max_{\|x\|=1}(Ax \cdot Ax)$.

$\|\|A\|\|_E$	The Euclidean norm of the matrix A: $\|\|A\|\|_E^2 = \Sigma_i\Sigma_j\|a_{ij}\|^2$ (for vectors, the Euclidean norm and the 2-norm are equal: $\|\|x\|\|_E = \|\|x\|\|_2$).
$q(x, y)$	The acute angle between vectors x and y: $\cos q(x, y) = \|x \cdot y\| / (\|\|x\|\|_2 \|\|y\|\|_2)$.

Computational Routines

In the sections that follow, the descriptions of LAPACK computational routines are given. These routines perform distinct computational tasks that can be used for:

Orthogonal Factorizations

Singular Value Decomposition

Symmetric Eigenvalue Problems

Generalized Symmetric-Definite Eigenvalue Problems

Nonsymmetric Eigenvalue Problems

Generalized Nonsymmetric Eigenvalue Problems

Generalized Singular Value Decomposition

See also the respective driver routines.

Orthogonal Factorizations

This section describes the LAPACK routines for the QR (RQ) and LQ (QL) factorization of matrices. Routines for the RZ factorization as well as for generalized QR and RQ factorizations are also included.

QR Factorization. Assume that A is an m-by-n matrix to be factored.

If $m \le n$, the QR factorization is given by

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix} = (Q_1, Q_2) \begin{pmatrix} R \\ 0 \end{pmatrix}$$

where R is an n-by-n upper triangular matrix with real diagonal elements, and Q is an m-by-m orthogonal (or unitary) matrix.

You can use the QR factorization for solving the following least-squares problem: minimize $||Ax - b||^2$ where A is a full-rank m-by-n matrix($m < n$). After factoring the matrix, compute the solution x by solving $Rx = (Q_1)^T b$.

If $m < n$, the QR factorization is given by

$A = QR = Q(R_1 R_2)$

where R is trapezoidal, R_1 is upper triangular and R_2 is rectangular.

The LAPACK routines do not form the matrix Q explicitly. Instead, Q is represented as a product of min(m, n) `elementary reflectors`. Routines are provided to work with Q in this representation.

LQ Factorization LQ factorization of an m-by-n matrix A is as follows. If $m \le n$,

$$A = (L, 0)\,Q = (L, 0)\begin{pmatrix} Q_1 \\ Q_2 \end{pmatrix} = (LQ1)$$

where L is an m-by-m lower triangular matrix with real diagonal elements, and Q is an n-by-n orthogonal (or unitary) matrix.

If $m > n$, the LQ factorization is

$$A = \begin{pmatrix} L_1 \\ L_2 \end{pmatrix} Q$$

where L_1 is an n-by-n lower triangular matrix, L_2 is rectangular, and Q is an n-by-n orthogonal (or unitary) matrix.

You can use the LQ factorization to find the minimum-norm solution of an underdetermined system of linear equations $Ax = b$ where A is an m-by-n matrix of rank m $(m < n)$. After factoring the matrix, compute the solution vector x as follows: solve $Ly = b$ for y, and then compute $x = (Q_1)^H y$.

Table 4-1 lists LAPACK routines (Fortran-77 interface) that perform orthogonal factorization of matrices. Respective routine names in Fortran-95 interface are without the first symbol (see Routine Naming Conventions).

Table 4-1 Computational Routines for Orthogonal Factorization

Matrix type, factorization	Factorize without pivoting	Factorize with pivoting	Generate matrix Q	Apply matrix Q
general matrices, QR factorization	?geqrf	?geqpf ?geqp3	?orgqr ?ungqr	?ormqr ?unmqr
general matrices, RQ factorization	?gerqf		?orgrq ?ungrq	?ormrq ?unmrq
general matrices, LQ factorization	?gelqf		?orglq ?unglq	?ormlq ?unmlq
general matrices, QL factorization	?geqlf		?orgql ?ungql	?ormql ?unmql
trapezoidal matrices, RZ factorization	?tzrzf			?ormrz ?unmrz
pair of matrices, generalized QR factorization	?ggqrf			
pair of matrices, generalized RQ factorization	?ggrqf			

?geqrf

Computes the QR factorization of a general m-by-n matrix.

Syntax

Fortran 77:

```
call sgeqrf(m, n, a, lda, tau, work, lwork, info)
call dgeqrf(m, n, a, lda, tau, work, lwork, info)
call cgeqrf(m, n, a, lda, tau, work, lwork, info)
call zgeqrf(m, n, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call geqrf(a [, tau] [,info])
```

Description

The routine forms the QR factorization of a general m-by-n matrix A (see Orthogonal Factorizations). No pivoting is performed.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of min(m, n) `elementary reflectors`. Routines are provided to work with Q in this representation.

Input Parameters

m	`INTEGER`. The number of rows in the matrix A ($m \geq 0$).
n	`INTEGER`. The number of columns in A ($n \geq 0$).

a, $work$	`REAL` for `sgeqrf`
	`DOUBLE PRECISION` for `dgeqrf`
	`COMPLEX` for `cgeqrf`
	`DOUBLE COMPLEX` for `zgeqrf`.
	Arrays: a(lda,*) contains the matrix A. The second dimension of a must be at least max(1, n).
	$work$ is a workspace array, its dimension `max(1, lwork)`.

lda	`INTEGER`. The first dimension of a; at least max(1, m).
$lwork$	`INTEGER`. The size of the $work$ array ($lwork \geq n$). If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by `xerbla`. See Application Notes for the suggested value of $lwork$.

Output Parameters

a	Overwritten by the factorization data as follows: If $m \geq n$, the elements below the diagonal are overwritten by the details of the unitary matrix Q, and the upper triangle is overwritten by the corresponding elements of the upper triangular matrix R.

	If *m* < *n*, the strictly lower triangular part is overwritten by the details of the unitary matrix *Q*, and the remaining elements are overwritten by the corresponding elements of the *m*-by-*n* upper trapezoidal matrix *R*.
tau	REAL for sgeqrf DOUBLE PRECISION for dgeqrf COMPLEX for cgeqrf DOUBLE COMPLEX for zgeqrf. Array, DIMENSION at least max (1, min(*m*, *n*)). Contains additional information on the matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine *geqrf* interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
tau	Holds the vector of length min(*m*,*n*)

Application Notes

For better performance, try using *lwork* = *n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed factorization is the exact factorization of a matrix $A + E$, where

$||E||_2 = O(\varepsilon)||A||_2$.

The approximate number of floating-point operations for real flavors is

$(4/3)n^3$	if $m = n$,
$(2/3)n^2(3m-n)$	if $m > n$,
$(2/3)m^2(3n-m)$	if $m < n$.

The number of operations for complex flavors is 4 times greater.

To solve a set of least-squares problems minimizing $||Ax - b||_2$ for all columns b of a given matrix B, you can call the following:

?geqrf (this routine)	to factorize $A = QR$;
?ormqr	to compute $C = Q^T*B$ (for real matrices);
?unmqr	to compute $C = Q^H*B$ (for complex matrices);
?trsm	(a BLAS routine) to solve $R*X = C$.

(The columns of the computed X are the least-squares solution vectors x.)

To compute the elements of Q explicitly, call

?orgqr	(for real matrices)
?ungqr	(for complex matrices).

?geqpf

Computes the QR factorization of a general m-by-n matrix with pivoting.

Syntax

Fortran 77:

```
call sgeqpf(m, n, a, lda, jpvt, tau, work, info)
call dgeqpf(m, n, a, lda, jpvt, tau, work, info)
call cgeqpf(m, n, a, lda, jpvt, tau, work, rwork, info)
call zgeqpf(m, n, a, lda, jpvt, tau, work, rwork, info)
```

Fortran 95:

```
call geqpf(a, jpvt [,tau] [,info])
```

Description

This routine is deprecated and has been replaced by routine ?geqp3.

The routine ?geqpf forms the *QR* factorization of a general *m*-by-*n* matrix *A* with column pivoting: *A*P = Q*R* (see Orthogonal Factorizations). Here *P* denotes an *n*-by-*n* permutation matrix.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) elementary reflectors. Routines are provided to work with *Q* in this representation.

Input Parameters

m	INTEGER. The number of rows in the matrix *A* (*m* ≥ 0).
n	INTEGER. The number of columns in *A* (*n* ≥ 0).
a, *work*	REAL for sgeqpf DOUBLE PRECISION for dgeqpf COMPLEX for cgeqpf DOUBLE COMPLEX for zgeqpf. Arrays: *a* (*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*).

	work (*lwork*) is a workspace array. The size of the *work* array must be at least max(1, 3**n*) for real flavors and at least max(1, *n*) for complex flavors.
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
jpvt	INTEGER. Array, DIMENSION at least max(1, *n*). On entry, if *jpvt*(*i*) > 0, the *i*-th column of *A* is moved to the beginning of *A*P* before the computation, and fixed in place during the computation. If *jpvt*(*i*) = 0, the *i*th column of *A* is a free column (that is, it may be interchanged during the computation with any other free column).
rwork	REAL for cgeqpf DOUBLE PRECISION for zgeqpf. A workspace array, DIMENSION at least max(1, 2**n*).

Output Parameters

a	Overwritten by the factorization data as follows: If *m* ≥ *n*, the elements below the diagonal are overwritten by the details of the unitary (orthogonal) matrix *Q*, and the upper triangle is overwritten by the corresponding elements of the upper triangular matrix *R*. If *m* < *n*, the strictly lower triangular part is overwritten by the details of the matrix *Q*, and the remaining elements are overwritten by the corresponding elements of the *m*-by-*n* upper trapezoidal matrix *R*.
tau	REAL for sgeqpf DOUBLE PRECISION for dgeqpf COMPLEX for cgeqpf DOUBLE COMPLEX for zgeqpf. Array, DIMENSION at least max (1, min(*m*, *n*)). Contains additional information on the matrix *Q*.
jpvt	Overwritten by details of the permutation matrix *P* in the factorization *A*P = Q*R*. More precisely, the columns of *A*P* are the columns of *A* in the following order: *jpvt*(1), *jpvt*(2), ..., *jpvt*(*n*).
info	INTEGER.

If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine geqpf interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
jpvt	Holds the vector of length (*n*).
tau	Holds the vector of length min(*m*,*n*)

Application Notes

The computed factorization is the exact factorization of a matrix $A + E$, where

$||E||_2 = O(\varepsilon)||A||_2$.

The approximate number of floating-point operations for real flavors is

$(4/3)n^3$	if $m = n$,
$(2/3)n^2(3m-n)$	if $m > n$,
$(2/3)m^2(3n-m)$	if $m < n$.

The number of operations for complex flavors is 4 times greater.

To solve a set of least-squares problems minimizing $||Ax - b||_2$ for all columns b of a given matrix B, you can call the following:

?geqpf (this routine)	to factorize *A*P = Q*R*;
?ormqr	to compute $C = Q^T*B$ (for real matrices);
?unmqr	to compute $C = Q^H*B$ (for complex matrices);
?trsm (a BLAS routine)	to solve *R*X = C*.

(The columns of the computed *X* are the permuted least-squares solution vectors *x*; the output array *jpvt* specifies the permutation order.)

To compute the elements of *Q* explicitly, call

?orgqr	(for real matrices)
?ungqr	(for complex matrices).

?geqp3

Computes the QR factorization of a general m-by-n matrix with column pivoting using Level 3 BLAS.

Syntax

Fortran 77:

```
call sgeqp3(m, n, a, lda, jpvt, tau, work, lwork, info)
call dgeqp3(m, n, a, lda, jpvt, tau, work, lwork, info)
call cgeqp3(m, n, a, lda, jpvt, tau, work, lwork, rwork, info)
call zgeqp3(m, n, a, lda, jpvt, tau, work, lwork, rwork, info)
```

Fortran 95:

```
call geqp3(a, jpvt [,tau] [,info])
```

Description

The routine forms the QR factorization of a general m-by-n matrix A with column pivoting: $AP = Q*R$ (see Orthogonal Factorizations) using Level 3 BLAS. Here P denotes an n-by-n permutation matrix. Use this routine instead of ?geqpf for better performance.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of min(m, n) elementary reflectors. Routines are provided to work with Q in this representation.

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
a, *work*	REAL for sgeqp3 DOUBLE PRECISION for dgeqp3 COMPLEX for cgeqp3 DOUBLE COMPLEX for zgeqp3. Arrays: *a* (*lda*,*) contains the matrix A.

The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The first dimension of *a*; at least max(1, *m*).

lwork — INTEGER. The size of the *work* array; must be at least max(1, 3**n*+1) for real flavors, and at least max(1, *n*+1) for complex flavors.
If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes below for details.

jpvt — INTEGER.
Array, DIMENSION at least max(1, *n*).
On entry, if *jpvt*(*i*) ≠ 0, the *i*th column of *A* is moved to the beginning of *AP* before the computation, and fixed in place during the computation.
If *jpvt*(*i*) = 0, the *i*-th column of *A* is a free column (that is, it may be interchanged during the computation with any other free column).

rwork — REAL for cgeqp3
DOUBLE PRECISION for zgeqp3.
A workspace array, DIMENSION at least max(1, 2**n*). Used in complex flavors only.

Output Parameters

a — Overwritten by the factorization data as follows:
If *m* ≥ *n*, the elements below the diagonal are overwritten by the details of the unitary (orthogonal) matrix *Q*, and the upper triangle is overwritten by the corresponding elements of the upper triangular matrix *R*.
If *m* < *n*, the strictly lower triangular part is overwritten by the details of the matrix *Q*, and the remaining elements are overwritten by the corresponding elements of the *m*-by-*n* upper trapezoidal matrix *R*.

tau — REAL for sgeqp3

	DOUBLE PRECISION for dgeqp3 COMPLEX for cgeqp3 DOUBLE COMPLEX for zgeqp3. Array, DIMENSION at least max (1, min(*m*, *n*)). Contains scalar factors of the elementary reflectors for the matrix *Q*.
jpvt	Overwritten by details of the permutation matrix *P* in the factorization *AP = Q*R*. More precisely, the columns of *AP* are the columns of *A* in the following order: *jpvt*(1), *jpvt*(2), ..., *jpvt*(*n*).
info	INTEGER. If *info = 0*, the execution is successful. If *info = -i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine geqp3 interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
jpvt	Holds the vector of length (*n*).
tau	Holds the vector of length min(*m*,*n*)

Application Notes

To solve a set of least-squares problems minimizing $||Ax - b||_2$ for all columns *b* of a given matrix *B*, you can call the following:

?geqp3 (this routine)	to factorize *AP = Q*R*;
?ormqr	to compute $C = Q^T*B$ (for real matrices);
?unmqr	to compute $C = Q^H*B$ (for complex matrices);
?trsm (a BLAS routine)	to solve *R*X = C*.

(The columns of the computed *X* are the permuted least-squares solution vectors *x*; the output array *jpvt* specifies the permutation order.)

To compute the elements of *Q* explicitly, call

?orgqr (for real matrices)

?ungqr (for complex matrices).

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?orgqr

Generates the real orthogonal matrix Q of the QR factorization formed by ?geqrf.

Syntax

Fortran 77:

```
call sorgqr(m, n, k, a, lda, tau, work, lwork, info)
call dorgqr(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call orgqr(a, tau [,info])
```

Description

The routine generates the whole or part of *m*-by-*m* orthogonal matrix *Q* of the *QR* factorization formed by the routines sgeqrf/dgeqrf or sgeqpf/dgeqpf. Use this routine after a call to sgeqrf/dgeqrf or sgeqpf/dgeqpf.

Usually Q is determined from the QR factorization of an m by p matrix A with $m \geq p$. To compute the whole matrix Q, use:

```
call ?orgqr(m, m, p, a, lda, tau, work, lwork, info)
```

To compute the leading p columns of Q (which form an orthonormal basis in the space spanned by the columns of A):

```
call ?orgqr(m, p, p, a, lda, tau, work, lwork, info)
```

To compute the matrix Q^k of the QR factorization of A's leading k columns:

```
call ?orgqr(m, m, k, a, lda, tau, work, lwork, info)
```

To compute the leading k columns of Q^k (which form an orthonormal basis in the space spanned by A's leading k columns):

```
call ?orgqr(m, k, k, a, lda, tau, work, lwork, info)
```

Input Parameters

m	INTEGER. The order of the orthogonal matrix Q ($m \geq 0$).
n	INTEGER. The number of columns of Q to be computed ($0 \leq n \leq m$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).
a, *tau*, *work*	REAL for sorgqr DOUBLE PRECISION for dorgqr Arrays: *a*(*lda*,*) and *tau*(*) are the arrays returned by sgeqrf / dgeqrf or sgeqpf / dgeqpf. The second dimension of *a* must be at least max(1, *n*). The dimension of *tau* must be at least max(1, *k*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array (*lwork* ≥ *n*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	Overwritten by *n* leading columns of the *m*-by-*m* orthogonal matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine orgqr interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
tau	Holds the vector of length (*k*)

Application Notes

For better performance, try using *lwork* = *n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed Q differs from an exactly orthogonal matrix by a matrix E such that

$||E||_2 = O(\varepsilon)||A||_2$ where ε is the machine precision.

The total number of floating-point operations is approximately $4*m*n*k - 2*(m + n)*k^2 + (4/3)*k^3$.

If $n = k$, the number is approximately $(2/3)*n^2*(3m - n)$.

The complex counterpart of this routine is ?ungqr.

?ormqr

Multiplies a real matrix by the orthogonal matrix Q of the QR factorization formed by ?geqrf *or* ?geqpf*.*

Syntax

Fortran 77:

```
call sormqr(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call dormqr(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call ormqr(a, tau, c [,side] [,trans] [,info])
```

Description

The routine multiplies a real matrix C by Q or Q^T, where Q is the orthogonal matrix Q of the QR factorization formed by the routines sgeqrf/dgeqrf or sgeqpf/dgeqpf.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q*C$, Q^T*C, $C*Q$, or $C*Q^T$ (overwriting the result on C).

Input Parameters

side	CHARACTER*1. Must be either 'L' or 'R'. If *side* ='L', Q or Q^T is applied to C from the left. If *side* ='R', Q or Q^T is applied to C from the right.
trans	CHARACTER*1. Must be either 'N' or 'T'. If *trans* ='N', the routine multiplies C by Q.

If *trans* ='T', the routine multiplies *C* by Q^T.

m — INTEGER. The number of rows in the matrix *C* (*m* ≥ 0).

n — INTEGER. The number of columns in *C* (*n* ≥ 0).

k — INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:

0 ≤ *k* ≤ *m* if *side* ='L';

0 ≤ *k* ≤ *n* if *side* ='R'.

a, *tau*, *c*, *work* — REAL for sgeqrf
DOUBLE PRECISION for dgeqrf.
Arrays:
a(*lda*,*) and *tau*(*) are the arrays returned by sgeqrf / dgeqrf or sgeqpf / dgeqpf. The second dimension of *a* must be at least max(1, *k*). The dimension of *tau* must be at least max(1, *k*).
c(*ldc*,*) contains the matrix *C*.
The second dimension of *c* must be at least max(1, *n*)
work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The first dimension of *a*. Constraints:

lda ≥ max(1, *m*) if *side* = 'L';

lda ≥ max(1, *n*) if *side* = 'R'.

ldc — INTEGER. The first dimension of *c*. Constraint:

ldc ≥ max(1, *m*).

lwork — INTEGER. The size of the *work* array. Constraints:

lwork ≥ max(1, *n*) if *side* = 'L';

lwork ≥ max(1, *m*) if *side* = 'R'.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.
See Application Notes for the suggested value of *lwork*.

Output Parameters

c	Overwritten by the product *Q*C*, Q^T**C*, *C*Q*, or *C**Q^T (as specified by *side* and *trans*).
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = –*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ormqr interface are the following:

a	Holds the matrix *A* of size (*r*,*k*). *r* = *m* if *side* = 'L'. *r* = *n* if *side* = 'R'.
tau	Holds the vector of length (*k*).
c	Holds the matrix *C* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
trans	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = *n*blocksize* (if *side* = 'L') or *lwork* = *m*blocksize* (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is ?unmqr.

?ungqr

Generates the complex unitary matrix Q of the QR factorization formed by ?geqrf.

Syntax

Fortran 77:

```
call cungqr(m, n, k, a, lda, tau, work, lwork, info)
call zungqr(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call ungqr(a, tau [,info])
```

Description

The routine generates the whole or part of *m*-by-*m* unitary matrix *Q* of the *QR* factorization formed by the routines cgeqrf/zgeqrf or cgeqpf/zgeqpf. Use this routine after a call to cgeqrf/zgeqrf or cgeqpf/zgeqpf.

Usually *Q* is determined from the *QR* factorization of an *m* by *p* matrix *A* with $m \geq p$. To compute the whole matrix *Q*, use:

```
call ?ungqr(m, m, p, a, lda, tau, work, lwork, info)
```

To compute the leading *p* columns of *Q* (which form an orthonormal basis in the space spanned by the columns of *A*):

```
call ?ungqr(m, p, p, a, lda, tau, work, lwork, info)
```

To compute the matrix Q^k of the *QR* factorization of *A*'s leading *k* columns:

```
call ?ungqr(m, m, k, a, lda, tau, work, lwork, info)
```

To compute the leading k columns of Q^k (which form an orthonormal basis in the space spanned by A's leading k columns):

```
call ?ungqr(m, k, k, a, lda, tau, work, lwork, info)
```

Input Parameters

m	INTEGER. The order of the unitary matrix Q ($m \geq 0$).
n	INTEGER. The number of columns of Q to be computed ($0 \leq n \leq m$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).
a, *tau*, *work*	COMPLEX for cungqr DOUBLE COMPLEX for zungqr Arrays: *a*(*lda*,*) and *tau*(*) are the arrays returned by cgeqrf/zgeqrf or cgeqpf/zgeqpf. The second dimension of *a* must be at least max(1, *n*). The dimension of *tau* must be at least max(1, *k*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array (*lwork* ≥ *n*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	Overwritten by *n* leading columns of the *m*-by-*m* unitary matrix Q.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `ungqr` interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
tau	Holds the vector of length (*k*).

Application Notes

For better performance, try using *lwork* =*n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the `blocked algorithm`.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed *Q* differs from an exactly unitary matrix by a matrix *E* such that $||E||_2 = O(\varepsilon)||A||_2$, where ε is the machine precision.

The total number of floating-point operations is approximately 16**m***n***k* - 8*(*m* + *n*)**k2* + (16/3)**k*^{3}.

If *n* = *k*, the number is approximately (8/3)**n*^{2}*(3*m* - *n*).

The real counterpart of this routine is `?orgqr`.

?unmqr

Multiplies a complex matrix by the unitary matrix Q of the QR factorization formed by ?geqrf.

Syntax

Fortran 77:

```
call cunmqr(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call zunmqr(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call unmqr(a, tau, c [,side] [,trans] [,info])
```

Description

The routine multiplies a rectangular complex matrix C by Q or Q_H, where Q is the unitary matrix Q of the QR factorization formed by the routines cgeqrf/zgeqrf or cgeqpf/zgeqpf.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q*C$, Q^H*C, $C*Q$, or $C*Q^H$ (overwriting the result on C).

Input Parameters

side	CHARACTER*1. Must be either 'L' or 'R'. If *side* = 'L', Q or Q_H is applied to C from the left. If *side* = 'R', Q or Q_H is applied to C from the right.
trans	CHARACTER*1. Must be either 'N' or 'C'. If *trans* = 'N', the routine multiplies C by Q. If *trans* = 'C', the routine multiplies C by Q^H.
m	INTEGER. The number of rows in the matrix C ($m \geq 0$).
n	INTEGER. The number of columns in C ($n \geq 0$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints: $0 \leq k \leq m$ if *side* = 'L'; $0 \leq k \leq n$ if *side* = 'R'.
a, *c*, *tau*, *work*	COMPLEX for cgeqrf

DOUBLE COMPLEX for zgeqrf.
Arrays:
a(*lda*,*) and *tau*(*) are the arrays returned by cgeqrf / zgeqrf or cgeqpf / zgeqpf.
The second dimension of *a* must be at least max(1, *k*).
The dimension of *tau* must be at least max(1, *k*).
c(*ldc*,*) contains the matrix *C*.
The second dimension of *c* must be at least max(1, *n*)
work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The first dimension of *a*. Constraints:

lda ≥ max(1, *m*) if *side* = 'L';

lda ≥ max(1, *n*) if *side* = 'R'.

ldc — INTEGER. The first dimension of *c*. Constraint:

ldc ≥ max(1, *m*).

lwork — INTEGER. The size of the *work* array. Constraints:

lwork ≥ max(1, *n*) if *side* = 'L';

lwork ≥ max(1, *m*) if *side* = 'R'.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.
See Application notes for the suggested value of *lwork*.

Output Parameters

c — Overwritten by the product *Q***C*, Q^H**C*, *C***Q*, or *C**Q^H (as specified by *side* and *trans*).

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `unmqr` interface are the following:

a	Holds the matrix *A* of size (*r*,*k*). *r* = *m* if *side* = 'L'. *r* = *n* if *side* = 'R'.
tau	Holds the vector of length (*k*).
c	Holds the matrix *C* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
trans	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = *n***blocksize* (if *side* = 'L') or *lwork* = *m***blocksize* (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is ?ormqr.

?gelqf

Computes the LQ factorization of a general m-by-n matrix.

Syntax

Fortran 77:

```
call sgelqf(m, n, a, lda, tau, work, lwork, info)
call dgelqf(m, n, a, lda, tau, work, lwork, info)
call cgelqf(m, n, a, lda, tau, work, lwork, info)
call zgelqf(m, n, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call gelqf(a [, tau] [,info])
```

Description

The routine forms the *LQ* factorization of a general *m*-by-*n* matrix *A* (see Orthogonal Factorizations). No pivoting is performed.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) elementary reflectors. Routines are provided to work with *Q* in this representation.

Input Parameters

m	INTEGER. The number of rows in the matrix *A* (*m* ≥ 0).
n	INTEGER. The number of columns in *A* (*n* ≥ 0).
a, *work*	REAL for sgelqf DOUBLE PRECISION for dgelqf COMPLEX for cgelqf DOUBLE COMPLEX for zgelqf. Arrays: *a*(*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).

lwork	INTEGER. The size of the *work* array; at least max(1, *m*). If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the factorization data as follows: If $m \le n$, the elements above the diagonal are overwritten by the details of the unitary (orthogonal) matrix Q, and the lower triangle is overwritten by the corresponding elements of the lower triangular matrix L. If $m > n$, the strictly upper triangular part is overwritten by the details of the matrix Q, and the remaining elements are overwritten by the corresponding elements of the m-by-n lower trapezoidal matrix L.
tau	REAL for sgelqf DOUBLE PRECISION for dgelqf COMPLEX for cgelqf DOUBLE COMPLEX for zgelqf. Array, DIMENSION at least max(1, min(*m*, *n*)). Contains additional information on the matrix Q.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = −*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gelqf interface are the following:

a	Holds the matrix A of size (m,n).

tau	Holds the vector of length min(*m*,*n*).

Application Notes

For better performance, try using *lwork* =*m***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed factorization is the exact factorization of a matrix $A + E$, where

$||E||_2 = O(\varepsilon) \ ||A||_2$.

The approximate number of floating-point operations for real flavors is

$(4/3)n^3$	if $m = n$,
$(2/3)n^2(3m-n)$	if $m > n$,
$(2/3)m^2(3n-m)$	if $m < n$.

The number of operations for complex flavors is 4 times greater.

To find the minimum-norm solution of an underdetermined least-squares problem minimizing $||Ax - b||_2$ for all columns b of a given matrix B, you can call the following:

?gelqf (this routine)	to factorize $A = L*Q$;
?trsm (a BLAS routine)	to solve $L*Y = B$ for Y;
?ormlq	to compute $X = (Q_1)^T*Y$ (for real matrices);
?unmlq	to compute $X = (Q_1)^H*Y$ (for complex matrices).

(The columns of the computed X are the minimum-norm solution vectors x. Here A is an m-by-n matrix with $m < n$; Q_1 denotes the first m columns of Q).

To compute the elements of Q explicitly, call

?orglq	(for real matrices)
?unglq	(for complex matrices).

?orglq

Generates the real orthogonal matrix Q of the LQ factorization formed by ?gelqf.

Syntax

Fortran 77:

```
call sorglq(m, n, k, a, lda, tau, work, lwork, info)
call dorglq(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call orglq(a, tau [,info])
```

Description

The routine generates the whole or part of n-by-n orthogonal matrix Q of the LQ factorization formed by the routines sgelqf/gelqf. Use this routine after a call to sgelqf/dgelqf.

Usually Q is determined from the LQ factorization of an p-by-n matrix A with $n \geq p$. To compute the whole matrix Q, use:

```
call ?orglq(n, n, p, a, lda, tau, work, lwork, info)
```

To compute the leading p rows of Q, which form an orthonormal basis in the space spanned by the rows of A, use:

```
call ?orglq(p, n, p, a, lda, tau, work, lwork, info)
```

To compute the matrix Q^k of the LQ factorization of A's leading k rows, use:

```
call ?orglq(n, n, k, a, lda, tau, work, lwork, info)
```

To compute the leading k rows of Q^k, which form an orthonormal basis in the space spanned by A's leading k rows, use:

```
call ?orgqr(k, n, k, a, lda, tau, work, lwork, info)
```

Input Parameters

m	INTEGER. The number of rows of *Q* to be computed ($0 \le m \le n$).
n	INTEGER. The order of the orthogonal matrix *Q* ($n \ge m$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q* ($0 \le k \le m$).
a, *tau*, *work*	REAL for sorglq DOUBLE PRECISION for dorglq Arrays: *a*(*lda*,*) and *tau*(*) are the arrays returned by sgelqf/dgelqf. The second dimension of *a* must be at least max(1, *n*). The dimension of *tau* must be at least max(1, *k*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array; at least max(1, *m*). If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	Overwritten by *m* leading rows of the *n*-by-*n* orthogonal matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = −*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `orglq` interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
tau	Holds the vector of length (*k*).

Application Notes

For better performance, try using *lwork* =*m***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the `blocked algorithm`.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed *Q* differs from an exactly orthogonal matrix by a matrix *E* such that $||E||_2 = O(\varepsilon)\ ||A||_2$, where *e* is the machine precision.

The total number of floating-point operations is approximately $4*m*n*k - 2*(m + n)*k^2 + (4/3)*k^3$.

If $m = k$, the number is approximately $(2/3)*m^2*(3n - m)$.

The complex counterpart of this routine is `?unglq`.

?ormlq

Multiplies a real matrix by the orthogonal matrix Q of the LQ factorization formed by ?gelqf.

Syntax

Fortran 77:

```
call sormlq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call dormlq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call ormlq(a, tau, c [,side] [,trans] [,info])
```

Description

The routine multiplies a real m-by-n matrix C by Q or Q^T, where Q is the orthogonal matrix Q of the LQ factorization formed by the routine sgelqf/dgelqf.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q*C$, Q^T*C, $C*Q$, or $C*Q^T$ (overwriting the result on C).

Input Parameters

side	CHARACTER*1. Must be either 'L' or 'R'. If *side* = 'L', Q or Q^T is applied to C from the left. If *side* = 'R', Q or Q^T is applied to C from the right.
trans	CHARACTER*1. Must be either 'N' or 'T'. If *trans* = 'N', the routine multiplies C by Q. If *trans* = 'T', the routine multiplies C by Q^T.
m	INTEGER. The number of rows in the matrix C ($m \geq 0$).
n	INTEGER. The number of columns in C ($n \geq 0$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints: $0 \leq k \leq m$ if *side* = 'L'; $0 \leq k \leq n$ if *side* = 'R'.
a, *c*, *tau*, *work*	REAL for sormlq

DOUBLE PRECISION for dormlq.
Arrays:
a(*lda*,*) and *tau*(*) are arrays returned by ?gelqf.
The second dimension of *a* must be:
at least max(1, *m*) if *side* = 'L';
at least max(1, *n*) if *side* = 'R'.
The dimension of *tau* must be at least max(1, *k*).
c(*ldc*,*) contains the matrix *C*.
The second dimension of *c* must be at least max(1, *n*)
work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The first dimension of *a*; *lda* ≥ max(1, *k*).

ldc — INTEGER. The first dimension of *c*; *ldc* ≥ max(1, *m*).

lwork — INTEGER. The size of the *work* array. Constraints:

lwork ≥ max(1, *n*) if *side* = 'L';

lwork ≥ max(1, *m*) if *side* = 'R'.

If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.
See Application Notes for the suggested value of *lwork*.

Output Parameters

c — Overwritten by the product *Q***C*, Q^T**C*, *C***Q*, or *C**Q^T (as specified by *side* and *trans*).

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = −*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `ormlq` interface are the following:

a	Holds the matrix *A* of size (*k*,*m*).
tau	Holds the vector of length (*k*).
c	Holds the matrix *C* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
trans	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = *n*blocksize* (if *side* = 'L') or *lwork* = *m*blocksize* (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork*= –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork*= –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is ?unmlq.

?unglq

Generates the complex unitary matrix Q of the LQ factorization formed by ?gelqf.

Syntax

Fortran 77:

```
call cunglq(m, n, k, a, lda, tau, work, lwork, info)
call zunglq(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call unglq(a, tau [,info])
```

Description

The routine generates the whole or part of n-by-n unitary matrix Q of the LQ factorization formed by the routines cgelqf/zgelqf. Use this routine after a call to cgelqf/zgelqf.

Usually Q is determined from the LQ factorization of an p-by-n matrix A with $n < p$. To compute the whole matrix Q, use:

```
call ?unglq(n, n, p, a, lda, tau, work, lwork, info)
```

To compute the leading p rows of Q, which form an orthonormal basis in the space spanned by the rows of A, use:

```
call ?unglq(p, n, p, a, lda, tau, work, lwork, info)
```

To compute the matrix Q^k of the LQ factorization of A's leading k rows, use:

```
call ?unglq(n, n, k, a, lda, tau, work, lwork, info)
```

To compute the leading k rows of Q^k, which form an orthonormal basis in the space spanned by A's leading k rows, use:

```
call ?ungqr(k, n, k, a, lda, tau, work, lwork, info)
```

Input Parameters

m	INTEGER. The number of rows of Q to be computed ($0 \le m \le n$).
n	INTEGER. The order of the unitary matrix Q ($n \ge m$).

k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q* (0 ≤ *k* ≤ *m*).
a, *tau*, *work*	COMPLEX for cunglq DOUBLE COMPLEX for zunglq Arrays: *a*(*lda*,*) and *tau*(*) are the arrays returned by sgelqf/dgelqf. The second dimension of *a* must be at least max(1, *n*). The dimension of *tau* must be at least max(1, *k*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array; at least max(1, *m*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	Overwritten by *m* leading rows of the *n*-by-*n* unitary matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine unglq interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
tau	Holds the vector of length (*k*).

Application Notes

For better performance, try using *lwork* = *m*blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed Q differs from an exactly unitary matrix by a matrix E such that $||E||_2 = O(\varepsilon)||A||_2$, where e is the machine precision.

The total number of floating-point operations is approximately $16*m*n*k - 8*(m + n)*k^2 + (16/3)*k^3$.

If $m = k$, the number is approximately $(8/3)*m^2*(3n - m)$.

The real counterpart of this routine is ?orglq.

?unmlq

Multiplies a complex matrix by the unitary matrix Q of the LQ factorization formed by ?gelqf.

Syntax

Fortran 77:

```
call cunmlq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call zunmlq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call unmlq(a, tau, c [,side] [,trans] [,info])
```

Description

The routine multiplies a real m-by-n matrix C by Q or Q^H, where Q is the unitary matrix Q of the LQ factorization formed by the routine cgelqf/zgelqf.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q*C$, Q^H*C, $C*Q$, or $C*Q^H$ (overwriting the result on C).

Input Parameters

side	CHARACTER*1. Must be either 'L' or 'R'. If *side* = 'L', Q or Q^H is applied to C from the left. If *side* = 'R', Q or Q^H is applied to C from the right.
trans	CHARACTER*1. Must be either 'N' or 'C'. If *trans* = 'N', the routine multiplies C by Q. If *trans* = 'C', the routine multiplies C by Q_H.
m	INTEGER. The number of rows in the matrix C ($m \geq 0$).
n	INTEGER. The number of columns in C ($n \geq 0$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints: $0 \leq k \leq m$ if *side* = 'L'; $0 \leq k \leq n$ if *side* = 'R'.
a, *c*, *tau*, *work*	COMPLEX for cunmlq DOUBLE COMPLEX for zunmlq. Arrays: *a*(*lda*,*) and *tau*(*) are arrays returned by ?gelqf. The second dimension of *a* must be: at least max(1, m) if *side* = 'L'; at least max(1, n) if *side* = 'R'. The dimension of *tau* must be at least max(1, k). *c*(*ldc*,*) contains the matrix C. The second dimension of *c* must be at least max(1, n) *work* is a workspace array, its dimension max(1, *lwork*).

lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *k*).
ldc	INTEGER. The first dimension of *c*; *ldc* ≥ max(1, *m*).
lwork	INTEGER. The size of the *work* array. Constraints: *lwork* ≥ max(1, *n*) if *side* = 'L'; *lwork* ≥ max(1, *m*) if *side* = 'R'. If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

c	Overwritten by the product $Q*C$, Q^H*C, $C*Q$, or $C*Q^H$ (as specified by *side* and *trans*).
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = −*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine unmlq interface are the following:

a	Holds the matrix A of size (*k*,*m*).
tau	Holds the vector of length (*k*).
c	Holds the matrix C of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
trans	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = *n*blocksize* (if *side* = 'L') or *lwork* = *m*blocksize* (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is ?ormlq.

?geqlf

Computes the QL factorization of a general m-by-n matrix.

Syntax

Fortran 77:

```
call sgeqlf(m, n, a, lda, tau, work, lwork, info)
call dgeqlf(m, n, a, lda, tau, work, lwork, info)
call cgeqlf(m, n, a, lda, tau, work, lwork, info)
call zgeqlf(m, n, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call geqlf(a [, tau] [,info])
```

Description

The routine forms the *QL* factorization of a general *m*-by-*n* matrix *A*. No pivoting is performed.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) `elementary reflectors`. Routines are provided to work with *Q* in this representation.

Input Parameters

m	`INTEGER`. The number of rows in the matrix *A* (*m* ≥ 0).
n	`INTEGER`. The number of columns in *A* (*n* ≥ 0).

a, *work*	`REAL` for `sgeqlf`
	`DOUBLE PRECISION` for `dgeqlf`
	`COMPLEX` for `cgeqlf`
	`DOUBLE COMPLEX` for `zgeqlf`.
	Arrays: *a*(*lda*,*) contains the matrix *A*.
	The second dimension of *a* must be at least max(1, *n*).
	work is a workspace array, its dimension `max(1, lwork)`.

lda	`INTEGER`. The first dimension of *a*; at least max(1, *m*).
lwork	`INTEGER`. The size of the *work* array; at least max(1, *n*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by `xerbla`. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	Overwritten on exit by the factorization data as follows: if *m* < *n*, the lower triangle of the subarray *a*(*m*-*n*+1:*m*, 1:*n*) contains the *n*-by-*n* lower triangular matrix *L*; if *m* ≤ *n*, the elements on and below the (*n*-*m*)-th superdiagonal contain the *m*-by-*n* lower trapezoidal matrix *L*; in both cases, the remaining elements, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.
tau	`REAL` for `sgeqlf` `DOUBLE PRECISION` for `dgeqlf`

	COMPLEX for cgeqlf DOUBLE COMPLEX for zgeqlf. Array, DIMENSION at least max(1, min(*m*, *n*)). Contains scalar factors of the elementary reflectors for the matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine geqlf interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
tau	Holds the vector of length min(*m*,*n*).

Application Notes

For better performance, try using *lwork* =*n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Related routines include:

?orgql	to generate matrix Q (for real matrices);
?ungql	to generate matrix Q (for complex matrices);
?ormql	to apply matrix Q (for real matrices);
?unmql	to apply matrix Q (for complex matrices).

?orgql

Generates the real matrix Q of the QL factorization formed by ?geqlf.

Syntax

Fortran 77:

```
call sorgql(m, n, k, a, lda, tau, work, lwork, info)
call dorgql(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call orgql(a, tau [,info])
```

Description

The routine generates an m-by-n real matrix Q with orthonormal columns, which is defined as the last n columns of a product of k elementary reflectors $H(i)$ of order m: $Q = H(k) *...* H(2)*H(1)$ as returned by the routines sgeqlf/dgeqlf. Use this routine after a call to sgeqlf/dgeqlf.

Input Parameters

m	INTEGER. The number of rows of the matrix Q ($m \geq 0$).
n	INTEGER. The number of columns of the matrix Q ($m \geq n \geq 0$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
a, *tau*, *work*	REAL for sorgql DOUBLE PRECISION for dorgql

	Arrays: *a*(*lda*,*), *tau*(*). On entry, the (*n* - *k* + *i*)th column of *a* must contain the vector which defines the elementary reflector *H*(*i*), for *i* = 1,2,...,*k*, as returned by sgeqlf/dgeqlf in the last *k* columns of its array argument *a*; *tau*(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by sgeqlf/dgeqlf; The second dimension of *a* must be at least max(1, *n*). The dimension of *tau* must be at least max(1, *k*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array; at least max(1, *n*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the *m*-by-*n* matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine orgql interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
tau	Holds the vector of length (*k*).

Application Notes

For better performance, try using *lwork* =*n*blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is ?ungql.

?ungql

Generates the complex matrix Q of the QL factorization formed by ?geqlf.

Syntax

Fortran 77:

```
call cungql(m, n, k, a, lda, tau, work, lwork, info)
call zungql(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call ungql(a, tau [,info])
```

Description

The routine generates an *m*-by-*n* complex matrix *Q* with orthonormal columns, which is defined as the last *n* columns of a product of *k* elementary reflectors *H*(*i*) of order *m*: *Q* = *H*(*k*) *...* *H*(2)*H*(1) as returned by the routines cgeqlf/zgeqlf . Use this routine after a call to cgeqlf/zgeqlf.

Input Parameters

m	INTEGER. The number of rows of the matrix *Q* (*m*≥ 0).
n	INTEGER. The number of columns of the matrix *Q* (*m*≥*n*≥ 0).
k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q* (*n*≥ *k*≥ 0).
a, *tau*, *work*	COMPLEX for cungql DOUBLE COMPLEX for zungql Arrays: *a*(*lda*,*), *tau*(*), *work*(*lwork*). On entry, the (*n* - *k* + *i*)th column of *a* must contain the vector which defines the elementary reflector *H*(*i*), for i = 1,2,...,*k*, as returned by cgeqlf/zgeqlf in the last *k* columns of its array argument *a*; *tau*(i) must contain the scalar factor of the elementaryreflector *H*(*i*), as returned by cgeqlf/zgeqlf; The second dimension of *a* must be at least max(1, *n*). The dimension of *tau* must be at least max(1, *k*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array; at least max(1, *n*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the *m*-by-*n* matrix *Q*.

work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ungql interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
tau	Holds the vector of length (*k*).

Application Notes

For better performance, try using *lwork* =*n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is ?orgql.

?ormql

Multiplies a real matrix by the orthogonal matrix Q of the QL factorization formed by ?geqlf.

Syntax

Fortran 77:

```
call sormql(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call dormql(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call ormql(a, tau, c [,side] [,trans] [,info])
```

Description

This routine multiplies a real m-by-n matrix C by Q or Q^T, where Q is the orthogonal matrix Q of the QL factorization formed by the routine sgeqlf/dgeqlf .

Depending on the parameters *side* and *trans*, the routine ?ormql can form one of the matrix products $Q*C$, Q^T*C, $C*Q$, or $C*Q^T$ (overwriting the result over C).

Input Parameters

side	CHARACTER*1. Must be either 'L' or 'R'. If *side* = 'L', Q or Q^T is applied to C from the left. If *side* = 'R', Q or Q^T is applied to C from the right.
trans	CHARACTER*1. Must be either 'N' or 'T'. If *trans* = 'N', the routine multiplies C by Q. If *trans* = 'T', the routine multiplies C by Q^T.
m	INTEGER. The number of rows in the matrix C ($m \geq 0$).
n	INTEGER. The number of columns in C ($n \geq 0$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints: $0 \leq k \leq m$ if *side* = 'L'; $0 \leq k \leq n$ if *side* = 'R'.
a, *tau*, *c*, *work*	REAL for sormql

DOUBLE PRECISION for dormql.
Arrays: *a*(*lda*,*), *tau*(*), *c*(*ldc*,*).
On entry, the *i*th column of *a* must contain the vector which defines the elementary reflector H_i, for i = 1,2,...,*k*, as returned by sgeqlf/dgeqlf in the last *k* columns of its array argument *a*.
The second dimension of *a* must be at least max(1, *k*).
tau(i) must contain the scalar factor of the elementary reflector H_i, as returned by sgeqlf/dgeqlf.
The dimension of *tau* must be at least max(1, *k*).
c(*ldc*,*) contains the *m*-by-*n* matrix *C*.
The second dimension of *c* must be at least max(1, *n*)
work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The first dimension of *a*;

if *side* = 'L', *lda* ≥ max(1, *m*);

if *side* = 'R', *lda* ≥ max(1, *n*).

ldc — INTEGER. The first dimension of *c*; *ldc* ≥ max(1, *m*).

lwork — INTEGER. The size of the *work* array. Constraints:

lwork ≥ max(1, *n*) if *side* = 'L';

lwork ≥ max(1, *m*) if *side* = 'R'.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.
See Application Notes for the suggested value of *lwork*.

Output Parameters

c — Overwritten by the product $Q*C$, Q^T*C, $C*Q$, or $C*Q^T$ (as specified by *side* and *trans*).

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.

If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ormql interface are the following:

a	Holds the matrix *A* of size (*r*,*k*). *r* = *m* if *side* = 'L'. *r* = *n* if *side* = 'R'.
tau	Holds the vector of length (*k*).
c	Holds the matrix *C* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
trans	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = *n***blocksize* (if *side* = 'L') or *lwork* = *m***blocksize* (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is ?unmql.

?unmql

Multiplies a complex matrix by the unitary matrix Q of the QL factorization formed by ?geqlf.

Syntax

Fortran 77:

```
call cunmql(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call zunmql(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call unmql(a, tau, c [,side] [,trans] [,info])
```

Description

The routine multiplies a complex m-by-n matrix C by Q or Q^H, where Q is the unitary matrix Q of the QL factorization formed by the routine cgeqlf/zgeqlf .

Depending on the parameters *side* and *trans*, the routine ?unmql can form one of the matrix products $Q*C$, Q^H*C, $C*Q$, or $C*Q^H$ (overwriting the result over C).

Input Parameters

side	CHARACTER*1. Must be either 'L' or 'R'. If *side* = 'L', Q or Q^H is applied to C from the left. If *side* = 'R', Q or Q^H is applied to C from the right.
trans	CHARACTER*1. Must be either 'N' or 'C'. If *trans* = 'N', the routine multiplies C by Q. If *trans* = 'C', the routine multiplies C by Q^H.
m	INTEGER. The number of rows in the matrix C ($m \geq 0$).
n	INTEGER. The number of columns in C ($n \geq 0$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints: $0 \leq k \leq m$ if *side* = 'L'; $0 \leq k \leq n$ if *side* = 'R'.
a, *tau*, *c*, *work*	COMPLEX for cunmql

DOUBLE COMPLEX for zunmql.
Arrays: *a*(*lda*,*), *tau*(*), *c*(*ldc*,*), *work*(*lwork*).
On entry, the *i*-th column of *a* must contain the vector which defines the elementary reflector *H*(*i*), for i = 1,2,...,*k*, as returned by cgeqlf/zgeqlf in the last *k* columns of its array argument *a*.
The second dimension of *a* must be at least max(1, *k*).
tau(i) must contain the scalar factor of the elementary reflector*H*(*i*), as returned by cgeqlf/zgeqlf.
The dimension of *tau* must be at least max(1, *k*).
c(*ldc*,*) contains the *m*-by-*n* matrix *C*.
The second dimension of *c* must be at least max(1, *n*)
work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The first dimension of *a*; *lda* ≥ max(1, *n*).

ldc — INTEGER. The first dimension of *c*; *ldc* ≥ max(1, *m*).

lwork — INTEGER. The size of the *work* array. Constraints:

lwork ≥ max(1, *n*) if *side* = 'L';

lwork ≥ max(1, *m*) if *side* = 'R'.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.
See Application Notes for the suggested value of *lwork*.

Output Parameters

c — Overwritten by the product *Q***C*, Q^H**C*, *C***Q*, or *C**Q^H (as specified by *side* and *trans*).

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `unmql` interface are the following:

a	Holds the matrix *A* of size (*r*,*k*). *r* = *m* if *side* = 'L'. *r* = *n* if *side* = 'R'.
tau	Holds the vector of length (*k*).
c	Holds the matrix *C* of size (*m*,*n*).
side	Must be 'L' or 'L'. The default value is 'L'.
trans	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = *n***blocksize* (if *side* = 'L') or *lwork* = *m***blocksize* (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is ?ormql.

?gerqf

Computes the RQ factorization of a general m-by-n matrix.

Syntax

Fortran 77:

```
call sgerqf(m, n, a, lda, tau, work, lwork, info)
call dgerqf(m, n, a, lda, tau, work, lwork, info)
call cgerqf(m, n, a, lda, tau, work, lwork, info)
call zgerqf(m, n, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call gerqf(a [, tau] [,info])
```

Description

The routine forms the *RQ* factorization of a general *m*-by-*n* matrix *A*. No pivoting is performed.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) elementary reflectors. Routines are provided to work with *Q* in this representation.

Input Parameters

m	INTEGER. The number of rows in the matrix *A* (*m* ≥ 0).
n	INTEGER. The number of columns in *A* (*n* ≥ 0).
a, *work*	REAL for sgerqf DOUBLE PRECISION for dgerqf COMPLEX for cgerqf DOUBLE COMPLEX for zgerqf. Arrays: *a*(*lda*,*) contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array;

lwork ≥ max(1, *m*).
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.
See Application Notes for the suggested value of *lwork*.

Output Parameters

a
Overwritten on exit by the factorization data as follows:
if *m* ≤ *n*, the upper triangle of the subarray *a*(1:*m*, *n*-*m*+1:*n*) contains the *m*-by-*m* upper triangular matrix *R*;
if *m* ≥ *n*, the elements on and above the (*m*-*n*)th subdiagonal contain the *m*-by-*n* upper trapezoidal matrix *R*;
in both cases, the remaining elements, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of min(*m*,*n*) elementary reflectors.

tau
REAL for sgerqf
DOUBLE PRECISION for dgerqf
COMPLEX for cgerqf
DOUBLE COMPLEX for zgerqf.
Array, DIMENSION at least max (1, min(*m*, *n*)).
Contains scalar factors of the elementary reflectors for the matrix *Q*.

work(1)
If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info
INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gerqf interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
tau	Holds the vector of length min(*m*,*n*).

Application Notes

For better performance, try using *lwork* =*m***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Related routines include:

?orgrq	to generate matrix Q (for real matrices);
?ungrq	to generate matrix Q (for complex matrices);
?ormrq	to apply matrix Q (for real matrices);
?unmrq	to apply matrix Q (for complex matrices).

?orgrq

Generates the real matrix Q of the RQ factorization formed by ?gerqf.

Syntax

Fortran 77:

```
call sorgrq(m, n, k, a, lda, tau, work, lwork, info)
call dorgrq(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call orgrq(a, tau [,info])
```

Description

The routine generates an *m*-by-*n* real matrix *Q* with orthonormal rows, which is defined as the last *m* rows of a product of *k* elementary reflectors *H*(*i*) of order *n*: *Q* = *H*(1)**H*(2)**...***H*(*k*) as returned by the routines sgerqf/dgerqf. Use this routine after a call to sgerqf/dgerqf.

Input Parameters

m	INTEGER. The number of rows of the matrix *Q* (*m*≥ 0).
n	INTEGER. The number of columns of the matrix *Q* (*n*≥ *m*).
k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q* (*m*≥ *k*≥ 0).
a, *tau*, *work*	REAL for sorgrq DOUBLE PRECISION for dorgrq Arrays: *a*(*lda*,*), *tau*(*). On entry, the (*m* - *k* + *i*)th row of *a* must contain the vector which defines the elementary reflector *H*(*i*), for i = 1,2,...,*k*, as returned by sgerqf/dgerqf in the last *k* rows of its array argument *a*; *tau*(i) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by sgerqf/dgerqf; The second dimension of *a* must be at least max(1, *n*).

	The dimension of *tau* must be at least max(1, *k*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array; at least max(1, *m*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the *m*-by-*n* matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine orgrq interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
tau	Holds the vector of length (*k*).

Application Notes

For better performance, try using *lwork* =*m***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is ?ungrq.

?ungrq

Generates the complex matrix Q of the RQ factorization formed by ?gerqf.

Syntax

Fortran 77:

```
call cungrq(m, n, k, a, lda, tau, work, lwork, info)
call zungrq(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call ungrq(a, tau [,info])
```

Description

The routine generates an *m*-by-*n* complex matrix *Q* with orthonormal rows, which is defined as the last *m* rows of a product of *k* elementary reflectors *H*(*i*) of order *n*: *Q* = *H*(1)*H** *H*(2)*H**...**H*(*k*)*H* as returned by the routines sgerqf/dgerqf. Use this routine after a call to sgerqf/dgerqf.

Input Parameters

m — INTEGER. The number of rows of the matrix *Q* (*m* ≥ 0).

n — INTEGER. The number of columns of the matrix *Q* (*n* ≥ *m*).

k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q* (*m*≥ *k*≥ 0).
a, *tau*, *work*	REAL for cungrq DOUBLE PRECISION for zungrq Arrays: *a*(*lda*,*), *tau*(*), *work*(*lwork*). On entry, the (*m* - *k* + *i*)th row of *a* must contain the vector which defines the elementary reflector *H*(*i*), for i = 1,2,...,*k*, as returned by sgerqf/dgerqf in the last *k* rows of its array argument *a*; *tau*(i) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by sgerqf/dgerqf; The second dimension of *a* must be at least max(1, *n*). The dimension of *tau* must be at least max(1, *k*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array; at least max(1, *m*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the *m*-by-*n* matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ungrq interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
tau	Holds the vector of length (*k*).

Application Notes

For better performance, try using *lwork* =*m***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is ?orgrq.

?ormrq

Multiplies a real matrix by the orthogonal matrix Q of the RQ factorization formed by ?gerqf.

Syntax

Fortran 77:

```
call sormrq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call dormrq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call ormrq(a, tau, c [,side] [,trans] [,info])
```

Description

The routine multiplies a real m-by-n matrix C by Q or Q^T, where Q is the real orthogonal matrix defined as a product of k elementary reflectors H_i : $Q = H_1 H_2 \ldots H_k$ as returned by the RQ factorization routine sgerqf/dgerqf .

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products QC, Q^TC, CQ, or CQ^T (overwriting the result over C).

Input Parameters

side	CHARACTER*1. Must be either 'L' or 'R'. If *side* = 'L', Q or Q^T is applied to C from the left. If *side* = 'R', Q or Q^T is applied to C from the right.
trans	CHARACTER*1. Must be either 'N' or 'T'. If *trans* = 'N', the routine multiplies C by Q. If *trans* = 'T', the routine multiplies C by Q^T.
m	INTEGER. The number of rows in the matrix C ($m \geq 0$).
n	INTEGER. The number of columns in C ($n \geq 0$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints: $0 \leq k \leq m$, if *side* = 'L'; $0 \leq k \leq n$, if *side* = 'R'.
a, *tau*, *c*, *work*	REAL for sormrq DOUBLE PRECISION for dormrq. Arrays: *a*(*lda*,*), *tau*(*), *c*(*ldc*,*). On entry, the *i*th row of *a* must contain the vector which defines the elementary reflector H_i, for i = 1,2,...,k, as returned by sgerqf/dgerqf in the last k rows of its array argument *a*. The second dimension of *a* must be at least max(1, m) if *side* = 'L', and at least max(1, n) if *side* = 'R'. *tau*(i) must contain the scalar factor of the elementary reflector H_i, as returned by sgerqf/dgerqf. The dimension of *tau* must be at least max(1, k). *c*(*ldc*,*) contains the m-by-n matrix C. The second dimension of *c* must be at least max(1, n)

	work is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *k*).
ldc	INTEGER. The first dimension of *c*; *ldc* ≥ max(1, *m*).
lwork	INTEGER. The size of the *work* array. Constraints: *lwork* ≥ max(1, *n*) if *side* = 'L'; *lwork* ≥ max(1, *m*) if *side* = 'R'. If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

c	Overwritten by the product QC, Q^TC, CQ, or CQ^T (as specified by *side* and *trans*).
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = −*i*, the *i*th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ormrq interface are the following:

a	Holds the matrix A of size (*k*,*m*).
tau	Holds the vector of length (*k*).
c	Holds the matrix C of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
trans	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = *n*blocksize* (if *side* = 'L') or *lwork* = *m*blocksize* (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is ?unmrq.

?unmrq

Multiplies a complex matrix by the unitary matrix Q of the RQ factorization formed by ?gerqf.

Syntax

Fortran 77:

```
call cunmrq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call zunmrq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call unmrq(a, tau, c [,side] [,trans] [,info])
```

Description

The routine multiplies a complex *m*-by-*n* matrix *C* by *Q* or Q^H, where *Q* is the complex unitary matrix defined as a product of *k* elementary reflectors *H(i)* of order *n*: *Q = H(1)H* H(2)H*...*H(k)H*as returned by the *RQ* factorization routine cgerqf/zgerqf .

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products *Q*C*, Q^H**C*, *C*Q*, or *C**Q^H (overwriting the result over *C*).

Input Parameters

side	CHARACTER*1. Must be either 'L' or 'R'. If *side* = 'L', *Q* or Q^H is applied to *C* from the left. If *side* = 'R', *Q* or Q^H is applied to *C* from the right.
trans	CHARACTER*1. Must be either 'N' or 'C'. If *trans* = 'N', the routine multiplies *C* by *Q*. If *trans* = 'C', the routine multiplies *C* by Q^H.
m	INTEGER. The number of rows in the matrix *C* (*m* ≥ 0).
n	INTEGER. The number of columns in *C* (*n* ≥ 0).
k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints: 0 ≤ *k* ≤ *m*, if *side* = 'L'; 0 ≤ *k* ≤ *n*, if *side* = 'R'.
a, *tau*, *c*, *work*	COMPLEX for cunmrq DOUBLE COMPLEX for zunmrq. Arrays: *a*(*lda*,*), *tau*(*), *c*(*ldc*,*), *work*(*lwork*). On entry, the *i*th row of *a* must contain the vector which defines the elementary reflector *H*(*i*), for i = 1,2,...,*k*, as returned by cgerqf/zgerqf in the last *k* rows of its array argument *a*. The second dimension of *a* must be at least max(1, *m*) if *side* = 'L', and at least max(1, *n*) if *side* = 'R'. *tau*(i) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by cgerqf/zgerqf. The dimension of *tau* must be at least max(1, *k*). *c*(*ldc*,*) contains the *m*-by-*n* matrix *C*. The second dimension of *c* must be at least max(1, *n*)

	work is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *k*) .
ldc	INTEGER. The first dimension of *c*; *ldc* ≥ max(1, *m*).
lwork	INTEGER. The size of the *work* array. Constraints: *lwork* ≥ max(1, *n*) if *side* = 'L'; *lwork* ≥ max(1, *m*) if *side* = 'R'. If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

c	Overwritten by the product *Q*C*, Q^H*C, *C*Q*, or $C*Q^H$ (as specified by *side* and *trans*).
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine unmrq interface are the following:

a	Holds the matrix *A* of size (*k*,*m*).
tau	Holds the vector of length (*k*).
c	Holds the matrix *C* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
trans	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = *n*blocksize* (if *side* = 'L') or *lwork* = *m*blocksize* (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is ?ormrq.

?tzrzf

Reduces the upper trapezoidal matrix A to upper triangular form.

Syntax

Fortran 77:

```
call stzrzf(m, n, a, lda, tau, work, lwork, info)
call dtzrzf(m, n, a, lda, tau, work, lwork, info)
call ctzrzf(m, n, a, lda, tau, work, lwork, info)
call ztzrzf(m, n, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call tzrzf(a [, tau] [,info])
```

Description

This routine reduces the m-by-n ($m \le n$) real/complex upper trapezoidal matrix A to upper triangular form by means of orthogonal/unitary transformations. The upper trapezoidal matrix A is factored as

$A = (R\ 0) * Z$,

where Z is an n-by-n orthogonal/unitary matrix and R is an m-by-m upper triangular matrix.

See ?larz that applies an elementary reflector returned by ?tzrzf to a general matrix.

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \ge 0$).
n	INTEGER. The number of columns in A ($n \ge m$).
a, *work*	REAL for stzrzf DOUBLE PRECISION for dtzrzf COMPLEX for ctzrzf DOUBLE COMPLEX for ztzrzf. Arrays: *a*(*lda*,*), *work*(*lwork*) .The leading m-by-n upper trapezoidal part of the array *a* contains the matrix A to be factorized. The second dimension of *a* must be at least max(1, n). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, m).
lwork	INTEGER. The size of the *work* array; *lwork* ≥ max(1, m). If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	Overwritten on exit by the factorization data as follows:

	the leading *m*-by-*m* upper triangular part of *a* contains the upper triangular matrix *R*, and elements *m* +1 to *n* of the first *m* rows of *a*, with the array *tau*, represent the orthogonal matrix *Z* as a product of *m* elementary reflectors.
tau	REAL for stzrzf DOUBLE PRECISION for dtzrzf COMPLEX for ctzrzf DOUBLE COMPLEX for ztzrzf. Array, DIMENSION at least max (1, *m*). Contains scalar factors of the elementary reflectors for the matrix *Z*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine tzrzf interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
tau	Holds the vector of length (*m*).

Application Notes

For better performance, try using *lwork* =*m***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Related routines include:

?ormrz	to apply matrix Q (for real matrices)
?unmrz	to apply matrix Q (for complex matrices).

?ormrz

Multiplies a real matrix by the orthogonal matrix defined from the factorization formed by ?tzrzf.

Syntax

Fortran 77:

```
call sormrz(side, trans, m, n, k, l, a, lda, tau, c, ldc, work, lwork, info)
call dormrz(side, trans, m, n, k, l, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call ormrz(a, tau, c, l [, side] [,trans] [,info])
```

Description

The routine multiplies a real *m*-by-*n* matrix *C* by *Q* or Q^T, where *Q* is the real orthogonal matrix defined as a product of *k* elementary reflectors *H(i)* of order *n*: *Q = H(1)* H(2)*...*H(k)* as returned by the factorization routine stzrzf/dtzrzf .

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products *Q*C*, Q^T**C*, *C*Q*, or *C**Q^T (overwriting the result over *C*).

The matrix *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side	CHARACTER*1. Must be either 'L' or 'R'. If *side* = 'L', *Q* or Q^T is applied to *C* from the left.

	If *side* = 'R', Q or Q^T is applied to C from the right.
trans	CHARACTER*1. Must be either 'N' or 'T'. If *trans* = 'N', the routine multiplies C by Q. If *trans* = 'T', the routine multiplies C by Q^T.
m	INTEGER. The number of rows in the matrix C ($m \geq 0$).
n	INTEGER. The number of columns in C ($n \geq 0$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints: $0 \leq k \leq m$, if *side* = 'L'; $0 \leq k \leq n$, if *side* = 'R'.
l	INTEGER. The number of columns of the matrix A containing the meaningful part of the Householder reflectors. Constraints: $0 \leq l \leq m$, if *side* = 'L'; $0 \leq l \leq n$, if *side* = 'R'.
a, *tau*, *c*, *work*	REAL for sormrz DOUBLE PRECISION for dormrz. Arrays: *a*(*lda*,*), *tau*(*), *c*(*ldc*,*). On entry, the *i*th row of *a* must contain the vector which defines the elementary reflector *H(i)*, for i = 1,2,...,*k*, as returned by stzrzf/dtzrzf in the last *k* rows of its array argument *a*. The second dimension of *a* must be at least max(1, *m*) if *side* = 'L', and at least max(1, *n*) if *side* = 'R'. *tau*(i) must contain the scalar factor of the elementary reflector *H(i)*, as returned by stzrzf/dtzrzf. The dimension of *tau* must be at least max(1, *k*). *c*(*ldc*,*) contains the *m*-by-*n* matrix C. The second dimension of *c* must be at least max(1, *n*) *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *k*) .
ldc	INTEGER. The first dimension of *c*; *ldc* ≥ max(1, *m*).
lwork	INTEGER. The size of the *work* array. Constraints:

lwork ≥ max(1, *n*) if *side* = 'L';

lwork ≥ max(1, *m*) if *side* = 'R'.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

c	Overwritten by the product $Q*C$, Q^T*C, $C*Q$, or $C*Q^T$ (as specified by *side* and *trans*).
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ormrz interface are the following:

a	Holds the matrix *A* of size (*k*,*m*).
tau	Holds the vector of length (*k*).
c	Holds the matrix *C* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
trans	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = *n***blocksize* (if *side* = 'L') or *lwork* = *m***blocksize* (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is ?unmrz.

?unmrz

Multiplies a complex matrix by the unitary matrix defined from the factorization formed by ?tzrzf.

Syntax

Fortran 77:

```
call cunmrz(side, trans, m, n, k, l, a, lda, tau, c, ldc, work, lwork, info)
call zunmrz(side, trans, m, n, k, l, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call unmrz(a, tau, c, l [,side] [,trans] [,info])
```

Description

The routine multiplies a complex *m*-by-*n* matrix *C* by *Q* or Q^H, where *Q* is the unitary matrix defined as a product of *k* elementary reflectors *H*(*i*):

Q = *H*(1)*H** *H*(2)*H**...**H*(*k*)*H* as returned by the factorization routine ctzrzf/ztzrzf .

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products *Q***C*, Q^H**C*, *C***Q*, or *C***Q^H** (overwriting the result over *C*).

The matrix *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side	CHARACTER*1. Must be either 'L' or 'R'. If *side* = 'L', Q or Q^H is applied to C from the left. If *side* = 'R', Q or Q^H is applied to C from the right.
trans	CHARACTER*1. Must be either 'N' or 'C'. If *trans* = 'N', the routine multiplies C by Q. If *trans* = 'C', the routine multiplies C by Q^H.
m	INTEGER. The number of rows in the matrix C ($m \geq 0$).
n	INTEGER. The number of columns in C ($n \geq 0$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints: $0 \leq k \leq m$, if *side* = 'L'; $0 \leq k \leq n$, if *side* = 'R'.
l	INTEGER. The number of columns of the matrix A containing the meaningful part of the Householder reflectors. Constraints: $0 \leq l \leq m$, if *side* = 'L'; $0 \leq l \leq n$, if *side* = 'R'.
a, *tau*, *c*, *work*	COMPLEX for cunmrz DOUBLE COMPLEX for zunmrz. Arrays: *a*(*lda*,*), *tau*(*), *c*(*ldc*,*), *work*(*lwork*). On entry, the *i*th row of *a* must contain the vector which defines the elementary reflector *H*(*i*), for i = 1,2,...,*k*, as returned by ctzrzf/ztzrzf in the last *k* rows of its array argument *a*. The second dimension of *a* must be at least max(1, *m*) if *side* = 'L', and at least max(1, *n*) if *side* = 'R'. *tau*(i) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by ctzrzf/ztzrzf. The dimension of *tau* must be at least max(1, *k*). *c*(*ldc*,*) contains the *m*-by-*n* matrix *C*. The second dimension of *c* must be at least max(1, *n*) *work* is a workspace array, its dimension max(1, *lwork*).

lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *k*).
ldc	INTEGER. The first dimension of *c*; *ldc* ≥ max(1, *m*).
lwork	INTEGER. The size of the *work* array. Constraints: *lwork* ≥ max(1, *n*) if *side* = 'L'; *lwork* ≥ max(1, *m*) if *side* = 'R'. If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

c	Overwritten by the product $Q*C$, Q^H*C, $C*Q$, or $C*Q^H$ (as specified by *side* and *trans*).
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = −*i*, the *i*th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine unmrz interface are the following:

a	Holds the matrix *A* of size (*k*,*m*).
tau	Holds the vector of length (*k*).
c	Holds the matrix *C* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
trans	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = *n*blocksize* (if *side* = 'L') or *lwork* = *m*blocksize* (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is ?ormrz.

?ggqrf

Computes the generalized QR factorization of two matrices.

Syntax

Fortran 77:

```
call sggqrf(n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call dggqrf(n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call cggqrf(n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call zggqrf(n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
```

Fortran 95:

```
call ggqrf(a, b [,taua] [,taub] [,info])
```

Description

The routine forms the generalized QR factorization of an n-by-m matrix A and an n-by-p matrix B as $A = Q*R$, $B = Q*T*Z$, where Q is an n-by-n orthogonal/unitary matrix, Z is a p-by-p orthogonal/unitary matrix, and R and T assume one of the forms:

$$R = \begin{array}{c} \\ m \\ n-m \end{array} \begin{array}{c} \begin{array}{c} m \end{array} \\ \begin{pmatrix} R_{11} \\ 0 \end{pmatrix} \end{array}, \quad \text{if } n \geq m$$

or

$$R = \begin{array}{c} \\ n \end{array} \begin{array}{c} \begin{array}{cc} n & m-n \end{array} \\ \begin{pmatrix} R_{11} & R_{12} \end{pmatrix} \end{array}, \quad \text{if } n < m$$

where R_{11} is upper triangular, and

$$T = \begin{array}{c} \\ n \end{array} \begin{array}{c} \begin{array}{cc} p-n & n \end{array} \\ \begin{pmatrix} 0 & T_{12} \end{pmatrix} \end{array}, \quad \text{if} \quad n \leq p,$$

$$T = \begin{array}{c} \\ n-p \\ p \end{array} \begin{array}{c} \begin{array}{c} p \end{array} \\ \begin{pmatrix} T_{11} \\ T_{21} \end{pmatrix} \end{array}, \quad \text{if } n > p,$$

where T_{12} or T_{21} is a p-by-p upper triangular matrix.

In particular, if B is square and nonsingular, the GQR factorization of A and B implicitly gives the QR factorization of $B^{-1}A$ as:

$B^{-1}*A = Z^{H}*(T^{-1}*R)$

Input Parameters

n	INTEGER. The number of rows of the matrices *A* and *B* (*n* ≥ 0).
m	INTEGER. The number of columns in *A* (*m* ≥ 0).
p	INTEGER. The number of columns in *B* (*p* ≥ 0).
a, *b*, *work*	REAL for sggqrf DOUBLE PRECISION for dggqrf COMPLEX for cggqrf DOUBLE COMPLEX for zggqrf. Arrays: *a*(*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1, *m*). *b*(*ldb*,*) contains the matrix *B*. The second dimension of *b* must be at least max(1, *p*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).
ldb	INTEGER. The first dimension of *b*; at least max(1, *n*).
lwork	INTEGER. The size of the *work* array; must be at least max(1, *n*, *m*, *p*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a, *b*	Overwritten by the factorization data as follows: on exit, the elements on and above the diagonal of the array *a* contain the min(*n*,*m*)-by-*m* upper trapezoidal matrix *R* (*R* is upper triangular if *n* ≥ *m*);the elements below the diagonal, with the array *taua*, represent the orthogonal/unitary matrix *Q* as a product of min(*n*,*m*) elementary reflectors ; if *n* ≤ *p*, the upper triangle of the subarray *b*(1:*n*, *p*-*n*+1:*p*) contains the *n*-by-*n* upper triangular matrix *T*;

	if $n > p$, the elements on and above the (n-p)th subdiagonal contain the n-by-p upper trapezoidal matrix T; the remaining elements, with the array *taub*, represent the orthogonal/unitary matrix Z as a product of elementary reflectors.

taua, *taub*	`REAL` for `sggqrf`
	`DOUBLE PRECISION` for `dggqrf`
	`COMPLEX` for `cggqrf`
	`DOUBLE COMPLEX` for `zggqrf`.
	Arrays, `DIMENSION` at least max (1, min(n, m)) for *taua* and at least max (1, min(n, p)) for *taub*. The array *taua* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Q.
	The array *taub* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Z.

work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = $-i$, the i-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `ggqrf` interface are the following:

a	Holds the matrix A of size (n,m).
b	Holds the matrix B of size (n,p).
taua	Holds the vector of length min(n,m).
taub	Holds the vector of length min(n,p).

Application Notes

For better performance, try using *lwork* ≥ max(*n*,*m*, *p*)*max(*nb1*,*nb2*,*nb3*), where *nb1* is the optimal blocksize for the *QR* factorization of an *n*-by-*m* matrix, *nb2* is the optimal blocksize for the *RQ* factorization of an *n*-by-*p* matrix, and *nb3* is the optimal blocksize for a call of ?ormqr/?unmqr.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?ggrqf

Computes the generalized RQ factorization of two matrices.

Syntax

Fortran 77:

```
call sggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
call dggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
call cggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
call zggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
```

Fortran 95:

```
call ggrqf(a, b [,taua] [,taub] [,info])
```

Description

The routine forms the generalized *RQ* factorization of an *m*-by-*n* matrix *A* and an *p*-by-*n* matrix *B* as *A = R*Q*, *B = Z*T*Q*, where *Q* is an *n*-by-*n* orthogonal/unitary matrix, *Z* is a *p*-by-*p* orthogonal/unitary matrix, and *R* and *T* assume one of the forms:

$$R = \begin{array}{c} \\ m \end{array} \begin{array}{c} \begin{array}{cc} n-m & m \end{array} \\ \begin{pmatrix} 0 & R_{12} \end{pmatrix} \end{array}, \quad \text{if } m \le n,$$

or

$$R = \begin{array}{c} m-n \\ n \end{array} \begin{array}{c} \begin{array}{c} n \end{array} \\ \begin{pmatrix} R_{11} \\ R_{21} \end{pmatrix} \end{array}, \quad \text{if } m > n,$$

where R_{11} or R_{21} is upper triangular, and

$$T = \begin{array}{c} \\ p-n \end{array} \begin{array}{c} \begin{array}{c} n \end{array} \\ \begin{pmatrix} T_{11} \\ 0 \end{pmatrix} \end{array}, \quad \text{if } p \ge n,$$

or

$$T = \begin{array}{c} p \end{array} \begin{array}{c} \begin{array}{cc} p & n-p \end{array} \\ \begin{pmatrix} T_{11} & T_{12} \end{pmatrix} \end{array}, \quad \text{if } p < n,$$

where T_{11} is upper triangular.

In particular, if B is square and nonsingular, the GRQ factorization of A and B implicitly gives the RQ factorization of $A*B^{-1}$ as:

$A*B^{-1} = (R*T^{-1})*Z^{H}$

Input Parameters

m	INTEGER. The number of rows of the matrix A ($m \geq 0$).
p	INTEGER. The number of rows in B ($p \geq 0$).
n	INTEGER. The number of columns of the matrices A and B ($n \geq 0$).
a, b, $work$	REAL for sggrqf DOUBLE PRECISION for dggrqf COMPLEX for cggrqf DOUBLE COMPLEX for zggrqf. Arrays: a(lda,*) contains the m-by-n matrix A. The second dimension of a must be at least max(1, n). b(ldb,*) contains the p-by-n matrix B. The second dimension of b must be at least max(1, n). $work$ is a workspace array, its dimension max(1, $lwork$).
lda	INTEGER. The first dimension of a; at least max(1, m).
ldb	INTEGER. The first dimension of b; at least max(1, p).
$lwork$	INTEGER. The size of the $work$ array; must be at least max(1, n, m, p). If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by xerbla. See Application Notes for the suggested value of $lwork$.

Output Parameters

a, b	Overwritten by the factorization data as follows: on exit, if $m \leq n$, the upper triangle of the subarray a(1:m, n-m+1:n) contains the m-by-m upper triangular matrix R;

	if $m > n$, the elements on and above the (m-n)th subdiagonal contain the m-by-n upper trapezoidal matrix R; the remaining elements, with the array *taua*, represent the orthogonal/unitary matrix Q as a product of elementary reflectors; the elements on and above the diagonal of the array *b* contain the min(p,n)-by-n upper trapezoidal matrix T (T is upper triangular if $p \geq n$); the elements below the diagonal, with the array *taub*, represent the orthogonal/unitary matrix Z as a product of elementary reflectors.

taua, *taub*	`REAL` for `sggrqf`
	`DOUBLE PRECISION` for `dggrqf`
	`COMPLEX` for `cggrqf`
	`DOUBLE COMPLEX` for `zggrqf`.
	Arrays, `DIMENSION` at least max (1, min(m, n)) for *taua* and at least max (1, min(p, n)) for *taub*.
	The array *taua* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Q.
	The array *taub* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Z.

work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	`INTEGER`. If *info* = 0, the execution is successful. If *info* = –*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `ggrqf` interface are the following:

a	Holds the matrix A of size (m,n).
b	Holds the matrix A of size (p,n).
taua	Holds the vector of length min(m,n).

taub Holds the vector of length min(*p*,*n*).

Application Notes

For better performance, try using

lwork ≥ max(*n*,*m*, *p*)*max(*nb1*,*nb2*,*nb3*),

where *nb1* is the optimal blocksize for the *RQ* factorization of an *m*-by-*n* matrix, *nb2* is the optimal blocksize for the *QR* factorization of an *p*-by-*n* matrix, and *nb3* is the optimal blocksize for a call of ?ormrq/?unmrq.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork*= -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork*= -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Singular Value Decomposition

This section describes LAPACK routines for computing the singular value decomposition (SVD) of a general *m*-by-*n* matrix *A*:

$A = U\Sigma V^H$.

In this decomposition, *U* and *V* are unitary (for complex *A*) or orthogonal (for real *A*); Σ is an *m*-by-*n* diagonal matrix with real diagonal elements σ_i:

$\sigma_1 < \sigma_2 < \ldots < \sigma_{\min(m,\ n)} < 0$.

The diagonal elements σ_i are singular values of *A*. The first min(*m*, *n*) columns of the matrices *U* and *V* are, respectively, left and right singular vectors of *A*. The singular values and singular vectors satisfy

$Av_i = \sigma_i u_i$ and $A^H ui = \sigma_i v_i$

where u_i and v_i are the i th columns of U and V, respectively.

To find the SVD of a general matrix A, call the LAPACK routine ?gebrd or ?gbbrd for reducing A to a bidiagonal matrix B by a unitary (orthogonal) transformation: $A = QBP^H$. Then call ?bdsqr, which forms the SVD of a bidiagonal matrix: $B = U_1 \Sigma V_1^H$.

Thus, the sought-for SVD of A is given by $A = U\Sigma V^H = (QU_1)\Sigma(V_1^H P^H)$.

Table 4-2 lists LAPACK routines (Fortran-77 interface) that perform singular value decomposition of matrices. Respective routine names in Fortran-95 interface are without the first symbol (see Routine Naming Conventions).

Table 4-2 Computational Routines for Singular Value Decomposition (SVD)

Operation	Real matrices	Complex matrices
Reduce A to a bidiagonal matrix B: $A = QBP^H$ (full storage)	?gebrd	?gebrd
Reduce A to a bidiagonal matrix B: $A = QBP^H$ (band storage)	?gbbrd	?gbbrd
Generate the orthogonal (unitary) matrix Q or P	?orgbr	?ungbr
Apply the orthogonal (unitary) matrix Q or P	?ormbr	?unmbr
Form singular value decomposition of the bidiagonal matrix B: $B = U\Sigma V^H$	?bdsqr ?bdsdc	?bdsqr

Figure 4-1 Decision Tree: Singular Value Decomposition

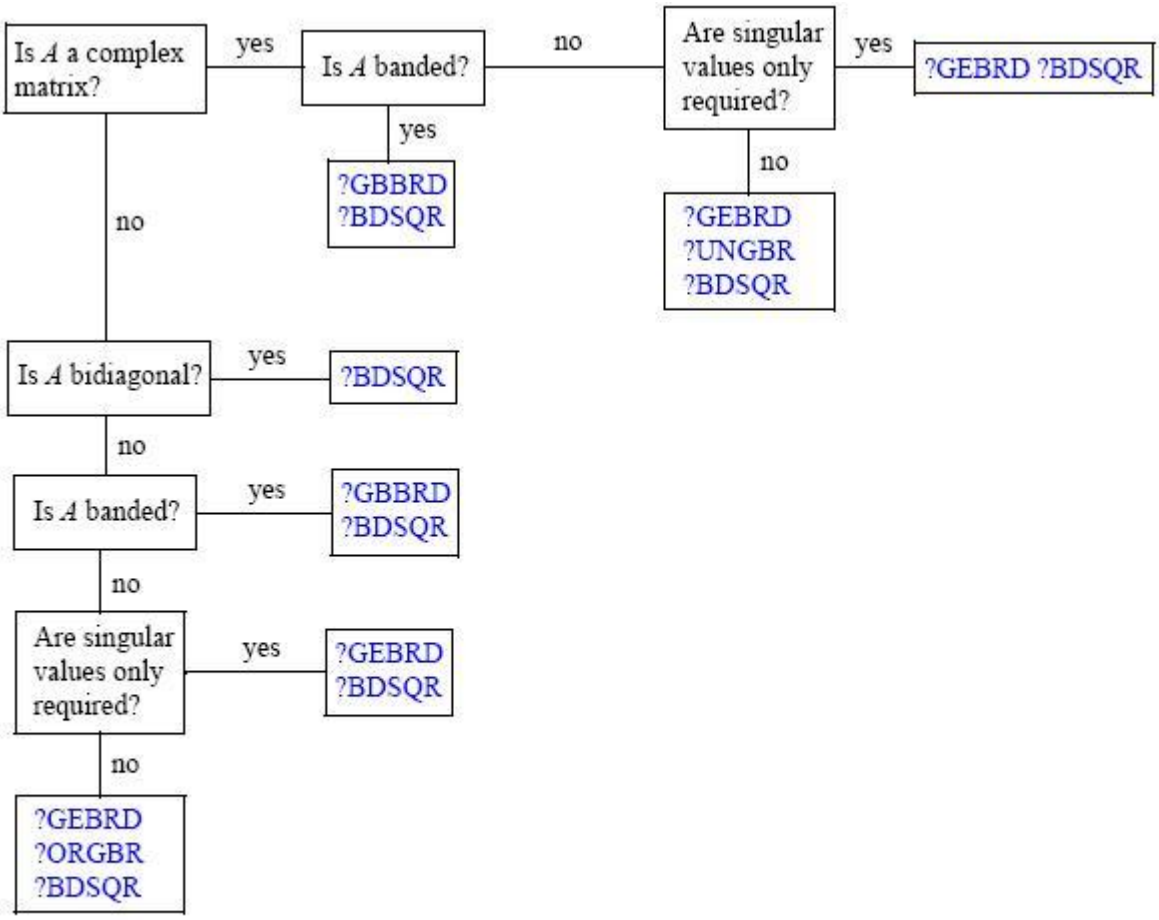


Figure 4-1 "Decision Tree: Singular Value Decomposition" presents a decision tree that helps you choose the right sequence of routines for SVD, depending on whether you need singular values only or singular vectors as well, whether `A` is real or complex, and so on.

You can use the SVD to find a minimum-norm solution to a (possibly) rank-deficient least-squares problem of minimizing $||Ax - b||^2$. The effective rank k of the matrix A can be determined as the number of singular values which exceed a suitable threshold. The minimum-norm solution is

$x = V_k(\Sigma_k)^{-1}c$,

where Σ_k is the leading k -by- k submatrix of Σ, the matrix V_k consists of the first k columns of $V = PV_1$, and the vector c consists of the first k elements of $U^Hb = U_1{}^HQ^Hb$.

?gebrd

Reduces a general matrix to bidiagonal form.

Syntax

Fortran 77:

```
call sgebrd(m, n, a, lda, d, e, tauq, taup, work, lwork, info)
call dgebrd(m, n, a, lda, d, e, tauq, taup, work, lwork, info)
call cgebrd(m, n, a, lda, d, e, tauq, taup, work, lwork, info)
call zgebrd(m, n, a, lda, d, e, tauq, taup, work, lwork, info)
```

Fortran 95:

```
call gebrd(a [, d] [,e] [,tauq] [,taup] [,info])
```

Description

The routine reduces a general m-by-n matrix A to a bidiagonal matrix B by an orthogonal (unitary) transformation.

If $m \geq n$, the reduction is given by

$$A = QBP^H = Q\begin{pmatrix} B_1 \\ 0 \end{pmatrix}P^H = Q_1B_1P^H,$$

where B_1 is an n-by-n upper diagonal matrix, Q and P are orthogonal or, for a complex A, unitary matrices; Q_1 consists of the first n columns of Q.

If $m < n$, the reduction is given by

$A = Q*B*P^H = Q*(B_10)*P^H = Q_1*B_1*P_1{}^H$,

where B_1 is an m-by-m lower diagonal matrix, Q and P are orthogonal or, for a complex A, unitary matrices; P_1 consists of the first m rows of P.

The routine does not form the matrices Q and P explicitly, but represents them as products of elementary reflectors. Routines are provided to work with the matrices Q and P in this representation:

If the matrix A is real,

- to compute Q and P explicitly, call ?orgbr.
- to multiply a general matrix by Q or P, call ?ormbr.

If the matrix A is complex,

- to compute Q and P explicitly, call ?ungbr.
- to multiply a general matrix by Q or P, call ?unmbr.

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
a, *work*	REAL for sgebrd DOUBLE PRECISION for dgebrd COMPLEX for cgebrd DOUBLE COMPLEX for zgebrd. Arrays: *a*(*lda*,*) contains the matrix A. The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
lwork	INTEGER. The dimension of *work*; at least max(1, *m*, *n*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	If $m \geq n$, the diagonal and first super-diagonal of *a* are overwritten by the upper bidiagonal matrix *B*. Elements below the diagonal are overwritten by details of *Q*, and the remaining elements are overwritten by details of *P*. If $m < n$, the diagonal and first sub-diagonal of *a* are overwritten by the lower bidiagonal matrix *B*. Elements above the diagonal are overwritten by details of *P*, and the remaining elements are overwritten by details of *Q*.
d	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Array, DIMENSION at least max(1, min(*m*, *n*)). Contains the diagonal elements of *B*.
e	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Array, DIMENSION at least max(1, min(*m*, *n*) - 1). Contains the off-diagonal elements of *B*.
tauq, *taup*	REAL for sgebrd DOUBLE PRECISION for dgebrd COMPLEX for cgebrd DOUBLE COMPLEX for zgebrd. Arrays, DIMENSION at least max (1, min(*m*, *n*)). Contain further details of the matrices *Q* and *P*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gebrd interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
d	Holds the vector of length min(*m*,*n*).
e	Holds the vector of length min(*m*,*n*)-1.
tauq	Holds the vector of length min(*m*,*n*).
taup	Holds the vector of length min(*m*,*n*).

Application Notes

For better performance, try using *lwork* = (*m* + *n*)***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrices *Q*, *B*, and *P* satisfy $QBP^H = A + E$, where $||E||_2 = c(n)\varepsilon ||A||_2$, $c(n)$ is a modestly increasing function of *n*, and ε is the machine precision.

The approximate number of floating-point operations for real flavors is

$(4/3)*n^2*(3*m - n)$ for $m \geq n$,

$(4/3)*m^2*(3*n - m)$ for $m < n$.

The number of operations for complex flavors is four times greater.

If *n* is much less than *m*, it can be more efficient to first form the *QR* factorization of *A* by calling ?geqrf and then reduce the factor *R* to bidiagonal form. This requires approximately $2*n^2*(m + n)$ floating-point operations.

If m is much less than n, it can be more efficient to first form the LQ factorization of A by calling ?gelqf and then reduce the factor L to bidiagonal form. This requires approximately $2*m^2*(m + n)$ floating-point operations.

?gbbrd

Reduces a general band matrix to bidiagonal form.

Syntax

Fortran 77:

```
call sgbbrd(vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt, ldpt, c, ldc, work, info)
call dgbbrd(vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt, ldpt, c, ldc, work, info)
call cgbbrd(vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt, ldpt, c, ldc, work, rwork, info)
call zgbbrd(vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt, ldpt, c, ldc, work, rwork, info)
```

Fortran 95:

```
call gbbrd(a [, c] [,d] [,e] [,q] [,pt] [,kl] [,m] [,info])
```

Description

This routine reduces an m-by-n band matrix A to upper bidiagonal matrix B: $A = Q*B*P^H$. Here the matrices Q and P are orthogonal (for real A) or unitary (for complex A). They are determined as products of Givens rotation matrices, and may be formed explicitly by the routine if required. The routine can also update a matrix C as follows: $C = Q^H*C$.

Input Parameters

vect	CHARACTER*1. Must be 'N' or 'Q' or 'P' or 'B'. If *vect* = 'N', neither Q nor P^H is generated. If *vect* = 'Q', the routine generates the matrix Q. If *vect* = 'P', the routine generates the matrix P^H. If *vect* = 'B', the routine generates both Q and P^H.
m	INTEGER. The number of rows in the matrix A ($m \geq 0$).

n	INTEGER. The number of columns in *A* (*n* ≥ 0).
ncc	INTEGER. The number of columns in *C* (*ncc* ≥ 0).
kl	INTEGER. The number of sub-diagonals within the band of *A* (*kl* ≥ 0).
ku	INTEGER. The number of super-diagonals within the band of *A* (*ku* ≥ 0).
ab, *c*, *work*	REAL for sgbbrd DOUBLE PRECISION for dgbbrd COMPLEX for cgbbrd DOUBLE COMPLEX for zgbbrd. Arrays: *ab*(*ldab*,*) contains the matrix *A* in band storage (see Matrix Storage Schemes). The second dimension of *a* must be at least max(1, *n*). *c*(*ldc*,*) contains an *m*-by-*ncc* matrix *C*. If *ncc* = 0, the array *c* is not referenced. The second dimension of *c* must be at least max(1, *ncc*). *work*(*) is a workspace array. The dimension of *work* must be at least 2*max(*m*, *n*) for real flavors, or max(*m*, *n*) for complex flavors.
ldab	INTEGER. The first dimension of the array *ab* (*ldab* ≥ *kl* + *ku* + 1).
ldq	INTEGER. The first dimension of the output array *q*. *ldq* ≥ max(1, *m*) if *vect* = 'Q' or 'B', *ldq* ≥ 1 otherwise.
ldpt	INTEGER. The first dimension of the output array *pt*. *ldpt* ≥ max(1, *n*) if *vect* = 'P' or 'B', *ldpt* ≥ 1 otherwise.
ldc	INTEGER. The first dimension of the array *c*. *ldc* ≥ max(1, *m*) if *ncc* > 0; *ldc* ≥ 1 if *ncc* = 0.
rwork	REAL for cgbbrd DOUBLE PRECISION for zgbbrd. A workspace array, DIMENSION at least max(*m*, *n*).

Output Parameters

ab	Overwritten by values generated during the reduction.
d	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Array, DIMENSION at least max(1, min(*m*, *n*)). Contains the diagonal elements of the matrix *B*.
e	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Array, DIMENSION at least max(1, min(*m*, *n*) - 1). Contains the off-diagonal elements of *B*.
q, *pt*	REAL for sgebrd DOUBLE PRECISION for dgebrd COMPLEX for cgebrd DOUBLE COMPLEX for zgebrd. Arrays: *q*(*ldq*,*) contains the output *m*-by-*m* matrix *Q*. The second dimension of *q* must be at least max(1, *m*). *p*(*ldpt*,*) contains the output *n*-by-*n* matrix P^T. The second dimension of *pt* must be at least max(1, *n*).
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gbbrd interface are the following:

a	Stands for argument *ab* in Fortan 77 interface . Holds the array *A* of size (*kl*+*ku*+1,*n*).
c	Holds the matrix *C* of size (*m*,*ncc*).
d	Holds the vector of length min(*m*,*n*).
e	Holds the vector of length min(*m*,*n*)-1.
q	Holds the matrix *Q* of size (*m*,*m*).

pt	Holds the matrix *PT* of size (*n*,*n*).
m	If omitted, assumed *m* = *n*.
kl	If omitted, assumed *kl* = *ku*.
ku	Restored as *ku* = *lda*-*kl*-1.
vect	Restored based on the presence of arguments *q* and pt as follows: *vect* = 'B', if both *q* and *pt* are present, *vect* = 'Q', if *q* is present and *pt* omitted, *vect* = 'P', if *q* is omitted and *pt* present, *vect* = 'N', if both *q* and *pt* are omitted.

Application Notes

The computed matrices *Q*, *B*, and *P* satisfy $Q*B*P^H = A + E$, where $||E||_2 = c(n)\varepsilon ||A||_2$, $c(n)$ is a modestly increasing function of *n*, and ε is the machine precision.

If *m* = *n*, the total number of floating-point operations for real flavors is approximately the sum of:

$6*n^2*(kl + ku)$ if *vect* = 'N' and *ncc* = 0,

$3*n^2*ncc*(kl + ku - 1)/(kl + ku)$ if *C* is updated, and

$3*n^3*(kl + ku - 1)/(kl + ku)$ if either *Q* or P^H is generated (double this if both).

To estimate the number of operations for complex flavors, use the same formulas with the coefficients 20 and 10 (instead of 6 and 3).

?orgbr

Generates the real orthogonal matrix Q or P^T determined by ?gebrd.

Syntax

Fortran 77:

```
call sorgbr(vect, m, n, k, a, lda, tau, work, lwork, info)
call dorgbr(vect, m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call orgbr(a, tau [,vect] [,info])
```

Description

The routine generates the whole or part of the orthogonal matrices Q and P_T formed by the routines sgebrd/dgebrd. Use this routine after a call to sgebrd/dgebrd. All valid combinations of arguments are described in Input parameters. In most cases you need the following:

To compute the whole m-by-m matrix Q:

```
call ?orgbr('Q', m, m, n, a ... )
```

(note that the array a must have at least m columns).

To form the n leading columns of Q if $m > n$:

```
call ?orgbr('Q', m, n, n, a ... )
```

To compute the whole n-by-n matrix P^T:

```
call ?orgbr('P', n, n, m, a ... )
```

(note that the array a must have at least n rows).

To form the m leading rows of P_T if $m < n$:

```
call ?orgbr('P', m, n, m, a ... )
```

Input Parameters

vect	CHARACTER*1. Must be 'Q' or 'P'. If *vect* = 'Q', the routine generates the matrix Q. If *vect* = 'P', the routine generates the matrix P^T.
m	INTEGER. The number of required rows of Q or P^T.
n	INTEGER. The number of required columns of Q or P^T.
k	INTEGER. One of the dimensions of A in ?gebrd: If *vect* = 'Q', the number of columns in A; If *vect* = 'P', the number of rows in A. Constraints: $m \geq 0$, $n \geq 0$, $k \geq 0$. For *vect* = 'Q': $k \leq n \leq m$ if $m > k$, or $m = n$ if $m \leq k$. For *vect* = 'P': $k \leq m \leq n$ if $n > k$, or $m = n$ if $n \leq k$.
a, *work*	REAL for sorgbr DOUBLE PRECISION for dorgbr. Arrays: *a*(*lda*,*) is the array a as returned by ?gebrd. The second dimension of a must be at least max(1, n).

	work is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
tau	REAL for sorgbr DOUBLE PRECISION for dorgbr. For *vect* = 'Q', the array *tauq* as returned by ?gebrd. For *vect* = 'P', the array *taup* as returned by ?gebrd. The dimension of *tau* must be at least max(1, min(*m*, *k*)) for *vect* = 'Q', or max(1, min(*m*, *k*)) for *vect* = 'P'.
lwork	INTEGER. The size of the *work* array. If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the orthogonal matrix Q or P^T (or the leading rows or columns thereof) as specified by *vect*, *m*, and *n*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine orgbr interface are the following:

a	Holds the matrix A of size (m,n).
tau	Holds the vector of length min(m,k) where $k = m$, if *vect* = 'P', $k = n$, if *vect* = 'Q'.
vect	Must be 'Q' or 'P'. The default value is 'Q'.

Application Notes

For better performance, try using *lwork* = min(*m*,*n*)***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $||E||_2 = O(\varepsilon)$.

The approximate numbers of floating-point operations for the cases listed in *Description* are as follows:

To form the whole of Q:

$(4/3)n(3m^2 - 3m*n + n^2)$ if $m > n$;

$(4/3)m^3$ if $m \le n$.

To form the n leading columns of Q when $m > n$:

$(2/3)n^2(3m - n^2)$ if $m > n$.

To form the whole of P_T:

$(4/3)n^3$ if $m \ge n$;

$(4/3)m(3n^2 - 3m*n + m^2)$ if $m < n$.

To form the m leading columns of P_T when $m < n$:

$(2/3)n^2(3m - n^2)$ if $m > n$.

The complex counterpart of this routine is ?ungbr.

?ormbr

Multiplies an arbitrary real matrix by the real orthogonal matrix Q or P_T determined by ?gebrd.

Syntax

Fortran 77:

```
call sormbr(vect, side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call dormbr(vect, side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call ormbr(a, tau, c [,vect] [,side] [,trans] [,info])
```

Description

Given an arbitrary real matrix C, this routine forms one of the matrix products $Q*C$, Q^T*C, $C*Q$, $C*Q^T$, $P*C$, P^T*C, $C*P$, or $C*P^T$, where Q and P are orthogonal matrices computed by a call to sgebrd/dgebrd. The routine overwrites the product on C.

Input Parameters

In the descriptions below, r denotes the order of Q or P^T:

If *side* = 'L', $r = m$; if *side* = 'R', $r = n$.

vect	CHARACTER*1. Must be 'Q' or 'P'. If *vect* = 'Q', then Q or Q^T is applied to C. If *vect* = 'P', then P or P_T is applied to C.
side	CHARACTER*1. Must be 'L' or 'R'. If *side* = 'L', multipliers are applied to C from the left. If *side* = 'R', they are applied to C from the right.
trans	CHARACTER*1. Must be 'N' or 'T'. If *trans* = 'N', then Q or P is applied to C. If *trans* = 'T', then Q^T or P^T is applied to C.
m	INTEGER. The number of rows in C.

n	INTEGER. The number of columns in *C*.
k	INTEGER. One of the dimensions of *A* in ?gebrd: If *vect* = 'Q', the number of columns in *A*; If *vect* = 'P', the number of rows in *A*. Constraints: $m \geq 0$, $n \geq 0$, $k \geq 0$.
a, *c*, *work*	REAL for sormbr DOUBLE PRECISION for dormbr. Arrays: *a*(*lda*,*) is the array *a* as returned by ?gebrd. Its second dimension must be at least max(1, min(*r*,*k*)) for *vect* = 'Q', or max(1, *r*)) for *vect* = 'P'. *c*(*ldc*,*) holds the matrix *C*. Its second dimension must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*. Constraints: *lda* ≥ max(1, *r*) if *vect* = 'Q'; *lda* ≥ max(1, min(*r*,*k*)) if *vect* = 'P'.
ldc	INTEGER. The first dimension of *c*; *ldc* ≥ max(1, *m*).
tau	REAL for sormbr DOUBLE PRECISION for dormbr. Array, DIMENSION at least max (1, min(*r*, *k*)). For *vect* = 'Q', the array *tauq* as returned by ?gebrd. For *vect* = 'P', the array *taup* as returned by ?gebrd.
lwork	INTEGER. The size of the *work* array. Constraints: *lwork* ≥ max(1, *n*) if *side* = 'L'; *lwork* ≥ max(1, *m*) if *side* = 'R'. If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

c	Overwritten by the product *Q*C*, Q^T**C*, *C*Q*, *C**Q^T, *P*C*, P^T**C*, *C*P*, or *C**P^T, as specified by *vect*, *side*, and *trans*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ormbr interface are the following:

a	Holds the matrix *A* of size (*r*,min *(nq,k)*) where *r* = *nq*, if *vect* = 'Q', *r* = min(*nq*,*k*), if *vect* = 'P', *nq* = *m*, if *side* = 'L', *nq* = *n*, if *side* = 'R', *k* = *m*, if *vect* = 'P', *k* = *n*, if *vect* = 'Q'.
tau	Holds the vector of length min(*nq*,*k*).
c	Holds the matrix *C* of size (*m*,*n*).
vect	Must be 'Q' or 'P'. The default value is 'Q'.
side	Must be 'L' or 'R'. The default value is 'L'.
trans	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using

lwork = *n*blocksize* for *side* = 'L', or

lwork = *m*blocksize* for *side* = 'R',

where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed product differs from the exact product by a matrix E such that $||E||_2 = O(\varepsilon)||C||_2$.

The total number of floating-point operations is approximately

$2*n*k(2*m - k)$ if *side* = 'L' and $m \geq k$;

$2*m*k(2*n - k)$ if *side* = 'R' and $n \geq k$;

$2*m^2*n$ if *side* = 'L' and $m < k$;

$2*n^2*m$ if *side* = 'R' and $n < k$.

The complex counterpart of this routine is ?unmbr.

?ungbr

Generates the complex unitary matrix Q or P_H determined by ?gebrd.

Syntax

Fortran 77:

```
call cungbr(vect, m, n, k, a, lda, tau, work, lwork, info)
call zungbr(vect, m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call ungbr(a, tau [,vect] [,info])
```

Description

The routine generates the whole or part of the unitary matrices Q and P_H formed by the routines cgebrd/zgebrd. Use this routine after a call to cgebrd/zgebrd. All valid combinations of arguments are described in Input Parameters;in most cases you need the following:

To compute the whole m-by-m matrix Q, use:

```
call ?ungbr('Q', m, m, n, a ... )
```

(note that the array a must have at least m columns).

To form the n leading columns of Q if $m > n$, use:

```
call ?ungbr('Q', m, n, n, a ... )
```

To compute the whole n-by-n matrix P^H, use:

```
call ?ungbr('P', n, n, m, a ... )
```

(note that the array a must have at least n rows).

To form the m leading rows of P_H if $m < n$, use:

```
call ?ungbr('P', m, n, m, a ... )
```

Input Parameters

vect — CHARACTER*1. Must be 'Q' or 'P'.
If *vect* = 'Q', the routine generates the matrix Q.
If *vect* = 'P', the routine generates the matrix P^H.

m	INTEGER. The number of required rows of Q or P^H.
n	INTEGER. The number of required columns of Q or P^H.
k	INTEGER. One of the dimensions of A in ?gebrd: If *vect* = 'Q', the number of columns in A; If *vect* = 'P', the number of rows in A. Constraints: $m \geq 0$, $n \geq 0$, $k \geq 0$. For *vect* = 'Q': $k \leq n \leq m$ if $m > k$, or $m = n$ if $m \leq k$. For *vect* = 'P': $k \leq m \leq n$ if $n > k$, or $m = n$ if $n \leq k$.
a, *work*	COMPLEX for cungbr DOUBLE COMPLEX for zungbr. Arrays: *a*(*lda*,*) is the array *a* as returned by ?gebrd. The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
tau	COMPLEX for cungbr DOUBLE COMPLEX for zungbr. For *vect* = 'Q', the array *tauq* as returned by ?gebrd. For *vect* = 'P', the array *taup* as returned by ?gebrd. The dimension of *tau* must be at least max(1, min(*m*, *k*)) for *vect* = 'Q', or max(1, min(*m*, *k*)) for *vect* = 'P'.
lwork	INTEGER. The size of the *work* array. Constraint: *lwork* < max(1, min(*m*, *n*)). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the orthogonal matrix Q or P^T (or the leading rows or columns thereof) as specified by *vect*, *m*, and *n*.

work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ungbr interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
tau	Holds the vector of length min(*m*,*k*) where *k* = *m*, if *vect* = 'P', *k* = *n*, if *vect* = 'Q'.
vect	Must be 'Q' or 'P'. The default value is 'Q'.

Application Notes

For better performance, try using *lwork* = min(*m*,*n*)**blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $||E||_2 = O(\varepsilon)$.

The approximate numbers of floating-point operations for the cases listed in *Description* are as follows:

To form the whole of Q:

$(16/3)n(3m^2 - 3m*n + n^2)$ if $m > n$;

$(16/3)m^3$ if $m \le n$.

To form the n leading columns of Q when $m > n$:

$(8/3)n^2(3m - n^2)$ if $m > n$.

To form the whole of P_T:

$(16/3)n^3$ if $m \ge n$;

$(16/3)m(3n^2 - 3m*n + m^2)$ if $m < n$.

To form the m leading columns of P_T when $m < n$:

$(8/3)n^2(3m - n^2)$ if $m > n$.

The real counterpart of this routine is ?orgbr.

?unmbr

Multiplies an arbitrary complex matrix by the unitary matrix Q or P determined by ?gebrd.

Syntax

Fortran 77:

```
call cunmbr(vect, side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call zunmbr(vect, side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call unmbr(a, tau, c [,vect] [,side] [,trans] [,info])
```

Description

Given an arbitrary complex matrix C, this routine forms one of the matrix products $Q*C$, Q^H*C, $C*Q$, $C*Q^H$, $P*C$, P^H*C, $C*P$, or $C*P^H$, where Q and P are orthogonal matrices computed by a call to cgebrd/zgebrd. The routine overwrites the product on C.

Input Parameters

In the descriptions below, r denotes the order of Q or P^H:

If $side$ = 'L', r = m; if $side$ = 'R', r = n.

vect	CHARACTER*1. Must be 'Q' or 'P'. If *vect* = 'Q', then Q or Q^H is applied to C. If *vect* = 'P', then P or P^H is applied to C.
side	CHARACTER*1. Must be 'L' or 'R'. If *side* = 'L', multipliers are applied to C from the left. If *side* = 'R', they are applied to C from the right.
trans	CHARACTER*1. Must be 'N' or 'C'. If *trans* = 'N', then Q or P is applied to C. If *trans* = 'C', then Q^H or P^H is applied to C.
m	INTEGER. The number of rows in C.
n	INTEGER. The number of columns in C.
k	INTEGER. One of the dimensions of A in ?gebrd: If *vect* = 'Q', the number of columns in A; If *vect* = 'P', the number of rows in A. Constraints: $m \geq 0$, $n \geq 0$, $k \geq 0$.
a, *c*, *work*	COMPLEX for cunmbr DOUBLE COMPLEX for zunmbr. Arrays: *a*(*lda*,*) is the array *a* as returned by ?gebrd. Its second dimension must be at least max(1, min(*r*,*k*)) for *vect* = 'Q', or max(1, *r*)) for *vect* = 'P'. *c*(*ldc*,*) holds the matrix C. Its second dimension must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*. Constraints:

	lda ≥ max(1, *r*) if *vect* = 'Q'; *lda* ≥ max(1, min(*r*,*k*)) if *vect* = 'P'.
ldc	INTEGER. The first dimension of *c*; *ldc* ≥ max(1, *m*).
tau	COMPLEX for cunmbr DOUBLE COMPLEX for zunmbr. Array, DIMENSION at least max (1, min(*r*, *k*)). For *vect* = 'Q', the array *tauq* as returned by ?gebrd. For *vect* = 'P', the array *taup* as returned by ?gebrd.
lwork	INTEGER. The size of the *work* array. *lwork* ≥ max(1, *n*) if *side* = 'L'; *lwork* ≥ max(1, *m*) if *side* = 'R'. *lwork* ≥ 1 if *n*=0 or *m*=0. For optimum performance *lwork* ≥ max(1,*n***nb*) if *side* = 'L', and *lwork* ≥ max(1,*m***nb*) if *side* = 'R', where *nb* is the optimal blocksize. (*nb* = 0 if *m* = 0 or *n* = 0.) If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

c	Overwritten by the product *Q***C*, Q^H**C*, *C***Q*, *C**Q^H, *P***C*, P^H**C*, *C***P*, or *C**P^H, as specified by *vect*, *side*, and *trans*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `unmbr` interface are the following:

a	Holds the matrix *A* of size (*r*,min *(nq,k)*) where *r* = *nq*, if *vect* = 'Q', *r* = min(*nq*,*k*), if *vect* = 'P', *nq* = *m*, if *side* = 'L', *nq* = *n*, if *side* = 'R', *k* = *m*, if *vect* = 'P', *k* = *n*, if *vect* = 'Q'.
tau	Holds the vector of length min(*nq*,*k*).
c	Holds the matrix *C* of size (*m*,*n*).
vect	Must be 'Q' or 'P'. The default value is 'Q'.
side	Must be 'L' or 'R'. The default value is 'L'.
trans	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, try using

lwork = *n***blocksize* for *side* = 'L', or

lwork = *m***blocksize* for *side* = 'R',

where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the `blocked algorithm`.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed product differs from the exact product by a matrix E such that $||E||_2 = O(\varepsilon)||C||_2$.

The total number of floating-point operations is approximately

$8*n*k(2*m - k)$ if *side* = 'L' and $m \geq k$;

$8*m*k(2*n - k)$ if *side* = 'R' and $n \geq k$;

$8*m^2*n$ if *side* = 'L' and $m < k$;

$8*n^2*m$ if *side* = 'R' and $n < k$.

The real counterpart of this routine is ?ormbr.

?bdsqr

Computes the singular value decomposition of a general matrix that has been reduced to bidiagonal form.

Syntax

Fortran 77:

```
call sbdsqr(uplo, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, work, info)
call dbdsqr(uplo, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, work, info)
call cbdsqr(uplo, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, work, info)
call zbdsqr(uplo, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, work, info)
```

Fortran 95:

```
call rbdsqr(d, e [,vt] [,u] [,c] [,uplo] [,info])
call bdsqr(d, e [,vt] [,u] [,c] [,uplo] [,info])
```

Description

This routine computes the singular values and, optionally, the right and/or left singular vectors from the Singular Value Decomposition (SVD) of a real n-by-n (upper or lower) bidiagonal matrix B using the implicit zero-shift QR algorithm. The SVD of B has the form $B = Q*S*P^H$ where S is the diagonal matrix of singular values, Q is an orthogonal matrix of left singular vectors, and P is an orthogonal matrix of right singular vectors. If left singular vectors are requested, this subroutine actually returns $U*Q$ instead of Q, and, if right singular vectors are requested, this subroutine returns P_H*VT instead of P_H, for given real/complex input matrices U and VT. When U and VT are the orthogonal/unitary matrices that reduce a general matrix A to bidiagonal form: $A = U*B*VT$, as computed by ?gebrd, then

$A = (U*Q)*S*(P^H*VT)$

is the SVD of A. Optionally, the subroutine may also compute Q_H*C for a given real/complex input matrix C.

See also ?lasq1, ?lasq2, ?lasq3, ?lasq4, ?lasq5, ?lasq6 used by this routine.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', B is an upper bidiagonal matrix. If *uplo* = 'L', B is a lower bidiagonal matrix.
n	INTEGER. The order of the matrix B ($n \geq 0$).
ncvt	INTEGER. The number of columns of the matrix VT, that is, the number of right singular vectors ($ncvt \geq 0$). Set *ncvt* = 0 if no right singular vectors are required.
nru	INTEGER. The number of rows in U, that is, the number of left singular vectors ($nru \geq 0$). Set *nru* = 0 if no left singular vectors are required.
ncc	INTEGER. The number of columns in the matrix C used for computing the product Q^H*C ($ncc \geq 0$). Set *ncc* = 0 if no matrix C is supplied.
d, *e*, *work*	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: *d*(*) contains the diagonal elements of B.

The dimension of *d* must be at least max(1, *n*).
e(*) contains the (*n*-1) off-diagonal elements of *B*.
The dimension of *e* must be at least max(1, *n*). *e*(*n*) is used for workspace.
work(*) is a workspace array.
The dimension of *work* must be at least max(1, 2**n*) if *ncvt* = *nru* = *ncc* = 0;
max(1, 4*(*n*-1)) otherwise.

vt, *u*, *c*
REAL for sbdsqr
DOUBLE PRECISION for dbdsqr
COMPLEX for cbdsqr
DOUBLE COMPLEX for zbdsqr.
Arrays:
vt(*ldvt*,*) contains an *n*-by-*ncvt* matrix *VT*.
The second dimension of *vt* must be at least max(1, *ncvt*).
vt is not referenced if *ncvt* = 0.
u(*ldu*,*) contains an *nru* by *n* unit matrix *U*.
The second dimension of *u* must be at least max(1, *n*).
u is not referenced if *nru* = 0.
c(*ldc*,*) contains the matrix *C* for computing the product $Q^H * C$.
The second dimension of *c* must be at least max(1, *ncc*).
The array is not referenced if *ncc* = 0.

ldvt
INTEGER. The first dimension of *vt*. Constraints:
ldvt ≥ max(1, *n*) if *ncvt* > 0;
ldvt ≥ 1 if *ncvt* = 0.

ldu
INTEGER. The first dimension of *u*. Constraint:
ldu ≥ max(1, *nru*).

ldc
INTEGER. The first dimension of *c*. Constraints:
ldc ≥ max(1, *n*) if *ncc* > 0; *ldc* ≥ 1 otherwise.

Output Parameters

d
On exit, if *info* = 0, overwritten by the singular values in decreasing order (see *info*).

e
On exit, if *info* = 0, *e* is destroyed. See also *info* below.

c	Overwritten by the product Q^H*C.
vt	On exit, this array is overwritten by P^H *VT.
u	On exit, this array is overwritten by *U* *Q .
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the algorithm failed to converge; *i* specifies how many off-diagonals did not converge. In this case, *d* and *e* contain on exit the diagonal and off-diagonal elements, respectively, of a bidiagonal matrix orthogonally equivalent to *B*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine bdsqr interface are the following:

d	Holds the vector of length (*n*).
e	Holds the vector of length (*n*).
vt	Holds the matrix *VT* of size (*n*, *ncvt*).
u	Holds the matrix *U* of size (*nru*,*n*).
c	Holds the matrix *C* of size (*n*,*ncc*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
ncvt	If argument *vt* is present, then *ncvt* is equal to the number of columns in matrix *VT*; otherwise, *ncvt* is set to zero.
nru	If argument *u* is present, then *nru* is equal to the number of rows in matrix *U*; otherwise, *nru* is set to zero.
ncc	If argument *c* is present, then *ncc* is equal to the number of columns in matrix *C*; otherwise, *ncc* is set to zero.

Note that two variants of Fortran 95 interface for bdsqr routine are needed because of an ambiguous choice between real and complex cases appear when *vt*, *u*, and *c* are omitted. Thus, the name rbdsqr is used in real cases (single or double precision), and the name bdsqr is used in complex cases (single or double precision).

Application Notes

Each singular value and singular vector is computed to high relative accuracy. However, the reduction to bidiagonal form (prior to calling the routine) may decrease the relative accuracy in the small singular values of the original matrix if its singular values vary widely in magnitude.

If s_i is an exact singular value of B, and s_i is the corresponding computed value, then

$|s_i - \sigma_i| \leq p(m,n)*\varepsilon*\sigma_i$

where $p(m, n)$ is a modestly increasing function of m and n, and ε is the machine precision.

If only singular values are computed, they are computed more accurately than when some singular vectors are also computed (that is, the function $p(m, n)$ is smaller).

If u_i is the corresponding exact left singular vector of B, and w_i is the corresponding computed left singular vector, then the angle $\theta(u_i, w_i)$ between them is bounded as follows:

$\theta(u_i, w_i) \leq p(m,n)*\varepsilon / \min_{i \neq j}(|\sigma i - \sigma j|/|\sigma_i + \sigma j|).$

Here $\min_{i \neq j}(|\sigma_i - \sigma_j|/|\sigma_i + \sigma_j|)$ is the relative gap between σ_i and the other singular values. A similar error bound holds for the right singular vectors.

The total number of real floating-point operations is roughly proportional to n^2 if only the singular values are computed. About $6n^2*nru$ additional operations ($12n^2*nru$ for complex flavors) are required to compute the left singular vectors and about $6n^2*ncvt$ operations ($12n^2*ncvt$ for complex flavors) to compute the right singular vectors.

?bdsdc

Computes the singular value decomposition of a real bidiagonal matrix using a divide and conquer method.

Syntax

Fortran 77:

```
call sbdsdc(uplo, compq, n, d, e, u, ldu, vt, ldvt, q, iq, work, iwork, info)
call dbdsdc(uplo, compq, n, d, e, u, ldu, vt, ldvt, q, iq, work, iwork, info)
```

Fortran 95:

```
call bdsdc(d, e [,u] [,vt] [,q] [,iq] [,uplo] [,info])
```

Description

This routine computes the Singular Value Decomposition (SVD) of a real n-by-n (upper or lower) bidiagonal matrix B: $B = U*\Sigma*V^T$, using a divide and conquer method, where Σ is a diagonal matrix with non-negative diagonal elements (the singular values of B), and U and V are orthogonal matrices of left and right singular vectors, respectively. ?bdsdc can be used to compute all singular values, and optionally, singular vectors or singular vectors in compact form.

This rotuine uses ?lasd0, ?lasd1, ?lasd2, ?lasd3, ?lasd4, ?lasd5, ?lasd6, ?lasd7, ?lasd8, ?lasd9, ?lasda, ?lasdq, ?lasdt.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', B is an upper bidiagonal matrix. If *uplo* = 'L', B is a lower bidiagonal matrix.
compq	CHARACTER*1. Must be 'N', 'P', or 'I'. If *compq* = 'N', compute singular values only. If *compq* = 'P', compute singular values and compute singular vectors in compact form. If *compq* = 'I', compute singular values and singular vectors.
n	INTEGER. The order of the matrix B ($n \geq 0$).
d, *e*, *work*	REAL for sbdsdc DOUBLE PRECISION for dbdsdc. Arrays: *d(*)* contains the n diagonal elements of the bidiagonal matrix B. The dimension of *d* must be at least max(1, n). *e*(*) contains the off-diagonal elements of the bidiagonal matrix B. The dimension of *e* must be at least max(1, n). *work*(*) is a workspace array. The dimension of *work* must be at least: max(1, 4*n), if *compq* = 'N'; max(1, 6*n), if *compq* = 'P'; max(1, 3*n^2+4*n), if *compq* = 'I'.

ldu	INTEGER. The first dimension of the output array *u*; *ldu* ≥ 1. If singular vectors are desired, then *ldu* ≥ max(1, *n*).
ldvt	INTEGER. The first dimension of the output array *vt*; *ldvt* ≥ 1. If singular vectors are desired, then *ldvt* ≥ max(1, *n*).
iwork	INTEGER. Workspace array, dimension at least max(1, 8**n*).

Output Parameters

d	If *info* = 0, overwritten by the singular values of *B*.
e	On exit, *e* is overwritten.
u, *vt*, *q*	REAL for sbdsdc DOUBLE PRECISION for dbdsdc. Arrays: *u*(*ldu*,*), *vt*(*ldvt*,*), *q*(*). If *compq* = 'I', then on exit *u* contains the left singular vectors of the bidiagonal matrix *B*, unless *info* ≠ 0 (see *info*). For other values of *compq*, *u* is not referenced. The second dimension of *u* must be at least max(1,*n*). if *compq* = 'I', then on exit *vt* contains the right singular vectors of the bidiagonal matrix *B*, unless *info* ≠ 0 (see *info*). For other values of *compq*, *vt* is not referenced. The second dimension of *vt* must be at least max(1,*n*). If *compq* = 'P', then on exit, if *info* = 0, *q* and *iq* contain the left and right singular vectors in a compact form. Specifically, *q* contains all the REAL (for sbdsdc) or DOUBLE PRECISION (for dbdsdc) data for singular vectors. For other values of *compq*, *q* is not referenced. See Application notes for details.
iq	INTEGER. Array: *iq*(*). If *compq* = 'P', then on exit, if *info* = 0, *q* and *iq* contain the left and right singular vectors in a compact form. Specifically, *iq* contains all the INTEGER data for singular vectors. For other values of *compq*, *iq* is not referenced. See Application notes for details.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, the algorithm failed to compute a singular value. The update process of divide and conquer failed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine bdsdc interface are the following:

d Holds the vector of length (*n*).

e Holds the vector of length (*n*).

u Holds the matrix *U* of size (*n*,*n*).

vt Holds the matrix *VT* of size (*n*,*n*).

q Holds the vector of length (*ldq*), where
ldq ≥ *n**(11 + 2**smlsiz* + 8**int*(*log_2*(*n*/(*smlsiz* + 1))))
and *smlsiz* is returned by *ilaenv* and is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25).

compq Restored based on the presence of arguments *u*, *vt*, *q*, and *iq* as follows:
compq = 'N', if none of *u*, *vt*, *q*, and *iq* are present,
compq = 'I', if both *u* and *vt* are present. Arguments *u* and *vt* must either be both present or both omitted,
compq = 'P', if both *q* and *iq* are present. Arguments *q* and *iq* must either be both present or both omitted.
Note that there will be an error condition if all of *u*, *vt*, *q*, and *iq* arguments are present simultaneously.

Symmetric Eigenvalue Problems

Symmetric eigenvalue problems are posed as follows: given an *n*-by-*n* real symmetric or complex Hermitian matrix *A*, find the eigenvaluesλ and the corresponding eigenvectors*z* that satisfy the equation

$Az = \lambda z$. (or, equivalently, $z^H A = \lambda z^H$) .

In such eigenvalue problems, all n eigenvalues are real not only for real symmetric but also for complex Hermitian matrices A, and there exists an orthonormal system of n eigenvectors. If A is a symmetric or Hermitian positive-definite matrix, all eigenvalues are positive.

To solve a symmetric eigenvalue problem with LAPACK, you usually need to reduce the matrix to tridiagonal form and then solve the eigenvalue problem with the tridiagonal matrix obtained. LAPACK includes routines for reducing the matrix to a tridiagonal form by an orthogonal (or unitary) similarity transformation $A = QTQ^H$ as well as for solving tridiagonal symmetric eigenvalue problems. These routines (for Fortran-77 interface) are listed in Table 4-3 . Respective routine names in Fortran-95 interface are without the first symbol (see Routine Naming Conventions).

There are different routines for symmetric eigenvalue problems, depending on whether you need all eigenvectors or only some of them or eigenvalues only, whether the matrix A is positive-definite or not, and so on.

These routines are based on three primary algorithms for computing eigenvalues and eigenvectors of symmetric problems: the divide and conquer algorithm, the QR algorithm, and bisection followed by inverse iteration. The divide and conquer algorithm is generally more efficient and is recommended for computing all eigenvalues and eigenvectors. Furthermore, to solve an eigenvalue problem using the divide and conquer algorithm, you need to call only one routine. In general, more than one routine has to be called if the QR algorithm or bisection followed by inverse iteration is used.

Decision tree in Figure 4-2 will help you choose the right routine or sequence of routines for eigenvalue problems with real symmetric matrices. A similar decision tree for complex Hermitian matrices is presented in Figure 4-3 .

Figure 4-2 Decision Tree: Real Symmetric Eigenvalue Problems

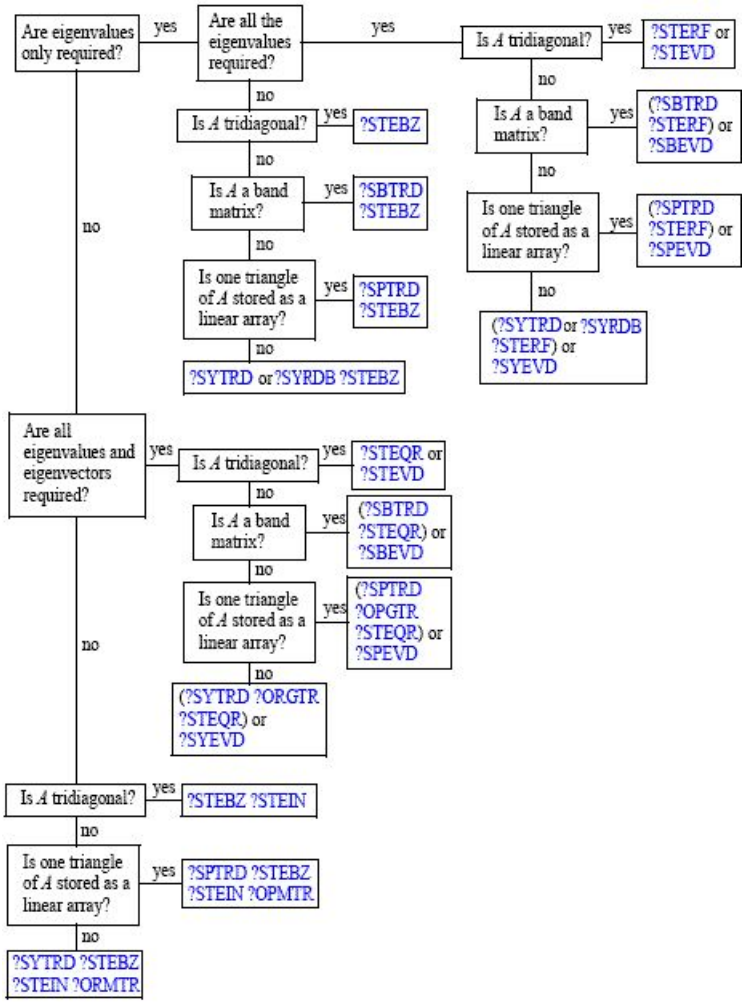


Figure 4-3 Decision Tree: Complex Hermitian Eigenvalue Problems

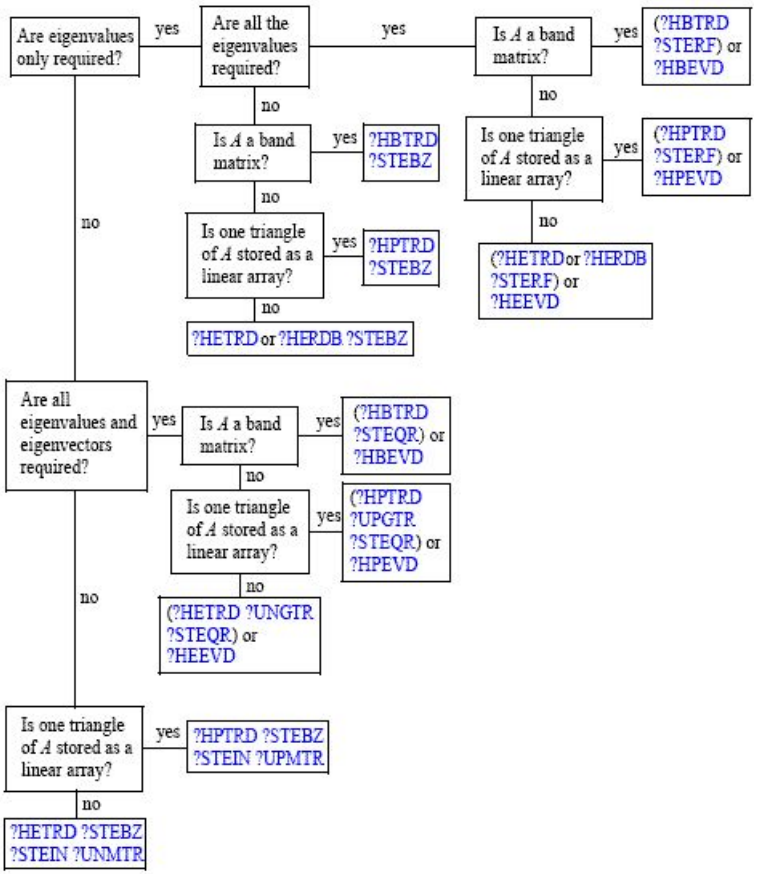


Table 4-3 Computational Routines for Solving Symmetric Eigenvalue Problems

Operation	Real symmetric matrices	Complex Hermitian matrices
Reduce to tridiagonal form $A = QTQ^H$ (full storage)	?sytrd ?syrdb	?hetrd ?herdb
Reduce to tridiagonal form $A = QTQ^H$ (packed storage)	?sptrd	?hptrd

Operation	Real symmetric matrices	Complex Hermitian matrices
Reduce to tridiagonal form $A = QTQ^H$ (band storage).	?sbtrd	?hbtrd
Generate matrix Q (full storage)	?orgtr	?ungtr
Generate matrix Q (packed storage)	?opgtr	?upgtr
Apply matrix Q (full storage)	?ormtr	?unmtr
Apply matrix Q (packed storage)	?opmtr	?upmtr
Find all eigenvalues of a tridiagonal matrix T	?sterf	
Find all eigenvalues and eigenvectors of a tridiagonal matrix T	?steqr ?stedc	?steqr ?stedc
Find all eigenvalues and eigenvectors of a tridiagonal positive-definite matrix T.	?pteqr	?pteqr
Find selected eigenvalues of a tridiagonal matrix T	?stebz ?stegr	?stegr
Find selected eigenvectors of a tridiagonal matrix T	?stein ?stegr	?stein ?stegr
Find selected eigenvalues and eigenvectors of f a real symmetric tridiagonal matrix T	?stemr	?stemr
Compute the reciprocal condition numbers for the eigenvectors	?disna	?disna

?sytrd

Reduces a real symmetric matrix to tridiagonal form.

Syntax

Fortran 77:

```
call ssytrd(uplo, n, a, lda, d, e, tau, work, lwork, info)
call dsytrd(uplo, n, a, lda, d, e, tau, work, lwork, info)
```

Fortran 95:

```
call sytrd(a, tau [,uplo] [,info])
```

Description

This routine reduces a real symmetric matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: $A = Q*T*Q^T$. The orthogonal matrix Q is not formed explicitly but is represented as a product of n-1 elementary reflectors. Routines are provided for working with Q in this representation. (They are described later in this section .)

This routine calls ?latrd to reduce a real symmetric matrix to tridiagonal form by an orthogonal similarity transformation.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *a* stores the upper triangular part of *A*. If *uplo* = 'L', *a* stores the lower triangular part of *A*.
n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
a, *work*	REAL for ssytrd DOUBLE PRECISION for dsytrd. *a*(*lda*,*) is an array containing either upper or lower triangular part of the matrix *A*, as specified by *uplo*. The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).
lwork	INTEGER. The size of the *work* array (*lwork* ≥ *n*).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a — Overwritten by the tridiagonal matrix *T* and details of the orthogonal matrix *Q*, as specified by *uplo*.

d, *e*, *tau* — REAL for ssytrd
DOUBLE PRECISION for dsytrd.
Arrays:
d(*) contains the diagonal elements of the matrix *T*.
The dimension of *d* must be at least max(1, *n*).
e(*) contains the off-diagonal elements of *T*.
The dimension of *e* must be at least max(1, *n*-1).
tau(*) stores further details of the orthogonal matrix *Q*.
The dimension of *tau* must be at least max(1, *n*-1).

work(1) — If *info*=0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine sytrd interface are the following:

a — Holds the matrix *A* of size (*n*,*n*).

tau — Holds the vector of length (*n*-1).

d — Holds the vector of length (*n*).

e — Holds the vector of length (*n*-1).

uplo — Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For better performance, try using *lwork* =*n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix T is exactly similar to a matrix $A + E$, where $||E||_2 = c(n)\varepsilon ||A||_2$, $c(n)$ is a modestly increasing function of n, and ε is the machine precision.

The approximate number of floating-point operations is $(4/3)n^3$.

After calling this routine, you can call the following:

?orgtr	to form the computed matrix Q explicitly
?ormtr	to multiply a real matrix by Q.

The complex counterpart of this routine is ?hetrd.

?syrdb

Reduces a real symmetric matrix to tridiagonal form with Successive Bandwidth Reduction approach.

Syntax

Fortran 77:

```
call ssyrdb(jobz, uplo, n, kd, a, lda, d, e, tau, z, ldz, work, lwork, info)
call dsyrdb(jobz, uplo, n, kd, a, lda, d, e, tau, z, ldz, work, lwork, info)
```

Description

This routine reduces a real symmetric matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: $A = Q*T*Q^T$ and optionally multuplies matrix Z by Q, or simply forms Q.

This routine reduces a full symmetric matrix to the banded symmetric form, and then to the tridiagonal symmetric form with a Successive Bandwidth Reduction approach after Prof. C.Bischof's works (see for instance, [Bischof92]). ?syrdb is functionally close to ?sytrd routine but the tridiagonal form may differ from those obtained by ?sytrd. Unlike ?sytrd, the orthogonal matrix Q cannot be restored from the details of matrix A on exit.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then only A is reduced to T. If *jobz* = 'V', then A is reduced to T and Z contains Q on exit. If *jobz* = 'U', then A is reduced to T and Z contains ZQ on exit.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *a* stores the upper triangular part of A. If *uplo* = 'L', *a* stores the lower triangular part of A.
n	INTEGER. The order of the matrix A ($n \geq 0$).
kd	INTEGER. The bandwidth of the banded matrix B ($kd \geq 1$).
a,*z*, *work*	REAL for ssyrdb.

DOUBLE PRECISION for dsyrdb.
a(*lda*,*) is an array containing either upper or lower triangular part of the matrix *A*, as specified by *uplo*.
The second dimension of *a* must be at least max(1, *n*).
z(*ldz*,*), the second dimension of *z* must be at least max(1, *n*).
If *jobz* = 'U', then the matrix *z* is multiplied by *Q*.
If *jobz* = 'V', then *z* content on input is discarded.
If *jobz* = 'N', then *z* is not referenced.
work(*lwork*) is a workspace array.

lda — INTEGER. The first dimension of *a*; at least max(1, *n*).

ldz — INTEGER. The first dimension of *z*; at least max(1, *n*). Not referenced if *jobz* = 'N'

lwork — INTEGER. The size of the *work* array (*lwork* ≥ (2*kd*+1)*n*+*kd*).
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.
See Application Notes for the suggested value of *lwork*.

Output Parameters

a — Overwritten by the banded matrix *B* and details of the orthogonal matrix Q_B to reduce *A* to *B* as specified by *uplo*.

z — On exit,
if *jobz* = 'U', then the matrix *z* is overwritten by *zQ*.
If *jobz* = 'V', then *z* contains *Q*.
If *jobz* = 'N', then *z* is not referenced.

d, *e*, *tau* — DOUBLE PRECISION.
Arrays:
d(*) contains the diagonal elements of the matrix *T*.
The dimension of *d* must be at least max(1, *n*).
e(*) contains the off-diagonal elements of *T*.
The dimension of *e* must be at least max(1, *n*-1).
tau(*) stores further details of the orthogonal matrix *Q*.
The dimension of *tau* must be at least max(1, *n*-*kd*-1).

work(1)	If *info*=0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* = *n**(3**kd*+3).

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

For better performance, try using *kd* equal to 40 if n ≤ 2000 and 64 otherwise.

Try using ?syrdb instead of ?sytrd on large matrices obtaining only eigenvalues - when no eigenvectors are needed, especially in multi-threaded environment. ?syrdb becomes faster beginning approximately with n = 1000, and much faster at larger matrices with a better scalability than ?sytrd.

Avoid applying ?syrdb for computing eigenvectors due to the two-step reduction, that is, the number of operations needed to apply orthogonal transformations to *Z* is doubled compared to the traditional one-step reduction. In that case it is better to apply ?sytrd and ?ormtr/?orgtr to obtain tridiagonal form along with the orthogonal transformation matrix *Q*.

?herdb

Reduces a complex Hermitian matrix to tridiagonal form with Successive Bandwidth Reduction approach.

Syntax

Fortran 77:

```
call cherdb(jobz, uplo, n, kd, a, lda, d, e, tau, z, ldz, work, lwork, info)
call zherdb(jobz, uplo, n, kd, a, lda, d, e, tau, z, ldz, work, lwork, info)
```

Description

This routine reduces a complex Hermitian matrix A to symmetric tridiagonal form T by a unitary similarity transformation: $A = Q*T*Q^T$ and optionally multuplies matrix Z by Q, or simply forms Q.

This routine reduces a full Hermitian matrix to the banded Hermitian form, and then to the tridiagonal symmetric form with a Successive Bandwidth Reduction approach after Prof. C.Bischof's works (see for instance, [Bischof92]). ?herdb is functionally close to ?hetrd routine but the tridiagonal form may differ from those obtained by ?hetrd. Unlike ?hetrd, the orthogonal matrix Q cannot be restored from the details of matrix A on exit.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then only A is reduced to T. If *jobz* = 'V', then A is reduced to T and Z contains Q on exit. If *jobz* = 'U', then A is reduced to T and Z contains ZQ on exit.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *a* stores the upper triangular part of A. If *uplo* = 'L', *a* stores the lower triangular part of A.
n	INTEGER. The order of the matrix A ($n \geq 0$).
kd	INTEGER. The bandwidth of the banded matrix B ($kd \geq 1$).
a,*z*, *work*	COMPLEX for cherdb.

DOUBLE COMPLEX for zherdb.
a(*lda*,*) is an array containing either upper or lower triangular part of the matrix *A*, as specified by *uplo*.
The second dimension of *a* must be at least max(1, *n*).
z(*ldz*,*), the second dimension of *z* must be at least max(1, *n*).
If *jobz* = 'U', then the matrix *z* is multiplied by *Q*.
If *jobz* = 'V', then *z* content on input is discarded.
If *jobz* = 'N', then *z* is not referenced.
work(*lwork*) is a workspace array.

lda — INTEGER. The first dimension of *a*; at least max(1, *n*).

ldz — INTEGER. The first dimension of *z*; at least max(1, *n*). Not referenced if *jobz* = 'N'

lwork — INTEGER. The size of the *work* array (*lwork* ≥ (2*kd*+1)*n*+*kd*).
If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.
See Application Notes for the suggested value of *lwork*.

Output Parameters

a — Overwritten by the banded matrix *B* and details of the unitary matrix Q_B to reduce *A* to *B* as specified by *uplo*.

z — On exit,
if *jobz* = 'U', then the matrix *z* is overwritten by *zQ*.
If *jobz* = 'V', then *z* contains *Q*.
If *jobz* = 'N', then *z* is not referenced.

d, *e* — COMPLEX for cherdb.
DOUBLE COMPLEX for zherdb.
Arrays:
d(*) contains the diagonal elements of the matrix *T*.
The dimension of *d* must be at least max(1, *n*).
e(*) contains the off-diagonal elements of *T*.
The dimension of *e* must be at least max(1, *n*-1).
tau(*) stores further details of the orthogonal matrix *Q*.

	The dimension of *tau* must be at least max(1, *n*-*kd*-1).
tau	COMPLEX for cherdb. DOUBLE COMPLEX for zherdb. Array, DIMENSION at least max(1, *n*-1) Stores further details of the unitary matrix Q_B. The dimension of *tau* must be at least max(1, *n*-*kd*-1).
work(1)	If *info*=0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

For better performance, try using *lwork* = *n**(3**kd*+3).

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* size, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*) on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

For better performance, try using *kd* equal to 40 if n ≤ 2000 and 64 otherwise.

Try using ?herdb instead of ?hetrd on large matrices obtaining only eigenvalues - when no eigenvectors are needed, especially in multi-threaded environment. ?herdb becomes faster beginning approximately with n = 1000, and much faster at larger matrices with a better scalability than ?hetrd.

Avoid applying ?herdb for computing eigenvectors due to the two-step reduction, that is, the number of operations needed to apply orthogonal transformations to *Z* is doubled compared to the traditional one-step reduction. In that case it is better to apply ?hetrd and ?unmtr/?ungtr to obtain tridiagonal form along with the unitary transformation matrix *Q*.

?orgtr

Generates the real orthogonal matrix Q determined by ?sytrd.

Syntax

Fortran 77:

```
call sorgtr(uplo, n, a, lda, tau, work, lwork, info)
call dorgtr(uplo, n, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call orgtr(a, tau [,uplo] [,info])
```

Description

The routine explicitly generates the *n*-by-*n* orthogonal matrix *Q* formed by ?sytrd when reducing a real symmetric matrix *A* to tridiagonal form: $A = Q*T*Q^T$. Use this routine after a call to ?sytrd.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Use the same *uplo* as supplied to ?sytrd.
n	INTEGER. The order of the matrix *Q* ($n \geq 0$).
a, *tau*, *work*	REAL for sorgtr DOUBLE PRECISION for dorgtr. Arrays: *a*(*lda*,*) is the array *a* as returned by ?sytrd. The second dimension of *a* must be at least max(1, *n*). *tau*(*) is the array *tau* as returned by ?sytrd. The dimension of *tau* must be at least max(1, *n*-1). *work* is a workspace array, its dimension max(1, *lwork*).

lda	INTEGER. The first dimension of *a*; at least max(1, *n*).
lwork	INTEGER. The size of the *work* array (*lwork* ≥ *n*). If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the orthogonal matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = −*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine orgtr interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
tau	Holds the vector of length (*n*−1).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For better performance, try using *lwork* = (*n*−1)**blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = −1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $||E||_2 = O(\varepsilon)$, where e is the machine precision.

The approximate number of floating-point operations is $(4/3)n^3$.

The complex counterpart of this routine is ?ungtr.

?ormtr

Multiplies a real matrix by the real orthogonal matrix Q determined by ?sytrd.

Syntax

Fortran 77:

```
call sormtr(side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info)
call dormtr(side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call ormtr(a, tau, c [,side] [,uplo] [,trans] [,info])
```

Description

The routine multiplies a real matrix C by Q or Q^T, where Q is the orthogonal matrix Q formed by ?sytrd when reducing a real symmetric matrix A to tridiagonal form: $A = Q*T*Q^T$. Use this routine after a call to ?sytrd.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q*C$, Q^T*C, $C*Q$, or $C*Q^T$(overwriting the result on C).

Input Parameters

In the descriptions below, *r* denotes the order of *Q*:

If *side* = 'L', *r* = *m*; if *side* = 'R', *r* = *n*.

side	CHARACTER*1. Must be either 'L' or 'R'. If *side* = 'L', *Q* or Q^T is applied to *C* from the left. If *side* = 'R', *Q* or Q^T is applied to *C* from the right.
uplo	CHARACTER*1. Must be 'U' or 'L'. Use the same *uplo* as supplied to ?sytrd.
trans	CHARACTER*1. Must be either 'N' or 'T'. If *trans* = 'N', the routine multiplies *C* by *Q*. If *trans* = 'T', the routine multiplies *C* by Q^T.
m	INTEGER. The number of rows in the matrix *C* (*m* ≥ 0).
n	INTEGER. The number of columns in *C* (*n* ≥ 0).
a, *c*, *tau*, *work*	REAL for sormtr DOUBLE PRECISION for dormtr *a*(*lda*,*) and *tau* are the arrays returned by ?sytrd. The second dimension of *a* must be at least max(1, *r*). The dimension of *tau* must be at least max(1, *r*-1). *c*(*ldc*,*) contains the matrix *C*. The second dimension of *c* must be at least max(1, *n*) *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *r*).
ldc	INTEGER. The first dimension of *c*; *ldc* ≥ max(1, *n*).
lwork	INTEGER. The size of the *work* array. Constraints: *lwork* ≥ max(1, *n*) if *side* = 'L'; *lwork* ≥ max(1, *m*) if *side* = 'R'. If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

c	Overwritten by the product *Q*C*, Q^T**C*, *C*Q*, or *C**Q^T (as specified by *side* and *trans*).
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ormtr interface are the following:

a	Holds the matrix *A* of size (*r*,*r*). *r* = *m* if *side* = 'L'. *r* = *n* if *side* = 'R'.
tau	Holds the vector of length (*r*-1).
c	Holds the matrix *C* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = *n*blocksize* for *side* = 'L', or *lwork* = *m*blocksize* for *side* = 'R', where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed product differs from the exact product by a matrix E such that $||E||^2 = O(\varepsilon)||C||^2$.

The total number of floating-point operations is approximately $2*m^2*n$, if *side* = 'L', or $2*n^2*m$, if *side* = 'R'.

The complex counterpart of this routine is ?unmtr.

?hetrd

Reduces a complex Hermitian matrix to tridiagonal form.

Syntax

Fortran 77:

```
call chetrd(uplo, n, a, lda, d, e, tau, work, lwork, info)
call zhetrd(uplo, n, a, lda, d, e, tau, work, lwork, info)
```

Fortran 95:

```
call hetrd(a, tau [,uplo] [,info])
```

Description

This routine reduces a complex Hermitian matrix A to symmetric tridiagonal form T by a unitary similarity transformation: $A = Q*T*Q^H$. The unitary matrix Q is not formed explicitly but is represented as a product of n-1 elementary reflectors. Routines are provided to work with Q in this representation. (They are described later in this section .)

This routine calls ?latrd to reduce a complex Hermitian matrix A to Hermitian tridiagonal form by a unitary similarity transformation.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *a* stores the upper triangular part of *A*. If *uplo* = 'L', *a* stores the lower triangular part of *A*.
n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
a, *work*	COMPLEX for chetrd DOUBLE COMPLEX for zhetrd. *a*(*lda*,*) is an array containing either upper or lower triangular part of the matrix *A*, as specified by *uplo*. The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).
lwork	INTEGER. The size of the *work* array (*lwork* ≥ *n*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the tridiagonal matrix *T* and details of the unitary matrix *Q*, as specified by *uplo*.
d, *e*	REAL for chetrd DOUBLE PRECISION for zhetrd. Arrays: *d*(*) contains the diagonal elements of the matrix *T*. The dimension of *d* must be at least max(1, *n*). *e*(*) contains the off-diagonal elements of *T*. The dimension of *e* must be at least max(1, *n*-1).
tau	COMPLEX for chetrd DOUBLE COMPLEX for zhetrd. Array, DIMENSION at least max(1, *n*-1). Stores further details of the unitary matrix *Q*.

work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = –*i*, the *i*th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hetrd interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
tau	Holds the vector of length (*n*–1).
d	Holds the vector of length (*n*).
e	Holds the vector of length (*n*–1).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For better performance, try using *lwork* =*n*blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix T is exactly similar to a matrix $A + E$, where $||E||_2 = c(n)\varepsilon ||A||_2$, $c(n)$ is a modestly increasing function of n, and ε is the machine precision.

The approximate number of floating-point operations is $(16/3)n^3$.

After calling this routine, you can call the following:

?ungtr	to form the computed matrix Q explicitly
?unmtr	to multiply a complex matrix by Q.

The real counterpart of this routine is ?sytrd.

?ungtr

Generates the complex unitary matrix Q determined by ?hetrd.

Syntax

Fortran 77:

```
call cungtr(uplo, n, a, lda, tau, work, lwork, info)
call zungtr(uplo, n, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call ungtr(a, tau [,uplo] [,info])
```

The routine explicitly generates the n-by-n unitary matrix Q formed by ?hetrd when reducing a complex Hermitian matrix A to tridiagonal form: $A = Q*T*Q^H$. Use this routine after a call to ?hetrd.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Use the same *uplo* as supplied to ?hetrd.
n	INTEGER. The order of the matrix Q ($n \geq 0$).
a, *tau*, *work*	COMPLEX for cungtr DOUBLE COMPLEX for zungtr. Arrays: *a*(*lda*,*) is the array *a* as returned by ?hetrd. The second dimension of *a* must be at least max(1, *n*).

	tau(*) is the array *tau* as returned by ?hetrd. The dimension of *tau* must be at least max(1, *n*-1). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).
lwork	INTEGER. The size of the *work* array (*lwork* ≥ *n*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the unitary matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ungtr interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
tau	Holds the vector of length (*n*-1).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For better performance, try using *lwork* = (*n*-1)**blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix Q differs from an exactly unitary matrix by a matrix E such that $||E||_2 = O(\varepsilon)$, where e is the machine precision.

The approximate number of floating-point operations is $(16/3)n^3$.

The real counterpart of this routine is ?orgtr.

?unmtr

Multiplies a complex matrix by the complex unitary matrix Q determined by ?hetrd.

Syntax

Fortran 77:

```
call cunmtr(side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info)
call zunmtr(side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call unmtr(a, tau, c [,side] [,uplo] [,trans] [,info])
```

Description

The routine multiplies a complex matrix C by Q or Q^H, where Q is the unitary matrix Q formed by ?hetrd when reducing a complex Hermitian matrix A to tridiagonal form: $A = Q*T*Q^H$. Use this routine after a call to ?hetrd.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q*C$, Q^H*C, $C*Q$, or $C*Q^H$ (overwriting the result on C).

Input Parameters

In the descriptions below, r denotes the order of Q:

If *side* = 'L', $r = m$; if *side* = 'R', $r = n$.

side	CHARACTER*1. Must be either 'L' or 'R'. If *side* = 'L', Q or Q^H is applied to C from the left. If *side* = 'R', Q or Q^H is applied to C from the right.
uplo	CHARACTER*1. Must be 'U' or 'L'. Use the same *uplo* as supplied to ?hetrd.
trans	CHARACTER*1. Must be either 'N' or 'T'. If *trans* = 'N', the routine multiplies C by Q. If *trans* = 'T', the routine multiplies C by Q^H.
m	INTEGER. The number of rows in the matrix C ($m \geq 0$).
n	INTEGER. The number of columns in C ($n \geq 0$).
a, *c*, *tau*, *work*	COMPLEX for cunmtr DOUBLE COMPLEX for zunmtr. *a*(*lda*,*) and *tau* are the arrays returned by ?hetrd. The second dimension of *a* must be at least max(1, *r*). The dimension of *tau* must be at least max(1, *r*-1). *c*(*ldc*,*) contains the matrix C. The second dimension of *c* must be at least max(1, *n*) *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *r*).
ldc	INTEGER. The first dimension of *c*; *ldc* ≥ max(1, *n*).
lwork	INTEGER. The size of the *work* array. Constraints: *lwork* ≥ max(1, *n*) if *side* = 'L'; *lwork* ≥ max(1, *m*) if *side* = 'R'.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

c	Overwritten by the product *Q*C*, Q^H**C*, *C*Q*, or *C**Q^H (as specified by *side* and *trans*).
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine unmtr interface are the following:

a	Holds the matrix *A* of size (*r*,*r*). *r* = *m* if *side* = 'L'. *r* = *n* if *side* = 'R'.
tau	Holds the vector of length (*r*-1).
c	Holds the matrix *C* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = *n*blocksize* (for *side* = 'L') or *lwork* = *m*blocksize* (for *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed product differs from the exact product by a matrix E such that $||E||_2 = O(\varepsilon)||C||_2$, where ε is the machine precision.

The total number of floating-point operations is approximately $8*m^2*n$ if *side* = 'L' or $8*n^2*m$ if *side* = 'R'.

The real counterpart of this routine is ?ormtr.

?sptrd

Reduces a real symmetric matrix to tridiagonal form using packed storage.

Syntax

Fortran 77:

```
call ssptrd(uplo, n, ap, d, e, tau, info)
call dsptrd(uplo, n, ap, d, e, tau, info)
```

Fortran 95:

```
call sptrd(a, tau [,uplo] [,info])
```

Description

This routine reduces a packed real symmetric matrix *A* to symmetric tridiagonal form *T* by an orthogonal similarity transformation: $A = Q*T*Q^T$. The orthogonal matrix *Q* is not formed explicitly but is represented as a product of *n*-1 elementary reflectors. Routines are provided for working with *Q* in this representation. (They are described later in this section .)

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *ap* stores the packed upper triangle of *A*. If *uplo* = 'L', *ap* stores the packed lower triangle of *A*.
n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
ap	REAL for ssptrd DOUBLE PRECISION for dsptrd. Array, DIMENSION at least max(1, *n*(*n*+1)/2). Contains either upper or lower triangle of *A* (as specified by *uplo*) in packed form.

Output Parameters

ap	Overwritten by the tridiagonal matrix *T* and details of the orthogonal matrix *Q*, as specified by *uplo*.
d, *e*, *tau*	REAL for ssptrd DOUBLE PRECISION for dsptrd. Arrays: *d*(*) contains the diagonal elements of the matrix *T*. The dimension of *d* must be at least max(1, *n*). *e*(*) contains the off-diagonal elements of *T*. The dimension of *e* must be at least max(1, *n*-1). *tau*(*) stores further details of the matrix *Q*. The dimension of *tau* must be at least max(1, *n*-1).
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `sptrd` interface are the following:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n*(n+1)/2*).
tau	Holds the vector of length (*n*-1).
d	Holds the vector of length (*n*).
e	Holds the vector of length (*n*-1).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

The computed matrix *T* is exactly similar to a matrix $A + E$, where $||E||_2 = c(n)\varepsilon ||A||_2$, $c(n)$ is a modestly increasing function of n, and ε is the machine precision. The approximate number of floating-point operations is $(4/3)n^3$.

After calling this routine, you can call the following:

`?opgtr`	to form the computed matrix *Q* explicitly
`?opmtr`	to multiply a real matrix by *Q*.

The complex counterpart of this routine is `?hptrd`.

?opgtr

Generates the real orthogonal matrix Q determined by `?sptrd`.

Syntax

Fortran 77:

```
call sopgtr(uplo, n, ap, tau, q, ldq, work, info)
call dopgtr(uplo, n, ap, tau, q, ldq, work, info)
```

Fortran 95:

```
call opgtr(a, tau, q [,uplo] [,info])
```

Description

The routine explicitly generates the n-by-n orthogonal matrix Q formed by ?sptrd when reducing a packed real symmetric matrix A to tridiagonal form: $A = Q*T*Q^T$. Use this routine after a call to ?sptrd.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Use the same *uplo* as supplied to ?sptrd.
n	INTEGER. The order of the matrix Q ($n \geq 0$).
ap, *tau*	REAL for sopgtr DOUBLE PRECISION for dopgtr. Arrays *ap* and *tau*, as returned by ?sptrd. The dimension of *ap* must be at least max(1, $n(n+1)/2$). The dimension of *tau* must be at least max(1, $n-1$).
ldq	INTEGER. The first dimension of the output array q; at least max(1, n).
work	REAL for sopgtr DOUBLE PRECISION for dopgtr. Workspace array, DIMENSION at least max(1, $n-1$).

Output Parameters

q	REAL for sopgtr DOUBLE PRECISION for dopgtr. Array, DIMENSION (*ldq*,*). Contains the computed matrix Q. The second dimension of q must be at least max(1, n).
info	INTEGER. If *info* = 0, the execution is successful. If *info* = $-i$, the ith parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `opgtr` interface are the following:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n*(n+1)/2*).
tau	Holds the vector of length (*n*–1).
q	Holds the matrix *Q* of size (*n*,*n*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

The computed matrix *Q* differs from an exactly orthogonal matrix by a matrix *E* such that $||E||_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of floating-point operations is $(4/3)n^3$.

The complex counterpart of this routine is `?upgtr`.

?opmtr

Multiplies a real matrix by the real orthogonal matrix Q determined by `?sptrd`.

Syntax

Fortran 77:

```
call sopmtr(side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
call dopmtr(side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
```

Fortran 95:

```
call opmtr(a, tau, c [,side] [,uplo] [,trans] [,info])
```

Description

The routine multiplies a real matrix C by Q or Q^T, where Q is the orthogonal matrix Q formed by ?sptrd when reducing a packed real symmetric matrix A to tridiagonal form: $A = QTQ^T$. Use this routine after a call to ?sptrd.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products Q

$*C$, Q^T*C, $C*Q$, or $C*Q^T$ (overwriting the result on C).

Input Parameters

In the descriptions below, r denotes the order of Q:

If *side* = 'L', $r = m$; if *side* = 'R', $r = n$.

side	CHARACTER*1. Must be either 'L' or 'R'. If *side* = 'L', Q or Q^T is applied to C from the left. If *side* = 'R', Q or Q^T is applied to C from the right.
uplo	CHARACTER*1. Must be 'U' or 'L'. Use the same *uplo* as supplied to ?sptrd.
trans	CHARACTER*1. Must be either 'N' or 'T'. If *trans* = 'N', the routine multiplies C by Q. If *trans* = 'T', the routine multiplies C by Q^T.
m	INTEGER. The number of rows in the matrix C ($m \geq 0$).
n	INTEGER. The number of columns in C ($n \geq 0$).
ap, *tau*, *c*, *work*	REAL for sopmtr DOUBLE PRECISION for dopmtr. *ap* and *tau* are the arrays returned by ?sptrd. The dimension of *ap* must be at least max(1, r(r+1)/2). The dimension of *tau* must be at least max(1, r-1). c(*ldc*,*) contains the matrix C. The second dimension of c must be at least max(1, n) *work*(*) is a workspace array. The dimension of *work* must be at least max(1, n) if *side* = 'L'; max(1, m) if *side* = 'R'.
ldc	INTEGER. The first dimension of c; *ldc* $\geq$ max(1, n).

Output Parameters

c	Overwritten by the product *Q*C*, Q^T**C*, *C*Q*, or *C*Q^T* (as specified by *side* and *trans*).
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine opmtr interface are the following:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*r*(r+1)/2*), where *r* = *m* if *side* = 'L'. *r* = *n* if *side* = 'R'.
tau	Holds the vector of length (*r*-1).
c	Holds the matrix *C* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
uplo	Must be 'U' or 'L'. The default value is 'U'.
trans	Must be 'N', 'C', or 'T'. The default value is 'N'.

Application Notes

The computed product differs from the exact product by a matrix *E* such that $||E||_2 = O(\varepsilon)||C||_2$, where ε is the machine precision.

The total number of floating-point operations is approximately $2*m^2*n$ if *side* = 'L' or $2*n^2*m$ if *side* = 'R'.

The complex counterpart of this routine is ?upmtr.

?hptrd

Reduces a complex Hermitian matrix to tridiagonal form using packed storage.

Syntax

Fortran 77:

```
call chptrd(uplo, n, ap, d, e, tau, info)
call zhptrd(uplo, n, ap, d, e, tau, info)
```

Fortran 95:

```
call hptrd(a, tau [,uplo] [,info])
```

Description

This routine reduces a packed complex Hermitian matrix A to symmetric tridiagonal form T by a unitary similarity transformation: $A = Q*T*Q^H$. The unitary matrix Q is not formed explicitly but is represented as a product of n-1 elementary reflectors. Routines are provided for working with Q in this representation. They are described later in this section .

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *ap* stores the packed upper triangle of *A*. If *uplo* = 'L', *ap* stores the packed lower triangle of *A*.
n	INTEGER. The order of the matrix *A* ($n \geq 0$).
ap	COMPLEX for chptrd DOUBLE COMPLEX for zhptrd. Array, DIMENSION at least max(1, $n(n+1)/2$). Contains either upper or lower triangle of *A* (as specified by *uplo*) in packed form.

Output Parameters

ap	Overwritten by the tridiagonal matrix *T* and details of the orthogonal matrix *Q*, as specified by *uplo*.
d, *e*	REAL for chptrd DOUBLE PRECISION for zhptrd.

	Arrays: *d*(*) contains the diagonal elements of the matrix *T*. The dimension of *d* must be at least max(1, *n*). *e*(*) contains the off-diagonal elements of *T*. The dimension of *e* must be at least max(1, *n*-1).
tau	COMPLEX for chptrd DOUBLE COMPLEX for zhptrd. Arrays, DIMENSION at least max(1, *n*-1). Contains further details of the orthogonal matrix *Q*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hptrd interface are the following:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).
tau	Holds the vector of length (*n*-1).
d	Holds the vector of length (*n*).
e	Holds the vector of length (*n*-1).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed matrix *T* is exactly similar to a matrix *A* + *E*, where $||E||_2 = c(n)\varepsilon ||A||_2$, $c(n)$ is a modestly increasing function of *n*, and ε is the machine precision.

The approximate number of floating-point operations is $(16/3)n^3$.

After calling this routine, you can call the following:

?upgtr	to form the computed matrix *Q* explicitly
?upmtr	to multiply a complex matrix by *Q*.

The real counterpart of this routine is ?sptrd.

?upgtr

Generates the complex unitary matrix Q determined by ?hptrd.

Syntax

Fortran 77:

```
call cupgtr(uplo, n, ap, tau, q, ldq, work, info)
call zupgtr(uplo, n, ap, tau, q, ldq, work, info)
```

Fortran 95:

```
call upgtr(a, tau, q [,uplo] [,info])
```

Description

The routine explicitly generates the n-by-n unitary matrix Q formed by ?hptrd when reducing a packed complex Hermitian matrix A to tridiagonal form: $A = Q*T*Q^H$. Use this routine after a call to ?hptrd.

Input Parameters

uplo	CHARACTER*1. Must be 'U' or 'L'. Use the same *uplo* as supplied to ?sptrd.
n	INTEGER. The order of the matrix Q ($n \geq 0$).
ap, *tau*	COMPLEX for cupgtr DOUBLE COMPLEX for zupgtr. Arrays *ap* and *tau*, as returned by ?hptrd. The dimension of *ap* must be at least max(1, $n(n+1)/2$). The dimension of *tau* must be at least max(1, $n-1$).
ldq	INTEGER. The first dimension of the output array q; at least max(1, n).
work	COMPLEX for cupgtr DOUBLE COMPLEX for zupgtr. Workspace array, DIMENSION at least max(1, $n-1$).

Output Parameters

q	COMPLEX for cupgtr DOUBLE COMPLEX for zupgtr. Array, DIMENSION (*ldq*,*). Contains the computed matrix *Q*. The second dimension of *q* must be at least max(1, *n*).
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine upgtr interface are the following:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).
tau	Holds the vector of length (*n*-1).
q	Holds the matrix *Q* of size (*n*,*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed matrix *Q* differs from an exactly orthogonal matrix by a matrix *E* such that $||E||_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of floating-point operations is (16/3)*n3*.

The real counterpart of this routine is ?opgtr.

?upmtr

Multiplies a complex matrix by the unitary matrix Q determined by ?hptrd.

Syntax

Fortran 77:

```
call cupmtr(side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
call zupmtr(side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
```

Fortran 95:

```
call upmtr(a, tau, c [,side] [,uplo] [,trans] [,info])
```

Description

The routine multiplies a complex matrix C by Q or Q^H, where Q is the unitary matrix Q formed by ?hptrd when reducing a packed complex Hermitian matrix A to tridiagonal form: $A = Q*T*Q^H$. Use this routine after a call to ?hptrd.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q*C$, Q^H*C, $C*Q$, or $C*Q^H$ (overwriting the result on C).

Input Parameters

In the descriptions below, r denotes the order of Q:

If *side* = 'L', $r = m$; if *side* = 'R', $r = n$.

side	CHARACTER*1. Must be either 'L' or 'R'. If *side* = 'L', Q or Q^H is applied to C from the left. If *side* = 'R', Q or Q^H is applied to C from the right.
uplo	CHARACTER*1. Must be 'U' or 'L'. Use the same *uplo* as supplied to ?hptrd.
trans	CHARACTER*1. Must be either 'N' or 'T'. If *trans* = 'N', the routine multiplies C by Q. If *trans* = 'T', the routine multiplies C by Q^H.
m	INTEGER. The number of rows in the matrix C ($m \geq 0$).
n	INTEGER. The number of columns in C ($n \geq 0$).

ap, *tau*, *c*,	COMPLEX for cupmtr DOUBLE COMPLEX for zupmtr. *ap* and *tau* are the arrays returned by ?hptrd. The dimension of *ap* must be at least max(1, *r*(*r*+1)/2). The dimension of *tau* must be at least max(1, *r*-1). *c*(*ldc*,*) contains the matrix *C*. The second dimension of *c* must be at least max(1, *n*) *work*(*) is a workspace array. The dimension of *work* must be at least max(1, *n*) if *side* = 'L'; max(1, *m*) if *side* = 'R'.
ldc	INTEGER. The first dimension of *c*; *ldc* ≥ max(1, *n*).

Output Parameters

c	Overwritten by the product *Q*C*, Q^H*C, *C*Q*, or $C*Q^H$ (as specified by *side* and *trans*).
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine upmtr interface are the following:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*r*(r+1)/2*), where *r* = *m* if *side* = 'L'. *r* = *n* if *side* = 'R'.
tau	Holds the vector of length (*r*-1).
c	Holds the matrix *C* of size (*m*,*n*).
side	Must be 'L' or 'R'. The default value is 'L'.
uplo	Must be 'U' or 'L'.The default value is 'U'.
trans	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

The computed product differs from the exact product by a matrix E such that $||E||_2 = O\varepsilon||C||_2$, where e is the machine precision.

The total number of floating-point operations is approximately $8*m^2*n$ if *side* = 'L' or $8*n^2*m$ if *side* = 'R'.

The real counterpart of this routine is ?opmtr.

?sbtrd

Reduces a real symmetric band matrix to tridiagonal form.

Syntax

Fortran 77:

```
call ssbtrd(vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
call dsbtrd(vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
```

Fortran 95:

```
call sbtrd(a [, q] [,vect] [,uplo] [,info])
```

Description

This routine reduces a real symmetric band matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: $A = Q*T*Q^T$. The orthogonal matrix Q is determined as a product of Givens rotations.

If required, the routine can also form the matrix Q explicitly.

Input Parameters

vect CHARACTER*1. Must be 'V' or 'N'.
If *vect* = 'V', the routine returns the explicit matrix Q.
If *vect* = 'N', the routine does not return Q.

uplo CHARACTER*1. Must be 'U' or 'L'.
If *uplo* = 'U', *ab* stores the upper triangular part of A.
If *uplo* = 'L', *ab* stores the lower triangular part of A.

n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
kd	INTEGER. The number of super- or sub-diagonals in *A* (*kd* < 0).
ab, *work*	REAL for ssbtrd DOUBLE PRECISION for dsbtrd. *ab* (*ldab*,*) is an array containing either upper or lower triangular part of the matrix *A* (as specified by *uplo*) in band storage format. The second dimension of *ab* must be at least max(1, *n*). *work*(*) is a workspace array. The dimension of *work* must be at least max(1, *n*).
ldab	INTEGER. The first dimension of *ab*; at least *kd*+1.
ldq	INTEGER. The first dimension of *q*. Constraints: *ldq* ≥ max(1, *n*) if *vect* = 'V'; *ldq* ≥ 1 if *vect* = 'N'.

Output Parameters

ab	On exit, the array *ab* is overwritten.
d, *e*, *q*	REAL for ssbtrd DOUBLE PRECISION for dsbtrd. Arrays: *d*(*) contains the diagonal elements of the matrix *T*. The dimension of *d* must be at least max(1, *n*). *e*(*) contains the off-diagonal elements of *T*. The dimension of *e* must be at least max(1, *n*-1). *q*(*ldq*,*) is not referenced if *vect* = 'N'. If *vect* = 'V', *q* contains the *n*-by-*n* matrix *Q*. The second dimension of *q* must be: at least max(1, *n*) if *vect* = 'V'; at least 1 if *vect* = 'N'.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `sbtrd` interface are the following:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*kd+1*,*n*).
q	Holds the matrix *Q* of size (*n*,*n*).
d	Holds the vector of length (*n*).
e	Holds the vector of length (*n*-1).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
vect	If omitted, this argument is restored based on the presence of argument *q* as follows: *vect* = `'V'`, if *q* is present, *vect* = `'N'`, if *q* is omitted. If present, *vect* must be equal to `'V'` or `'U'` and the argument *q* must also be present. Note that there will be an error condition if *vect* is present and *q* omitted.

Application Notes

The computed matrix *T* is exactly similar to a matrix *A* + *E*, where $||E||_2 = c(n)\varepsilon ||A||_2$, $c(n)$ is a modestly increasing function of *n*, and ε is the machine precision. The computed matrix *Q* differs from an exactly orthogonal matrix by a matrix *E* such that $||E||_2 = O(\varepsilon)$.

The total number of floating-point operations is approximately $6n^2*kd$ if *vect* = `'N'`, with $3n^3*(kd-1)/kd$ additional operations if *vect* = `'V'`.

The complex counterpart of this routine is `?hbtrd`.

?hbtrd

Reduces a complex Hermitian band matrix to tridiagonal form.

Syntax

Fortran 77:

```
call chbtrd(vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
call zhbtrd(vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
```

Fortran 95:

```
call hbtrd(a [, q] [,vect] [,uplo] [,info])
```

Description

This routine reduces a complex Hermitian band matrix A to symmetric tridiagonal form T by a unitary similarity transformation: $A = Q*T*Q^H$. The unitary matrix Q is determined as a product of Givens rotations.

If required, the routine can also form the matrix Q explicitly.

Input Parameters

vect	CHARACTER*1. Must be 'V' or 'N'. If *vect* = 'V', the routine returns the explicit matrix Q. If *vect* = 'N', the routine does not return Q.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *ab* stores the upper triangular part of A. If *uplo* = 'L', *ab* stores the lower triangular part of A.
n	INTEGER. The order of the matrix A ($n \geq 0$).
kd	INTEGER. The number of super- or sub-diagonals in A ($kd \geq 0$).
ab, *work*	COMPLEX for chbtrd DOUBLE COMPLEX for zhbtrd. *ab* (*ldab*,*) is an array containing either upper or lower triangular part of the matrix A (as specified by *uplo*) in band storage format.

The second dimension of *ab* must be at least max(1, *n*).
work(*) is a workspace array.
The dimension of *work* must be at least max(1, *n*).

ldab INTEGER. The first dimension of *ab*; at least *kd*+1.

ldq INTEGER. The first dimension of *q*. Constraints:

ldq ≥ max(1, *n*) if *vect* = 'V';

ldq ≥ 1 if *vect* = 'N'.

Output Parameters

ab On exit, the array *ab* is overwritten.

d, *e* REAL for chbtrd
DOUBLE PRECISION for zhbtrd.
Arrays:
d(*) contains the diagonal elements of the matrix *T*.
The dimension of *d* must be at least max(1, *n*).
e(*) contains the off-diagonal elements of *T*.
The dimension of *e* must be at least max(1, *n*-1).

q COMPLEX for chbtrd
DOUBLE COMPLEX for zhbtrd.
Array, DIMENSION (*ldq*,*).
If *vect* = 'N', *q* is not referenced.
If *vect* = 'V', *q* contains the *n*-by-*n* matrix *Q*.
The second dimension of *q* must be:
at least max(1, *n*) if *vect* = 'V';
at least 1 if *vect* = 'N'.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hbtrd interface are the following:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*kd+1*,*n*).
q	Holds the matrix *Q* of size (*n*,*n*).
d	Holds the vector of length (*n*).
e	Holds the vector of length (*n*-1).
uplo	Must be 'U' or 'L'. The default value is 'U'.
vect	If omitted, this argument is restored based on the presence of argument *q* as follows: *vect* = 'V', if *q* is present, *vect* = 'N', if *q* is omitted. If present, *vect* must be equal to 'V' or 'U' and the argument *q* must also be present. Note that there will be an error condition if *vect* is present and *q* omitted.

Application Notes

The computed matrix *T* is exactly similar to a matrix *A* + *E*, where $||E||_2 = c(n)\varepsilon ||A||_2$, $c(n)$ is a modestly increasing function of *n*, and ε is the machine precision. The computed matrix *Q* differs from an exactly unitary matrix by a matrix *E* such that $||E||_2 = O(\varepsilon)$.

The total number of floating-point operations is approximately $20n^2*kd$ if *vect* = 'N', with $10n^3*(kd-1)/kd$ additional operations if *vect* = 'V'.

The real counterpart of this routine is ?sbtrd.

?sterf

Computes all eigenvalues of a real symmetric tridiagonal matrix using QR algorithm.

Syntax

Fortran 77:

```
call ssterf(n, d, e, info)
call dsterf(n, d, e, info)
```

Fortran 95:

```
call sterf(d, e [,info])
```

Description

This routine computes all the eigenvalues of a real symmetric tridiagonal matrix *T* (which can be obtained by reducing a symmetric or Hermitian matrix to tridiagonal form). The routine uses a square-root-free variant of the *QR* algorithm.

If you need not only the eigenvalues but also the eigenvectors, call ?steqr.

Input Parameters

n	INTEGER. The order of the matrix *T* (*n* ≥ 0).
d, *e*	REAL for ssterf DOUBLE PRECISION for dsterf. Arrays: *d*(*) contains the diagonal elements of *T*. The dimension of *d* must be at least max(1, *n*). *e*(*) contains the off-diagonal elements of *T*. The dimension of *e* must be at least max(1, *n*-1).

Output Parameters

d	The *n* eigenvalues in ascending order, unless *info* > 0. See also *info*.
e	On exit, the array is overwritten; see *info*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *i*, the algorithm failed to find all the eigenvalues after 30*n* iterations: *i* off-diagonal elements have not converged to zero. On exit, *d* and *e* contain, respectively, the diagonal and off-diagonal elements of a tridiagonal matrix orthogonally similar to *T*. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine sterf interface are the following:

d	Holds the vector of length (*n*).
e	Holds the vector of length (*n*–1).

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T + E$ such that $||E||_2 = O(\varepsilon)\ ||T||_2$, where ε is the machine precision.

If λ_i is an exact eigenvalue, and m_i is the corresponding computed value, then

$|\mu i - \lambda i| \le c(n)\varepsilon\ ||T||_2$

where $c(n)$ is a modestly increasing function of n.

The total number of floating-point operations depends on how rapidly the algorithm converges. Typically, it is about $14n^2$.

?steqr

Computes all eigenvalues and eigenvectors of a symmetric or Hermitian matrix reduced to tridiagonal form (QR algorithm).

Syntax

Fortran 77:

```
call ssteqr(compz, n, d, e, z, ldz, work, info)
call dsteqr(compz, n, d, e, z, ldz, work, info)
call csteqr(compz, n, d, e, z, ldz, work, info)
call zsteqr(compz, n, d, e, z, ldz, work, info)
```

Fortran 95:

```
call rsteqr(d, e [,z] [,compz] [,info])
call steqr(d, e [,z] [,compz] [,info])
```

Description

This routine computes all the eigenvalues and (optionally) all the eigenvectors of a real symmetric tridiagonal matrix T. In other words, the routine can compute the spectral factorization: $T = Z\Lambda Z^T$. Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i; Z is an orthogonal matrix whose columns are eigenvectors. Thus,

$Tz_i = \lambda_i z_i$ for $i = 1, 2, \ldots, n$.

The routine normalizes the eigenvectors so that $||z_i||_2 = 1$.

You can also use the routine for computing the eigenvalues and eigenvectors of an arbitrary real symmetric (or complex Hermitian) matrix A reduced to tridiagonal form T: $A = Q*T*Q^H$. In this case, the spectral factorization is as follows: $A = Q*T*Q^H = (Q*Z)\Lambda(Q*Z)^H$. Before calling ?steqr, you must reduce A to tridiagonal form and generate the explicit matrix Q by calling the following routines:

	for real matrices:	for complex matrices:
full storage	?sytrd, ?orgtr	?hetrd, ?ungtr
packed storage	?sptrd, ?opgtr	?hptrd, ?upgtr
band storage	?sbtrd *(vect*='V')	?hbtrd *(vect*='V')

If you need eigenvalues only, it's more efficient to call ?sterf. If T is positive-definite, ?pteqr can compute small eigenvalues more accurately than ?steqr.

To solve the problem by a single call, use one of the divide and conquer routines ?stevd, ?syevd, ?spevd, or ?sbevd for real symmetric matrices or ?heevd, ?hpevd, or ?hbevd for complex Hermitian matrices.

Input Parameters

compz — CHARACTER*1. Must be 'N' or 'I' or 'V'.
If *compz* = 'N', the routine computes eigenvalues only.
If *compz* = 'I', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix T.
If *compz* = 'V', the routine computes the eigenvalues and eigenvectors of A (and the array z must contain the matrix Q on entry).

n	INTEGER. The order of the matrix T ($n \geq 0$).
d, *e*, *work*	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: *d*(*) contains the diagonal elements of T. The dimension of *d* must be at least max(1, *n*). *e*(*) contains the off-diagonal elements of T. The dimension of *e* must be at least max(1, *n*-1). *work*(*) is a workspace array. The dimension of *work* must be: at least 1 if *compz* = 'N'; at least max(1, 2**n*-2) if *compz* = 'V' or 'I'.
z	REAL for ssteqr DOUBLE PRECISION for dsteqr COMPLEX for csteqr DOUBLE COMPLEX for zsteqr. Array, DIMENSION (*ldz*, *) If *compz* = 'N' or 'I', *z* need not be set. If *vect* = 'V', *z* must contain the *n*-by-*n* matrix Q. The second dimension of *z* must be: at least 1 if *compz* = 'N'; at least max(1, *n*) if *compz* = 'V' or 'I'. *work* (*lwork*) is a workspace array.
ldz	INTEGER. The first dimension of *z*. Constraints: *ldz* ≥ 1 if *compz* = 'N'; *ldz* ≥ max(1, *n*) if *compz* = 'V' or 'I'.

Output Parameters

d	The *n* eigenvalues in ascending order, unless *info* > 0. See also *info*.
e	On exit, the array is overwritten; see *info*.
z	If *info* = 0, contains the *n* orthonormal eigenvectors, stored by columns. (The *i*-th column corresponds to the *i*th eigenvalue.)
info	INTEGER.

If *info* = 0, the execution is successful.
If *info* = *i*, the algorithm failed to find all the eigenvalues after 30*n* iterations: *i* off-diagonal elements have not converged to zero. On exit, *d* and *e* contain, respectively, the diagonal and off-diagonal elements of a tridiagonal matrix orthogonally similar to *T*.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine steqr interface are the following:

d — Holds the vector of length (*n*).

e — Holds the vector of length (*n*-1).

z — Holds the matrix *Z* of size (*n*,*n*).

compz — If omitted, this argument is restored based on the presence of argument *z* as follows:
compz = 'I', if *z* is present,
compz = 'N', if *z* is omitted.
If present, *compz* must be equal to 'I' or 'V' and the argument *z* must also be present. Note that there will be an error condition if *compz* is present and *z* omitted.
Note that two variants of Fortran 95 interface for steqr routine are needed because of an ambiguous choice between real and complex cases appear when *z* is omitted. Thus, the name rsteqr is used in real cases (single or double precision), and the name steqr is used in complex cases (single or double precision).

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T + E$ such that $||E||_2 = O(\varepsilon)\ ||T||_2$, where ε is the machine precision.

If λ_i is an exact eigenvalue, and μ_i is the corresponding computed value, then

$|\mu_i - \lambda_i| \leq c(n)\varepsilon\ ||T||_2$

where $c(n)$ is a modestly increasing function of n.

If z_i is the corresponding exact eigenvector, and w_i is the corresponding computed vector, then the angle $\theta(z_i, w_i)$ between them is bounded as follows:

$\theta(z_i, w_i) \leq c(n)\varepsilon \, ||T||_2 / \min_{i \neq j}|\lambda_i - \lambda_j|$.

The total number of floating-point operations depends on how rapidly the algorithm converges. Typically, it is about

$24n^2$ if *compz* = 'N';

$7n^3$ (for complex flavors, $14n^3$) if *compz* = 'V' or 'I'.

?stemr

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

Fortran 77:

```
call sstemr(jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz,
tryrac, work, lwork, iwork, liwork, info)
call dstemr(jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz,
tryrac, work, lwork, iwork, liwork, info)
call cstemr(jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz,
tryrac, work, lwork, iwork, liwork, info)
call zstemr(jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz,
tryrac, work, lwork, iwork, liwork, info)
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T. Any such unreduced matrix has a well defined set of pairwise different real eigenvalues, the corresponding real eigenvectors are pairwise orthogonal.

The spectrum may be computed either completely or partially by specifying either an interval (vl,vu] or a range of indices il:iu for the desired eigenvalues.

Depending on the number of desired eigenvalues, these are computed either by bisection or the *dqds* algorithm. Numerically orthogonal eigenvectors are computed by the use of various suitable $L*D*L^T$ factorizations near clusters of close eigenvalues (referred to as RRRs, Relatively Robust Representations). An informal sketch of the algorithm follows.

For each unreduced block (submatrix) of T,

a. Compute $T - \sigma I = L*D*L^T$, so that L and D define all the wanted eigenvalues to high relative accuracy. This means that small relative changes in the entries of L and D cause only small relative changes in the eigenvalues and eigenvectors. The standard (unfactored) representation of the tridiagonal matrix T does not have this property in general.

b. Compute the eigenvalues to suitable accuracy. If the eigenvectors are desired, the algorithm attains full accuracy of the computed eigenvalues only right before the corresponding vectors have to be computed, see steps c and d.

c. For each cluster of close eigenvalues, select a new shift close to the cluster, find a new factorization, and refine the shifted eigenvalues to suitable accuracy.

d. For each eigenvalue with a large enough relative separation compute the corresponding eigenvector by forming a rank revealing twisted factorization. Go back to step c for any clusters that remain.

For more details, see: [Dhillon04], [Dhillon04-02], [Dhillon97]

The routine works only on machines which follow IEEE-754 floating-point standard in their handling of infinities and NaNs (NaN stands for "not a number"). This permits the use of efficient inner loops avoiding a check for zero divisors.

LAPACK routines can be used to reduce a complex Hermitean matrix to real symmetric tridiagonal form.

(Any complex Hermitean tridiagonal matrix has real values on its diagonal and potentially complex numbers on its off-diagonals. By applying a similarity transform with an appropriate diagonal matrix $\mathrm{diag}(1, e^{i\phi_1}, \ldots, e^{i\phi_{n-1}})$, the complex Hermitean matrix can be transformed into a real symmetric matrix and complex arithmetic can be entirely avoided.) While the eigenvectors of the real symmetric tridiagonal matrix are real, the eigenvectors of original complex Hermitean matrix have complex entries in general. Since LAPACK drivers overwrite the matrix data with the eigenvectors, `zstemr` accepts complex workspace to facilitate interoperability with `zunmtr` or `zupmtr`.

Input Parameters

jobz — `CHARACTER*1`. Must be `'N'` or `'V'`.
If *jobz* = `'N'`, then only eigenvalues are computed.

If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

range — CHARACTER*1. Must be 'A' or 'V' or 'I'.
If *range* = 'A', the routine computes all eigenvalues.
If *range* = 'V', the routine computes all eigenvalues in the half-open interval: (*vl*, *vu*].
If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.

n — INTEGER. The order of the matrix *T* (*n*≥0).

d — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Array, DIMENSION (*n*).
Contains *n* diagonal elements of the tridiagonal matrix *T*.

e — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Array, DIMENSION (*n*–1).
Contains (*n*–1) off-diagonal elements of the tridiagonal matrix *T* in elements 1 to *n*–1 of *e*. *e*(*n*) need not be set on input, but is used internally as workspace.

vl, *vu* — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: *vl*<*vu*.
If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu* — INTEGER.
If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: 1≤*il*≤*iu*≤*n*, if *n*>0.
If *range* = 'A' or 'V', *il* and *iu* are not referenced.

ldz — INTEGER. The leading dimension of the output array *z*.
if *jobz* = 'V', then *ldz* ≥ max(1, *n*);
ldz ≥ 1 otherwise.

nzc — INTEGER. The number of eigenvectors to be held in the array *z*.
If *range* = 'A', then *nzc*≥max(1, *n*);

If *range* = 'V', then *nzc* is greater than or equal to the number of eigenvalues in the half-open interval: (*vl*, *vu*].

If *range* = 'I', then *nzc*≥*il*+*iu*+1.

This value is returned as the first entry of the array *z*, and no error message related to *nzc* is issued by the routine xerbla.

tryrac LOGICAL.

If *tryrac* = .TRUE., it indicates that the code should check whether the tridiagonal matrix defines its eigenvalues to high relative accuracy. If so, the code uses relative-accuracy preserving algorithms that might be (a bit) slower depending on the matrix. If the matrix does not define its eigenvalues to high relative accuracy, the code can uses possibly faster algorithms.

If *tryrac* = .FALSE., the code is not required to guarantee relatively accurate eigenvalues and can use the fastest possible techniques.

work REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, DIMENSION (*lwork*).

lwork INTEGER.

The dimension of the array *work*,

lwork≥max(1, 18**n*).

If *lwork*=−1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

iwork INTEGER.

Workspace array, DIMENSION (*liwork*).

liwork INTEGER.

The dimension of the array *iwork*.

lwork≥max(1, 10**n*) if the eigenvectors are desired, and

lwork≥max(1, 8**n*) if only the eigenvalues are to be computed.

If *liwork*=-1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by xerbla.

Output Parameters

e — On exit, the array *e* is overwritten.

m — INTEGER.
The total number of eigenvalues found, $0 \le m \le n$.
If *range* = 'A', then $m=n$, and if If *range* = 'I', then $m=iu-il+1$.

w — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Array, DIMENSION (*n*).
The first *m* elements contain the selected eigenvalues in ascending order.

z — REAL for sstemr
DOUBLE PRECISION for dstemr
COMPLEX for cstemr
DOUBLE COMPLEX for zstemr.
Array *z*(*ldz*, *), the second dimension of *z* must be at least max(1, *m*).
If *jobz* = 'V', and *info* = 0, then the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the selected eigenvalues, with the i-th column of *z* holding the eigenvector associated with *w*(i).
If *jobz* = 'N', then *z* is not referenced.
Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z* ; if *range* = 'V', the exact value of *m* is not known in advance and an can be computed with a workspace query by setting *nzc*=-1, see description of the parameter *nzc*.

isuppz — INTEGER.
Array, DIMENSION (2*max(1, *m*)).

	The support of the eigenvectors in *z*, that is the indices indicating the nonzero elements in *z*. The i-th computed eigenvector is nonzero only in elements *isuppz*(2*i-1) through *isuppz*(2*i). This is relevant in the case when the matrix is split. *isuppz* is only accessed when *jobz* = 'V' and *n*>0.
tryrac	On exit, TRUE. *tryrac* is set to .FALSE. if the matrix does not define its eigenvalues to high relative accuracy.
work(1)	On exit, if *info* = 0, then *work*(1) returns the optimal (and minimal) size of *lwork*.
iwork(1)	On exit, if *info* = 0, then *iwork*(1) returns the optimal size of *liwork*.
info	INTEGER. If = 0, the execution is successful. If *info* = -i, the i-th parameter had an illegal value. If *info* = 1, internal error in ?larre occurred, if *info* = 2, internal error in ?larrv occurred.

?stedc

Computes all eigenvalues and eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method.

Syntax

Fortran 77:

call sstedc(*compz*, *n*, *d*, *e*, *z*, *ldz*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call dstedc(*compz*, *n*, *d*, *e*, *z*, *ldz*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call cstedc(*compz*, *n*, *d*, *e*, *z*, *ldz*, *work*, *lwork*, *rwork*, *lrwork*, *iwork*, *liwork*, *info*)

call zstedc(*compz*, *n*, *d*, *e*, *z*, *ldz*, *work*, *lwork*, *rwork*, *lrwork*, *iwork*, *liwork*, *info*)

Fortran 95:

```
call rstedc(d, e [,z] [,compz] [,info])
call stedc(d, e [,z] [,compz] [,info])
```

Description

This routine computes all the eigenvalues and (optionally) all the eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method. The eigenvectors of a full or band real symmetric or complex Hermitian matrix can also be found if ?sytrd/?hetrd or ?sptrd/?hptrd or ?sbtrd/?hbtrd has been used to reduce this matrix to tridiagonal form.

See also ?laed0, ?laed1, ?laed2, ?laed3, ?laed4, ?laed5, ?laed6, ?laed7, ?laed8, ?laed9, and ?laeda used by this function.

Input Parameters

compz	CHARACTER*1. Must be 'N' or 'I' or 'V'. If *compz* = 'N', the routine computes eigenvalues only. If *compz* = 'I', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix. If *compz* = 'V', the routine computes the eigenvalues and eigenvectors of original symmetric/Hermitian matrix. On entry, the array *z* must contain the orthogonal/unitary matrix used to reduce the original matrix to tridiagonal form.
n	INTEGER. The order of the symmetric tridiagonal matrix (*n* ≥ 0).
d, *e*, *rwork*	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: *d*(*) contains the diagonal elements of the tridiagonal matrix. The dimension of *d* must be at least max(1, *n*). *e*(*) contains the subdiagonal elements of the tridiagonal matrix. The dimension of *e* must be at least max(1, *n*-1). *rwork* is a workspace array, its dimension max(1, *lrwork*).
z, *work*	REAL for sstedc DOUBLE PRECISION for dstedc

COMPLEX for cstedc
DOUBLE COMPLEX for zstedc.
Arrays: *z*(*ldz*, *), *work*(*).
If *compz* = 'V', then, on entry, *z* must contain the orthogonal/unitary matrix used to reduce the original matrix to tridiagonal form.
The second dimension of *z* must be at least max(1, *n*).
work is a workspace array, its dimension max(1, *lwork*).

ldz — INTEGER. The first dimension of *z*. Constraints:
ldz ≥ 1 if *compz* = 'N';
ldz ≥ max(1, *n*) if *compz* = 'V' or 'I'.

lwork — INTEGER. The dimension of the array *work*.
If *compz* = 'N'or 'I', or *n* ≤ 1, *lwork* must be at least 1.
If *compz* = 'V' and *n* > 1, *lwork* must be at least *n***n*.
Note that for *compz* = 'V', and if *n* is less than or equal to the minimum divide size, usually 25, then *lwork* need only be 1.
If *lwork* = –1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla.
See Application Notes for the required value of *lwork*.

lrwork — INTEGER. The dimension of the array *rwork* (used for complex flavors only).
If *compz* = 'N', or *n* ≤ 1, *lrwork* must be at least 1.
If *compz* = 'V' and *n* > 1, *lrwork* must be at least (1+3**n*+2**n**lg(*n*)+3**n***n*), where lg(*n*) is the smallest integer *k* such that 2***k*≥*n*.
If *compz* = 'I' and *n* > 1, *lrwork* must be at least (1+4**n*+2**n***n*).
Note that for *compz* = 'V'or 'I', and if *n* is less than or equal to the minimum divide size, usually 25, then *lrwork* need only be max(1, 2*(*n*–1)).

If *lrwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See Application Notes for the required value of *lrwork*.

iwork INTEGER. Workspace array, its dimension max(1, *liwork*).

liwork INTEGER. The dimension of the array *iwork*.

If *compz* = 'N', or $n \leq 1$, *liwork* must be at least 1.
If *compz* = 'V' and $n > 1$, *liwork* must be at least (6+6**n*+5**n**lg(*n*), where lg(*n*) is the smallest integer *k* such that 2***k*≥*n*.
If *compz* = 'I' and $n > 1$, *liwork* must be at least (3+5**n*).
Note that for *compz* = 'V'or 'I', and if *n* is less than or equal to the minimum divide size, usually 25, then *liwork* need only be 1.
If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See Application Notes for the required value of *liwork*.

Output Parameters

d The *n* eigenvalues in ascending order, unless *info* ≠ 0. See also *info*.

e On exit, the array is overwritten; see *info*.

z If *info* = 0, then if *compz* = 'V', *z* contains the orthonormal eigenvectors of the original symmetric/Hermitian matrix, and if *compz* = 'I', *z* contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If *compz* = 'N', *z* is not referenced.

work(1) On exit, if *info* = 0, then *work(1)* returns the optimal *lwork*.

rwork(1)	On exit, if *info* = 0, then *rwork(1)* returns the optimal *lrwork* (for complex flavors only).
iwork(1)	On exit, if *info* = 0, then *iwork(1)* returns the optimal *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns *i*/(*n*+1) through mod(*i*, *n*+1).

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine stedc interface are the following:

d	Holds the vector of length (*n*).
e	Holds the vector of length (*n*-1).
z	Holds the matrix *Z* of size (*n*,*n*).
compz	If omitted, this argument is restored based on the presence of argument *z* as follows: *compz* = 'I', if *z* is present, *compz* = 'N', if *z* is omitted. If present, *compz* must be equal to 'I' or 'V' and the argument *z* must also be present. Note that there will be an error condition if *compz* is present and *z* omitted.

Note that two variants of Fortran 95 interface for stedc routine are needed because of an ambiguous choice between real and complex cases appear when *z* and *work* are omitted. Thus, the name rstedc is used in real cases (single or double precision), and the name stedc is used in complex cases (single or double precision).

Application Notes

The required size of workspace arrays must be as follows.

For sstedc/dstedc:

If *compz* = 'N' or *n* ≤ 1 then *lwork* must be at least 1.

If *compz* = 'V' and $n > 1$ then *lwork* must be at least $(1 + 3n + 2n \cdot \lg n + 3n^2)$, where lg($n$) = smallest integer k such that $2^k \geq n$.

If *compz* = 'I' and $n > 1$ then *lwork* must be at least $(1 + 4n + n^2)$.

If *compz* = 'N' or $n \leq 1$ then *liwork* must be at least 1.

If *compz* = 'V' and $n > 1$ then *liwork* must be at least $(6 + 6n + 5n \cdot \lg n)$.

If *compz* = 'I' and $n > 1$ then *liwork* must be at least $(3 + 5n)$.

For cstedc/zstedc:

If *compz* = 'N' or'I', or $n \leq 1$, *lwork* must be at least 1.

If *compz* = 'V' and $n > 1$, *lwork* must be at least n^2.

If *compz* = 'N' or $n \leq 1$, *lrwork* must be at least 1.

If *compz* = 'V' and $n > 1$, *lrwork* must be at least $(1 + 3n + 2n \cdot \lg n + 3n^2)$, where lg($n$) = smallest integer k such that $2^k \geq n$.

If *compz* = 'I' and $n > 1$, *lrwork* must be at least$(1 + 4n + 2n^2)$.

The required value of *liwork* for complex flavors is the same as for real flavors.

You may set *lwork* (or *liwork* or *lrwork*, if supplied) to -1. The routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*, *lrwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?stegr

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

Fortran 77:

```
call sstegr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work, lwork, iwork, liwork, info)
call dstegr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work, lwork, iwork, liwork, info)
call cstegr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work, lwork, iwork, liwork, info)
call zstegr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call rstegr(d, e, w [,z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol] [,info])
call stegr(d, e, w [,z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol] [,info])
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T. Any such unreduced matrix has a well defined set of pairwise different real eigenvalues, the corresponding real eigenvectors are pairwise orthogonal.

The spectrum may be computed either completely or partially by specifying either an interval $(vl, vu]$ or a range of indices $il:iu$ for the desired eigenvalues.

?sregr is a compatability wrapper around the improved ?stemr routine. See it descriprion for further details.

Note that the *abstol* parameter no longer provides any benefit and hence is no longer used.

See also auxiliary ?lasq2 ?lasq5, ?lasq6 , used by this routine.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *job* = 'N', then only eigenvalues are computed. If *job* = 'V', then eigenvalues and eigenvectors are computed.
range	CHARACTER*1. Must be 'A' or 'V' or 'I'. If *range* = 'A', the routine computes all eigenvalues. If *range* = 'V', the routine computes eigenvalues *lambda*(*i*) in the half-open interval: *vl*< *lambda*(*i*) ≤ *vu*. If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.
n	INTEGER. The order of the matrix *T* (*n* ≥ 0).
d, *e*, *work*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays: *d*(*) contains the diagonal elements of *T*. The dimension of *d* must be at least max(1, *n*). *e*(*) contains the subdiagonal elements of *T* in elements 1 to *n*-1; *e*(*n*) need not be set on input, but it is used as a workspace. The dimension of *e* must be at least max(1, *n*). *work*(*lwork*) is a workspace array.
vl, *vu*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: *vl*< *vu*. If *range* = 'A' or 'I', *vl* and *vu* are not referenced.
il, *iu*	INTEGER. If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: 1 ≤ *il* ≤ *iu* ≤ *n*, if *n* > 0. If *range* = 'A' or 'V', *il* and *iu* are not referenced.
abstol	REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Unused. Was the absolute error tolerance for the eigenvalues/eigenvectors in previous versions.

ldz — INTEGER. The leading dimension of the output array *z*.
Constraints:
ldz < 1 if *jobz* = 'N';
ldz < max(1, *n*) *jobz* = 'V', an.

lwork — INTEGER.
The dimension of the array *work*,

lwork≥max(1, 18**n*) if *jobz* = 'V', and

lwork≥max(1, 12**n*) if *jobz* = 'N'.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes below for details.

iwork — INTEGER.
Workspace array, DIMENSION (*liwork*).

liwork — INTEGER.

The dimension of the array *iwork*, *lwork* ≥ max(1, 10**n*)

if the eigenvectors are desired, and *lwork* ≥ max(1, 8**n*) if only the eigenvalues are to be computed..
If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by xerbla. See Application Notes below for details.

Output Parameters

d, *e* — On exit, *d* and *e* are overwritten.

m — INTEGER. The total number of eigenvalues found,

0 ≤ *m* ≤ *n*.

If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu*-*il*+1.

w — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.

	Array, DIMENSION at least max(1, *n*). The selected eigenvalues in ascending order, stored in *w*(1) to *w*(*m*).
z	REAL for sstegr DOUBLE PRECISION for dstegr COMPLEX for cstegr DOUBLE COMPLEX for zstegr. Array *z*(*ldz*, *), the second dimension of *z* must be at least max(1, *m*). If *jobz* = 'V', and if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). If *jobz* = 'N', then *z* is not referenced. Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z* ; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used. Supplying *n* columns is always safe.
isuppz	INTEGER. Array, DIMENSION at least (2*max(1, *m*)). The support of the eigenvectors in *z*, that is the indices indicating the nonzero elements in *z*. The *i*-th computed eigenvector is nonzero only in elements *isuppz*(2**i*–1) through *isuppz*(2**i*). This is relevant in the case when the matrix is split. *isuppz* is only accessed when *jobz* = 'V', and *n* > 0.
work(1)	On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.
iwork(1)	On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = –*i*, the *i*-th parameter had an illegal value. If *info* = 1*x*, internal error in ?larre occurred, If *info* = 2*x*, internal error in ?larrv occurred. Here the digit *x* = abs(*iinfo*) < 10, where *iinfo* is the non-zero error code returned by ?larre or ?larrv, respectively.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `stegr` interface are the following:

d	Holds the vector of length (*n*).
e	Holds the vector of length (*n*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*,*m*).
isuppz	Holds the vector of length (*2*m*).
vl	Default value for this argument is *vl* = – HUGE (*vl*) where HUGE(*a*) means the largest machine number of the same precision as argument *a*.
vu	Default value for this argument is *vu* = HUGE (*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this argument is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Note that two variants of Fortran 95 interface for `stegr` routine are needed because of an ambiguous choice between real and complex cases appear when *z* is omitted. Thus, the name `rstegr` is used in real cases (single or double precision), and the name `stegr` is used in complex cases (single or double precision).

Application Notes

Currently ?stegr is only set up to find *all* the *n* eigenvalues and eigenvectors of *T* in O(*n2*) time, that is, only *range* = 'A' is supported.

Currently the routine ?stein is called when an appropriate s_i cannot be chosen in step (c) above. ?stein invokes modified Gram-Schmidt when eigenvalues are close.

?stegr works only on machines which follow IEEE-754 floating-point standard in their handling of infinities and NaNs. Normal execution of ?stegr may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not conform to the IEEE-754 standard.

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If you choose the first option and set any of admissible *lwork* (or *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = -1 (*liwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?pteqr

Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric positive-definite tridiagonal matrix.

Syntax

Fortran 77:

```
call spteqr(compz, n, d, e, z, ldz, work, info)
call dpteqr(compz, n, d, e, z, ldz, work, info)
call cpteqr(compz, n, d, e, z, ldz, work, info)
call zpteqr(compz, n, d, e, z, ldz, work, info)
```

Fortran 95:

```
call rpteqr(d, e [,z] [,compz] [,info])
call pteqr(d, e [,z] [,compz] [,info])
```

Description

This routine computes all the eigenvalues and (optionally) all the eigenvectors of a real symmetric positive-definite tridiagonal matrix T. In other words, the routine can compute the spectral factorization: $T = Z\Lambda Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i; Z is an orthogonal matrix whose columns are eigenvectors. Thus,

$Tz_i = \lambda_i z_i$ for $i = 1, 2, \ldots, n$.

(The routine normalizes the eigenvectors so that $||z_i||_2 = 1$.)

You can also use the routine for computing the eigenvalues and eigenvectors of real symmetric (or complex Hermitian) positive-definite matrices A reduced to tridiagonal form T: $A = Q*T*Q^H$.

In this case, the spectral factorization is as follows: $A = Q*T*Q^H = (QZ)\Lambda(QZ)^H$. Before calling ?pteqr, you must reduce A to tridiagonal form and generate the explicit matrix Q by calling the following routines:

	for real matrices:	for complex matrices:
full storage	?sytrd, ?orgtr	?hetrd, ?ungtr
packed storage	?sptrd, ?opgtr	?hptrd, ?upgtr
band storage	?sbtrd *(vect*='V')	?hbtrd *(vect*='V')

The routine first factorizes T as $L*D*L^H$ where L is a unit lower bidiagonal matrix, and D is a diagonal matrix. Then it forms the bidiagonal matrix $B = L*D^{1/2}$ and calls ?bdsqr to compute the singular values of B, which are the same as the eigenvalues of T.

Input Parameters

compz — CHARACTER*1. Must be 'N' or 'I' or 'V'.
If *compz* = 'N', the routine computes eigenvalues only.
If *compz* = 'I', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix T.
If *compz* = 'V', the routine computes the eigenvalues and eigenvectors of A (and the array z must contain the matrix Q on entry).

n — INTEGER. The order of the matrix T ($n \geq 0$).

d, *e*, *work* — REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
Arrays:
d(*) contains the diagonal elements of T.
The dimension of d must be at least max(1, n).
e(*) contains the off-diagonal elements of T.
The dimension of e must be at least max(1, n-1).
work(*) is a workspace array.
The dimension of *work* must be:
at least 1 if *compz* = 'N';
at least max(1, 4*n-4) if *compz* = 'V' or 'I'.

z — REAL for spteqr
DOUBLE PRECISION for dpteqr
COMPLEX for cpteqr
DOUBLE COMPLEX for zpteqr.
Array, DIMENSION (*ldz*,*)

If *compz* = 'N' or 'I', *z* need not be set.
If *vect* = 'V', *z* must contains the *n*-by-*n* matrix *Q*.
The second dimension of *z* must be:
at least 1 if *compz* = 'N';
at least max(1, *n*) if *compz* = 'V' or 'I'.

ldz — INTEGER. The first dimension of *z*. Constraints:
ldz ≥ 1 if *compz* = 'N';
ldz ≥ max(1, *n*) if *compz* = 'V' or 'I'.

Output Parameters

d — The *n* eigenvalues in descending order, unless *info* > 0. See also *info*.

e — On exit, the array is overwritten.

z — If *info* = 0, contains the *n* orthonormal eigenvectors, stored by columns. (The *i*th column corresponds to the *i*th eigenvalue.)

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = *i*, the leading minor of order *i* (and hence *T* itself) is not positive-definite.
If *info* = *n* + *i*, the algorithm for computing singular values failed to converge; *i* off-diagonal elements have not converged to zero.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine pteqr interface are the following:

d — Holds the vector of length (*n*).

e — Holds the vector of length (*n*-1).

z — Holds the matrix *Z* of size (*n*,*n*).

compz	If omitted, this argument is restored based on the presence of argument *z* as follows: *compz* = 'I', if *z* is present, *compz* = 'N', if *z* is omitted. If present, *compz* must be equal to 'I' or 'V' and the argument *z* must also be present. Note that there will be an error condition if *compz* is present and *z* omitted.

Note that two variants of Fortran 95 interface for pteqr routine are needed because of an ambiguous choice between real and complex cases appear when *z* is omitted. Thus, the name rpteqr is used in real cases (single or double precision), and the name pteqr is used in complex cases (single or double precision).

Application Notes

If λ_i is an exact eigenvalue, and m_i is the corresponding computed value, then

$|\mu_i - \lambda_i| \le c(n)\varepsilon K\lambda_i$

where $c(n)$ is a modestly increasing function of n, ε is the machine precision, and $K = ||DTD||_2 ||(DTD)^{-1}||_2$, D is diagonal with $d_{ii} = t_{ii}^{-1/2}$.

If z_i is the corresponding exact eigenvector, and w_i is the corresponding computed vector, then the angle $\theta(z_i, w_i)$ between them is bounded as follows: $\theta(u_i, w_i) \le c(n)\varepsilon K / \min_{i \neq j}(|\lambda_i - \lambda_j|/|\lambda_i + \lambda_j|)$.

Here $\min_{i \neq j}(|\lambda_i - \lambda_j|/|\lambda_i + \lambda_j|)$ is the relative gap between λ_i and the other eigenvalues.

The total number of floating-point operations depends on how rapidly the algorithm converges.

Typically, it is about

$30n^2$ if *compz* = 'N';

$6n^3$ (for complex flavors, $12n^3$) if *compz* = 'V' or 'I'.

?stebz

Computes selected eigenvalues of a real symmetric tridiagonal matrix by bisection.

Syntax

Fortran 77:

```
call sstebz (range, order, n, vl, vu, il, iu, abstol, d, e, m, nsplit, w,
iblock, isplit, work, iwork, info)
```

```
call dstebz (range, order, n, vl, vu, il, iu, abstol, d, e, m, nsplit, w,
iblock, isplit, work, iwork, info)
```

Fortran 95:

```
call stebz(d, e, m, nsplit, w, iblock, isplit [, order] [,vl] [,vu] [,il]
[,iu] [,abstol] [,info])
```

Description

This routine computes some (or all) of the eigenvalues of a real symmetric tridiagonal matrix T by bisection. The routine searches for zero or negligible off-diagonal elements to see if T splits into block-diagonal form $T = \mathrm{diag}(T_1, T_2, \ldots)$. Then it performs bisection on each of the blocks T_i and returns the block index of each computed eigenvalue, so that a subsequent call to ?stein can also take advantage of the block structure.

See also ?laebz.

Input Parameters

range — CHARACTER*1. Must be 'A' or 'V' or 'I'.
If *range* = 'A', the routine computes all eigenvalues.
If *range* = 'V', the routine computes eigenvalues *lambda*(*i*) in the half-open interval: $vl < lambda(i) \leq vu$.
If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.

order — CHARACTER*1. Must be 'B' or 'E'.
If *order* = 'B', the eigenvalues are to be ordered from smallest to largest within each split-off block.

	If *order* = 'E', the eigenvalues for the entire matrix are to be ordered from smallest to largest.
n	INTEGER. The order of the matrix *T* (*n* ≥ 0).
vl, *vu*	REAL for sstebz DOUBLE PRECISION for dstebz. If *range* = 'V', the routine computes eigenvalues *lambda*(*i*) in the half-open interval: *vl* < *lambda*(*i*) ≤ *vu*. If *range* = 'A' or 'I', *vl* and *vu* are not referenced.
il, *iu*	INTEGER. Constraint: 1 ≤ *il* ≤ *iu* ≤ *n*. If *range* = 'I', the routine computes eigenvalues *lambda*(*i*) such that *il*≤ *i* ≤ *iu* (assuming that the eigenvalues *lambda*(*i*) are in ascending order). If *range* = 'A' or 'V', *il* and *iu* are not referenced.
abstol	REAL for sstebz DOUBLE PRECISION for dstebz. The absolute tolerance to which each eigenvalue is required. An eigenvalue (or cluster) is considered to have converged if it lies in an interval of width *abstol*. If *abstol* ≤ 0.0, then the tolerance is taken as eps*|*T*|, where eps is the machine precision, and |*T*| is the 1-norm of the matrix *T*.
d, *e*, *work*	REAL for sstebz DOUBLE PRECISION for dstebz. Arrays: *d*(*) contains the diagonal elements of *T*. The dimension of *d* must be at least max(1, *n*). *e*(*) contains the off-diagonal elements of *T*. The dimension of *e* must be at least max(1, *n*-1). *work*(*) is a workspace array. The dimension of *work* must be at least max(1, 4*n*).
iwork	INTEGER. Workspace. Array, DIMENSION at least max(1, 3*n*).

Output Parameters

m	INTEGER. The actual number of eigenvalues found.
nsplit	INTEGER. The number of diagonal blocks detected in *T*.
w	REAL for sstebz DOUBLE PRECISION for dstebz. Array, DIMENSION at least max(1, *n*). The computed eigenvalues, stored in *w*(1) to *w*(*m*).
iblock, *isplit*	INTEGER. Arrays, DIMENSION at least max(1, *n*). A positive value *iblock*(*i*) is the block number of the eigenvalue stored in *w*(*i*) (see also *info*). The leading *nsplit* elements of *isplit* contain points at which *T* splits into blocks T_i as follows: the block T_1 contains rows/columns 1 to *isplit*(1); the block T_2 contains rows/columns *isplit*(1)+1 to *isplit*(2), and so on.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = 1, for *range* = 'A' or 'V', the algorithm failed to compute some of the required eigenvalues to the desired accuracy; *iblock*(*i*)<0 indicates that the eigenvalue stored in *w*(*i*) failed to converge. If *info* = 2, for *range* = 'I', the algorithm failed to compute some of the required eigenvalues. Try calling the routine again with *range* = 'A'. If *info* = 3: for *range* = 'A' or 'V', same as *info* = 1; for *range* = 'I', same as *info* = 2. If *info* = 4, no eigenvalues have been computed. The floating-point arithmetic on the computer is not behaving as expected. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine stebz interface are the following:

d	Holds the vector of length (*n*).
e	Holds the vector of length (*n*–1).
w	Holds the vector of length (*n*).
iblock	Holds the vector of length (*n*).
isplit	Holds the vector of length (*n*).
order	Must be 'B' or 'E'. The default value is 'B'.
vl	Default value for this argument is *vl* = - HUGE (*vl*) where HUGE(*a*) means the largest machine number of the same precision as argument *a*.
vu	Default value for this argument is *vu* = HUGE (*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this argument is *abstol* = 0.0_WP.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

The eigenvalues of *T* are computed to high relative accuracy which means that if they vary widely in magnitude, then any small eigenvalues will be computed more accurately than, for example, with the standard *QR* method. However, the reduction to tridiagonal form (prior to calling the routine) may exclude the possibility of obtaining high relative accuracy in the small eigenvalues of the original matrix if its eigenvalues vary widely in magnitude.

?stein

Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix.

Syntax

Fortran 77:

```
call sstein(n, d, e, m, w, iblock, isplit, z, ldz, work, iwork, ifailv, info)
call dstein(n, d, e, m, w, iblock, isplit, z, ldz, work, iwork, ifailv, info)
call cstein(n, d, e, m, w, iblock, isplit, z, ldz, work, iwork, ifailv, info)
call zstein(n, d, e, m, w, iblock, isplit, z, ldz, work, iwork, ifailv, info)
```

Fortran 95:

```
call stein(d, e, w, iblock, isplit, z [,ifailv] [,info])
```

Description

This routine computes the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, by inverse iteration. It is designed to be used in particular after the specified eigenvalues have been computed by ?stebz with *order* = 'B', but may also be used when the eigenvalues have been computed by other routines.

If you use this routine after ?stebz, it can take advantage of the block structure by performing inverse iteration on each block T_i separately, which is more efficient than using the whole matrix T.

If T has been formed by reduction of a full symmetric or Hermitian matrix A to tridiagonal form, you can transform eigenvectors of T to eigenvectors of A by calling ?ormtr or ?opmtr (for real flavors) or by calling ?unmtr or ?upmtr (for complex flavors).

Input Parameters

n	INTEGER. The order of the matrix T ($n \geq 0$).
m	INTEGER. The number of eigenvectors to be returned.
d, *e*, *w*	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays:

	d(*) contains the diagonal elements of *T*. The dimension of *d* must be at least max(1, *n*). *e*(*) contains the sub-diagonal elements of *T* stored in elements 1 to *n*-1 The dimension of *e* must be at least max(1, *n*-1). *w*(*) contains the eigenvalues of *T*, stored in *w*(1) to *w*(*m*) (as returned by ?stebz). Eigenvalues of T_1 must be supplied first, in non-decreasing order; then those of T_2, again in non-decreasing order, and so on. Constraint: if *iblock*(*i*) = *iblock*(*i*+1), *w*(*i*) ≤ *w*(*i*+1). The dimension of *w* must be at least max(1, *n*).
iblock, *isplit*	INTEGER. Arrays, DIMENSION at least max(1, *n*). The arrays *iblock* and *isplit*, as returned by ?stebz with *order* = 'B'. If you did not call ?stebz with *order* = 'B', set all elements of *iblock* to 1, and *isplit*(1) to *n*.)
ldz	INTEGER. The first dimension of the output array *z*; *ldz* ≥ max(1, *n*).
work	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Workspace array, DIMENSION at least max(1, 5*n*).
iwork	INTEGER. Workspace array, DIMENSION at least max(1, *n*).

Output Parameters

z	REAL for sstein DOUBLE PRECISION for dstein COMPLEX for cstein DOUBLE COMPLEX for zstein. Array, DIMENSION (*ldz*, *). If *info* = 0, *z* contains the *m* orthonormal eigenvectors, stored by columns. (The *i*th column corresponds to the *i*-th specified eigenvalue.)
ifailv	INTEGER. Array, DIMENSION at least max(1, *m*).

If *info* = *i* > 0, the first *i* elements of *ifailv* contain the indices of any eigenvectors that failed to converge.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = *i*, then *i* eigenvectors (as indicated by the parameter *ifailv*) each failed to converge in 5 iterations. The current iterates are stored in the corresponding columns of the array *z*.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `stein` interface are the following:

d — Holds the vector of length (*n*).

e — Holds the vector of length (*n*).

w — Holds the vector of length (*n*).

iblock — Holds the vector of length (*n*).

isplit — Holds the vector of length (*n*).

z — Holds the matrix *Z* of size (*n*,*m*).

ifailv — Holds the vector of length (*m*).

Application Notes

Each computed eigenvector z_i is an exact eigenvector of a matrix $T + E_i$, where $||E_i||_2 = O(\varepsilon)\ ||T||_2$. However, a set of eigenvectors computed by this routine may not be orthogonal to so high a degree of accuracy as those computed by `?steqr`.

?disna

Computes the reciprocal condition numbers for the eigenvectors of a symmetric/ Hermitian matrix or for the left or right singular vectors of a general matrix.

Syntax

Fortran 77:

```
call sdisna(job, m, n, d, sep, info)
call ddisna(job, m, n, d, sep, info)
```

Fortran 95:

```
call disna(d, sep [,job] [,minmn] [,info])
```

Description

This routine computes the reciprocal condition numbers for the eigenvectors of a real symmetric or complex Hermitian matrix or for the left or right singular vectors of a general *m*-by-*n* matrix.

The reciprocal condition number is the 'gap' between the corresponding eigenvalue or singular value and the nearest other one.

The bound on the error, measured by angle in radians, in the *i*-th computed vector is given by

```
slamch('E')*(anorm/sep(i))
```

where *anorm* = $||A||_2$ = max(|*d*(j)|). *sep*(*i*) is not allowed to be smaller than `slamch('E')*anorm` in order to limit the size of the error bound.

`?disna` may also be used to compute error bounds for eigenvectors of the generalized symmetric definite eigenproblem.

Input Parameters

job — `CHARACTER*1`. Must be `'E'`,`'L'`, or `'R'`. Specifies for which problem the reciprocal condition numbers should be computed:
job = `'E'`: for the eigenvectors of a symmetric/Hermitian matrix;
job = `'L'`: for the left singular vectors of a general matrix;

	job = 'R': for the right singular vectors of a general matrix.
m	INTEGER. The number of rows of the matrix ($m \geq 0$).
n	INTEGER. If *job* = 'L', or 'R', the number of columns of the matrix ($n \geq 0$). Ignored if *job* = 'E'.
d	REAL for sdisna DOUBLE PRECISION for ddisna. Array, dimension at least max(1,*m*) if *job* = 'E', and at least max(1, min(*m*,*n*)) if *job* = 'L' or 'R'. This array must contain the eigenvalues (if *job* = 'E') or singular values (if *job* = 'L' or 'R') of the matrix, in either increasing or decreasing order. If singular values, they must be non-negative.

Output Parameters

sep	REAL for sdisna DOUBLE PRECISION for ddisna. Array, dimension at least max(1,*m*) if *job* = 'E', and at least max(1, min(*m*,*n*)) if *job* = 'L' or 'R'. The reciprocal condition numbers of the vectors.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine disna interface are the following:

d	Holds the vector of length min(*m*,*n*).
sep	Holds the vector of length min(*m*,*n*).
job	Must be 'E', 'L', or 'R'. The default value is 'E'.
minmn	Indicates which of the values *m* or *n* is smaller. Must be either 'M' or 'N', the default is 'M'.

If *job* = 'E', this argument is superfluous, If *job* = 'L' or 'R', this argument is used by the routine.

Generalized Symmetric-Definite Eigenvalue Problems

Generalized symmetric-definite eigenvalue problems are as follows: find the eigenvalues λ and the corresponding eigenvectors z that satisfy one of these equations:

$Az = \lambda Bz$, $ABz = \lambda z$, or $BAz = \lambda z$,

where A is an n-by-n symmetric or Hermitian matrix, and B is an n-by-n symmetric positive-definite or Hermitian positive-definite matrix.

In these problems, there exist n real eigenvectors corresponding to real eigenvalues (even for complex Hermitian matrices A and B).

Routines described in this section allow you to reduce the above generalized problems to standard symmetric eigenvalue problem $Cy = \lambda y$, which you can solve by calling LAPACK routines described earlier in this chapter (see Symmetric Eigenvalue Problems).

Different routines allow the matrices to be stored either conventionally or in packed storage. Prior to reduction, the positive-definite matrix B must first be factorized using either ?potrf or ?pptrf.

The reduction routine for the banded matrices A and B uses a split Cholesky factorization for which a specific routine ?pbstf is provided. This refinement halves the amount of work required to form matrix C.

Table 4-4 lists LAPACK routines (Fortran-77 interface) that can be used to solve generalized symmetric-definite eigenvalue problems. Respective routine names in Fortran-95 interface are without the first symbol (see Routine Naming Conventions).

Table 4-4 Computational Routines for Reducing Generalized Eigenproblems to Standard Problems

Matrix type	Reduce to standard problems (full storage)	Reduce to standard problems (packed storage)	Reduce to standard problems (band matrices)	Factorize band matrix
real symmetric matrices	?sygst	?spgst	?sbgst	?pbstf

Matrix type	Reduce to standard problems (full storage)	Reduce to standard problems (packed storage)	Reduce to standard problems (band matrices)	Factorize band matrix
complex Hermitian matrices	?hegst	?hpgst	?hbgst	?pbstf

?sygst

Reduces a real symmetric-definite generalized eigenvalue problem to the standard form.

Syntax

Fortran 77:

```
call ssygst(itype, uplo, n, a, lda, b, ldb, info)
call dsygst(itype, uplo, n, a, lda, b, ldb, info)
```

Fortran 95:

```
call sygst(a, b [,itype] [,uplo] [,info])
```

Description

This routine reduces real symmetric-definite generalized eigenproblems

$Az = \lambda Bz$, $ABz = \lambda_z$, or $BAz = \lambda z$

to the standard form $Cy = \lambda_y$. Here A is a real symmetric matrix, and B is a real symmetric positive-definite matrix. Before calling this routine, call ?potrf to compute the Cholesky factorization: $B = U^T U$ or $B = LL^T$.

Input Parameters

itype INTEGER. Must be 1 or 2 or 3.
If *itype* = 1, the generalized eigenproblem is *A*z = lambda*B*z*
for *uplo* = 'U': C = inv(U^T)*A*inv(U), z = inv(U)*y;
for *uplo* = 'L': C = inv(L)*A*inv(L^T), z = inv(L^T)*y.

If *itype* = 2, the generalized eigenproblem is *A*B*z = lambda*z*
for *uplo* = 'U': *C* = *U*A*U*^{T}, *z* = inv(*U*)**y*;
for *uplo* = 'L': *C* = *L*^{T}**A*L*, *z* = inv(*L*^{T})**y*.
If *itype* = 3, the generalized eigenproblem is *B*A*z = lambda*z*
for *uplo* = 'U': *C* = *U*A*U*^{T}, *z* = *U*^{T}**y*;
for *uplo* = 'L': *C* = *L*^{T}**A*L*, *z* = *L*y*.

uplo — CHARACTER*1. Must be 'U' or 'L'.
If *uplo* = 'U', the array *a* stores the upper triangle of *A*; you must supply *B* in the factored form *B* = *U*^{T}**U*.
If *uplo* = 'L', the array *a* stores the lower triangle of *A*; you must supply *B* in the factored form *B* = *L*L*^{T}.

n — INTEGER. The order of the matrices *A* and *B* (*n* ≥ 0).

a, *b* — REAL for ssygst
DOUBLE PRECISION for dsygst.
Arrays:
a(*lda*,*) contains the upper or lower triangle of *A*.
The second dimension of *a* must be at least max(1, *n*).
b(*ldb*,*) contains the Cholesky-factored matrix *B*:
B = *U*^{T}**U* or *B* = *L*L*^{T} (as returned by ?potrf).
The second dimension of *b* must be at least max(1, *n*).

lda — INTEGER. The first dimension of *a*; at least max(1, *n*).

ldb — INTEGER. The first dimension of *b*; at least max(1, *n*).

Output Parameters

a — The upper or lower triangle of *A* is overwritten by the upper or lower triangle of *C*, as specified by the arguments *itype* and *uplo*.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `sygst` interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
b	Holds the matrix *B* of size (*n*,*n*).
itype	Must be 1, 2, or 3. The default value is 1.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

Forming the reduced matrix *C* is a stable procedure. However, it involves implicit multiplication by B^{-1} (if *itype* = 1) or *B* (if *itype* = 2 or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is *n3*.

?hegst

Reduces a complex Hermitian-definite generalized eigenvalue problem to the standard form.

Syntax

Fortran 77:

```
call chegst(itype, uplo, n, a, lda, b, ldb, info)
call zhegst(itype, uplo, n, a, lda, b, ldb, info)
```

Fortran 95:

```
call hegst(a, b [,itype] [,uplo] [,info])
```

Description

This routine reduces complex Hermitian-definite generalized eigenvalue problems

$Az = \lambda Bz$, $ABz = \lambda z$, or $BAz = \lambda z$

to the standard form $Cy = \lambda y$. Here the matrix A is complex Hermitian, and B is complex Hermitian positive-definite. Before calling this routine, you must call ?potrf to compute the Cholesky factorization: $B = U^H*U$ or $B = L*L^H$.

Input Parameters

itype	INTEGER. Must be 1 or 2 or 3. If *itype* = 1, the generalized eigenproblem is *A*z = lambda*B*z* for *uplo* = 'U': $C = \text{inv}(U^H)*A*\text{inv}(U)$, $z = \text{inv}(U)*y$; for *uplo* = 'L': $C = \text{inv}(L)*A*\text{inv}(L^H)$, $z = \text{inv}(L^H)*y$. If *itype* = 2, the generalized eigenproblem is *A*B*z = lambda*z* for *uplo* = 'U': $C = U*A*U^H$, $z = \text{inv}(U)*y$; for *uplo* = 'L': $C = L^H*A*L$, $z = \text{inv}(L^H)*y$. If *itype* = 3, the generalized eigenproblem is *B*A*z = lambda*z* for *uplo* = 'U': $C = U*A*U^H$, $z = U^H*y$; for *uplo* = 'L': $C = L^H*A*L$, $z = L*y$.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', the array *a* stores the upper triangle of *A*; you must supply *B* in the factored form $B = U^H*U$. If *uplo* = 'L', the array *a* stores the lower triangle of *A*; you must supply *B* in the factored form $B = L*L^H$.
n	INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).
a, *b*	COMPLEX for chegstDOUBLE COMPLEX for zhegst. Arrays: *a*(*lda*,*) contains the upper or lower triangle of *A*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the Cholesky-factored matrix *B*: $B = U^H*U$ or $B = L*L^H$ (as returned by ?potrf). The second dimension of *b* must be at least max(1, *n*).
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).
ldb	INTEGER. The first dimension of *b*; at least max(1, *n*).

Output Parameters

a	The upper or lower triangle of *A* is overwritten by the upper or lower triangle of *C*, as specified by the arguments *itype* and *uplo*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hegst interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
b	Holds the matrix *B* of size (*n*,*n*).
itype	Must be 1, 2, or 3. The default value is 1.
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

Forming the reduced matrix *C* is a stable procedure. However, it involves implicit multiplication by inv(*B*) (if *itype* = 1) or *B* (if *itype* = 2 or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3.

?spgst

Reduces a real symmetric-definite generalized eigenvalue problem to the standard form using packed storage.

Syntax

Fortran 77:

```
call sspgst(itype, uplo, n, ap, bp, info)
call dspgst(itype, uplo, n, ap, bp, info)
```

Fortran 95:

```
call spgst(a, b [,itype] [,uplo] [,info])
```

Description

This routine reduces real symmetric-definite generalized eigenproblems

$Az = \lambda Bz$, $ABz = \lambda z$, or $BAz = \lambda z$

to the standard form $Cy = \lambda y$, using packed matrix storage. Here A is a real symmetric matrix, and B is a real symmetric positive-definite matrix. Before calling this routine, call ?pptrf to compute the Cholesky factorization: $B = U^T*U$ or $B = L*L^T$.

Input Parameters

itype — INTEGER. Must be 1 or 2 or 3.
If *itype* = 1, the generalized eigenproblem is *A*z = lambda*B*z*
for *uplo* = 'U': $C = \text{inv}(U^T)*A*\text{inv}(U)$, $z = \text{inv}(U)*y$;
for *uplo* = 'L': $C = \text{inv}(L)*A*\text{inv}(L^T)$, $z = \text{inv}(L^T)*y$.
If *itype* = 2, the generalized eigenproblem is *A*B*z = lambda*z*
for *uplo* = 'U': $C = U*A*U^T$, $z = \text{inv}(U)*y$;
for *uplo* = 'L': $C = L^T*A*L$, $z = \text{inv}(L^T)*y$.
If *itype* = 3, the generalized eigenproblem is *B*A*z = lambda*z*
for *uplo* = 'U': $C = U*A*U^T$, $z = U^T*y$;
for *uplo* = 'L': $C = L^T*A*L$, $z = L*y$.

uplo — CHARACTER*1. Must be 'U' or 'L'.
If *uplo* = 'U', *ap* stores the packed upper triangle of *A*; you must supply *B* in the factored form $B = U^T*U$.
If *uplo* = 'L', *ap* stores the packed lower triangle of *A*; you must supply *B* in the factored form $B = L*L^T$.

n — INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

ap, *bp* — REAL for sspgst
DOUBLE PRECISION for dspgst.
Arrays:
ap(*) contains the packed upper or lower triangle of *A*.
The dimension of *ap* must be at least max(1, *n**(*n*+1)/2).

b — p(*) contains the packed Cholesky factor of *B* (as returned by ?pptrf with the same *uplo* value).
The dimension of *bp* must be at least max(1, *n**(*n*+1)/2).

Output Parameters

ap — The upper or lower triangle of *A* is overwritten by the upper or lower triangle of *C*, as specified by the arguments *itype* and *uplo*.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine spgst interface are the following:

a — Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n*(n+1)/2*).

b — Stands for argument *bp* in Fortan 77 interface. Holds the array *B* of size (*n*(n+1)/2*).

itype — Must be 1, 2, or 3. The default value is 1.

uplo — Must be 'U' or 'L'. The default value is 'U'.

Application Notes

Forming the reduced matrix C is a stable procedure. However, it involves implicit multiplication by B^{-1} (if $itype = 1$) or B (if $itype = 2$ or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if B is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3.

?hpgst

Reduces a complex Hermitian-definite generalized eigenvalue problem to the standard form using packed storage.

Syntax

Fortran 77:

```
call chpgst(itype, uplo, n, ap, bp, info)
call zhpgst(itype, uplo, n, ap, bp, info)
```

Fortran 95:

```
call hpgst(a, b [,itype] [,uplo] [,info])
```

Description

This routine reduces real symmetric-definite generalized eigenproblems

$Az = \lambda Bz$, $ABz = \lambda z$, or $BAz = \lambda z$

to the standard form $Cy = \lambda y$, using packed matrix storage. Here A is a real symmetric matrix, and B is a real symmetric positive-definite matrix. Before calling this routine, you must call ?pptrf to compute the Cholesky factorization: $B = U^H*U$ or $B = L*L^H$.

Input Parameters

itype	INTEGER. Must be 1 or 2 or 3. If $itype = 1$, the generalized eigenproblem is $A*z = lambda*B*z$ for $uplo$ = 'U': $C = \text{inv}(U^H)*A*\text{inv}(U)$, $z = \text{inv}(U)*y$; for $uplo$ = 'L': $C = \text{inv}(L)*A*\text{inv}(L^H)$, $z = \text{inv}(L^H)*y$.

If *itype* = 2, the generalized eigenproblem is *A*B*z = lambda*z*
for *uplo* = 'U': *C* = *U*A*U^H^*, *z* = inv(*U*)*y*;
for *uplo* = 'L': *C* = *L^H^*A*L*, *z* = inv(*L^H^*)*y*.
If *itype* = 3, the generalized eigenproblem is *B*A*z = lambda*z*
for *uplo* = 'U': *C* = *U*A*U^H^*, *z* = *U^H^*y*;
for *uplo* = 'L': *C* = *L^H^*A*L*, *z* = *L*y*.

uplo — CHARACTER*1. Must be 'U' or 'L'.
If *uplo* = 'U', *ap* stores the packed upper triangle of *A*; you must supply *B* in the factored form *B* = *U^H^*U*.
If *uplo* = 'L', *ap* stores the packed lower triangle of *A*; you must supply *B* in the factored form B = *L*L^H^*.

n — INTEGER. The order of the matrices *A* and *B* (*n* ≥ 0).

ap, *bp* — COMPLEX for chpgstDOUBLE COMPLEX for zhpgst.
Arrays:
ap(*) contains the packed upper or lower triangle of *A*.
The dimension of *a* must be at least max(1, *n**(*n*+1)/2).
bp(*) contains the packed Cholesky factor of *B* (as returned by ?pptrf with the same *uplo* value).
The dimension of *b* must be at least max(1, *n**(*n*+1)/2).

Output Parameters

ap — The upper or lower triangle of *A* is overwritten by the upper or lower triangle of *C*, as specified by the arguments *itype* and *uplo*.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hpgst interface are the following:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n*(n+1)/2*).
b	Stands for argument *bp* in Fortan 77 interface. Holds the array *B* of size (*n*(n+1)/2*).
itype	Must be 1, 2, or 3. The default value is 1.
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

Forming the reduced matrix *C* is a stable procedure. However, it involves implicit multiplication by inv(*B*) (if *itype* = 1) or *B* (if *itype* = 2 or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3.

?sbgst

Reduces a real symmetric-definite generalized eigenproblem for banded matrices to the standard form using the factorization performed by ?pbstf*.*

Syntax

Fortran 77:

```
call ssbgst(vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx, work, info)
call dsbgst(vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx, work, info)
```

Fortran 95:

```
call sbgst(a, b [,x] [,uplo] [,info])
```

Description

To reduce the real symmetric-definite generalized eigenproblem $Az = \lambda Bz$ to the standard form $Cy = \lambda y$, where *A*, *B* and *C* are banded, this routine must be preceded by a call to spbstf/dpbstf, which computes the split Cholesky factorization of the positive-definite matrix *B*: $B = S^T*S$. The split Cholesky factorization, compared with the ordinary Cholesky factorization, allows the work to be approximately halved.

This routine overwrites *A* with $C = X^T*A*X$, where $X = \mathrm{inv}(S)*Q$ and *Q* is an orthogonal matrix chosen (implicitly) to preserve the bandwidth of *A*. The routine also has an option to allow the accumulation of *X*, and then, if *z* is an eigenvector of *C*, *Xz* is an eigenvector of the original system.

Input Parameters

vect	CHARACTER*1. Must be 'N' or 'V'. If *vect* = 'N', then matrix *X* is not returned; If *vect* = 'V', then matrix *X* is returned.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *ab* stores the upper triangular part of *A*. If *uplo* = 'L', *ab* stores the lower triangular part of *A*.
n	INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).
ka	INTEGER. The number of super- or sub-diagonals in *A* ($ka \geq 0$).
kb	INTEGER. The number of super- or sub-diagonals in *B* ($ka \geq kb \geq 0$).
ab, *bb*, *work*	REAL for ssbgst DOUBLE PRECISION for dsbgst *ab* (*ldab*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A* (as specified by *uplo*) in band storage format. The second dimension of the array *ab* must be at least max(1, *n*). *bb* (*ldbb*,*) is an array containing the banded split Cholesky factor of *B* as specified by *uplo*, *n* and *kb* and returned by spbstf/dpbstf. The second dimension of the array *bb* must be at least max(1, *n*). *work(*)* is a workspace array, dimension at least max(1, 2**n*)
ldab	INTEGER. The first dimension of the array *ab*; must be at least *ka*+1.
ldbb	INTEGER. The first dimension of the array *bb*; must be at least *kb*+1.

ldx	The first dimension of the output array x. Constraints: if *vect* = 'N', then *ldx* ≥ 1; if *vect* = 'V', then *ldx* ≥ max(1, *n*).

Output Parameters

ab	On exit, this array is overwritten by the upper or lower triangle of *C* as specified by *uplo*.
x	REAL for ssbgst DOUBLE PRECISION for dsbgst Array. If *vect* = 'V', then *x* (*ldx*,*) contains the *n*-by-*n* matrix *X* = inv(*S*)**Q*. If *vect* = 'N', then *x* is not referenced. The second dimension of x must be: at least max(1, *n*), if *vect* = 'V'; at least 1, if *vect* = 'N'.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine sbgst interface are the following:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*ka+1*,*n*).
b	Stands for argument *bb* in Fortan 77 interface. Holds the array *B* of size (*kb+1*,*n*).
x	Holds the matrix *X* of size (*n*,*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
vect	Restored based on the presence of the argument *x* as follows: *vect* = 'V', if *x* is present, *vect* = 'N', if *x* is omitted.

Application Notes

Forming the reduced matrix C involves implicit multiplication by B^{-1}. When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if B is ill-conditioned with respect to inversion. The total number of floating-point operations is approximately $6n^2*kb$, when `vect = 'N'`. Additional $(3/2)n^3*(kb/ka)$ operations are required when `vect = 'V'`. All these estimates assume that both ka and kb are much less than n.

?hbgst

Reduces a complex Hermitian-definite generalized eigenproblem for banded matrices to the standard form using the factorization performed by `?pbstf`.

Syntax

Fortran 77:

```
call chbgst(vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx, work, rwork, info)
```

```
call zhbgst(vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx, work, rwork, info)
```

Fortran 95:

```
call hbgst(a, b [,x] [,uplo] [,info])
```

Description

To reduce the complex Hermitian-definite generalized eigenproblem $Az = \lambda Bz$ to the standard form $Cy = \lambda y$, where A, B and C are banded, this routine must be preceded by a call to cpbstf/zpbstf, which computes the split Cholesky factorization of the positive-definite matrix B: $B = S^H*S$. The split Cholesky factorization, compared with the ordinary Cholesky factorization, allows the work to be approximately halved.

This routine overwrites A with $C = X^H*A*X$, where $X = \text{inv}(S)*Q$, and Q is a unitary matrix chosen (implicitly) to preserve the bandwidth of A. The routine also has an option to allow the accumulation of X, and then, if z is an eigenvector of C, Xz is an eigenvector of the original system.

Input Parameters

vect	CHARACTER*1. Must be 'N' or 'V'. If *vect* = 'N', then matrix *X* is not returned; If *vect* = 'V', then matrix *X* is returned.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *ab* stores the upper triangular part of *A*. If *uplo* = 'L', *ab* stores the lower triangular part of *A*.
n	INTEGER. The order of the matrices *A* and *B* (*n* ≥ 0).
ka	INTEGER. The number of super- or sub-diagonals in *A* (*ka* ≥ 0).
kb	INTEGER. The number of super- or sub-diagonals in *B* (*ka* ≥ *kb* ≥ 0).
ab, *bb*, *work*	COMPLEX for chbgstDOUBLE COMPLEX for zhbgst *ab* (*ldab*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by *uplo*) in band storage format. The second dimension of the array *ab* must be at least max(1, *n*). *bb* (*ldbb*,*) is an array containing the banded split Cholesky factor of *B* as specified by *uplo*, *n* and *kb* and returned by cpbstf/zpbstf. The second dimension of the array *bb* must be at least max(1, *n*). *work(*)* is a workspace array, dimension at least max(1, *n*)
ldab	INTEGER. The first dimension of the array *ab*; must be at least *ka*+1.
ldbb	INTEGER. The first dimension of the array *bb*; must be at least *kb*+1.
ldx	The first dimension of the output array x. Constraints: if *vect* = 'N', then *ldx* ≥ 1; if *vect* = 'V', then *ldx* ≥ max(1, *n*).
rwork	REAL for chbgst DOUBLE PRECISION for zhbgst

Workspace array, `dimension` at least max(1, *n*)

Output Parameters

ab	On exit, this array is overwritten by the upper or lower triangle of *C* as specified by *uplo*.

x	`COMPLEX` for `chbgst`
	`DOUBLE COMPLEX` for `zhbgst`
	Array.
	If *vect* = `'V'`, then *x* (*ldx*,*) contains the *n*-by-*n* matrix *X* = `inv(`*S*`)*`*Q*.
	If *vect* = `'N'`, then *x* is not referenced.
	The second dimension of x must be:
	at least max(1, *n*), if *vect* = `'V'`;
	at least 1, if *vect* = `'N'`.

info	`INTEGER`. If *info* = `0`, the execution is successful. If *info* = `-`*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `hbgst` interface are the following:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*ka+1*,*n*).
b	Stands for argument *bb* in Fortan 77 interface. Holds the array *B* of size (*kb+1*,*n*).
x	Holds the matrix *X* of size (*n*,*n*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
vect	Restored based on the presence of the argument *x* as follows: *vect* = `'V'`, if *x* is present, *vect* = `'N'`, if *x* is omitted.

Application Notes

Forming the reduced matrix *C* involves implicit multiplication by inv(*B*). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion. The total number of floating-point operations is approximately $20n^2*kb$, when *vect* = 'N'. Additional $5n^3*(kb/ka)$ operations are required when *vect* = 'V'. All these estimates assume that both *ka* and *kb* are much less than *n*.

?pbstf

Computes a split Cholesky factorization of a real symmetric or complex Hermitian positive-definite banded matrix used in ?sbgst/?hbgst *.*

Syntax

Fortran 77:

```
call spbstf(uplo, n, kb, bb, ldbb, info)
call dpbstf(uplo, n, kb, bb, ldbb, info)
call cpbstf(uplo, n, kb, bb, ldbb, info)
call zpbstf(uplo, n, kb, bb, ldbb, info)
```

Fortran 95:

```
call pbstf(b [, uplo] [,info])
```

Description

This routine computes a split Cholesky factorization of a real symmetric or complex Hermitian positive-definite band matrix *B*. It is to be used in conjunction with ?sbgst/?hbgst.

The factorization has the form $B = S^T*S$ (or $B = S^H*S$ for complex flavors), where *S* is a band matrix of the same bandwidth as *B* and the following structure: S is upper triangular in the first (*n*+*kb*)/2 rows and lower triangular in the remaining rows.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
If *uplo* = 'U', *bb* stores the upper triangular part of *B*.
If *uplo* = 'L', *bb* stores the lower triangular part of *B*.

n	INTEGER. The order of the matrix *B* (*n* ≥ 0).
kb	INTEGER. The number of super- or sub-diagonals in *B* (*kb* ≥ 0).
bb	REAL for spbstf DOUBLE PRECISION for dpbstf COMPLEX for cpbstf DOUBLE COMPLEX for zpbstf. *bb* (*ldbb*,*) is an array containing either upper or lower triangular part of the matrix *B* (as specified by *uplo*) in band storage format. The second dimension of the array *bb* must be at least max(1, *n*).
ldbb	INTEGER. The first dimension of *bb*; must be at least *kb*+1.

Output Parameters

bb	On exit, this array is overwritten by the elements of the split Cholesky factor *S*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *i*, then the factorization could not be completed, because the updated element b_{ii} would be the square root of a negative number; hence the matrix *B* is not positive-definite. If *info* = -*i*, the *i*th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine pbstf interface are the following:

b	Stands for argument *bb* in Fortan 77 interface. Holds the array *B* of size (*kb+1*,*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed factor S is the exact factor of a perturbed matrix $B + E$, where

$$|E| \le c(kb + 1)\varepsilon |S^H||S|, \quad |e_{ij}| \le c(kb + 1)\varepsilon \sqrt{b_{ii} b_{jj}}$$

$c(n)$ is a modest linear function of n, and ε is the machine precision.

The total number of floating-point operations for real flavors is approximately $n(kb+1)^2$. The number of operations for complex flavors is 4 times greater. All these estimates assume that kb is much less than n.

After calling this routine, you can call ?sbgst/?hbgst to solve the generalized eigenproblem $Az = \lambda Bz$, where A and B are banded and B is positive-definite.

Nonsymmetric Eigenvalue Problems

This section describes LAPACK routines for solving nonsymmetric eigenvalue problems, computing the Schur factorization of general matrices, as well as performing a number of related computational tasks.

A nonsymmetric eigenvalue problem is as follows: given a nonsymmetric (or non-Hermitian) matrix A, find the eigenvalues λ and the corresponding eigenvectors z that satisfy the equation

$Az = \lambda z$ (right eigenvectors z)

or the equation

$z^H A = \lambda z^H$ (left eigenvectors z).

Nonsymmetric eigenvalue problems have the following properties:

- The number of eigenvectors may be less than the matrix order (but is not less than the number of distinct eigenvalues of A).
- Eigenvalues may be complex even for a real matrix A.
- If a real nonsymmetric matrix has a complex eigenvalue $a+bi$ corresponding to an eigenvector z, then $a-bi$ is also an eigenvalue. The eigenvalue $a-bi$ corresponds to the eigenvector whose elements are complex conjugate to the elements of z.

To solve a nonsymmetric eigenvalue problem with LAPACK, you usually need to reduce the matrix to the upper Hessenberg form and then solve the eigenvalue problem with the Hessenberg matrix obtained. Table 4-5 lists LAPACK routines (Fortran-77 interface) for reducing the matrix to the upper Hessenberg form by an orthogonal (or unitary) similarity transformation $A = QHQ^H$ as well as routines for solving eigenvalue problems with Hessenberg matrices, forming the Schur factorization of such matrices and computing the corresponding condition numbers. Respective routine names in Fortran-95 interface are without the first symbol (see Routine Naming Conventions).

Decision tree in Figure 4-4 helps you choose the right routine or sequence of routines for an eigenvalue problem with a real nonsymmetric matrix. If you need to solve an eigenvalue problem with a complex non-Hermitian matrix, use the decision tree shown in Figure 4-5 .

Table 4-5 Computational Routines for Solving Nonsymmetric Eigenvalue Problems

Operation performed	Routines for real matrices	Routines for complex matrices
Reduce to Hessenberg form $A = QHQ^H$	?gehrd,	?gehrd
Generate the matrix Q	?orghr	?unghr
Apply the matrix Q	?ormhr	?unmhr
Balance matrix	?gebal	?gebal
Transform eigenvectors of balanced matrix to those of the original matrix	?gebak	?gebak
Find eigenvalues and Schur factorization (QR algorithm)	?hseqr	?hseqr
Find eigenvectors from Hessenberg form (inverse iteration)	?hsein	?hsein
Find eigenvectors from Schur factorization	?trevc	?trevc
Estimate sensitivities of eigenvalues and eigenvectors	?trsna	?trsna

Operation performed	Routines for real matrices	Routines for complex matrices
Reorder Schur factorization	?trexc	?trexc
Reorder Schur factorization, find the invariant subspace and estimate sensitivities	?trsen	?trsen
Solves Sylvester's equation.	?trsyl	?trsyl

Figure 4-4 Decision Tree: Real Nonsymmetric Eigenvalue Problems

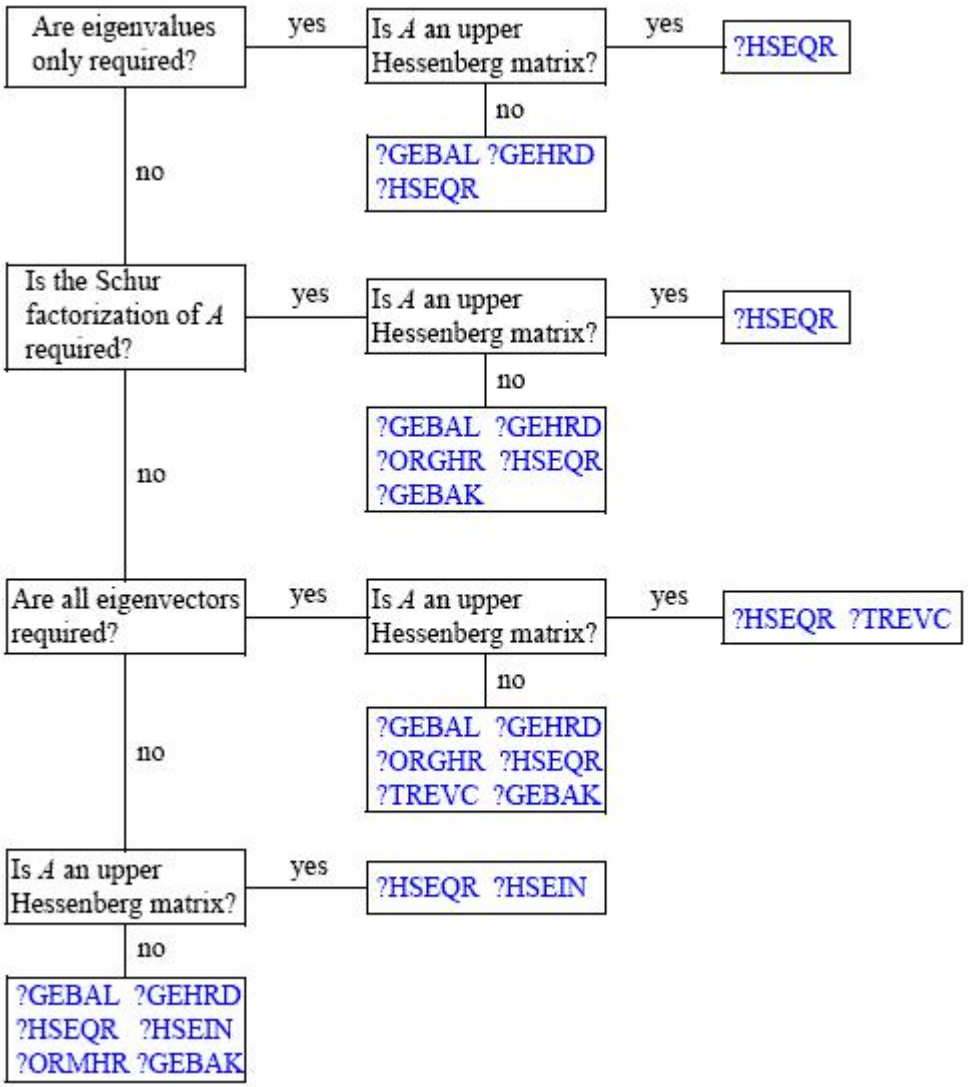
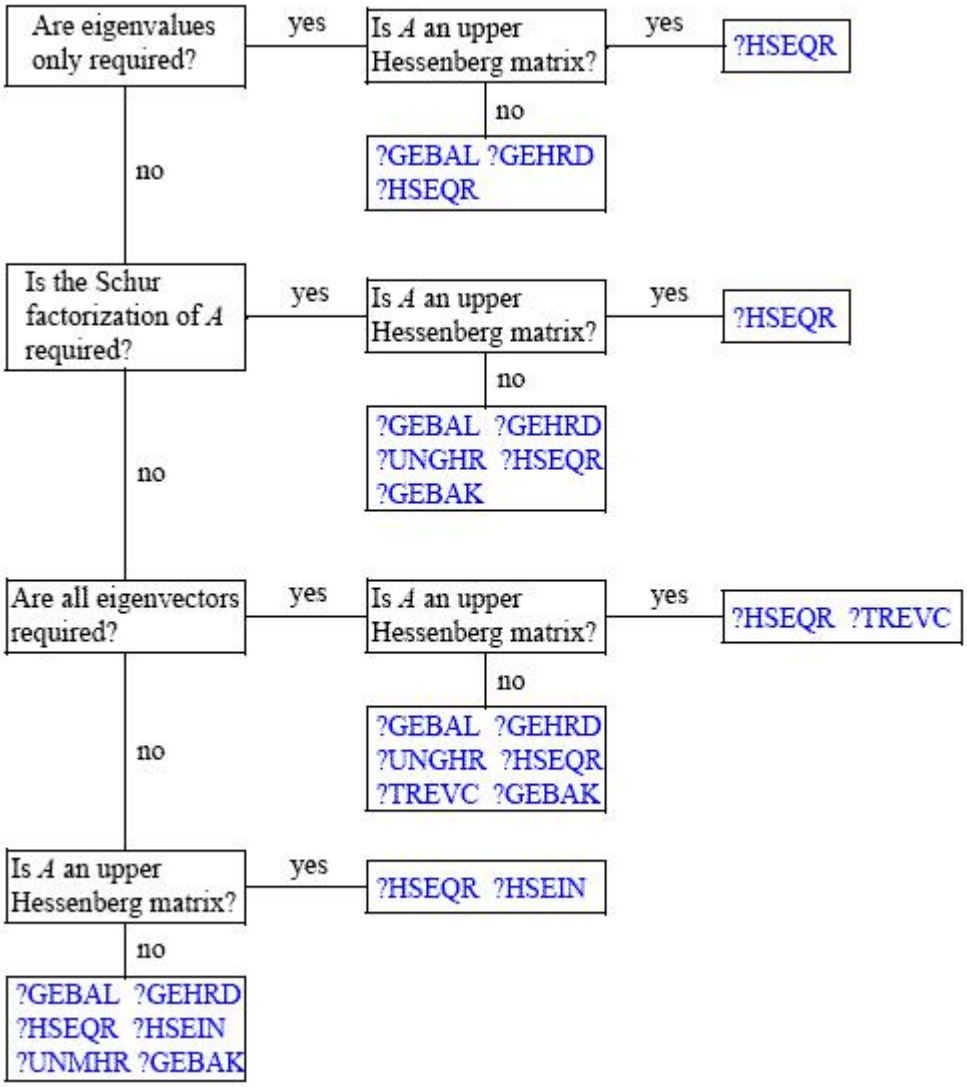


Figure 4-5 Decision Tree: Complex Non-Hermitian Eigenvalue Problems



?gehrd

Reduces a general matrix to upper Hessenberg form.

Syntax

Fortran 77:

```
call sgehrd(n, ilo, ihi, a, lda, tau, work, lwork, info)
call dgehrd(n, ilo, ihi, a, lda, tau, work, lwork, info)
call cgehrd(n, ilo, ihi, a, lda, tau, work, lwork, info)
call zgehrd(n, ilo, ihi, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call gehrd(a [, tau] [,ilo] [,ihi] [,info])
```

Description

The routine reduces a general matrix A to upper Hessenberg form H by an orthogonal or unitary similarity transformation $A = Q*H*Q^H$. Here H has real subdiagonal elements.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of elementary reflectors. Routines are provided to work with Q in this representation.

Input Parameters

n	INTEGER. The order of the matrix A ($n \geq 0$).
ilo, *ihi*	INTEGER. If A has been output by ?gebal, then *ilo* and *ihi* must contain the values returned by that routine. Otherwise $ilo = 1$ and $ihi = n$. (If $n > 0$, then $1 \leq ilo \leq ihi \leq n$; if $n = 0$, $ilo = 1$ and $ihi = 0$.)
a, *work*	REAL for sgehrd DOUBLE PRECISION for dgehrd COMPLEX for cgehrd DOUBLE COMPLEX for zgehrd. Arrays: *a* (*lda*,*) contains the matrix A. The second dimension of *a* must be at least max(1, *n*).

	work (*lwork*) is a workspace array.
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).
lwork	INTEGER. The size of the *work* array; at least max(1, *n*). If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the upper Hessenberg matrix *H* and details of the matrix *Q*. The subdiagonal elements of *H* are real.
tau	REAL for sgehrd DOUBLE PRECISION for dgehrd COMPLEX for cgehrd DOUBLE COMPLEX for zgehrd. Array, DIMENSION at least max (1, *n*-1). Contains additional information on the matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = −*i* , the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gehrd interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
tau	Holds the vector of length (*n*–1).
ilo	Default value for this argument is *ilo* = 1.
ihi	Default value for this argument is *ihi* = *n*.

Application Notes

For better performance, try using *lwork* = *n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed Hessenberg matrix H is exactly similar to a nearby matrix $A + E$, where $||E||_2 < c(n)\varepsilon||A||_2$, $c(n)$ is a modestly increasing function of n, and ε is the machine precision.

The approximate number of floating-point operations for real flavors is $(2/3)(ihi - ilo)^2(2ihi + 2ilo + 3n)$; for complex flavors it is 4 times greater.

?orghr

Generates the real orthogonal matrix Q determined by ?gehrd.

Syntax

Fortran 77:

```
call sorghr(n, ilo, ihi, a, lda, tau, work, lwork, info)
call dorghr(n, ilo, ihi, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call orghr(a, tau [,ilo] [,ihi] [,info])
```

Description

This routine explicitly generates the orthogonal matrix *Q* that has been determined by a preceding call to sgehrd/dgehrd. (The routine ?gehrd reduces a real general matrix *A* to upper Hessenberg form *H* by an orthogonal similarity transformation, $A = Q*H*Q^T$, and represents the matrix *Q* as a product of *ihi-ilo* elementary reflectors. Here *ilo* and *ihi* are values determined by sgebal/dgebal when balancing the matrix;if the matrix has not been balanced, *ilo* = 1 and *ihi* = *n*.)

The matrix *Q* generated by ?orghr has the structure:

$$Q = \begin{bmatrix} I & 0 & 0 \\ 0 & Q_{22} & 0 \\ 0 & 0 & I \end{bmatrix}$$

where Q_{22} occupies rows and columns *ilo* to *ihi*.

Input Parameters

n	INTEGER. The order of the matrix *Q* (*n* ≥ 0).
ilo, *ihi*	INTEGER. These must be the same parameters *ilo* and *ihi*, respectively, as supplied to ?gehrd. (If *n* > 0, then 1 ≤ *ilo* ≤ *ihi* ≤ *n*; if *n* = *0*, *ilo* = 1 and *ihi* = 0.)
a, *tau*, *work*	REAL for sorghr DOUBLE PRECISION for dorghr Arrays: *a*(*lda*,*) contains details of the vectors which define the elementary reflectors, as returned by ?gehrd. The second dimension of *a* must be at least max(1, *n*). *tau*(*) contains further details of the elementary reflectors, as returned by ?gehrd. The dimension of *tau* must be at least max (1, *n*-1). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).
lwork	INTEGER. The size of the *work* array;

lwork ≥ max(1, *ihi*-*ilo*).
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.
See Application Notes for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the *n*-by-*n* orthogonal matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine orghr interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
tau	Holds the vector of length (*n*-1).
ilo	Default value for this argument is *ilo* = 1.
ihi	Default value for this argument is *ihi* = *n*.

Application Notes

For better performance, try using *lwork* =(*ihi*-*ilo*)***blocksize* where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from the exact result by a matrix *E* such that $||E||_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of floating-point operations is $(4/3)(ihi-ilo)^3$.

The complex counterpart of this routine is ?unghr.

?ormhr

Multiplies an arbitrary real matrix C by the real orthogonal matrix Q determined by ?gehrd.

Syntax

Fortran 77:

```
call sormhr(side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc, work, lwork, info)
call dormhr(side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call ormhr(a, tau, c [,ilo] [,ihi] [,side] [,trans] [,info])
```

Description

This routine multiplies a matrix *C* by the orthogonal matrix *Q* that has been determined by a preceding call to sgehrd/dgehrd. (The routine ?gehrd reduces a real general matrix *A* to upper Hessenberg form *H* by an orthogonal similarity transformation, $A = Q*H*Q^T$, and represents

the matrix *Q* as a product of *ihi*-*ilo* elementary reflectors. Here *ilo* and *ihi* are values determined by sgebal/dgebal when balancing the matrix;if the matrix has not been balanced, *ilo* = 1 and *ihi* = *n*.)

With ?ormhr, you can form one of the matrix products *Q***C*, Q^T**C*, *C***Q*, or *C**Q^T, overwriting the result on *C* (which may be any real rectangular matrix).

A common application of ?ormhr is to transform a matrix *V* of eigenvectors of *H* to the matrix *QV* of eigenvectors of *A*.

Input Parameters

side	CHARACTER*1. Must be 'L' or 'R'. If *side* = 'L', then the routine forms *Q***C* or Q^T**C*. If *side* = 'R', then the routine forms *C***Q* or *C**Q^T.
trans	CHARACTER*1. Must be 'N' or 'T'. If *trans* = 'N' , then *Q* is applied to *C*. If *trans* = 'T', then Q^T is applied to *C*.
m	INTEGER. The number of rows in *C* (*m* ≥ 0).
n	INTEGER. The number of columns in *C* (*n* ≥ 0).
ilo, *ihi*	INTEGER. These must be the same parameters *ilo* and *ihi*, respectively, as supplied to ?gehrd. If *m* > 0 and *side* = 'L', then 1 ≤ *ilo* ≤ *ihi* ≤ *m*. If *m* = 0 and *side* = 'L' , then *ilo* = 1 and *ihi* = 0. If *n* > 0 and *side* = 'R', then 1 ≤ *ilo* ≤ *ihi* ≤ *n*. If *n* = 0 and *side* = 'R' , then *ilo* = 1 and *ihi* = 0.
a, *tau*, *c*, *work*	REAL for sormhr DOUBLE PRECISION for dormhr Arrays: *a*(*lda*,*) contains details of the vectors which define the elementary reflectors, as returned by ?gehrd. The second dimension of *a* must be at least max(1, *m*) if *side* = 'L' and at least max(1, *n*) if *side* = 'R'. *tau*(*) contains further details of the elementary reflectors, as returned by ?gehrd . The dimension of *tau* must be at least max (1, *m*-1) if *side* = 'L' and at least max (1, *n*-1) if *side* = 'R'.

	c(*ldc*,*) contains the *m* by *n* matrix *C*. The second dimension of *c* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*) if *side* = 'L' and at least max (1, *n*) if *side* = 'R'.
ldc	INTEGER. The first dimension of *c*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array. If *side* = 'L', *lwork* ≥ max(1, *n*). If *side* = 'R', *lwork* ≥ max(1, *m*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

c	*C* is overwritten by $Q*C$ or Q^T*C or $C*Q^T$ or $C*Q$ as specified by *side* and *trans*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ormhr interface are the following:

a	Holds the matrix *A* of size (*r*,*r*). *r* = *m* if *side* = 'L'. *r* = *n* if *side* = 'R'.
tau	Holds the vector of length (*r*-1).

c	Holds the matrix *C* of size (*m*,*n*).
ilo	Default value for this argument is *ilo* = 1.
ihi	Default value for this argument is *ihi* = *n*.
side	Must be 'L' or 'R'. The default value is 'L'.
trans	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, *lwork* should be at least *n***blocksize* if *side* = 'L' and at least *m***blocksize* if *side* = 'R', where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from the exact result by a matrix *E* such that $||E||_2 = O(\varepsilon)||C||_2$, where ε is the machine precision.

The approximate number of floating-point operations is

$2n(ihi-ilo)^2$ if *side* = 'L';

$2m(ihi-ilo)^2$ if *side* = 'R'.

The complex counterpart of this routine is ?unmhr.

?unghr

Generates the complex unitary matrix Q determined by ?gehrd.

Syntax

Fortran 77:

```
call cunghr(n, ilo, ihi, a, lda, tau, work, lwork, info)
call zunghr(n, ilo, ihi, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call unghr(a, tau [,ilo] [,ihi] [,info])
```

Description

This routine is intended to be used following a call to cgehrd/zgehrd, which reduces a complex matrix A to upper Hessenberg form H by a unitary similarity transformation: $A = Q*H*Q^H$. ?gehrd represents the matrix Q as a product of $ihi-ilo$ elementary reflectors. Here ilo and ihi are values determined by cgebal/zgebal when balancing the matrix; if the matrix has not been balanced, $ilo = 1$ and $ihi = n$.

Use the routine ?unghr to generate Q explicitly as a square matrix. The matrix Q has the structure:

$$Q = \begin{bmatrix} I & 0 & 0 \\ 0 & Q_{22} & 0 \\ 0 & 0 & I \end{bmatrix}$$

where Q_{22} occupies rows and columns ilo to ihi.

Input Parameters

n	INTEGER. The order of the matrix Q ($n \geq 0$).

ilo, *ihi*	INTEGER. These must be the same parameters *ilo* and *ihi*, respectively, as supplied to ?gehrd . (If $n > 0$, then $1 \le ilo \le ihi \le n$. If $n = 0$, then $ilo = 1$ and $ihi = 0$.)
a, *tau*, *work*	COMPLEX for cunghr DOUBLE COMPLEX for zunghr. Arrays: *a*(*lda*,*) contains details of the vectors which define the elementary reflectors, as returned by ?gehrd. The second dimension of *a* must be at least max(1, *n*). *tau*(*) contains further details of the elementary reflectors, as returned by ?gehrd . The dimension of *tau* must be at least max (1, *n*-1). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).
lwork	INTEGER. The size of the *work* array; *lwork* ≥ max(1, *ihi*-*ilo*). If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	Overwritten by the *n*-by-*n* unitary matrix *Q*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = −*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `unghr` interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
tau	Holds the vector of length (*n*–1).
ilo	Default value for this argument is *ilo* = 1.
ihi	Default value for this argument is *ihi* = *n*.

Application Notes

For better performance, try using *lwork* = (*ihi*-*ilo*)**blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix Q differs from the exact result by a matrix E such that $||E||_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of real floating-point operations is $(16/3)(ihi-ilo)^3$.

The real counterpart of this routine is `?orghr`.

?unmhr

Multiplies an arbitrary complex matrix C by the complex unitary matrix Q determined by ?gehrd.

Syntax

Fortran 77:

```
call cunmhr(side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc, work, lwork, info)
call zunmhr(side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call unmhr(a, tau, c [,ilo] [,ihi] [,side] [,trans] [,info])
```

Description

This routine multiplies a matrix C by the unitary matrix Q that has been determined by a preceding call to cgehrd/zgehrd. (The routine ?gehrd reduces a real general matrix A to upper Hessenberg form H by an orthogonal similarity transformation, $A = Q*H*Q^H$, and represents the matrix Q as a product of ihi-iloelementary reflectors. Here ilo and ihi are values determined by cgebal/zgebal when balancing the matrix; if the matrix has not been balanced, $ilo = 1$ and $ihi = n$.)

With ?unmhr, you can form one of the matrix products $Q*C$, Q^H*C, $C*Q$, or $C*Q^H$, overwriting the result on C (which may be any complex rectangular matrix). A common application of this routine is to transform a matrix V of eigenvectors of H to the matrix QV of eigenvectors of A.

Input Parameters

side	CHARACTER*1. Must be 'L' or 'R'. If *side* = 'L', then the routine forms $Q*C$ or Q^H*C. If *side* = 'R', then the routine forms $C*Q$ or $C*Q^H$.
trans	CHARACTER*1. Must be 'N' or 'C'. If *trans* = 'N' , then Q is applied to C. If *trans* = 'T', then Q^H is applied to C.
m	INTEGER. The number of rows in C ($m \geq 0$).

n	INTEGER. The number of columns in *C* (*n*≥ 0).
ilo, *ihi*	INTEGER. These must be the same parameters *ilo* and *ihi*, respectively, as supplied to ?gehrd . If *m* > 0 and *side* = 'L', then 1 ≤*ilo*≤*ihi*≤*m*. If *m* = 0 and *side* = 'L' , then *ilo* = 1 and *ihi* = 0. If *n* > 0 and *side* = 'R', then 1 ≤*ilo*≤*ihi*≤*n*. If *n* = 0 and *side* = 'R' , then *ilo* =1 and *ihi* = 0.
a, *tau*, *c*, *work*	COMPLEX for cunmhr DOUBLE COMPLEX for zunmhr. Arrays: *a* (*lda*,*) contains details of the vectors which define the elementary reflectors, as returned by ?gehrd. The second dimension of *a* must be at least max(1, *m*) if *side* = 'L' and at least max(1, *n*) if *side* = 'R'. *tau*(*) contains further details of the elementary reflectors, as returned by ?gehrd. The dimension of *tau* must be at least max (1, *m*-1) if *side* = 'L' and at least max (1, *n*-1) if *side* = 'R'. *c* (*ldc*,*) contains the *m*-by-*n* matrix *C*. The second dimension of *c* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*) if *side* = 'L' and at least max (1, *n*) if *side* = 'R'.
ldc	INTEGER. The first dimension of *c*; at least max(1, *m*).
lwork	INTEGER. The size of the *work* array. If *side* = 'L', *lwork*≥ max(1,*n*). If *side* = 'R', *lwork*≥ max(1,*m*). If *lwork* = –1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

c	*C* is overwritten by *Q*C*, or Q^H**C*, or *C*Q*^{H}, or *C*Q* as specified by *side* and *trans*.
work(1)	If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine unmhr interface are the following:

a	Holds the matrix *A* of size (*r*,*r*). *r* = *m* if *side* = 'L'. *r* = *n* if *side* = 'R'.
tau	Holds the vector of length (*r*-1).
c	Holds the matrix *C* of size (*m*,*n*).
ilo	Default value for this argument is *ilo* = 1.
ihi	Default value for this argument is *ihi* = *n*.
side	Must be 'L' or 'R'. The default value is 'L'.
trans	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, *lwork* should be at least *n***blocksize* if *side* = 'L' and at least *m***blocksize* if *side* = 'R', where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from the exact result by a matrix *E* such that $||E||_2 = O(\varepsilon)||C||_2$, where ε is the machine precision.

The approximate number of floating-point operations is

$8n(ihi-ilo)^2$ if *side* = 'L';

$8m(ihi-ilo)^2$ if *side* = 'R'.

The real counterpart of this routine is ?ormhr.

?gebal

Balances a general matrix to improve the accuracy of computed eigenvalues and eigenvectors.

Syntax

Fortran 77:

```
call sgebal(job, n, a, lda, ilo, ihi, scale, info)
call dgebal(job, n, a, lda, ilo, ihi, scale, info)
call cgebal(job, n, a, lda, ilo, ihi, scale, info)
call zgebal(job, n, a, lda, ilo, ihi, scale, info)
```

Fortran 95:

```
call gebal(a [, scale] [,ilo] [,ihi] [,job] [,info])
```

Description

This routine *balances* a matrix *A* by performing either or both of the following two similarity transformations:

(1) The routine first attempts to permute *A* to block upper triangular form:

$$PAP^T = A' = \begin{bmatrix} A'_{11} & A'_{12} & A'_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & 0 & A'_{33} \end{bmatrix}$$

where *P* is a permutation matrix, and A'_{11} and A'_{33} are upper triangular. The diagonal elements of A'_{11} and A'_{33} are eigenvalues of *A*. The rest of the eigenvalues of *A* are the eigenvalues of the central diagonal block A'_{22}, in rows and columns *ilo* to *ihi*. Subsequent operations to compute the eigenvalues of *A* (or its Schur factorization) need only be applied to these rows and columns; this can save a significant amount of work if *ilo* > 1 and *ihi* < *n*.

If no suitable permutation exists (as is often the case), the routine sets *ilo* = 1 and *ihi* = *n*, and A'_{22} is the whole of *A*.

(2) The routine applies a diagonal similarity transformation to *A'*, to make the rows and columns of A'_{22} as close in norm as possible:

$$A'' = DA'D^{-1} = \begin{bmatrix} I & 0 & 0 \\ 0 & D_{22} & 0 \\ 0 & 0 & I \end{bmatrix} \times \begin{bmatrix} A'_{11} & A'_{12} & A'_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & 0 & A'_{33} \end{bmatrix} \times \begin{bmatrix} I & 0 & 0 \\ 0 & D_{22}^{-1} & 0 \\ 0 & 0 & I \end{bmatrix}$$

This scaling can reduce the norm of the matrix (that is, $||A'_2{}'_2|| < ||A'_{22}||$), and hence reduce the effect of rounding errors on the accuracy of computed eigenvalues and eigenvectors.

Input Parameters

job CHARACTER*1. Must be 'N' or 'P' or 'S' or 'B'.

	If *job* = 'N', then *A* is neither permuted nor scaled (but *ilo*, *ihi*, and *scale* get their values). If *job* = 'P', then *A* is permuted but not scaled. If *job* = 'S', then *A* is scaled but not permuted. If *job* = 'B', then *A* is both scaled and permuted.
n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
a	REAL for sgebal DOUBLE PRECISION for dgebal COMPLEX for cgebal DOUBLE COMPLEX for zgebal. Arrays: *a* (*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*). *a* is not referenced if *job* = 'N'.
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).

Output Parameters

a	Overwritten by the balanced matrix (*a* is not referenced if *job* = 'N').
ilo, *ihi*	INTEGER. The values *ilo* and *ihi* such that on exit *a*(*i*,*j*) is zero if *i* > *j* and 1 ≤ *j* < *ilo* or *ihi* < *i* ≤ *n*. If *job* = 'N' or 'S', then *ilo* = 1 and *ihi* = *n*.
scale	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors Array, DIMENSION at least max(1, *n*). Contains details of the permutations and scaling factors. More precisely, if p_j is the index of the row and column interchanged with row and column *j*, and d_j is the scaling factor used to balance row and column *j*, then *scale*(*j*) = p_j for *j* = 1, 2,..., *ilo*-1, *ihi*+1,..., *n*; *scale*(*j*) = d_j for *j* = *ilo*, *ilo* + 1,..., *ihi*. The order in which the interchanges are made is *n* to *ihi*+1, then 1 to *ilo*-1.
info	INTEGER.

If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gebal interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
scale	Holds the vector of length (*n*).
ilo	Default value for this argument is *ilo* = 1.
ihi	Default value for this argument is *ihi* = *n*.
job	Must be 'B', 'S', 'P', or 'N'. The default value is 'B'.

Application Notes

The errors are negligible, compared with those in subsequent computations.

If the matrix *A* is balanced by this routine, then any eigenvectors computed subsequently are eigenvectors of the matrix *A*'' and hence you must call ?gebak to transform them back to eigenvectors of *A*.

If the Schur vectors of *A* are required, do not call this routine with *job* = 'S' or 'B', because then the balancing transformation is not orthogonal (not unitary for complex flavors).

If you call this routine with *job* = 'P', then any Schur vectors computed subsequently are Schur vectors of the matrix *A*'', and you need to call ?gebak (with *side* = 'R') to transform them back to Schur vectors of *A*.

The total number of floating-point operations is proportional to n^2.

?gebak

Transforms eigenvectors of a balanced matrix to those of the original nonsymmetric matrix.

Syntax

Fortran 77:

```
call sgebak(job, side, n, ilo, ihi, scale, m, v, ldv, info)
call dgebak(job, side, n, ilo, ihi, scale, m, v, ldv, info)
call cgebak(job, side, n, ilo, ihi, scale, m, v, ldv, info)
call zgebak(job, side, n, ilo, ihi, scale, m, v, ldv, info)
```

Fortran 95:

```
call gebak(v, scale [,ilo] [,ihi] [,job] [,side] [,info])
```

Description

This routine is intended to be used after a matrix A has been balanced by a call to ?gebal, and eigenvectors of the balanced matrix A''_{22} have subsequently been computed. For a description of balancing, see ?gebal. The balanced matrix A'' is obtained as $A'' = D*P*A*P^T*\mathrm{inv}(D)$, where P is a permutation matrix and D is a diagonal scaling matrix. This routine transforms the eigenvectors as follows:

if x is a right eigenvector of A'', then $P^T*\mathrm{inv}(D)*x$ is a right eigenvector of A; if x is a left eigenvector of A'', then P^T*D*y is a left eigenvector of A.

Input Parameters

job	CHARACTER*1. Must be 'N' or 'P' or 'S' or 'B'. The same parameter *job* as supplied to ?gebal.
side	CHARACTER*1. Must be 'L' or 'R'. If *side* = 'L' , then left eigenvectors are transformed. If *side* = 'R', then right eigenvectors are transformed.
n	INTEGER. The number of rows of the matrix of eigenvectors ($n \geq 0$).
ilo, *ihi*	INTEGER. The values *ilo* and *ihi*, as returned by ?gebal. (If $n > 0$, then $1 \leq ilo \leq ihi \leq n$;

	if $n = 0$, then $ilo = 1$ and $ihi = 0$.)
scale	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors Array, DIMENSION at least max(1, n). Contains details of the permutations and/or the scaling factors used to balance the original general matrix, as returned by ?gebal.
m	INTEGER. The number of columns of the matrix of eigenvectors ($m \geq 0$).
v	REAL for sgebak DOUBLE PRECISION for dgebak COMPLEX for cgebak DOUBLE COMPLEX for zgebak. Arrays: v (ldv,*) contains the matrix of left or right eigenvectors to be transformed. The second dimension of v must be at least max(1, m).
ldv	INTEGER. The first dimension of v; at least max(1, n).

Output Parameters

v	Overwritten by the transformed eigenvectors.
info	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gebak interface are the following:

v	Holds the matrix V of size (n,m).
scale	Holds the vector of length (n).
ilo	Default value for this argument is $ilo = 1$.
ihi	Default value for this argument is $ihi = n$.

job	Must be 'B', 'S', 'P', or 'N'. The default value is 'B'.
side	Must be 'L' or 'R'. The default value is 'L'.

Application Notes

The errors in this routine are negligible.

The approximate number of floating-point operations is approximately proportional to *m*n*.

?hseqr

Computes all eigenvalues and (optionally) the Schur factorization of a matrix reduced to Hessenberg form.

Syntax

Fortran 77:

```
call shseqr(job, compz, n, ilo, ihi, h, ldh, wr, wi, z, ldz, work, lwork, info)
call dhseqr(job, compz, n, ilo, ihi, h, ldh, wr, wi, z, ldz, work, lwork, info)
call chseqr(job, compz, n, ilo, ihi, h, ldh, w, z, ldz, work, lwork, info)
call zhseqr(job, compz, n, ilo, ihi, h, ldh, w, z, ldz, work, lwork, info)
```

Fortran 95:

```
call hseqr(h, wr, wi [,ilo] [,ihi] [,z] [,job] [,compz] [,info])
call hseqr(h, w [,ilo] [,ihi] [,z] [,job] [,compz] [,info])
```

Description

This routine computes all the eigenvalues, and optionally the Schur factorization, of an upper Hessenberg matrix H: $H = ZTZ^H$, where T is an upper triangular (or, for real flavors, quasi-triangular) matrix (the Schur form of H), and Z is the unitary or orthogonal matrix whose columns are the Schur vectors z_i.

You can also use this routine to compute the Schur factorization of a general matrix A which has been reduced to upper Hessenberg form H:

$A = QHQ^H$, where Q is unitary (orthogonal for real flavors);

$A = (QZ)H(QZ)^H$.

In this case, after reducing A to Hessenberg form by ?gehrd, call ?orghr to form Q explicitly and then pass Q to ?hseqr with $compz$ = 'V'.

You can also call ?gebal to balance the original matrix before reducing it to Hessenberg form by ?hseqr, so that the Hessenberg matrix H will have the structure:

$$\begin{bmatrix} H_{11} & H_{12} & H_{13} \\ 0 & H_{22} & H_{23} \\ 0 & 0 & H_{33} \end{bmatrix}$$

where H_{11} and H_{33} are upper triangular.

If so, only the central diagonal block H_{22} (in rows and columns ilo to ihi) needs to be further reduced to Schur form (the blocks H_{12} and H_{23} are also affected). Therefore the values of ilo and ihi can be supplied to ?hseqr directly. Also, after calling this routine you must call ?gebak to permute the Schur vectors of the balanced matrix to those of the original matrix.

If ?gebal has not been called, however, then ilo must be set to 1 and ihi to n. Note that if the Schur factorization of A is required, ?gebal must not be called with job = 'S' or 'B', because the balancing transformation is not unitary (for real flavors, it is not orthogonal).

?hseqr uses a multishift form of the upper Hessenberg QR algorithm. The Schur vectors are normalized so that $||z_i||_2 = 1$, but are determined only to within a complex factor of absolute value 1 (for the real flavors, to within a factor ±1).

Input Parameters

job	CHARACTER*1. Must be 'E' or 'S'. If job = 'E', then eigenvalues only are required. If job = 'S', then the Schur form T is required.
$compz$	CHARACTER*1. Must be 'N' or 'I' or 'V'. If $compz$ = 'N', then no Schur vectors are computed (and the array z is not referenced). If $compz$ = 'I', then the Schur vectors of H are computed (and the array z is initialized by the routine).

	If *compz* = 'V', then the Schur vectors of *A* are computed (and the array *z* must contain the matrix *Q* on entry).
n	INTEGER. The order of the matrix *H* (*n* ≥ 0).
ilo, *ihi*	INTEGER. If *A* has been balanced by ?gebal, then *ilo* and *ihi* must contain the values returned by ?gebal. Otherwise, *ilo* must be set to 1 and *ihi* to *n*.
h, *z*, *work*	REAL for shseqr DOUBLE PRECISION for dhseqr COMPLEX for chseqr DOUBLE COMPLEX for zhseqr. Arrays: *h*(*ldh*,*) The *n*-by-*n* upper Hessenberg matrix *H*. The second dimension of *h* must be at least max(1, *n*). *z*(*ldz*,*) If *compz* = 'V', then *z* must contain the matrix *Q* from the reduction to Hessenberg form. If *compz* = 'I', then *z* need not be set. If *compz* = 'N', then *z* is not referenced. The second dimension of *z* must be at least max(1, *n*) if *compz* = 'V' or 'I'; at least 1 if *compz* = 'N'. *work*(*lwork*) is a workspace array. The dimension of *work* must be at least max (1, *n*).
ldh	INTEGER. The first dimension of *h*; at least max(1, *n*).
ldz	INTEGER. The first dimension of *z*; If *compz* = 'N', then *ldz* ≥ 1. If *compz* = 'V' or 'I', then *ldz* ≥ max(1, *n*).
lwork	INTEGER. The dimension of the array *work*. *lwork* ≥ max(1, *n*) is sufficient, but *lwork* typically as large as 6**n* may be required for optimal performance. A workspace query to determine the optimal workspace size is recommended.

If *lwork* = -1, then a workspace query is assumed; the routine only estimates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for details.

Output Parameters

w
COMPLEX for chseqr
DOUBLE COMPLEX for zhseqr.
Array, DIMENSION at least max (1, *n*). Contains the computed eigenvalues, unless *info*>0. The eigenvalues are stored in the same order as on the diagonal of the Schur form *T* (if computed).

wr, *wi*
REAL for shseqr
DOUBLE PRECISION for dhseqr
Arrays, DIMENSION at least max (1, *n*) each.
Contain the real and imaginary parts, respectively, of the computed eigenvalues, unless *info* > 0. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first. The eigenvalues are stored in the same order as on the diagonal of the Schur form *T* (if computed).

h
If *info* = 0 and *job* = 'S', *h* contains the upper triangular matrix *T* from the Schur decomposition (the Schur form).
If *info* = 0 and *job* = 'E', the contents of *h* are unspecified on exit. (The output value of *h* when *info* > 0 is given under the description of *info* below.)

z
If *compz* = 'V' and *info* = 0, then *z* contains *Q***Z*.
If *compz* = 'I' and *info* = 0, then *z* contains the unitary or orthogonal matrix *Z* of the Schur vectors of *H*.
If *compz* = 'N', then *z* is not referenced.

work(1)
On exit, if *info* = 0, then *work(1)* returns the optimal *lwork*.

info
INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, elements 1,2, ..., *ilo*-1 and *i*+1, *i*+2, ..., *n* of *wr* and *wi* contain the real and imaginary parts of those eigenvalues that have been succesively found.
If *info* > 0, and *job* = 'E', then on exit, the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns *ilo* through *info* of the final output value of *H*.
If *info* > 0, and *job* = 'S', then on exit (initial value of *H*)**U* = *U**(final value of *H*), where *U* is a unitary matrix. The finalvalue of *H* is upper Hessenberg and triangular in rows and columns *info*+1 through *ihi*.
If *info* > 0, and *compz* = 'V', then on exit (final value of *Z*) = (initial value of *Z*)**U*, where *U* is the unitary matrix (regardless of the value of *job*).
If *info* > 0, and *compz* = 'I', then on exit (final value of *Z*) = *U*, where *U* is the unitary matrix (regardless of the value of *job*).
If *info* > 0, and *compz* = 'N', then *Z* is not accessed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hseqr interface are the following:

h	Holds the matrix *H* of size (*n*,*n*).
wr	Holds the vector of length (*n*). Used in real flavors only.
wi	Holds the vector of length (*n*). Used in real flavors only.
w	Holds the vector of length (*n*). Used in complex flavors only.
z	Holds the matrix *Z* of size (*n*,*n*).
job	Must be 'E' or 'S'. The default value is 'E'.
compz	If omitted, this argument is restored based on the presence of argument *z* as follows: *compz* = 'I', if *z* is present, *compz* = 'N', if *z* is omitted. If present, *compz* must be equal to 'I' or 'V' and the argument *z* must also be present. Note that there will be an error condition if *compz* is present and *z* omitted.

Application Notes

The computed Schur factorization is the exact factorization of a nearby matrix $H + E$, where $||E||_2 < O(\varepsilon) ||H||_2/s_i$, and ε is the machine precision.

If λ_i is an exact eigenvalue, and μ is the corresponding computed value, then $|\lambda_i - \mu_i| \leq c(n)\varepsilon||H||_2/s_i$ where $c(n)$ is a modestly increasing function of n, and s_i is the reciprocal condition number of λ_i. You can compute the condition numbers s_i by calling `?trsna`.

The total number of floating-point operations depends on how rapidly the algorithm converges; typical numbers are as follows.

If only eigenvalues are computed:	$7n^3$ for real flavors $25n^3$ for complex flavors.
If the Schur form is computed:	$10n^3$ for real flavors $35n^3$ for complex flavors.
If the full Schur factorization is computed:	$20n^3$ for real flavors $70n^3$ for complex flavors.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hsein

Computes selected eigenvectors of an upper Hessenberg matrix that correspond to specified eigenvalues.

Syntax

Fortran 77:

```
call shsein(job, eigsrc, initv, select, n, h, ldh, wr, wi, vl, ldvl, vr, ldvr,
mm, m, work, ifaill, ifailr, info)

call dhsein(job, eigsrc, initv, select, n, h, ldh, wr, wi, vl, ldvl, vr, ldvr,
mm, m, work, ifaill, ifailr, info)

call chsein(job, eigsrc, initv, select, n, h, ldh, w, vl, ldvl, vr, ldvr, mm,
m, work, rwork, ifaill, ifailr, info)

call zhsein(job, eigsrc, initv, select, n, h, ldh, w, vl, ldvl, vr, ldvr, mm,
m, work, rwork, ifaill, ifailr, info)
```

Fortran 95:

```
call hsein(h, wr, wi, select [, vl] [,vr] [,ifaill] [,ifailr] [,initv]
[,eigsrc] [,m] [,info])

call hsein(h, w, select [,vl] [,vr] [,ifaill] [,ifailr] [,initv] [,eigsrc]
[,m] [,info])
```

Description

This routine computes left and/or right eigenvectors of an upper Hessenberg matrix H, corresponding to selected eigenvalues.

The right eigenvector x and the left eigenvector y, corresponding to an eigenvalue λ, are defined by: $Hx = \lambda x$ and $y^H H = \lambda y^H$ (or $H^H y = \lambda^* y$). Here λ^* denotes the conjugate of λ.

The eigenvectors are computed by inverse iteration. They are scaled so that, for a real eigenvector x, $\max|x_i| = 1$, and for a complex eigenvector, $\max(|\mathrm{Re}x_i| + |\mathrm{Im}x_i|) = 1$.

If H has been formed by reduction of a general matrix A to upper Hessenberg form, then eigenvectors of H may be transformed to eigenvectors of A by ?ormhr or ?unmhr.

Input Parameters

job — CHARACTER*1. Must be 'R' or 'L' or 'B'.
If *job* = 'R', then only right eigenvectors are computed.
If *job* = 'L', then only left eigenvectors are computed.
If *job* = 'B', then all eigenvectors are computed.

eigsrc — CHARACTER*1. Must be 'Q' or 'N'.
If *eigsrc* = 'Q', then the eigenvalues of *H* were found using ?hseqr; thus if *H* has any zero sub-diagonal elements (and so is block triangular), then the *j*-th eigenvalue can be assumed to be an eigenvalue of the block containing the *j*-th row/column. This property allows the routine to perform inverse iteration on just one diagonal block. If *eigsrc* = 'N', then no such assumption is made and the routine performs inverse iteration using the whole matrix.

initv — CHARACTER*1. Must be 'N' or 'U'.
If *initv* = 'N', then no initial estimates for the selected eigenvectors are supplied.
If *initv* = 'U', then initial estimates for the selected eigenvectors are supplied in *vl* and/or *vr*.

select — LOGICAL.
Array, DIMENSION at least max (1, *n*). Specifies which eigenvectors are to be computed.
For real flavors:
To obtain the real eigenvector corresponding to the real eigenvalue *wr*(*j*), set *select*(*j*) to .TRUE..
To select the complex eigenvector corresponding to the complex eigenvalue (*wr*(*j*), *wi*(*j*)) with complex conjugate (*wr*(*j*+1), *wi*(*j*+1)), set *select*(*j*) and/or *select*(*j*+1) to .TRUE.; the eigenvector corresponding to the first eigenvalue in the pair is computed.
For complex flavors:
To select the eigenvector corresponding to the eigenvalue *w*(*j*), set *select*(*j*) to .TRUE..

n — INTEGER. The order of the matrix *H* (*n* ≥ 0).

h, *vl*, *vr*, — REAL for shsein
DOUBLE PRECISION for dhsein

	COMPLEX for chsein DOUBLE COMPLEX for zhsein. Arrays: *h*(*ldh*,*) The *n*-by-*n* upper Hessenberg matrix *H*. The second dimension of *h* must be at least max(1, *n*). (*ldvl*,*) If *initv* = 'V' and *job* = 'L' or 'B', then *vl* must contain starting vectors for inverse iteration for the left eigenvectors. Each starting vector must be stored in the same column or columns as will be used to store the corresponding eigenvector. If *initv* = 'N', then *vl* need not be set. The second dimension of *vl* must be at least max(1, *mm*) if *job* = 'L' or 'B' and at least 1 if *job* = 'R'. The array *vl* is not referenced if *job* = 'R'. *vr*(*ldvr*,*) If *initv* = 'V' and *job* = 'R' or 'B', then *vr* must contain starting vectors for inverse iteration for the right eigenvectors. Each starting vector must be stored in the same column or columns as will be used to store the corresponding eigenvector. If *initv* = 'N', then *vr* need not be set. The second dimension of *vr* must be at least max(1, *mm*) if *job* = 'R' or 'B' and at least 1 if *job* = 'L'. The array *vr* is not referenced if *job* = 'L'. *work*(*) is a workspace array. DIMENSION at least max (1, *n**(*n+2*)) for real flavors and at least max (1, *n***n*) for complex flavors.
ldh	INTEGER. The first dimension of *h*; at least max(1, *n*).
w	COMPLEX for chsein DOUBLE COMPLEX for zhsein. Array, DIMENSION at least max (1, *n*). Contains the eigenvalues of the matrix *H*. If *eigsrc* = 'Q', the array must be exactly as returned by ?hseqr.
wr, *wi*	REAL for shsein DOUBLE PRECISION for dhsein Arrays, DIMENSION at least max (1, *n*) each.

Contain the real and imaginary parts, respectively, of the eigenvalues of the matrix *H*. Complex conjugate pairs of values must be stored in consecutive elements of the arrays. If *eigsrc* = 'Q', the arrays must be exactly as returned by ?hseqr.

ldvl — INTEGER. The first dimension of *vl*.

If *job* = 'L' or 'B', *ldvl* ≥ max(1,*n*).

If *job* = 'R', *ldvl* ≥ 1.

ldvr — INTEGER. The first dimension of *vr*.

If *job* = 'R' or 'B', *ldvr* ≥ max(1,*n*).

If *job* = 'L', *ldvr* ≥1.

mm — INTEGER. The number of columns in *vl* and/or *vr*.
Must be at least *m*, the actual number of columns required (see Output Parameters below).
For real flavors, *m* is obtained by counting 1 for each selected real eigenvector and 2 for each selected complex eigenvector (see *select*).
For complex flavors, *m* is the number of selected eigenvectors (see *select*).
Constraint:

0 ≤ *mm* ≤ *n*.

rwork — REAL for chsein
DOUBLE PRECISION for zhsein.
Array, DIMENSION at least max (1, *n*).

Output Parameters

select — Overwritten for real flavors only.
If a complex eigenvector was selected as specified above, then *select*(*j*) is set to .TRUE. and *select*(*j*+1) to .FALSE.

w — The real parts of some elements of *w* may be modified, as close eigenvalues are perturbed slightly in searching for independent eigenvectors.

wr	Some elements of *wr* may be modified, as close eigenvalues are perturbed slightly in searching for independent eigenvectors.
vl, *vr*	If *job* = 'L' or 'B', *vl* contains the computed left eigenvectors (as specified by *select*). If *job* = 'R' or 'B', *vr* contains the computed right eigenvectors (as specified by *select*). The eigenvectors are stored consecutively in the columns of the array, in the same order as their eigenvalues. For real flavors: a real eigenvector corresponding to a selected real eigenvalue occupies one column; a complex eigenvector corresponding to a selected complex eigenvalue occupies two columns: the first column holds the real part and the second column holds the imaginary part.
m	INTEGER. For real flavors: the number of columns of *vl* and/or *vr* required to store the selected eigenvectors. For complex flavors: the number of selected eigenvectors.

ifaill, *ifailr*	INTEGER.
	Arrays, DIMENSION at least max(1, *mm*) each.
	ifaill(*i*) = 0 if the *i*th column of *vl* converged;
	ifaill(*i*) = *j* > 0 if the eigenvector stored in the *i*-th column of *vl* (corresponding to the *j*th eigenvalue) failed to converge.
	ifailr(*i*) = 0 if the *i*th column of *vr* converged;
	ifailr(*i*) = *j* > 0 if the eigenvector stored in the *i*-th column of *vr* (corresponding to the *j*th eigenvalue) failed to converge.
	For real flavors: if the *i*th and (*i*+1)th columns of *vl* contain a selected complex eigenvector, then *ifaill*(*i*) and *ifaill*(*i*+1) are set to the same value. A similar rule holds for *vr* and *ifailr*.
	The array *ifaill* is not referenced if *job* = 'R'. The array *ifailr* is not referenced if *job* = 'L'.

info	INTEGER. If *info* = 0, the execution is successful. If *info* = –*i*, the *i*-th parameter had an illegal value.

If *info* > 0, then *i* eigenvectors (as indicated by the parameters *ifaill* and/or *ifailr* above) failed to converge. The corresponding columns of *vl* and/or *vr* contain no useful information.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hsein interface are the following:

h	Holds the matrix *H* of size (*n*,*n*).
wr	Holds the vector of length (*n*). Used in real flavors only.
wi	Holds the vector of length (*n*). Used in real flavors only.
w	Holds the vector of length (*n*). Used in complex flavors only.
select	Holds the vector of length (*n*).
vl	Holds the matrix *VL* of size (*n*,*mm*).
vr	Holds the matrix *VR* of size (*n*,*mm*).
ifaill	Holds the vector of length (*mm*). Note that there will be an error condition if *ifaill* is present and *vl* is omitted.
ifailr	Holds the vector of length (*mm*). Note that there will be an error condition if *ifailr* is present and *vr* is omitted.
initv	Must be 'N' or 'U'. The default value is 'N'.
eigsrc	Must be 'N' or 'Q'. The default value is 'N'.
job	Restored based on the presence of arguments *vl* and *vr* as follows: *job* = 'B', if both *vl* and *vr* are present, *job* = 'L', if *vl* is present and *vr* omitted, *job* = 'R', if *vl* is omitted and *vr* present, Note that there will be an error condition if both *vl* and *vr* are omitted.

Application Notes

Each computed right eigenvector *x* i is the exact eigenvector of a nearby matrix $A + E_i$, such that $||E_i|| < O(\varepsilon)||A||$. Hence the residual is small:

$$||Ax_i - \lambda_i x_i|| = O(\varepsilon)||A||.$$

However, eigenvectors corresponding to close or coincident eigenvalues may not accurately span the relevant subspaces.

Similar remarks apply to computed left eigenvectors.

?trevc

Computes selected eigenvectors of an upper (quasi-) triangular matrix computed by ?hseqr.

Syntax

Fortran 77:

call strevc(*side*, *howmny*, *select*, *n*, *t*, *ldt*, *vl*, *ldvl*, *vr*, *ldvr*, *mm*, *m*, *work*, *info*)

call dtrevc(*side*, *howmny*, *select*, *n*, *t*, *ldt*, *vl*, *ldvl*, *vr*, *ldvr*, *mm*, *m*, *work*, *info*)

call ctrevc(*side*, *howmny*, *select*, *n*, *t*, *ldt*, *vl*, *ldvl*, *vr*, *ldvr*, *mm*, *m*, *work*, *rwork*, *info*)

call ztrevc(*side*, *howmny*, *select*, *n*, *t*, *ldt*, *vl*, *ldvl*, *vr*, *ldvr*, *mm*, *m*, *work*, *rwork*, *info*)

Fortran 95:

call trevc(*t* [, *howmny*] [,*select*] [,*vl*] [,*vr*] [,*m*] [,*info*])

Description

This routine computes some or all of the right and/or left eigenvectors of an upper triangular matrix T (or, for real flavors, an upper quasi-triangular matrix T). Matrices of this type are produced by the Schur factorization of a general matrix: $A = Q*T*Q^H$, as computed by ?hseqr.

The right eigenvector x and the left eigenvector y of T corresponding to an eigenvalue w, are defined by:

$T*x = w*x$, $y^H*T = w*y^H$, where y^H denotes the conjugate transpose of y.

The eigenvalues are not input to this routine, but are read directly from the diagonal blocks of T.

This routine returns the matrices X and/or Y of right and left eigenvectors of T, or the products $Q*X$ and/or $Q*Y$, where Q is an input matrix.

If *Q* is the orthogonal/unitary factor that reduces a matrix *A* to Schur form *T*, then *Q***X* and *Q***Y* are the matrices of right and left eigenvectors of *A*.

Input Parameters

side — CHARACTER*1. Must be 'R' or 'L' or 'B'.
If *side* = 'R', then only right eigenvectors are computed.
If *side* = 'L', then only left eigenvectors are computed.
If *side* = 'B', then all eigenvectors are computed.

howmny — CHARACTER*1. Must be 'A' or 'B' or 'S'.
If *howmny* = 'A', then all eigenvectors (as specified by *side*) are computed.
If *howmny* = 'B', then all eigenvectors (as specified by *side*) are computed and backtransformed by the matrices supplied in *vl* and *vr*.
If *howmny* = 'S', then selected eigenvectors (as specified by *side* and *select*) are computed.

select — LOGICAL.
Array, DIMENSION at least max (1, *n*).
If *howmny* = 'S', *select* specifies which eigenvectors are to be computed.
If *howmny* = 'A' or 'B', *select* is not referenced.
For real flavors:
If *omega*(*j*) is a real eigenvalue, the corresponding real eigenvector is computed if *select*(*j*) is .TRUE..
If *omega*(*j*) and *omega*(*j*+1) are the real and imaginary parts of a complex eigenvalue, the corresponding complex eigenvector is computed if either *select*(*j*) or *select*(*j*+1) is .TRUE., and on exit *select*(*j*) is set to .TRUE.and *select*(*j*+1) is set to .FALSE..
For complex flavors:
The eigenvector corresponding to the *j*-th eigenvalue is computed if *select*(*j*) is .TRUE..

n — INTEGER. The order of the matrix *T* (*n* ≥ 0).

t, *vl*, *vr*, — REAL for strevc
DOUBLE PRECISION for dtrevc
COMPLEX for ctrevc
DOUBLE COMPLEX for ztrevc.

	Arrays: *t*(*ldt*,*) contains the *n*-by-*n* matrix *T* in Schur canonical form. The second dimension of *t* must be at least max(1, *n*). *vl*(*ldvl*,*) If *howmny* = 'B' and *side* = 'L' or 'B', then *vl* must contain an *n*-by-*n* matrix *Q* (usually the matrix of Schur vectors returned by ?hseqr). If *howmny* = 'A' or 'S', then *vl* need not be set. The second dimension of *vl* must be at least max(1, *mm*) if *side* = 'L' or 'B' and at least 1 if *side* = 'R'. The array *vl* is not referenced if *side* = 'R'. *vr* (*ldvr*,*) If *howmny* = 'B' and *side* = 'R' or 'B', then *vr* must contain an *n*-by-*n* matrix *Q* (usually the matrix of Schur vectors returned by ?hseqr). . If *howmny* = 'A' or 'S', then *vr* need not be set. The second dimension of *vr* must be at least max(1, *mm*) if *side* = 'R' or 'B' and at least 1 if *side* = 'L'. The array *vr* is not referenced if *side* = 'L'. *work*(*) is a workspace array. DIMENSION at least max (1, 3**n*) for real flavors and at least max (1, 2**n*) for complex flavors.
ldt	INTEGER. The first dimension of *t*; at least max(1, *n*).
ldvl	INTEGER. The first dimension of *vl*. If *side* = 'L' or 'B', *ldvl* ≥ *n*. If *side* = 'R', *ldvl* ≥ 1.
ldvr	INTEGER. The first dimension of *vr*. If *side* = 'R' or 'B', *ldvr* ≥ *n*. If *side* = 'L', *ldvr* ≥ 1.
mm	INTEGER. The number of columns in the arrays *vl* and/or *vr*. Must be at least *m* (the precise number of columns required). If *howmny* = 'A' or 'B', *m* = *n*.

If *howmny* = 'S': for real flavors, *m* is obtained by counting 1 for each selected real eigenvector and 2 for each selected complex eigenvector;
for complex flavors, *m* is the number of selected eigenvectors (see *select*).

Constraint: $0 \le m \le n$.

rwork — REAL for ctrevc
DOUBLE PRECISION for ztrevc.
Workspace array, DIMENSION at least max (1, *n*).

Output Parameters

select — If a complex eigenvector of a real matrix was selected as specified above, then *select*(*j*) is set to .TRUE. and *select*(*j*+1) to .FALSE.

vl, *vr* — If *side* = 'L' or 'B', *vl* contains the computed left eigenvectors (as specified by *howmny* and *select*).
If *side* = 'R' or 'B', *vr* contains the computed right eigenvectors (as specified by *howmny* and *select*).
The eigenvectors are stored consecutively in the columns of the array, in the same order as their eigenvalues.
For real flavors: corresponding to each real eigenvalue is a real eigenvector, occupying one column;corresponding to each complex conjugate pair of eigenvalues is a complex eigenvector, occupying two columns; the first column holds the real part and the second column holds the imaginary part.

m — INTEGER.
For complex flavors: the number of selected eigenvectors.
If *howmny* = 'A' or 'B', *m* is set to *n*.
For real flavors: the number of columns of *vl* and/or *vr* actually used to store the selected eigenvectors.
If *howmny* = 'A' or 'B', *m* is set to *n*.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `trevc` interface are the following:

t	Holds the matrix *T* of size (*n*,*n*).
select	Holds the vector of length (*n*).
vl	Holds the matrix *VL* of size (*n*,*mm*).
vr	Holds the matrix *VR* of size (*n*,*mm*).
side	If omitted, this argument is restored based on the presence of arguments *vl* and *vr* as follows: *side* = 'B', if both *vl* and *vr* are present, *side* = 'L', if *vr* is omitted, *side* = 'R', if *vl* is omitted. Note that there will be an error condition if both *vl* and *vr* are omitted.
howmny	If omitted, this argument is restored based on the presence of argument *select* as follows: *howmny* = 'V', if *q* is present, *howmny* = 'N', if *q* is omitted. If present, *vect* = 'V' or 'U' and the argument *q* must also be present. Note that there will be an error condition if both *select* and *howmny* are present.

Application Notes

If x i is an exact right eigenvector and y_i is the corresponding computed eigenvector, then the angle $\theta(y_i, x_i)$ between them is bounded as follows: $\theta(y_i, x_i) \leq (c(n)\varepsilon||T||_2)/\mathrm{sep}_i$ where sep_i is the reciprocal condition number of x_i. The condition number sep_i may be computed by calling `?trsna`.

?trsna

Estimates condition numbers for specified eigenvalues and right eigenvectors of an upper (quasi-) triangular matrix.

Syntax

Fortran 77:

```
call strsna(job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, s, sep, mm, m, work, ldwork, iwork, info)
call dtrsna(job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, s, sep, mm, m, work, ldwork, iwork, info)
call ctrsna(job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, s, sep, mm, m, work, ldwork, rwork, info)
call ztrsna(job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, s, sep, mm, m, work, ldwork, rwork, info)
```

Fortran 95:

```
call trsna(t [, s] [,sep] [,vl] [,vr] [,select] [,m] [,info])
```

Description

This routine estimates condition numbers for specified eigenvalues and/or right eigenvectors of an upper triangular matrix T (or, for real flavors, upper quasi-triangular matrix T in canonical Schur form). These are the same as the condition numbers of the eigenvalues and right eigenvectors of an original matrix $A = Z*T*Z^H$ (with unitary or, for real flavors, orthogonal Z), from which T may have been derived.

The routine computes the reciprocal of the condition number of an eigenvalue $lambda(i)$ as $s_i = |v_H u| / (||u||_E ||v||_E)$, where u and v are the right and left eigenvectors of T, respectively, corresponding to $lambda(i)$. This reciprocal condition number always lies between zero (ill-conditioned) and one (well-conditioned).

An approximate error estimate for a computed eigenvalue $lambda(i)$ is then given by $\varepsilon ||T|| / s_i$, where ε is the machine precision.

To estimate the reciprocal of the condition number of the right eigenvector corresponding to *lambda*(*i*), the routine first calls ?trexc to reorder the eigenvalues so that *lambda*(*j*) is in the leading position:

$$T = Q \begin{bmatrix} \lambda_i & C^H \\ 0 & T_{22} \end{bmatrix} Q^H$$

The reciprocal condition number of the eigenvector is then estimated as sep_i, the smallest singular value of the matrix T_{22} - *lambda*(*i*)*I. This number ranges from zero (ill-conditioned) to very large (well-conditioned).

An approximate error estimate for a computed right eigenvector u corresponding to *lambda*(*i*) is then given by $\varepsilon||T||/sep_i$.

Input Parameters

job	CHARACTER*1. Must be 'E' or 'V' or 'B'. If *job* = 'E', then condition numbers for eigenvalues only are computed. If *job* = 'V', then condition numbers for eigenvectors only are computed. If *job* = 'B', then condition numbers for both eigenvalues and eigenvectors are computed.
howmny	CHARACTER*1. Must be 'A' or 'S'. If *howmny* = 'A' , then the condition numbers for all eigenpairs are computed. If *howmny* = 'S' , then condition numbers for selected eigenpairs (as specified by *select*) are computed.
select	LOGICAL. Array, DIMENSION at least max (1, *n*) if *howmny* = 'S' and at least 1 otherwise. Specifies the eigenpairs for which condition numbers are to be computed if *howmny*= 'S'. For real flavors:

To select condition numbers for the eigenpair corresponding to the real eigenvalue *lambda*(*j*), *select*(*j*) must be set .TRUE.;
to select condition numbers for the eigenpair corresponding to a complex conjugate pair of eigenvalues *lambda*(*j*) and *lambda*(*j*+1), *select*(*j*) and/or *select*(*j*+1) must be set .TRUE.
For complex flavors
To select condition numbers for the eigenpair corresponding to the eigenvalue *lambda*(*j*), *select*(*j*) must be set .TRUE. *select* is not referenced if *howmny* = 'A'.

n — INTEGER. The order of the matrix *T* (*n* ≥ 0).

t, *vl*, *vr*, *work* — REAL for strsna
DOUBLE PRECISION for dtrsna
COMPLEX for ctrsna
DOUBLE COMPLEX for ztrsna.
Arrays:
t(*ldt*,*) contains the *n*-by-*n* matrix *T*.
The second dimension of *t* must be at least max(1, *n*).
vl(*ldvl*,*)
If *job* = 'E' or 'B', then *vl* must contain the left eigenvectors of *T* (or of any matrix $Q*T*Q^H$ with *Q* unitary or orthogonal) corresponding to the eigenpairs specified by *howmny* and *select*. The eigenvectors must be stored in consecutive columns of *vl*, as returned by ?trevc or ?hsein.
The second dimension of *vl* must be at least max(1, *mm*) if *job* = 'E' or 'B' and at least 1 if *job* = 'V'.
The array *vl* is not referenced if *job* = 'V'.
vr(*ldvr*,*)
If *job* = 'E' or 'B', then *vr* must contain the right eigenvectors of *T* (or of any matrix $Q*T*Q^H$ with *Q* unitary or orthogonal) corresponding to the eigenpairs specified by *howmny* and *select*. The eigenvectors must be stored in consecutive columns of *vr*, as returned by ?trevc or ?hsein.

The second dimension of *vr* must be at least max(1, *mm*) if *job* = 'E' or 'B' and at least 1 if *job* = 'V'.
The array *vr* is not referenced if *job* = 'V'.
work is a workspace array, its dimension (*ldwork*,*n*+6).
The array *work* is not referenced if *job* = 'E'.

ldt — INTEGER. The first dimension of *t*; at least max(1, *n*).

ldvl — INTEGER. The first dimension of *vl*.

If *job* = 'E' or 'B', *ldvl* ≥ max(1,*n*).

If *job* = 'V', *ldvl* ≥ 1.

ldvr — INTEGER. The first dimension of *vr*.

If *job* = 'E' or 'B', *ldvr* ≥ max(1,*n*).

If *job* = 'R', *ldvr* ≥ 1.

mm — INTEGER. The number of elements in the arrays *s* and *sep*, and the number of columns in *vl* and *vr* (if used). Must be at least *m* (the precise number required).
If *howmny* = 'A', *m* = *n*;
if *howmny* = 'S', for real flavors *m* is obtained by counting 1 for each selected real eigenvalue and 2 for each selected complex conjugate pair of eigenvalues.
for complex flavors *m* is the number of selected eigenpairs (see *select*). Constraint:

0 ≤ *m* ≤ *n*.

ldwork — INTEGER. The first dimension of *work*.

If *job* = 'V' or 'B', *ldwork* ≥ max(1,*n*).

If *job* = 'E', *ldwork* ≥ 1.

rwork — REAL for ctrsna, ztrsna.
Array, DIMENSION at least max (1, *n*).

iwork — INTEGER for strsna, dtrsna.
Array, DIMENSION at least max (1, *n*).

Output Parameters

s — REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.

Array, DIMENSION at least max(1, *mm*) if *job* = 'E' or 'B' and at least 1 if *job* = 'V'.
Contains the reciprocal condition numbers of the selected eigenvalues if *job* = 'E' or 'B', stored in consecutive elements of the array. Thus *s*(*j*), *sep*(*j*) and the *j*-th columns of *vl* and *vr* all correspond to the same eigenpair (but not in general the *j* th eigenpair unless all eigenpairs have been selected).
For real flavors: for a complex conjugate pair of eigenvalues, two consecutive elements of S are set to the same value.
The array *s* is not referenced if *job* = 'V'.

sep

REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
Array, DIMENSION at least max(1, *mm*) if *job* = 'V' or 'B' and at least 1 if *job* = 'E'. Contains the estimated reciprocal condition numbers of the selected right eigenvectors if *job* = 'V' or 'B', stored in consecutive elements of the array.
For real flavors: for a complex eigenvector, two consecutive elements of *sep* are set to the same value; if the eigenvalues cannot be reordered to compute *sep*(*j*), then *sep*(*j*) is set to zero; this can only occur when the true value would be very small anyway. The array *sep* is not referenced if *job* = 'E'.

m

INTEGER.
For complex flavors: the number of selected eigenpairs.
If *howmny* = 'A', *m* is set to *n*.
For real flavors: the number of elements of *s* and/or *sep* actually used to store the estimated condition numbers.
If *howmny* = 'A', *m* is set to *n*.

info

INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `trsna` interface are the following:

t	Holds the matrix *T* of size (*n*,*n*).
s	Holds the vector of length (*mm*).
sep	Holds the vector of length (*mm*).
vl	Holds the matrix *VL* of size (*n*,*mm*).
vr	Holds the matrix *VR* of size (*n*,*mm*).
select	Holds the vector of length (*n*).
job	Restored based on the presence of arguments *s* and sep as follows: *job* = 'B', if both *s* and *sep* are present, *job* = 'E', if *s* is present and *sep* omitted, *job* = 'V', if *s* is omitted and *sep* present. Note an error condition if both *s* and *sep* are omitted.
howmny	Restored based on the presence of the argument *select* as follows: *howmny* = 'S', if *select* is present, *howmny* = 'A', if *select* is omitted.

Note that the arguments *s*, *vl*, and *vr* must either be all present or all omitted.

Otherwise, an error condition is observed.

Application Notes

The computed values sep_i may overestimate the true value, but seldom by a factor of more than 3.

?trexc

Reorders the Schur factorization of a general matrix.

Syntax

Fortran 77:

```
call strexc(compq, n, t, ldt, q, ldq, ifst, ilst, work, info)
call dtrexc(compq, n, t, ldt, q, ldq, ifst, ilst, work, info)
call ctrexc(compq, n, t, ldt, q, ldq, ifst, ilst, info)
call ztrexc(compq, n, t, ldt, q, ldq, ifst, ilst, info)
```

Fortran 95:

```
call trexc(t, ifst, ilst [,q] [,info])
```

Description

This routine reorders the Schur factorization of a general matrix $A = Q*T*Q^H$, so that the diagonal element or block of T with row index $ifst$ is moved to row $ilst$.

The reordered Schur form S is computed by an unitary (or, for real flavors, orthogonal) similarity transformation: $S = Z^H*T*Z$. Optionally the updated matrix P of Schur vectors is computed as $P = Q*Z$, giving $A = P*S*P^H$.

Input Parameters

compq	CHARACTER*1. Must be 'V' or 'N'. If *compq* = 'V', then the Schur vectors (Q) are updated. If *compq* = 'N', then no Schur vectors are updated.
n	INTEGER. The order of the matrix T ($n \geq 0$).
t, *q*	REAL for strexc DOUBLE PRECISION for dtrexc COMPLEX for ctrexc DOUBLE COMPLEX for ztrexc. Arrays: *t*(*ldt*,*) contains the n-by-n matrix T. The second dimension of *t* must be at least max(1, n). *q*(*ldq*,*) If *compq* = 'V', then *q* must contain Q (Schur vectors). If *compq* = 'N', then *q* is not referenced. The second dimension of *q* must be at least max(1, n) if *compq* = 'V' and at least 1 if *compq* = 'N'.
ldt	INTEGER. The first dimension of *t*; at least max(1, n).
ldq	INTEGER. The first dimension of *q*; If *compq* = 'N', then *ldq*≥ 1. If *compq* = 'V', then *ldq*≥ max(1, *n*).
ifst, *ilst*	INTEGER. $1 \leq ifst \leq n$; $1 \leq ilst \leq n$.

	Must specify the reordering of the diagonal elements (or blocks, which is possible for real flavors) of the matrix T. The element (or block) with row index $ifst$ is moved to row $ilst$ by a sequence of exchanges between adjacent elements (or blocks).
work	REAL for strexc DOUBLE PRECISION for dtrexc. Array, DIMENSION at least max (1, n).

Output Parameters

t	Overwritten by the updated matrix S.
q	If $compq$ = 'V', q contains the updated matrix of Schur vectors.
ifst, *ilst*	Overwritten for real flavors only. If $ifst$ pointed to the second row of a 2 by 2 block on entry, it is changed to point to the first row; $ilst$ always points to the first row of the block in its final position (which may differ from its input value by ±1).
info	INTEGER. If $info$ = 0, the execution is successful. If $info$ = $-i$, the i-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine trexc interface are the following:

t	Holds the matrix T of size (n,n).
q	Holds the matrix Q of size (n,n).
compq	Restored based on the presence of the argument q as follows: $compq$ = 'V', if q is present, $compq$ = 'N', if q is omitted.

Application Notes

The computed matrix S is exactly similar to a matrix $T + E$, where $||E||_2 = O(\varepsilon)||T||_2$, and ε is the machine precision.

Note that if a 2 by 2 diagonal block is involved in the re-ordering, its off-diagonal elements are in general changed; the diagonal elements and the eigenvalues of the block are unchanged unless the block is sufficiently ill-conditioned, in which case they may be noticeably altered. It is possible for a 2 by 2 block to break into two 1 by 1 blocks, that is, for a pair of complex eigenvalues to become purely real.

The values of eigenvalues however are never changed by the re-ordering.

The approximate number of floating-point operations is

for real flavors: $6n(ifst-ilst)$ if *compq* = 'N';
$12n(ifst-ilst)$ if *compq* = 'V';

for complex flavors: $20n(ifst-ilst)$ if *compq* = 'N';
$40n(ifst-ilst)$ if *compq* = 'V'.

?trsen

Reorders the Schur factorization of a matrix and (optionally) computes the reciprocal condition numbers and invariant subspace for the selected cluster of eigenvalues.

Syntax

Fortran 77:

```
call strsen(job, compq, select, n, t, ldt, q, ldq, wr, wi, m, s, sep, work, lwork, iwork, liwork, info)
call dtrsen(job, compq, select, n, t, ldt, q, ldq, wr, wi, m, s, sep, work, lwork, iwork, liwork, info)
call ctrsen(job, compq, select, n, t, ldt, q, ldq, w, m, s, sep, work, lwork, info)
call ztrsen(job, compq, select, n, t, ldt, q, ldq, w, m, s, sep, work, lwork, info)
```

Fortran 95:

```
call trsen(t, select [,wr] [,wi] [,m] [,s] [,sep] [,q] [,info])
call trsen(t, select [,w] [,m] [,s] [,sep] [,q] [,info])
```

Description

This routine reorders the Schur factorization of a general matrix $A = Q*T*Q^H$ so that a selected cluster of eigenvalues appears in the leading diagonal elements (or, for real flavors, diagonal blocks) of the Schur form. The reordered Schur form R is computed by an unitary (orthogonal) similarity transformation: $R = Z^H*T*Z$. Optionally the updated matrix P of Schur vectors is computed as $P = Q*Z$, giving $A = P*R*P^H$.

Let

$$R = \begin{bmatrix} T_{11} & T_{12} \\ 0 & T_{13} \end{bmatrix}$$

where the selected eigenvalues are precisely the eigenvalues of the leading m-by-m submatrix T_{11}. Let P be correspondingly partitioned as (Q 1 Q_2) where Q_1 consists of the first m columns of Q. Then $A*Q$ 1 = $Q_1 T_{11}$, and so the m columns of Q_1 form an orthonormal basis for the invariant subspace corresponding to the selected cluster of eigenvalues.

Optionally the routine also computes estimates of the reciprocal condition numbers of the average of the cluster of eigenvalues and of the invariant subspace.

Input Parameters

job	CHARACTER*1. Must be 'N' or 'E' or 'V' or 'B'. If *job* = 'N', then no condition numbers are required. If *job* = 'E', then only the condition number for the cluster of eigenvalues is computed. If *job* = 'V', then only the condition number for the invariant subspace is computed. If *job* = 'B', then condition numbers for both the cluster and the invariant subspace are computed.
compq	CHARACTER*1. Must be 'V' or 'N'. If *compq* = 'V', then Q of the Schur vectors is updated.

If *compq* = 'N', then no Schur vectors are updated.

select — LOGICAL.
Array, DIMENSION at least max (1, *n*).
Specifies the eigenvalues in the selected cluster. To select an eigenvalue *lambda*(*j*), *select*(*j*) must be .TRUE.
For real flavors: to select a complex conjugate pair of eigenvalues *lambda*(*j*) and *lambda*(*j*+1) (corresponding 2 by 2 diagonal block), *select*(*j*) and/or *select*(*j*+1) must be .TRUE.; the complex conjugate *lambda*(*j*)and *lambda*(*j*+1) must be either both included in the cluster or both excluded.

n — INTEGER. The order of the matrix *T* (*n* ≥ 0).

t, *q*, *work* — REAL for strsen
DOUBLE PRECISION for dtrsen
COMPLEX for ctrsen
DOUBLE COMPLEX for ztrsen.
Arrays:
t (*ldt*,*) The *n*-by-*n* *T*.
The second dimension of *t* must be at least max(1, *n*).

q (*ldq*,*)
If *compq* = 'V', then *q* must contain *Q* of Schur vectors.
If *compq* = 'N', then *q* is not referenced.
The second dimension of *q* must be at least max(1, *n*) if *compq* = 'V' and at least 1 if *compq* = 'N'.
work is a workspace array, its dimension max(1, *lwork*).

ldt — INTEGER. The first dimension of *t*; at least max(1, *n*).

ldq — INTEGER. The first dimension of *q*;

If *compq* = 'N', then *ldq* ≥ 1.

If *compq* = 'V', then *ldq* ≥ max(1, *n*).

lwork — INTEGER. The dimension of the array *work*.

If *job* = 'V' or 'B', *lwork* ≥ max(1,2**m**(*n*-*m*)).

If *job* = 'E', then *lwork* ≥ max(1, *m**(*n*-*m*))

If *job* = 'N', then *lwork* ≥ 1 for complex flavors and *lwork* ≥ max(1,*n*) for real flavors.
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for details.

iwork — INTEGER.*iwork*(*liwork*) is a workspace array. The array *iwork* is not referenced if *job* = 'N' or 'E'.
The actual amount of workspace required cannot exceed $n^2/2$ if *job* = 'V' or 'B'.

liwork — INTEGER.
The dimension of the array *iwork*.
If *job* = 'V' or 'B', *liwork* ≥ max(1,2*m*(*n*-*m*)).
If *job* = 'E' or 'E', *liwork* ≥ 1.
If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by xerbla. See Application Notes for details.

Output Parameters

t — Overwritten by the updated matrix *R*.

q — If *compq* = 'V', *q* contains the updated matrix of Schur vectors; the first m columns of the *Q* form an orthogonal basis for the specified invariant subspace.

w — COMPLEX for ctrsen
DOUBLE COMPLEX for ztrsen.
Array, DIMENSION at least max(1, *n*). The recorded eigenvalues of *R*. The eigenvalues are stored in the same order as on the diagonal of *R*.

wr, *wi* — REAL for strsen
DOUBLE PRECISION for dtrsen

Arrays, DIMENSION at least max(1, *n*). Contain the real and imaginary parts, respectively, of the reordered eigenvalues of *R*. The eigenvalues are stored in the same order as on the diagonal of *R*. Note that if a complex eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.

m

INTEGER.
For complex flavors: the number of the specified invariant subspaces, which is the same as the number of selected eigenvalues (see *select*).
For real flavors: the dimension of the specified invariant subspace. The value of *m* is obtained by counting 1 for each selected real eigenvalue and 2 for each selected complex conjugate pair of eigenvalues (see *select*).

Constraint: $0 \le m \le n$.

s

REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
If *job* = 'E' or 'B', *s* is a lower bound on the reciprocal condition number of the average of the selected cluster of eigenvalues.
If *m* = 0 or *n*, then *s* = 1.
For real flavors: if *info* = 1, then *s* is set to zero.*s* is not referenced if *job* = 'N' or 'V'.

sep

REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors.
If *job* = 'V' or 'B', *sep* is the estimated reciprocal condition number of the specified invariant subspace.
If *m* = 0 or *n*, then *sep* = |*T*|.
For real flavors: if *info* = 1, then *sep* is set to zero.
sep is not referenced if *job* = 'N' or 'E'.

work(1)

On exit, if *info* = 0, then *work*(1) returns the optimal size of *lwork*.

iwork(1)

On exit, if *info* = 0, then *iwork*(1) returns the optimal size of *liwork*.

info

INTEGER.
If *info* = 0, the execution is successful.
If *info* = –*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `trsen` interface are the following:

t	Holds the matrix *T* of size (*n*,*n*).
select	Holds the vector of length (*n*).
wr	Holds the vector of length (*n*). Used in real flavors only.
wi	Holds the vector of length (*n*). Used in real flavors only.
w	Holds the vector of length (*n*). Used in complex flavors only.
q	Holds the matrix *Q* of size (*n*,*n*).
compq	Restored based on the presence of the argument *q* as follows: *compq* = 'V', if *q* is present, *compq* = 'N', if *q* is omitted.
job	Restored based on the presence of arguments *s* and sep as follows: *job* = 'B', if both *s* and *sep* are present, *job* = 'E', if *s* is present and *sep* omitted, *job* = 'V', if *s* is omitted and *sep* present, *job* = 'N', if both *s* and *sep* are omitted.

Application Notes

The computed matrix *R* is exactly similar to a matrix $T + E$, where $||E||_2 = O(\varepsilon)||T||_2$, and ε is the machine precision. The computed *s* cannot underestimate the true reciprocal condition number by more than a factor of $(\min(m, n\text{-}m))_{1/2}$; *sep* may differ from the true value by $(m*n\text{-}m^2)_{1/2}$. The angle between the computed invariant subspace and the true subspace is $O(\varepsilon)||A||_2/sep$. Note that if a 2 by 2 diagonal block is involved in the re-ordering, its off-diagonal elements are in general changed; the diagonal elements and the eigenvalues of the block are unchanged unless the block is sufficiently ill-conditioned, in which case they may be noticeably altered. It is possible for a 2 by 2 block to break into two 1 by 1 blocks, that is, for a pair of complex eigenvalues to become purely real. The values of eigenvalues however are never changed by the re-ordering.

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If you choose the first option and set any of admissible *lwork* (or *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?trsyl

Solves Sylvester equation for real quasi-triangular or complex triangular matrices.

Syntax

Fortran 77:

```
call strsyl(trana, tranb, isgn, m, n, a, lda, b, ldb, c, ldc, scale, info)
call dtrsyl(trana, tranb, isgn, m, n, a, lda, b, ldb, c, ldc, scale, info)
call ctrsyl(trana, tranb, isgn, m, n, a, lda, b, ldb, c, ldc, scale, info)
call ztrsyl(trana, tranb, isgn, m, n, a, lda, b, ldb, c, ldc, scale, info)
```

Fortran 95:

```
call trsyl(a, b, c, scale [, trana] [,tranb] [,isgn] [,info])
```

Description

This routine solves the Sylvester matrix equation op(*A*)*X* ± *X* op(*B*) = α*C*, where op(*A*) = *A* or A^H, and the matrices *A* and *B* are upper triangular (or, for real flavors, upper quasi-triangular in canonical Schur form); α ≤ 1 is a scale factor determined by the routine to avoid overflow in *X*; *A* is *m*-by-*m*, *B* is *n*-by-*n*, and *C* and *X* are both *m*-by-*n*. The matrix *X* is obtained by a straightforward process of back substitution.

The equation has a unique solution if and only if $\alpha_i \pm \beta_i \neq 0$, where $\{\alpha_i\}$ and $\{\beta_i\}$ are the eigenvalues of A and B, respectively, and the sign (+ or -) is the same as that used in the equation to be solved.

Input Parameters

trana	CHARACTER*1. Must be 'N' or 'T' or 'C'. If *trana* = 'N' , then op(*A*) = *A*. If *trana* = 'T', then op(*A*) = A^T (real flavors only). If *trana* = 'C' then op(*A*) = A^H.
tranb	CHARACTER*1. Must be 'N' or 'T' or 'C'. If *tranb* = 'N' , then op(*B*) = *B*. If *tranb* = 'T', then op(*B*) = B^T (real flavors only). If *tranb* = 'C', then op(*B*) = B^H.
isgn	INTEGER. Indicates the form of the Sylvester equation. If *isgn* = +1, op(*A*)**X* + *X**op(*B*) = alpha**C*. If *isgn* = -1, op(*A*)**X* - *X**op(*B*) = alpha**C*.
m	INTEGER. The order of *A*, and the number of rows in *X* and *C* ($m \geq 0$).
n	INTEGER. The order of *B*, and the number of columns in *X* and *C* ($n \geq 0$).
a, *b*, *c*	REAL for strsyl DOUBLE PRECISION for dtrsyl COMPLEX for ctrsyl DOUBLE COMPLEX for ztrsyl. Arrays: *a*(*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1, *m*). *b*(*ldb*,*) contains the matrix *B*. The second dimension of *b* must be at least max(1, *n*). *c*(*ldc*,*) contains the matrix *C*. The second dimension of *c* must be at least max(1, *n*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
ldb	INTEGER. The first dimension of *b*; at least max(1, *n*).
ldc	INTEGER. The first dimension of *c*; at least max(1, *n*).

Output Parameters

c	Overwritten by the solution matrix *X*.
scale	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. The value of the scale factor α.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = 1, *A* and *B* have common or close eigenvalues perturbed values were used to solve the equation.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine trsyl interface are the following:

a	Holds the matrix *A* of size (*m*,*m*).
b	Holds the matrix *B* of size (*n*,*n*).
c	Holds the matrix *C* of size (*m*,*n*).
trana	Must be 'N', 'C', or 'T'. The default value is 'N'.
tranb	Must be 'N', 'C', or 'T'. The default value is 'N'.
isgn	Must be +1 or -1. The default value is +1.

Application Notes

Let *X* be the exact, *Y* the corresponding computed solution, and *R* the residual matrix: *R* = *C* - (*AY* ± *YB*). Then the residual is always small:

$$||R||_F = O(\varepsilon) \quad (||A||_F + ||B||_F) \quad ||Y||_F.$$

However, *Y* is not necessarily the exact solution of a slightly perturbed equation; in other words, the solution is not backwards stable.

For the forward error, the following bound holds:

$$||Y - X||_F \leq ||R||_F / \mathrm{sep}(A, B)$$

but this may be a considerable overestimate. See [Golub96] for a definition of sep(*A*, *B*).

The approximate number of floating-point operations for real flavors is *m***n**(*m* + *n*). For complex flavors it is 4 times greater.

Generalized Nonsymmetric Eigenvalue Problems

This section describes LAPACK routines for solving generalized nonsymmetric eigenvalue problems, reordering the generalized Schur factorization of a pair of matrices, as well as performing a number of related computational tasks.

A `generalized nonsymmetric eigenvalue problem` is as follows: given a pair of nonsymmetric (or non-Hermitian) *n*by-*n* matrices *A* and *B*, find the `generalized eigenvalues` λ and the corresponding `generalized eigenvectors` *x* and *y* that satisfy the equations

$Ax = \lambda Bx$ (right generalized eigenvectors *x*)

and

$y^H A = \lambda_y{}^H B$ (left generalized eigenvectors *y*).

Table 4-6 lists LAPACK routines (Fortran-77 interface) used to solve the generalized nonsymmetric eigenvalue problems and the generalized Sylvester equation. Respective routine names in Fortran-95 interface are without the first symbol (see Routine Naming Conventions).

Table 4-6 Computational Routines for Solving Generalized Nonsymmetric Eigenvalue Problems

Routine name	Operation performed
`?gghrd`	Reduces a pair of matrices to generalized upper Hessenberg form using orthogonal/unitary transformations.
`?ggbal`	Balances a pair of general real or complex matrices.
`?ggbak`	Forms the right or left eigenvectors of a generalized eigenvalue problem.
`?hgeqz`	Implements the QZ method for finding the generalized eigenvalues of the matrix pair (H,T).
`?tgevc`	Computes some or all of the right and/or left generalized eigenvectors of a pair of upper triangular matrices

Routine name	Operation performed
?tgexc	Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that one diagonal block of (A,B) moves to another row index.
?tgsen	Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that a selected cluster of eigenvalues appears in the leading diagonal blocks of (A,B).
?tgsyl	Solves the generalized Sylvester equation.
?tgsna	Estimates reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a pair of matrices in generalized real Schur canonical form.

?gghrd

Reduces a pair of matrices to generalized upper Hessenberg form using orthogonal/unitary transformations.

Syntax

Fortran 77:

```
call sgghrd(compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, info)
call dgghrd(compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, info)
call cgghrd(compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, info)
call zgghrd(compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, info)
```

Fortran 95:

```
call gghrd(a, b [,ilo] [,ihi] [,q] [,z] [,compq] [,compz] [,info])
```

Description

This routine reduces a pair of real/complex matrices (A,B) to generalized upper Hessenberg form using orthogonal/unitary transformations, where A is a general matrix and B is upper triangular. The form of the generalized eigenvalue problem is $A*x = \lambda*B*x$, and B is typically made upper triangular by computing its QR factorization and moving the orthogonal matrix Q to the left side of the equation.

This routine simultaneously reduces A to a Hessenberg matrix H:

$Q^H*A*Z = H$

and transforms B to another upper triangular matrix T:

$Q^H*B*Z = T$

in order to reduce the problem to its standard form $H*y = \lambda*T*y$, where $y = Z^H*x$.

The orthogonal/unitary matrices Q and Z are determined as products of Givens rotations. They may either be formed explicitly, or they may be postmultiplied into input matrices Q_1 and $_1$, so that

$Q_1*A*Z*Z_1^H = (Q_1*Q)*H*(Z_1*Z)^H$

$Q_1*B*Z_1^H = (Q_1*Q)*T*(Z_1*Z)^H$

If Q_1 is the orthogonal/unitary matrix from the QR factorization of B in the original equation $Ax = \lambda Bx$, then the routine `?gghrd` reduces the original problem to generalized Hessenberg form.

Input Parameters

compq	`CHARACTER*1`. Must be `'N'`, `'I'`, or `'V'`. If *compq* = `'N'`, matrix Q is not computed. If *compq* = `'I'`, Q is initialized to the unit matrix, and the orthogonal/unitary matrix Q is returned; If *compq* = `'V'`, Q must contain an orthogonal/unitary matrix Q_1 on entry, and the product Q_1*Q is returned.
compz	`CHARACTER*1`. Must be `'N'`, `'I'`, or `'V'`. If *compz* = `'N'`, matrix Z is not computed. If *compz* = `'I'`, Z is initialized to the unit matrix, and the orthogonal/unitary matrix Z is returned; If *compz* = `'V'`, Z must contain an orthogonal/unitary matrix Z_1 on entry, and the product Z_1*Z is returned.
n	`INTEGER`. The order of the matrices A and B ($n \geq 0$).
ilo, *ihi*	`INTEGER`. *ilo* and *ihi* mark the rows and columns of A which are to be reduced. It is assumed that A is already upper triangular in rows and columns 1:*ilo*-1 and *ihi*+1:*n*. Values of *ilo* and *ihi* are normally set by a previous call to `?ggbal`; otherwise they should be set to 1 and *n* respectively.

Constraint:
If $n > 0$, then $1 \le ilo \le ihi \le n$;
if $n = 0$, then $ilo = 1$ and $ihi = 0$.

a, *b*, *q*, *z*
`REAL` for `sgghrd`
`DOUBLE PRECISION` for `dgghrd`
`COMPLEX` for `cgghrd`
`DOUBLE COMPLEX` for `zgghrd`.
Arrays:
a(*lda*,*) contains the n-by-n general matrix A. The second dimension of *a* must be at least max(1, n).
b(*ldb*,*) contains the n-by-n upper triangular matrix B.
The second dimension of *b* must be at least max(1, n).
q (*ldq*,*)
If *compq* = `'N'`, then *q* is not referenced.
If *compq* = `'V'`, then *q* must contain the orthogonal/unitary matrix Q_1, typically from the QR factorization of B.
The second dimension of *q* must be at least max(1, n).
z (*ldz*,*)
If *compq* = `'N'`, then *z* is not referenced.
If *compq* = `'V'`, then *z* must contain the orthogonal/unitary matrix Z_1.
The second dimension of *z* must be at least max(1, n).

lda
`INTEGER`. The first dimension of *a*; at least max(1, n).

ldb
`INTEGER`. The first dimension of *b*; at least max(1, n).

ldq
`INTEGER`. The first dimension of *q*;
If *compq* = `'N'`, then $ldq \ge 1$.
If *compq* = `'I'`or `'V'`, then $ldq \ge$ max(1, n).

ldz
`INTEGER`. The first dimension of *z*;
If *compq* = `'N'`, then $ldz \ge 1$.
If *compq* = `'I'`or `'V'`, then $ldz \ge$ max(1, n).

Output Parameters

a
On exit, the upper triangle and the first subdiagonal of A are overwritten with the upper Hessenberg matrix H, and the rest is set to zero.

b	On exit, overwritten by the upper triangular matrix $T = Q^H * B * Z$. The elements below the diagonal are set to zero.
q	If *compq* = 'I', then *q* contains the orthogonal/unitary matrix Q, ; If *compq* = 'V', then *q* is overwritten by the product $Q_1 * Q$.
z	If *compq* = 'I', then *z* contains the orthogonal/unitary matrix Z; If *compq* = 'V', then *z* is overwritten by the product $Z_1 * Z$.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gghrd interface are the following:

a	Holds the matrix A of size (n,n).
b	Holds the matrix B of size (n,n).
q	Holds the matrix Q of size (n,n).
z	Holds the matrix Z of size (n,n).
ilo	Default value for this argument is *ilo* = 1.
ihi	Default value for this argument is *ihi* = *n*.
compq	If omitted, this argument is restored based on the presence of argument *q* as follows: *compq* = 'I', if *q* is present, *compq* = 'N', if *q* is omitted. If present, *compq* must be equal to 'I' or 'V' and the argument *q* must also be present. Note that there will be an error condition if *compq* is present and *q* omitted.
compz	If omitted, this argument is restored based on the presence of argument *z* as follows: *compz* = 'I', if *z* is present, *compz* = 'N', if *z* is omitted. If present, *compz* must be equal to 'I' or 'V' and the argument *z* must also be present. Note that there will be an error condition if *compz* is present and *z* omitted.

?ggbal

Balances a pair of general real or complex matrices.

Syntax

Fortran 77:

```
call sggbal(job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale, work, info)
call dggbal(job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale, work, info)
call cggbal(job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale, work, info)
call zggbal(job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale, work, info)
```

Fortran 95:

```
call ggbal(a, b [,ilo] [,ihi] [,lscale] [,rscale] [,job] [,info])
```

Description

This routine balances a pair of general real/complex matrices (*A*,*B*). This involves, first, permuting *A* and *B* by similarity transformations to isolate eigenvalues in the first 1 to *ilo*-1 and last *ihi*+1 to *n* elements on the diagonal;and second, applying a diagonal similarity transformation to rows and columns *ilo* to *ihi* to make the rows and columns as close in norm as possible. Both steps are optional. Balancing may reduce the 1-norm of the matrices, and improve the accuracy of the computed eigenvalues and/or eigenvectors in the generalized eigenvalue problem $Ax = \lambda Bx$.

Input Parameters

job	CHARACTER*1. Specifies the operations to be performed on *A* and *B*. Must be 'N' or 'P' or 'S' or 'B'. If *job* = 'N', then no operations are done; simply set *ilo* =1, *ihi*=*n*, *lscale*(i) =1.0 and *rscale*(i)=1.0 for i = 1,..., *n*. If *job* = 'P', then permute only. If *job* = 'S', then scale only. If *job* = 'B', then both permute and scale.
n	INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).
a, *b*	REAL for sggbal

DOUBLE PRECISION for dggbal
COMPLEX for cggbal
DOUBLE COMPLEX for zggbal.
Arrays:
a(*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*).
b(*ldb*,*) contains the matrix *B*. The second dimension of *b* must be at least max(1, *n*).

lda — INTEGER. The first dimension of *a*; at least max(1, *n*).

ldb — INTEGER. The first dimension of *b*; at least max(1, *n*).

work — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Workspace array, DIMENSION at least max(1, 6*n*) when *job* = 'S'or 'B', or at least 1 when *job* = 'N'or 'P'.

Output Parameters

a, *b* — Overwritten by the balanced matrices *A* and *B*, respectively. If *job* = 'N', *a* and *b* are not referenced.

ilo, *ihi* — INTEGER. *ilo* and *ihi* are set to integers such that on exit *a*(*i*,*j*)=0 and *b*(*i*,*j*)=0 if *i*>*j* and *j*=1,...,*ilo*-1 or *i*=*ihi*+1,..., *n*.
If *job* = 'N'or 'S', then *ilo* = 1 and *ihi* = *n*.

lscale, *rscale* — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least max(1, *n*).
lscale contains details of the permutations and scaling factors applied to the left side of *A* and *B*.
If P_j is the index of the row interchanged with row *j*, and D_j is the scaling factor applied to row *j*, then
lscale(*J*) = P_j, for *j* = 1,..., *ilo*-1
= D_j, for *j* = *ilo*,...,*ihi*
= P_j, for *j* = *ihi*+1,..., *n*.
rscale contains details of the permutations and scaling factors applied to the right side of *A* and *B*.
If P_j is the index of the column interchanged with column *j*, and D_j is the scaling factor applied to column *j*, then

	rscale(*j*) = P_j, for *j* = 1,..., *ilo*-1 = D_j, for *j* = *ilo*,...,*ihi* = P_j, for *j* = *ihi*+1,..., *n* The order in which the interchanges are made is *n* to *ihi*+1, then 1 to *ilo*-1.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ggbal interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
b	Holds the matrix *B* of size (*n*,*n*).
lscale	Holds the vector of length (*n*).
rscale	Holds the vector of length (*n*).
ilo	Default value for this argument is *ilo* = 1.
ihi	Default value for this argument is *ihi* = *n*.
job	Must be 'B', 'S', 'P', or 'N'. The default value is 'B'.

?ggbak

Forms the right or left eigenvectors of a generalized eigenvalue problem.

Syntax

Fortran 77:

```
call sggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call dggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call cggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call zggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
```

Fortran 95:

```
call ggbak(v [, ilo] [,ihi] [,lscale] [,rscale] [,job] [,info])
```

Description

This routine forms the right or left eigenvectors of a real/complex generalized eigenvalue problem

$Ax = \lambda Bx$

by backward transformation on the computed eigenvectors of the balanced pair of matrices output by ?ggbal.

Input Parameters

job	CHARACTER*1. Specifies the type of backward transformation required. Must be 'N', 'P', 'S', or 'B'. If *job* = 'N', then no operations are done; return. If *job* = 'P', then do backward transformation for permutation only. If *job* = 'S', then do backward transformation for scaling only. If *job* = 'B', then do backward transformation for both permutation and scaling. This argument must be the same as the argument *job* supplied to ?ggbal.
side	CHARACTER*1. Must be 'L' or 'R'. If *side* = 'L' , then *v* contains left eigenvectors. If *side* = 'R' , then *v* contains right eigenvectors.
n	INTEGER. The number of rows of the matrix *V* (*n* ≥ 0).
ilo, *ihi*	INTEGER. The integers *ilo* and *ihi* determined by ?gebal. Constraint: If *n* > 0, then 1 ≤ *ilo* ≤ *ihi* ≤ *n*; if *n* = 0, then *ilo* = 1 and *ihi* = 0.
lscale, *rscale*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1, *n*). The array *lscale* contains details of the permutations and/or scaling factors applied to the left side of *A* and *B*, as returned by ?ggbal.

	The array *rscale* contains details of the permutations and/or scaling factors applied to the right side of *A* and *B*, as returned by ?ggbal.
m	INTEGER. The number of columns of the matrix *V* ($m \geq 0$).
v	REAL for sggbak DOUBLE PRECISION for dggbak COMPLEX for cggbak DOUBLE COMPLEX for zggbak. Array *v*(*ldv*,*). Contains the matrix of right or left eigenvectors to be transformed, as returned by ?tgevc. The second dimension of *v* must be at least max(1, *m*).
ldv	INTEGER. The first dimension of *v*; at least max(1, *n*).

Output Parameters

v	Overwritten by the transformed eigenvectors
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ggbak interface are the following:

v	Holds the matrix *V* of size (*n*,*m*).
lscale	Holds the vector of length (*n*).
rscale	Holds the vector of length (*n*).
ilo	Default value for this argument is *ilo* = 1.
ihi	Default value for this argument is *ihi* = *n*.
job	Must be 'B', 'S', 'P', or 'N'. The default value is 'B'.
side	If omitted, this argument is restored based on the presence of arguments *lscale* and *rscale* as follows: *side* = 'L', if *lscale* is present and *rscale* omitted,

side = 'R', if *lscale* is omitted and *rscale* present.
Note that there will be an error condition if both *lscale* and *rscale* are present or if they both are omitted.

?hgeqz

Implements the QZ method for finding the generalized eigenvalues of the matrix pair (H,T).

Syntax

Fortran 77:

```
call shgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alphar, alphai,
beta, q, ldq, z, ldz, work, lwork, info)
```

```
call dhgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alphar, alphai,
beta, q, ldq, z, ldz, work, lwork, info)
```

```
call chgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alpha, beta, q,
ldq, z, ldz, work, lwork, rwork, info)
```

```
call zhgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alpha, beta, q,
ldq, z, ldz, work, lwork, rwork, info)
```

Fortran 95:

```
call hgeqz(h, t [,ilo] [,ihi] [,alphar] [,alphai] [,beta] [,q] [,z] [,job]
[,compq] [,compz] [,info])
```

```
call hgeqz(h, t [,ilo] [,ihi] [,alpha] [,beta] [,q] [,z] [,job] [,compq] [,
compz] [,info])
```

Description

This routine computes the eigenvalues of a real/complex matrix pair (*H*,*T*), where *H* is an upper Hessenberg matrix and *T* is upper triangular, using the double-shift version (for real flavors) or single-shift version (for complex flavors) of the *QZ* method. Matrix pairs of this type are produced by the reduction to generalized upper Hessenberg form of a real/complex matrix pair (*A*,*B*):

$A = Q_1 * H * Z_1^H, B = Q_1 * T * Z_1^H,$

as computed by ?gghrd.

For real flavors:

If *job* = 'S', then the Hessenberg-triangular pair (*H*,*T*) is also reduced to generalized Schur form,

$H = Q*S*Z^T$, $T = Q*P*Z^T$,

where *Q* and *Z* are orthogonal matrices, *P* is an upper triangular matrix, and *S* is a quasi-triangular matrix with 1-by-1 and 2-by-2 diagonal blocks. The 1-by-1 blocks correspond to real eigenvalues of the matrix pair (*H*,*T*) and the 2-by-2 blocks correspond to complex conjugate pairs of eigenvalues.

Additionally, the 2-by-2 upper triangular diagonal blocks of *P* corresponding to 2-by-2 blocks of *S* are reduced to positive diagonal form, that is, if *S*(j+1,j) is non-zero, then *P*(j+1,j) = *P*(j,j+1) = 0, *P*(j,j) > 0, and *P*(j+1,j+1) > 0.

For complex flavors:

If *job* = 'S', then the Hessenberg-triangular pair (*H*,*T*) is also reduced to generalized Schur form,

$H = Q*\ S*Z^H$, $T = Q*P*Z^H$,

where *Q* and *Z* are unitary matrices, and *S* and *P* are upper triangular.

For all function flavors:

Optionally, the orthogonal/unitary matrix *Q* from the generalized Schur factorization may be postmultiplied into an input matrix Q_1, and the orthogonal/unitary matrix *Z* may be postmultiplied into an input matrix Z_1.

If Q_1 and Z_1 are the orthogonal/unitary matrices from ?gghrd that reduced the matrix pair (*A*,*B*) to generalized upper Hessenberg form, then the output matrices Q_1Q and Z 1Z are the orthogonal/unitary factors from the generalized Schur factorization of (*A*,*B*):

$A = (Q_1Q)*S\ *(Z_1Z)^H$, $B = (Q_1Q)\ *P\ *(Z_1Z)^H$.

To avoid overflow, eigenvalues of the matrix pair (*H*,*T*) (equivalently, of (*A*,*B*)) are computed as a pair of values (*alpha*,*beta*>). For chgeqz/zhgeqz, *alpha* and *beta* are complex, and for shgeqz/dhgeqz, *alpha* is complex and *beta* real. If *beta* is nonzero, λ = *alpha* / *beta* is an eigenvalue of the generalized nonsymmetric eigenvalue problem (GNEP)

$Ax = \lambda Bx$

and if *alpha* is nonzero, μ = *beta* / *alpha* is an eigenvalue of the alternate form of the GNEP

$\mu Ay = By$.

Real eigenvalues (for real flavors) or the values of *alpha* and *beta* for the i-th eigenvalue (for complex flavors) can be read directly from the generalized Schur form:

alpha = *S*(i,i), *beta* = *P*(i,i).

Input Parameters

job	CHARACTER*1. Specifies the operations to be performed. Must be 'E' or 'S'. If *job* = 'E', then compute eigenvalues only; If *job* = 'S', then compute eigenvalues and the Schur form.
compq	CHARACTER*1. Must be 'N', 'I', or 'V'. If *compq* = 'N', left Schur vectors (*q*) are not computed; If *compq* = 'I', *q* is initialized to the unit matrix and the matrix of left Schur vectors of (*H*,*T*) is returned; If *compq* = 'V', *q* must contain an orthogonal/unitary matrix Q_1 on entry and the product Q_1*Q is returned.
compz	CHARACTER*1. Must be 'N', 'I', or 'V'. If *compz* = 'N', right Schur vectors (*z*) are not computed; If *compz* = 'I', *z* is initialized to the unit matrix and the matrix of right Schur vectors of (*H*,*T*) is returned; If *compz* = 'V', *z* must contain an orthogonal/unitary matrix Z_1 on entry and the product Z_1*Z is returned.
n	INTEGER. The order of the matrices *H*, *T*, *Q*, and *Z* ($n \geq 0$).
ilo, *ihi*	INTEGER. *ilo* and *ihi* mark the rows and columns of *H* which are in Hessenberg form. It is assumed that *H* is already upper triangular in rows and columns 1:*ilo*-1 and *ihi*+1:*n*. Constraint: If $n > 0$, then $1 \leq ilo \leq ihi \leq n$; if $n = 0$, then $ilo = 1$ and $ihi = 0$.
h, *t*, *q*, *z*, *work*	REAL for shgeqz DOUBLE PRECISION for dhgeqz COMPLEX for chgeqz

DOUBLE COMPLEX for zhgeqz.
Arrays:
On entry, *h*(*ldh*,*) contains the *n*-by-*n* upper Hessenberg matrix *H*.
The second dimension of *h* must be at least max(1, *n*).
On entry, *t*(*ldt*,*) contains the *n*-by-*n* upper triangular matrix *T*.
The second dimension of *t* must be at least max(1, *n*).
q (*ldq*,*):
On entry, if *compq* = 'V', this array contains the orthogonal/unitary matrix Q_1 used in the reduction of (*A*,*B*) to generalized Hessenberg form.
If *compq* = 'N', then *q* is not referenced.
The second dimension of *q* must be at least max(1, *n*).
z (*ldz*,*):
On entry, if *compz* = 'V', this array contains the orthogonal/unitary matrix Z_1 used in the reduction of (*A*,*B*) to generalized Hessenberg form.
If *compz* = 'N', then *z* is not referenced.
The second dimension of *z* must be at least max(1, *n*).
work is a workspace array, its dimension max(1, *lwork*).

ldh INTEGER. The first dimension of *h*; at least max(1, *n*).

ldt INTEGER. The first dimension of *t*; at least max(1, *n*).

ldq INTEGER. The first dimension of *q*;

If *compq* = 'N', then *ldq* ≥ 1.

If *compq* = 'I'or 'V', then *ldq* ≥ max(1, *n*).

ldz INTEGER. The first dimension of *z*;

If *compq* = 'N', then *ldz* ≥ 1.

If *compq* = 'I'or 'V', then *ldz* ≥ max(1, *n*).

lwork INTEGER. The dimension of the array *work*.

lwork ≥ max(1, *n*).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for details.

rwork — REAL for chgeqz
DOUBLE PRECISION for zhgeqz.
Workspace array, DIMENSION at least max(1, *n*). Used in complex flavors only.

Output Parameters

h — For real flavors:
If *job* = 'S', then, on exit, *h* contains the upper quasi-triangular matrix *S* from the generalized Schur factorization; 2-by-2 diagonal blocks (corresponding to complex conjugate pairs of eigenvalues) are returned in standard form, with *h*(i,i) = *h*(i+1, i+1) and *h*(i+1, i) * *h*(i, i+1) < 0.
If *job* = 'E', then on exit the diagonal blocks of *h* match those of *S*, but the rest of *h* is unspecified.
For complex flavors:
If *job* = 'S', then, on exit, *h* contains the upper triangular matrix *S* from the generalized Schur factorization.
If *job* = 'E', then on exit the diagonal of *h* matches that of *S*, but the rest of *h* is unspecified.

t — If *job* = 'S', then, on exit, *t* contains the upper triangular matrix *P* from the generalized Schur factorization.
For real flavors:
2-by-2 diagonal blocks of *P* corresponding to 2-by-2 blocks of *S* are reduced to positive diagonal form, that is, if *h*(j+1,j) is non-zero, then *t*(j+1,j)=*t*(j,j+1)=0 and *t*(j,j) and *t*(j+1,j+1) will be positive.
If *job* = 'E', then on exit the diagonal blocks of *t* match those of *P*, but the rest of *t* is unspecified.
For complex flavors:
if *job* = 'E', then on exit the diagonal of *t* matches that of *P*, but the rest of *t* is unspecified.

alphar, *alphai*
REAL for shgeqz;
DOUBLE PRECISION for dhgeqz.
Arrays, DIMENSION at least max(1, *n*). The real and imaginary parts, respectively, of each scalar *alpha* defining an eigenvalue of GNEP.
If *alphai*(j) is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-th eigenvalues are a complex conjugate pair, with
alphai(j+1) = -*alphai*(j).

alpha
COMPLEX for chgeqz;
DOUBLE COMPLEX for zhgeqz.
Array, DIMENSION at least max(1, *n*).
The complex scalars *alpha* that define the eigenvalues of GNEP. *alphai*(i) = *S*(i,i) in the generalized Schur factorization.

beta
REAL for shgeqz
DOUBLE PRECISION for dhgeqz
COMPLEX for chgeqz
DOUBLE COMPLEX for zhgeqz.
Array, DIMENSION at least max(1, *n*).
For real flavors:
The scalars *beta* that define the eigenvalues of GNEP.
Together, the quantities *alpha* = (*alphar*(j), *alphai*(j)) and *beta* = *beta*(j) represent the j-th eigenvalue of the matrix pair (*A*,*B*), in one of the forms *lambda* = *alpha*/*beta* or *mu* = *beta*/*alpha*. Since either *lambda* or *mu* may overflow, they should not, in general, be computed.
For complex flavors:
The real non-negative scalars *beta* that define the eigenvalues of GNEP.
beta(i) = *P*(i,i) in the generalized Schur factorization.
Together, the quantities *alpha* = *alpha*(j) and *beta* = *beta*(j) represent the j-th eigenvalue of the matrix pair (*A*,*B*), in one of the forms *lambda* = *alpha*/*beta* or *mu* = *beta*/*alpha*. Since either *lambda* or *mu* may overflow, they should not, in general, be computed.

q	On exit, if *compq* = 'I', *q* is overwritten by the orthogonal/unitary matrix of left Schur vectors of the pair (*H*,*T*), and if *compq* = 'V', *q* is overwritten by the orthogonal/unitary matrix of left Schur vectors of (*A*,*B*).
z	On exit, if *compz* = 'I', *z* is overwritten by the orthogonal/unitary matrix of right Schur vectors of the pair (*H*,*T*), and if *compz* = 'V', *z* is overwritten by the orthogonal/unitary matrix of right Schur vectors of (*A*,*B*).
work(1)	If *info* ≥ 0, on exit, *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = 1,..., *n*, the *QZ* iteration did not converge. (*H*,*T*) is not in Schur form, but *alphar*(i), *alphai*(i) (for real flavors), *alpha*(i) (for complex flavors), and *beta*(i), i=*info*+1,..., *n* should be correct. If *info* = *n*+1,...,2*n*, the shift calculation failed. (*H*,*T*) is not in Schur form, but *alphar*(i), *alphai*(i) (for real flavors), *alpha*(i) (for complex flavors), and *beta*(i), i =*info*-*n*+1,..., *n* should be correct.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hgeqz interface are the following:

h	Holds the matrix *H* of size (*n*,*n*).
t	Holds the matrix *T* of size (*n*,*n*).
alphar	Holds the vector of length (*n*). Used in real flavors only.
alphai	Holds the vector of length (*n*). Used in real flavors only.
alpha	Holds the vector of length (*n*). Used in complex flavors only.
beta	Holds the vector of length (*n*).

q	Holds the matrix *Q* of size (*n*,*n*).
z	Holds the matrix *Z* of size (*n*,*n*).
ilo	Default value for this argument is *ilo* = 1.
ihi	Default value for this argument is *ihi* = *n*.
job	Must be 'E' or 'S'. The default value is 'E'.
compq	If omitted, this argument is restored based on the presence of argument *q* as follows: *compq* = 'I', if *q* is present, *compq* = 'N', if *q* is omitted. If present, *compq* must be equal to 'I' or 'V' and the argument *q* must also be present. Note that there will be an error condition if *compq* is present and *q* omitted.
compz	If omitted, this argument is restored based on the presence of argument *z* as follows: *compz* = 'I', if *z* is present, *compz* = 'N', if *z* is omitted. If present, *compz* must be equal to 'I' or 'V' and the argument *z* must also be present. Note an error condition if *compz* is present and *z* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tgevc

Computes some or all of the right and/or left generalized eigenvectors of a pair of upper triangular matrices.

Syntax

Fortran 77:

```
call stgevc(side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr, ldvr, mm, m, work, info)
call dtgevc(side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr, ldvr, mm, m, work, info)
call ctgevc(side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr, ldvr, mm, m, work, rwork, info)
call ztgevc(side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr, ldvr, mm, m, work, rwork, info)
```

Fortran 95:

```
call tgevc(s, p [,howmny] [,select] [,vl] [,vr] [,m] [,info])
```

Description

This routine computes some or all of the right and/or left eigenvectors of a pair of real/complex matrices (S,P), where S is quasi-triangular (for real flavors) or upper triangular (for complex flavors) and P is upper triangular.

Matrix pairs of this type are produced by the generalized Schur factorization of a real/complex matrix pair (A,B):

$A = Q*S*Z^H$, $B = Q*P*Z^H$

as computed by ?gghrd plus ?hgeqz.

The right eigenvector x and the left eigenvector y of (S,P) corresponding to an eigenvalue w are defined by:

$S*x = w*P*x$, $y^H*S = w*y^H*P$

The eigenvalues are not input to this routine, but are computed directly from the diagonal blocks or diagonal elements of S and P.

This routine returns the matrices *X* and/or *Y* of right and left eigenvectors of (*S*,*P*), or the products *Z***X* and/or *Q***Y*, where *Z* and *Q* are input matrices.

If *Q* and *Z* are the orthogonal/unitary factors from the generalized Schur factorization of a matrix pair (*A*,*B*), then *Z***X* and *Q***Y* are the matrices of right and left eigenvectors of (*A*,*B*).

Input Parameters

side	CHARACTER*1. Must be 'R', 'L', or 'B'. If *side* = 'R' , compute right eigenvectors only. If *side* = 'L' , compute left eigenvectors only. If *side* = 'B', compute both right and left eigenvectors.
howmny	CHARACTER*1. Must be 'A' , 'B', or 'S'. If *howmny* = 'A' , compute all right and/or left eigenvectors. If *howmny* = 'B' , compute all right and/or left eigenvectors, backtransformed by the matrices in *vr* and/or *vl*. If *howmny* = 'S' , compute selected right and/or left eigenvectors, specified by the logical array *select*.
select	LOGICAL. Array, DIMENSION at least max (1, *n*). If *howmny* = 'S', *select* specifies the eigenvectors to be computed. If *howmny* = 'A'or 'B', *select* is not referenced. For real flavors: If *omega*(*j*) is a real eigenvalue, the corresponding real eigenvector is computed if *select*(*j*) is .TRUE.. If *omega*(*j*) and *omega*(*j*+1) are the real and imaginary parts of a complex eigenvalue, the corresponding complex eigenvector is computed if either *select*(*j*) or *select*(*j*+1) is .TRUE., and on exit *select*(*j*) is set to .TRUE.and *select*(*j*+1) is set to .FALSE.. For complex flavors: The eigenvector corresponding to the *j*-th eigenvalue is computed if *select*(*j*) is .TRUE..
n	INTEGER. The order of the matrices *S* and *P* (*n* ≥ 0).
s, *p*, *vl*, *vr*, *work*	REAL for stgevc

DOUBLE PRECISION for dtgevc
COMPLEX for ctgevc
DOUBLE COMPLEX for ztgevc.
Arrays:
s(*lds*,*) contains the matrix *S* from a generalized Schur factorization as computed by ?hgeqz. This matrix is upper quasi-triangular for real flavors, and upper triangular for complex flavors.
The second dimension of *s* must be at least max(1, *n*).
p(*ldp*,*) contains the upper triangular matrix *P* from a generalized Schur factorization as computed by ?hgeqz. For real flavors, 2-by-2 diagonal blocks of *P* corresponding to 2-by-2 blocks of *S* must be in positive diagonal form. For complex flavors, *P* must have real diagonal elements.
The second dimension of *p* must be at least max(1, *n*).
If *side* = 'L' or 'B' and *howmny* = 'B', *vl*(*ldvl*,*) must contain an *n*-by-*n* matrix *Q* (usually the orthogonal/unitary matrix *Q* of left Schur vectors returned by ?hgeqz). The second dimension of *vl* must be at least max(1, *mm*).
If *side* = 'R' , *vl* is not referenced.
If *side* = 'R' or 'B' and *howmny* = 'B', *vr*(*ldvr*,*) must contain an *n*-by-*n* matrix *Z* (usually the orthogonal/unitary matrix *Z* of right Schur vectors returned by ?hgeqz). The second dimension of *vr* must be at least max(1, *mm*).
If *side* = 'L', *vr* is not referenced.
work(*) is a workspace array.
DIMENSION at least max (1, 6**n*) for real flavors and at least max (1, 2**n*) for complex flavors.

lds — INTEGER. The first dimension of *s*; at least max(1, *n*).

ldp — INTEGER. The first dimension of *p*; at least max(1, *n*).

ldvl — INTEGER. The first dimension of *vl*;
If *side* = 'L' or 'B', then *ldvl* ≥*n*.
If *side* = 'R', then *ldvl* ≥ 1.

ldvr — INTEGER. The first dimension of *vr*;
If *side* = 'R' or 'B', then *ldvr* ≥*n*.
If *side* = 'L', then *ldvr* ≥ 1.

mm INTEGER. The number of columns in the arrays *vl* and/or *vr* (*mm* ≥ *m*).

rwork REAL for ctgevc DOUBLE PRECISION for ztgevc. Workspace array, DIMENSION at least max (1, 2**n*). Used in complex flavors only.

Output Parameters

vl On exit, if *side* = 'L' or 'B', *vl* contains:
if *howmny* = 'A', the matrix *Y* of left eigenvectors of (*S*,*P*);
if *howmny* = 'B', the matrix *Q***Y*;
if *howmny* = 'S', the left eigenvectors of (*S*,*P*) specified by *select*, stored consecutively in the columns of *vl*, in the same order as their eigenvalues.
For real flavors:
A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part, and the second the imaginary part.

vr On exit, if *side* = 'R' or 'B', *vr* contains:
if *howmny* = 'A', the matrix *X* of right eigenvectors of (*S*,*P*);
if *howmny* = 'B', the matrix *Z***X*;
if *howmny* = 'S', the right eigenvectors of (*S*,*P*) specified by *select*, stored consecutively in the columns of *vr*, in the same order as their eigenvalues.
For real flavors:
A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part, and the second the imaginary part.

m INTEGER. The number of columns in the arrays *vl* and/or *vr* actually used to store the eigenvectors.
If *howmny* = 'A' or 'B', *m* is set to *n*.
For real flavors:
Each selected real eigenvector occupies one column and each selected complex eigenvector occupies two columns.
For complex flavors:
Each selected eigenvector occupies one column.

info INTEGER.
If *info* = 0, the execution is successful.

If *info* = *-i*, the *i*-th parameter had an illegal value.
For real flavors:
if *info* = *i*>0, the 2-by-2 block (*i*:*i*+1) does not have a complex eigenvalue.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `tgevc` interface are the following:

s — Holds the matrix *S* of size (*n*,*n*).

p — Holds the matrix *P* of size (*n*,*n*).

select — Holds the vector of length (*n*).

vl — Holds the matrix *VL* of size (*n*,*mm*).

vr — Holds the matrix *VR* of size (*n*,*mm*).

side — Restored based on the presence of arguments *vl* and *vr* as follows:
side = 'B', if both *vl* and *vr* are present,
side = 'L', if *vl* is present and *vr* omitted,
side = 'R', if *vl* is omitted and *vr* present,
Note that there will be an error condition if both *vl* and *vr* are omitted.

howmny — If omitted, this argument is restored based on the presence of argument *select* as follows:
howmny = 'S', if *select* is present,
howmny = 'A', if *select* is omitted.
If present, *howmny* must be equal to 'A' or 'B' and the argument *select* must be omitted.
Note that there will be an error condition if both *howmny* and *select* are present.

?tgexc

Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that one diagonal block of (A,B) moves to another row index.

Syntax

Fortran 77:

```
call stgexc(wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, ifst, ilst, work, lwork, info)
call dtgexc(wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, ifst, ilst, work, lwork, info)
call ctgexc(wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, ifst, ilst, info)
call ztgexc(wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, ifst, ilst, info)
```

Fortran 95:

```
call tgexc(a, b [,ifst] [,ilst] [,z] [,q] [,info])
```

Description

This routine reorders the generalized real-Schur/Schur decomposition of a real/complex matrix pair (*A*,*B*) using an orthogonal/unitary equivalence transformation

$(A, B) = Q\ (A, B)\ Z^H$,

so that the diagonal block of (*A*, *B*) with row index *ifst* is moved to row *ilst*. Matrix pair (*A*, *B*) must be in a generalized real-Schur/Schur canonical form (as returned by ?gges), that is, *A* is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks and *B* is upper triangular. Optionally, the matrices *Q* and *Z* of generalized Schur vectors are updated.

Q(in) * *A*(in) * *Z*(in)' = *Q*(out) * *A*(out) * *Z*(out)'

Q(in) * *B*(in) * *Z*(in)' = *Q*(out) * *B*(out) * *Z*(out)'.

Input Parameters

wantq, *wantz*	LOGICAL. If *wantq* = .TRUE., update the left transformation matrix *Q*; If *wantq* = .FALSE., do not update *Q*;

	If *wantz* = .TRUE., update the right transformation matrix *Z*; If *wantz* = .FALSE., do not update *Z*.
n	INTEGER. The order of the matrices *A* and *B* (*n* ≥ 0).
a, *b*, *q*,	REAL for stgexc DOUBLE PRECISION for dtgexc COMPLEX for ctgexc DOUBLE COMPLEX for ztgexc. Arrays: *a*(*lda*,*) contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the matrix *B*. The second dimension of *b* must be at least max(1, *n*). *q* (*ldq*,*) If *wantq* = .FALSE., then *q* is not referenced. If *wantq* = .TRUE., then *q* must contain the orthogonal/unitary matrix *Q*. The second dimension of *q* must be at least max(1, *n*). *z* (*ldz*,*) If *wantz* = .FALSE., then *z* is not referenced. If *wantz* = .TRUE., then *z* must contain the orthogonal/unitary matrix *Z*. The second dimension of *z* must be at least max(1, *n*).
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).
ldb	INTEGER. The first dimension of *b*; at least max(1, *n*).
ldq	INTEGER. The first dimension of *q*; If *wantq* = .FALSE., then *ldq* ≥ 1. If *wantq* = .TRUE., then *ldq* ≥ max(1, *n*).
ldz	INTEGER. The first dimension of *z*; If *wantz* = .FALSE., then *ldz* ≥ 1. If *wantz* = .TRUE., then *ldz* ≥ max(1, *n*).
ifst, *ilst*	INTEGER. Specify the reordering of the diagonal blocks of (*A*, *B*). The block with row index *ifst* is moved to row *ilst*, by a sequence of swapping between adjacent blocks. Constraint: 1 ≤ *ifst*, *ilst* ≤ *n*.

work	REAL for stgexc; DOUBLE PRECISION for dtgexc. Workspace array, DIMENSION (*lwork*). Used in real flavors only.
lwork	INTEGER. The dimension of *work*; must be at least $4n + 16$. If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for details.

Output Parameters

a, *b*	Overwritten by the updated matrices *A* and *B*.
ifst, *ilst*	Overwritten for real flavors only. If *ifst* pointed to the second row of a 2 by 2 block on entry, it is changed to point to the first row; *ilst* always points to the first row of the block in its final position (which may differ from its input value by ±1).
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = 1, the transformed matrix pair (*A*, *B*) would be too far from generalized Schur form; the problem is ill-conditioned. (*A*, *B*) may have been partially reordered, and *ilst* points to the first row of the current position of the block being moved.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine tgexc interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
b	Holds the matrix *B* of size (*n*,*n*).
z	Holds the matrix *Z* of size (*n*,*n*).

q	Holds the matrix *Q* of size (*n*,*n*).
wantq	Restored based on the presence of the argument *q* as follows: *wantq* = .TRUE., if *q* is present, *wantq* = .FALSE., if *q* is omitted.
wantz	Restored based on the presence of the argument *z* as follows: *wantz* = .TRUE., if *z* is present, *wantz* = .FALSE., if z is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tgsen

Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that a selected cluster of eigenvalues appears in the leading diagonal blocks of (A,B).

Syntax

Fortran 77:

```
call stgsen(ijob, wantq, wantz, select, n, a, lda, b, ldb, alphar, alphai,
beta, q, ldq, z, ldz, m, pl, pr, dif, work, lwork, iwork, liwork, info)

call dtgsen(ijob, wantq, wantz, select, n, a, lda, b, ldb, alphar, alphai,
beta, q, ldq, z, ldz, m, pl, pr, dif, work, lwork, iwork, liwork, info)

call ctgsen(ijob, wantq, wantz, select, n, a, lda, b, ldb, alpha, beta, q,
ldq, z, ldz, m, pl, pr, dif, work, lwork, iwork, liwork, info)

call ztgsen(ijob, wantq, wantz, select, n, a, lda, b, ldb, alpha, beta, q,
ldq, z, ldz, m, pl, pr, dif, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call tgsen(a, b, select [,alphar] [,alphai] [,beta] [,ijob] [,q] [,z] [,pl]
[,pr] [,dif] [,m] [,info])

call tgsen(a, b, select [,alpha] [,beta] [,ijob] [,q] [,z] [,pl] [,pr] [, dif]
[,m] [,info])
```

Description

This routine reorders the generalized real-Schur/Schur decomposition of a real/complex matrix pair (*A*, *B*) (in terms of an orthogonal/unitary equivalence transformation *Q* *(*A*, *B*) * *Z*), so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the pair (*A*, *B*). The leading columns of *Q* and *Z* form orthonormal/unitary bases of the corresponding left and right eigenspaces (deflating subspaces).

(*A*, *B*) must be in generalized real-Schur/Schur canonical form (as returned by ?gges), that is, *A* and *B* are both upper triangular.

?tgsen also computes the generalized eigenvalues

ω_j = (*alphar*(j) + *alphai*(j)*i)/*beta*(j) (for real flavors)

ω_j = *alpha*(j)/*beta*(j) (for complex flavors)

of the reordered matrix pair (*A*, *B*).

Optionally, the routine computes the estimates of reciprocal condition numbers for eigenvalues and eigenspaces. These are Difu[(A_{11}, B_{11}), (A_{22}, B_{22})] and Difl[(A_{11}, B_{11}), (A_{22}, B_{22})], that is, the separation(s) between the matrix pairs (A_{11}, B_{11}) and (A_{22}, B_{22}) that correspond to the selected cluster and the eigenvalues outside the cluster, respectively, and norms of "projections" onto left and right eigenspaces with respect to the selected cluster in the (1,1)-block.

Input Parameters

ijob	INTEGER. Specifies whether condition numbers are required for the cluster of eigenvalues (*pl* and *pr*) or the deflating subspaces Difu and Difl. If *ijob* =0, only reorder with respect to *select*; If *ijob* =1, reciprocal of norms of "projections" onto left and right eigenspaces with respect to the selected cluster (*pl* and *pr*); If *ijob* =2, compute upper bounds on Difu and Difl, using F-norm-based estimate (*dif* (1:2)); If *ijob* =3, compute estimate of Difu and Difl, sing 1-norm-based estimate (*dif* (1:2)). This option is about 5 times as expensive as *ijob* =2; If *ijob* =4,>compute *pl*, *pr* and *dif* (i.e., options 0, 1 and 2 above). This is an economic version to get it all; If *ijob* =5, compute *pl*, *pr* and *dif* (i.e., options 0, 1 and 3 above).
wantq, *wantz*	LOGICAL. If *wantq* = .TRUE., update the left transformation matrix *Q*; If *wantq* = .FALSE., do not update *Q*; If *wantz* = .TRUE., update the right transformation matrix *Z*; If *wantz* = .FALSE., do not update *Z*.
select	LOGICAL. Array, DIMENSION at least max (1, *n*). Specifies the eigenvalues in the selected cluster.

	To select an eigenvalue *omega*(*j*), *select*(*j*) must be .TRUE. For real flavors: to select a complex conjugate pair of eigenvalues *omega*(*j*) and *omega*(*j*+1) (corresponding 2 by 2 diagonal block), *select*(*j*) and/or *select*(*j*+1) must be set to .TRUE.; the complex conjugate *omega*(*j*) and *omega*(*j*+1) must be either both included in the cluster or both excluded.
n	INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).
a, *b*, *q*, *z*, *work*	REAL for stgsen DOUBLE PRECISION for dtgsen COMPLEX for ctgsen DOUBLE COMPLEX for ztgsen. Arrays: *a*(*lda*,*) contains the matrix *A*. For real flavors: *A* is upper quasi-triangular, with (*A*, *B*) in generalized real Schur canonical form. For complex flavors: *A* is upper triangular, in generalized Schur canonical form. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the matrix *B*. For real flavors: *B* is upper triangular, with (*A*, *B*) in generalized real Schur canonical form. For complex flavors: *B* is upper triangular, in generalized Schur canonical form. The second dimension of *b* must be at least max(1, *n*). *q* (*ldq*,*) If *wantq* = .TRUE., then *q* is an *n*-by-*n* matrix; If *wantq* = .FALSE., then *q* is not referenced. The second dimension of *q* must be at least max(1, *n*). *z* (*ldz*,*) If *wantz* = .TRUE., then *z* is an *n*-by-*n* matrix; If *wantz* = .FALSE., then *z* is not referenced. The second dimension of *z* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*). If *ijob*=0, *work* is not referenced.
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).

ldb INTEGER. The first dimension of *b*; at least max(1, *n*).

ldq INTEGER. The first dimension of *q*; *ldq* ≥ 1.
If *wantq* = .TRUE., then *ldq* ≥ max(1, *n*).

ldz INTEGER. The first dimension of *z*; *ldz* ≥ 1.
If *wantz* = .TRUE., then *ldz* ≥ max(1, *n*).

lwork INTEGER. The dimension of the array *work*.
For real flavors:
If *ijob* = 1, 2, or 4, *lwork* ≥ max(4*n*+16, 2*m*(*n*-*m*)).
If *ijob* = 3 or 5, *lwork* ≥ max(4*n*+16, 4*m*(*n*-*m*)).
For complex flavors:
If *ijob* = 1, 2, or 4, *lwork* ≥ max(1, 2*m*(*n*-*m*)).
If *ijob* = 3 or 5, *lwork* ≥ max(1, 4*m*(*n*-*m*)).
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for details.

iwork INTEGER. Workspace array, its dimension max(1, *liwork*).
If *ijob* =0, *iwork* is not referenced.

liwork INTEGER. The dimension of the array *iwork*.
For real flavors:
If *ijob* = 1, 2, or 4, *liwork* ≥ *n*+6.
If *ijob* = 3 or 5, *liwork* ≥ max(*n*+6, 2*m*(*n*-*m*)).
For complex flavors:
If *ijob* = 1, 2, or 4, *liwork* ≥ *n*+2.
If *ijob* = 3 or 5, *liwork* ≥ max(*n*+2, 2*m*(*n*-*m*)).
If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by xerbla. See Application Notes for details.

Output Parameters

a, *b*	Overwritten by the reordered matrices *A* and *B*, respectively.
alphar, *alphai*	REAL for stgsen; DOUBLE PRECISION for dtgsen. Arrays, DIMENSION at least max(1, *n*). Contain values that form generalized eigenvalues in real flavors. See *beta*.
alpha	COMPLEX for ctgsen; DOUBLE COMPLEX for ztgsen. Array, DIMENSION at least max(1, *n*). Contain values that form generalized eigenvalues in complex flavors. See *beta*.
beta	REAL for stgsen DOUBLE PRECISION for dtgsen COMPLEX for ctgsen DOUBLE COMPLEX for ztgsen. Array, DIMENSION at least max(1, *n*). For real flavors: On exit, (*alphar*(j) + *alphai*(j)*i)/*beta*(j), j=1,..., *n*, will be the generalized eigenvalues. *alphar*(j) + *alphai*(j)*i and *beta*(j), j=1,..., *n* are the diagonals of the complex Schur form (*S*,*T*) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (*A*,*B*) were further reduced to triangular form using complex unitary transformations. If *alphai*(j) is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with *alphai*(j+1) negative. For complex flavors: The diagonal elements of *A* and *B*, respectively, when the pair (*A*,*B*) has been reduced to generalized Schur form. *alpha*(i)/*beta*(i), i=1,..., *n* are the generalized eigenvalues.
q	If *wantq* = .TRUE., then, on exit, *Q* has been postmultiplied by the left orthogonal transformation matrix which reorder (*A*, *B*). The leading *m* columns of *Q* form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces).

z	If *wantz* = .TRUE., then, on exit, *Z* has been postmultiplied by the left orthogonal transformation matrix which reorder (*A*, *B*). The leading *m* columns of *Z* form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces).
m	INTEGER. The dimension of the specified pair of left and right eigen-spaces (deflating subspaces); $0 \le m \le n$.
pl, *pr*	REAL for single precision flavors; DOUBLE PRECISION for double precision flavors. If *ijob* = 1, 4, or 5, *pl* and *pr* are lower bounds on the reciprocal of the norm of "projections" onto left and right eigenspaces with respect to the selected cluster. $0 < pl, pr \le 1$. If $m = 0$ or $m = n$, $pl = pr = 1$. If *ijob* = 0, 2 or 3, *pl* and *pr* are not referenced
dif	REAL for single precision flavors;DOUBLE PRECISION for double precision flavors. Array, DIMENSION (2). If *ijob* ≥ 2, *dif*(1:2) store the estimates ofDifu and Difl. If *ijob* = 2 or 4, *dif*(1:2) are F-norm-based upper bounds on Difu and Difl. If *ijob* = 3 or 5, *dif*(1:2) are 1-norm-based estimates of Difu and Difl. If $m = 0$ or n, *dif*(1:2) = F-norm([*A*, *B*]). If *ijob* = 0 or 1, *dif* is not referenced.
work(1)	If *ijob* is not 0 and *info* = 0, on exit, *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
iwork(1)	If *ijob* is not 0 and *info* = 0, on exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance. Use this *liwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = 1, Reordering of (*A*, *B*) failed because the transformed matrix pair (*A*, *B*) would be too far from generalized Schur form; the problem is very ill-conditioned. (*A*, *B*) may have been partially reordered.
If requested, 0 is returned in *dif*(*), *pl* and *pr*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `tgsen` interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
b	Holds the matrix *B* of size (*n*,*n*).
select	Holds the vector of length (*n*).
alphar	Holds the vector of length (*n*). Used in real flavors only.
alphai	Holds the vector of length (*n*). Used in real flavors only.
alpha	Holds the vector of length (*n*). Used in complex flavors only.
beta	Holds the vector of length (*n*).
q	Holds the matrix *Q* of size (*n*,*n*).
z	Holds the matrix *Z* of size (*n*,*n*).
dif	Holds the vector of length (2).
ijob	Must be 0, 1, 2, 3, 4, or 5. The default value is 0.
wantq	Restored based on the presence of the argument *q* as follows: *wantq* = .TRUE., if *q* is present, *wantq* = .FALSE., if *q* is omitted.
wantz	Restored based on the presence of the argument *z* as follows: *wantz* = .TRUE., if *z* is present, *wantz* = .FALSE., if z is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = –1 (*liwork* = –1).

If you choose the first option and set any of admissible *lwork* (or *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tgsyl

Solves the generalized Sylvester equation.

Syntax

Fortran 77:

```
call stgsyl(trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf,
scale, dif, work, lwork, iwork, info)

call dtgsyl(trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf,
scale, dif, work, lwork, iwork, info)

call ctgsyl(trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf,
scale, dif, work, lwork, iwork, info)

call ztgsyl(trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf,
scale, dif, work, lwork, iwork, info)
```

Fortran 95:

```
call tgsyl(a, b, c, d, e, f [,ijob] [,trans] [,scale] [,dif] [,info])
```

Description

This routine solves the generalized Sylvester equation:

*A R - L B = scale * C*

*D R - L E = scale * F*

where *R* and *L* are unknown *m*-by-*n* matrices, (*A*, *D*), (*B*, *E*) and (*C*, *F*) are given matrix pairs of size *m*-by-*m*, *n*-by-*n* and *m*-by-*n*, respectively, with real/complex entries. (*A*, *D*) and (*B*, *E*) must be in generalized real-Schur/Schur canonical form, that is, *A*, *B* are upper quasi-triangular/triangular and *D*, *E* are upper triangular.

The solution (*R*, *L*) overwrites (*C*, *F*). The factor *scale*, $0 \le scale \le 1$, is an output scaling factor chosen to avoid overflow.

In matrix notation the above equation is equivalent to the following: solve $Zx = scale * b$, where *Z* is defined as

$$Z = \begin{pmatrix} kron(I_n, A) - kron(B', I_m) \\ kron(I_n, D) - kron(E', I_m) \end{pmatrix}$$

Here I_k is the identity matrix of size *k* and *X*' is the transpose/conjugate-transpose of *X*. *kron*(*X*, *Y*) is the Kronecker product between the matrices *X* and *Y*.

If *trans* = 'T' (for real flavors), or *trans* = 'C' (for complex flavors), the routine ?tgsyl solves the transposed/conjugate-transposed system *Z* ' *y* = *scale* * *b*, which is equivalent to solve for *R* and *L* in

A' *R* + *D*' *L* = *scale* * *C*

R B' + *L E*' = *scale* * (-*F*)

This case (*trans* = 'T' for stgsyl/dtgsyl or *trans* = 'C' for ctgsyl/ztgsyl) is used to compute an one-norm-based estimate of Dif[(*A*, *D*), (*B*, *E*)], the separation between the matrix pairs (*A*,*D*) and (*B*,*E*), using slacon/clacon.

If *ijob* < 1, ?tgsyl computes a Frobenius norm-based estimate of Dif[(*A*, *D*), (*B*,*E*)]. That is, the reciprocal of a lower bound on the reciprocal of the smallest singular value of *Z*. This is a level 3 BLAS algorithm.

Input Parameters

trans — CHARACTER*1. Must be 'N', 'T', or 'C'.
If *trans* = 'N', solve the generalized Sylvester equation.
If *trans* = 'T', solve the 'transposed' system (for real flavors only).

	If *trans* = 'C', solve the ' conjugate transposed' system (for complex flavors only).
ijob	INTEGER. Specifies what kind of functionality to be performed: If *ijob* =0 , solve the generalized Sylvester equation only; If *ijob* =1, perform the functionality of *ijob* =0 and *ijob* =3; If *ijob* =2, perform the functionality of *ijob* =0 and *ijob* =4; If *ijob* =3, only an estimate of Dif[(*A*, *D*), (*B*, *E*)] is computed (look ahead strategy is used); If *ijob* =4, only an estimate of Dif[(*A*, *D*), (*B*,*E*)] is computed (?gecon on sub-systems is used). If *trans* = 'T' or 'C', *ijob* is not referenced.
m	INTEGER. The order of the matrices *A* and *D*, and the row dimension of the matrices *C*, *F*, *R* and *L*.
n	INTEGER. The order of the matrices *B* and *E*, and the column dimension of the matrices *C*, *F*, *R* and *L*.
a, *b*, *c*, *d*, *e*, *f*, *work*	REAL for stgsyl DOUBLE PRECISION for dtgsyl COMPLEX for ctgsyl DOUBLE COMPLEX for ztgsyl. Arrays: *a*(*lda*,*) contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix *A*. The second dimension of *a* must be at least max(1, *m*). *b*(*ldb*,*) contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix *B*. The second dimension of *b* must be at least max(1, *n*). *c* (*ldc*,*) contains the right-hand-side of the first matrix equation in the generalized Sylvester equation (as defined by *trans*) The second dimension of *c* must be at least max(1, *n*). *d* (*ldd*,*) contains the upper triangular matrix *D*. The second dimension of *d* must be at least max(1, *m*). *e* (*lde*,*) contains the upper triangular matrix *E*. The second dimension of *e* must be at least max(1, *n*).

	f (*ldf*,*) contains the right-hand-side of the second matrix equation in the generalized Sylvester equation (as defined by *trans*) The second dimension of *f* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
ldb	INTEGER. The first dimension of *b*; at least max(1, *n*).
ldc	INTEGER. The first dimension of *c*; at least max(1, *m*).
ldd	INTEGER. The first dimension of *d*; at least max(1, *m*).
lde	INTEGER. The first dimension of *e*; at least max(1, *n*).
ldf	INTEGER. The first dimension of *f*; at least max(1, *m*).
lwork	INTEGER. The dimension of the array *work*. *lwork* ≥ 1. If *ijob* = 1 or 2 and *trans* = 'N', *lwork* ≥ max(1, 2**m***n*). If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for details.
iwork	INTEGER. Workspace array, DIMENSION at least (*m*+*n*+6) for real flavors, and at least (*m*+*n*+2) for complex flavors. If *ijob*=0, *iwork* is not referenced.

Output Parameters

c	If *ijob*=0, 1, or 2, overwritten by the solution *R*. If *ijob*=3 or 4 and *trans* = 'N', *c* holds *R*, the solution achieved during the computation of the Dif-estimate.
f	If *ijob*=0, 1, or 2, overwritten by the solution *L*. If *ijob*=3 or 4 and *trans* = 'N', *f* holds *L*, the solution achieved during the computation of the Dif-estimate.
dif	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors.

	On exit, *dif* is the reciprocal of a lower bound of the reciprocal of the Dif-function, that is, *dif* is an upper bound of Dif[(*A*, *D*), (*B*, *E*)] = sigma_min(*Z*), where *Z* as in (2). If *ijob* = 0, or *trans* = 'T' (for real flavors), or *trans* = 'C' (for complex flavors), *dif* is not touched.
scale	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. On exit, *scale* is the scaling factor in the generalized Sylvester equation. If 0 < *scale* < 1, *c* and *f* hold the solutions *R* and *L*, respectively, to a slightly perturbed system but the input matrices *A*, *B*, *D* and *E* have not been changed. If *scale* = 0, *c* and *f* hold the solutions *R* and *L*, respectively, to the homogeneous system with *C* = *F* = 0. Normally, *scale* = 1.
work(1)	If *info* = 0, *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* > 0, (*A*, *D*) and (*B*, *E*) have common or close eigenvalues.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine tgsyl interface are the following:

a	Holds the matrix *A* of size (*m*,*m*).
b	Holds the matrix *B* of size (*n*,*n*).
c	Holds the matrix *C* of size (*m*,*n*).
d	Holds the matrix *D* of size (*m*,*m*).
e	Holds the matrix *E* of size (*n*,*n*).
f	Holds the matrix *F* of size (*m*,*n*).

ijob	Must be 0, 1, 2, 3, or 4. The default value is 0.
trans	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tgsna

Estimates reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a pair of matrices in generalized real Schur canonical form.

Syntax

Fortran 77:

```
call stgsna(job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr, ldvr, s,
dif, mm, m, work, lwork, iwork, info)
call dtgsna(job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr, ldvr, s,
dif, mm, m, work, lwork, iwork, info)
call ctgsna(job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr, ldvr, s,
dif, mm, m, work, lwork, iwork, info)
call ztgsna(job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr, ldvr, s,
dif, mm, m, work, lwork, iwork, info)
```

Fortran 95:

```
call tgsna(a, b [,s] [,dif] [,vl] [,vr] [,select] [,m] [,info])
```

Description

The real flavors stgsna/dtgsna of this routine estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (*A*, *B*) in generalized real Schur canonical form (or of any matrix pair ($Q*A*Z^T$, $Q*B*Z^T$) with orthogonal matrices *Q* and *Z*.

(*A*, *B*) must be in generalized real Schur form (as returned by sgges/dgges), that is, *A* is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks. B is upper triangular.

The complex flavors ctgsna/ztgsna estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (*A*, *B*). (*A*, *B*) must be in generalized Schur canonical form , that is, *A* and *B* are both upper triangular.

Input Parameters

job — CHARACTER*1. Specifies whether condition numbers are required for eigenvalues or eigenvectors. Must be 'E' or 'V' or 'B'.
If *job* = 'E', for eigenvalues only (compute *s*).
If *job* = 'V', for eigenvectors only (compute *dif*).
If *job* = 'B', for both eigenvalues and eigenvectors (compute both *s* and *dif*).

howmny — CHARACTER*1. Must be 'A' or 'S'.
If *howmny* = 'A' , compute condition numbers for all eigenpairs.
If *howmny* = 'S' , compute condition numbers for selected eigenpairs specified by the logical array *select*.

select — LOGICAL.
Array, DIMENSION at least max (1, *n*).
If *howmny* = 'S', *select* specifies the eigenpairs for which condition numbers are required.
If *howmny* = 'A', *select* is not referenced.
For real flavors:
To select condition numbers for the eigenpair corresponding to a real eigenvalue omega(*j*), *select*(*j*) must be set to .TRUE.; to select condition numbers corresponding to a

complex conjugate pair of eigenvalues omega(*j*) and omega(*j*+1), either *select*(*j*) or *select*(*j*+1) must be set to .TRUE.
For complex flavors:
To select condition numbers for the corresponding *j*-th eigenvalue and/or eigenvector, *select*(*j*) must be set to .TRUE..

n — INTEGER. The order of the square matrix pair (*A*, *B*) (*n* ≥ 0).

a, *b*, *vl*, *vr*, *work* — REAL for stgsna
DOUBLE PRECISION for dtgsna
COMPLEX for ctgsna
DOUBLE COMPLEX for ztgsna.
Arrays:
a(*lda*,*) contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix *A* in the pair (*A*, *B*).
The second dimension of *a* must be at least max(1, *n*).
b(*ldb*,*) contains the upper triangular matrix *B* in the pair (*A*, *B*). The second dimension of *b* must be at least max(1, *n*).
If *job* = 'E' or 'B', *vl*(*ldvl*,*) must contain left eigenvectors of (*A*, *B*), corresponding to the eigenpairs specified by *howmny* and *select*. The eigenvectors must be stored in consecutive columns of *vl*, as returned by ?tgevc.
If *job* = 'V', *vl* is not referenced. The second dimension of *vl* must be at least max(1, *m*).
If *job* = 'E' or 'B', *vr*(*ldvr*,*) must contain right eigenvectors of (*A*, *B*), corresponding to the eigenpairs specified by *howmny* and *select*. The eigenvectors must be stored in consecutive columns of *vr*, as returned by ?tgevc.
If *job* = 'V', *vr* is not referenced. The second dimension of *vr* must be at least max(1, *m*).
work is a workspace array, its dimension max(1, *lwork*).
If *job* = 'E', *work* is not referenced.

lda — INTEGER. The first dimension of *a*; at least max(1, *n*).

ldb INTEGER. The first dimension of *b*; at least max(1, *n*).

ldvl INTEGER. The first dimension of *vl*; *ldvl* ≥ 1.
If *job* = 'E' or 'B', then *ldvl* ≥ max(1, *n*).

ldvr INTEGER. The first dimension of *vr*; *ldvr* ≥ 1.
If *job* = 'E' or 'B', then *ldvr* ≥ max(1, *n*).

mm INTEGER. The number of elements in the arrays *s* and *dif* (*mm* ≥ *m*).

lwork INTEGER. The dimension of the array *work*.
lwork ≥ max(1, *n*).
If *job* = 'V' or 'B', *lwork* ≥ 2**n**(*n*+2)+16 for real flavors, and *lwork* ≥ max(1, 2**n***n*) for complex flavors.
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for details.

iwork INTEGER. Workspace array, DIMENSION at least (*n*+6) for real flavors, and at least (*n*+2) for complex flavors.
If *ijob* = 'E', *iwork* is not referenced.

Output Parameters

s REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
Array, DIMENSION (*mm*).
If *job* = 'E' or 'B', contains the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array.
If *job* = 'V', *s* is not referenced.
For real flavors:
For a complex conjugate pair of eigenvalues two consecutive elements of *s* are set to the same value. Thus, *s*(j), *dif*(j), and the j-th columns of *vl* and *vr* all correspond to the same eigenpair (but not in general the j-th eigenpair, unless all eigenpairs are selected).

dif	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Array, DIMENSION (*mm*). If *job* = 'V' or 'B', contains the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array. If the eigenvalues cannot be reordered to compute *dif*(j), *dif*(j) is set to 0; this can only occur when the true value would be very small anyway. If *job* = 'E', *dif* is not referenced. For real flavors: For a complex eigenvector, two consecutive elements of *dif* are set to the same value. For complex flavors: For each eigenvalue/vector specified by *select*, *dif* stores a Frobenius norm-based estimate of Difl.
m	INTEGER. The number of elements in the arrays *s* and *dif* used to store the specified condition numbers; for each selected eigenvalue one element is used. If *howmny* = 'A', *m* is set to *n*.
work(1)	*work*(1) If *job* is not 'E' and *info* = 0, on exit, *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine tgsna interface are the following:

a	Holds the matrix *A* of size (*n*,*n*).
b	Holds the matrix *B* of size (*n*,*n*).
s	Holds the vector of length (*mm*).

dif	Holds the vector of length (*mm*).
vl	Holds the matrix *VL* of size (*n*,*mm*).
vr	Holds the matrix *VR* of size (*n*,*mm*).
select	Holds the vector of length (*n*).
howmny	Restored based on the presence of the argument *select* as follows: *howmny* = 'S', if *select* is present, *howmny* = 'A', if *select* is omitted.
job	Restored based on the presence of arguments *s* and dif as follows: *job* = 'B', if both *s* and *dif* are present, *job* = 'E', if *s* is present and *dif* omitted, *job* = 'V', if *s* is omitted and *dif* present, Note that there will be an error condition if both *s* and *dif* are omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Generalized Singular Value Decomposition

This section describes LAPACK computational routines used for finding the generalized singular value decomposition (GSVD) of two matrices *A* and *B* as

$U^H A Q = D_1 * (0\ R)$,

$V^H B Q = D_2 * (0\ R)$,

where *U*, *V*, and *Q* are orthogonal/unitary matrices, *R* is a nonsingular upper triangular matrix, and D_1, D_2 are "diagonal" matrices of the structure detailed in the routines description section.

Table 4-7 lists LAPACK routines (Fortran-77 interface) that perform generalized singular value decomposition of matrices. Respective routine names in Fortran-95 interface are without the first symbol (see Routine Naming Conventions).

Table 4-7 Computational Routines for Generalized Singular Value Decomposition

Routine name	Operation performed
?ggsvp	Computes the preprocessing decomposition for the generalized SVD
?tgsja	Computes the generalized SVD of two upper triangular or trapezoidal matrices

You can use routines listed in the above table as well as the driver routine ?ggsvd to find the GSVD of a pair of general rectangular matrices.

?ggsvp

Computes the preprocessing decomposition for the generalized SVD.

Syntax

Fortran 77:

```
call sggsvp(jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u,
ldu, v, ldv, q, ldq, iwork, tau, work, info)
call dggsvp(jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u,
ldu, v, ldv, q, ldq, iwork, tau, work, info)
call cggsvp (jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u,
ldu, v, ldv, q, ldq, iwork, rwork, tau, work, info)
call zggsvp(jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u,
ldu, v, ldv, q, ldq, iwork, rwork, tau, work, info)
```

Fortran 95:

```
call ggsvp(a, b, tola, tolb [, k] [,l] [,u] [,v] [,q] [,info])
```

Description

This routine computes orthogonal matrices U, V and Q such that

$$U^H A Q = \begin{array}{c} \\ k \\ l \\ m-k-l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \\ 0 & 0 & 0 \end{pmatrix} \end{array}, \quad \text{if } m-k-l \geq 0$$

$$= \begin{array}{c} \\ k \\ m-k \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \end{pmatrix} \end{array}, \quad \text{if } m-k-l < 0$$

$$V^H B Q = \begin{array}{c} \\ l \\ p-l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & 0 & B_{13} \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

where the k-by-k matrix A_{12} and l-by-l matrix B_{13} are nonsingular upper triangular; A_{23} is l-by-l upper triangular if $m-k-l \geq 0$, otherwise A_{23} is $(m-k)$-by-l upper trapezoidal. The sum $k+l$ is equal to the effective numerical rank of the $(m+p)$-by-n matrix $(A^H, B^H)^H$.

This decomposition is the preprocessing step for computing the Generalized Singular Value Decomposition (GSVD), see subroutine `?ggsvd`.

Input Parameters

jobu	CHARACTER*1. Must be 'U' or 'N'. If *jobu* = 'U', orthogonal/unitary matrix U is computed. If *jobu* = 'N', U is not computed.
jobv	CHARACTER*1. Must be 'V' or 'N'.

	If *jobv* = 'V', orthogonal/unitary matrix *V* is computed. If *jobv* = 'N', *V* is not computed.
jobq	CHARACTER*1. Must be 'Q' or 'N'. If *jobq* = 'Q', orthogonal/unitary matrix *Q* is computed. If *jobq* = 'N', *Q* is not computed.
m	INTEGER. The number of rows of the matrix *A* (*m* ≥ 0).
p	INTEGER. The number of rows of the matrix *B* (*p* ≥ 0).
n	INTEGER. The number of columns of the matrices *A* and *B* (*n* ≥ 0).
a, *b*, *tau*, *work*	REAL for sggsvp DOUBLE PRECISION for dggsvp COMPLEX for cggsvp DOUBLE COMPLEX for zggsvp. Arrays: *a*(*lda*,*) contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the *p*-by-*n* matrix *B*. The second dimension of *b* must be at least max(1, *n*). *tau*(*) is a workspace array. The dimension of *tau* must be at least max(1, *n*). *work*(*) is a workspace array. The dimension of *work* must be at least max(1, 3*n*, *m*, *p*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
ldb	INTEGER. The first dimension of *b*; at least max(1, *p*).
tola, *tolb*	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. *tola* and *tolb* are the thresholds to determine the effective numerical rank of matrix *B* and a subblock of *A*. Generally, they are set to *tola* = max(*m*, *n*)*||*A*||*MACHEPS, *tolb* = max(*p*, *n*)*||*B*||*MACHEPS. The size of *tola* and *tolb* may affect the size of backward errors of the decomposition.
ldu	INTEGER. The first dimension of the output array *u* . *ldu* ≥ max(1, *m*) if *jobu* = 'U'; *ldu* ≥ 1 otherwise.

ldv	INTEGER. The first dimension of the output array *v*. *ldv* ≥ max(1, *p*) if *jobv* = 'V'; *ldv* ≥ 1 otherwise.
ldq	INTEGER. The first dimension of the output array *q*. *ldq* ≥ max(1, *n*) if *jobq* = 'Q'; *ldq* ≥ 1 otherwise.
iwork	INTEGER. Workspace array, DIMENSION at least max(1, *n*).
rwork	REAL for cggsvp DOUBLE PRECISION for zggsvp. Workspace array, DIMENSION at least max(1, 2*n*). Used in complex flavors only.

Output Parameters

a	Overwritten by the triangular (or trapezoidal) matrix described in the Description section.
b	Overwritten by the triangular matrix described in the Description section.
k, *l*	INTEGER. On exit, *k* and *l* specify the dimension of subblocks. The sum *k* + *l* is equal to effective numerical rank of $(A^H, B^H)^H$.
u, *v*, *q*	REAL for sggsvp DOUBLE PRECISION for dggsvp COMPLEX for cggsvp DOUBLE COMPLEX for zggsvp. Arrays: If *jobu* = 'U', *u*(*ldu*,*) contains the orthogonal/unitary matrix *U*. The second dimension of *u* must be at least max(1, *m*). If *jobu* = 'N', *u* is not referenced. If *jobv* = 'V', *v*(*ldv*,*) contains the orthogonal/unitary matrix *V*. The second dimension of *v* must be at least max(1, *m*). If *jobv* = 'N', *v* is not referenced. If *jobq* = 'Q', *q*(*ldq*,*) contains the orthogonal/unitary matrix *Q*. The second dimension of *q* must be at least max(1, *n*). If *jobq* = 'N', *q* is not referenced.

info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ggsvp interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
b	Holds the matrix *B* of size (*p*,*n*).
u	Holds the matrix *U* of size (*m*,*m*).
v	Holds the matrix *V* of size (*p*,*m*).
q	Holds the matrix *Q* of size (*n*,*n*).
jobu	Restored based on the presence of the argument *u* as follows: *jobu* = 'U', if *u* is present, *jobu* = 'N', if *u* is omitted.
jobv	Restored based on the presence of the argument *v* as follows: *jobz* = 'V', if *v* is present, *jobz* = 'N', if *v* is omitted.
jobq	Restored based on the presence of the argument *q* as follows: *jobz* = 'Q', if *q* is present, *jobz* = 'N', if *q* is omitted.

?tgsja

Computes the generalized SVD of two upper triangular or trapezoidal matrices.

Syntax

Fortran 77:

```
call stgsja(jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola, tolb, alpha,
beta, u, ldu, v, ldv, q, ldq, work, ncycle, info)

call dtgsja(jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola, tolb, alpha,
beta, u, ldu, v, ldv, q, ldq, work, ncycle, info)

call ctgsja(jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola, tolb, alpha,
beta, u, ldu, v, ldv, q, ldq, work, ncycle, info)

call ztgsja(jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola, tolb, alpha,
beta, u, ldu, v, ldv, q, ldq, work, ncycle, info)
```

Fortran 95:

```
call tgsja(a, b, tola, tolb, k, l [,u] [,v] [,q] [,jobu] [,jobv] [,jobq]
[,alpha] [,beta] [,ncycle] [,info])
```

Description

This routine computes the generalized singular value decomposition (GSVD) of two real/complex upper triangular (or trapezoidal) matrices A and B. On entry, it is assumed that matrices A and B have the following forms, which may be obtained by the preprocessing subroutine ?ggsvp from a general m-by-n matrix A and p-by-n matrix B:

$$A = \begin{array}{c} \\ k \\ l \\ m-k-l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \\ 0 & 0 & 0 \end{pmatrix} \end{array}, \quad \text{if } m-k-l \geq 0$$

$$= \begin{array}{c} \\ k \\ m-k \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \end{pmatrix} \end{array}, \quad \text{if } m-k-l < 0$$

$$B = \begin{array}{c} \\ l \\ p-l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & 0 & B_{13} \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

where the k-by-k matrix A_{12} and l-by-l matrix B_{13} are nonsingular upper triangular; A_{23} is l-by-l upper triangular if $m-k-l \geq 0$, otherwise A_{23} is $(m-k)$-by-l upper trapezoidal.

On exit,

$U^H A Q = D_1 * (0 \ R)$, $V^H B Q = D_2 * (0 \ R)$,

where U, V and Q are orthogonal/unitary matrices, R is a nonsingular upper triangular matrix, and D_1 and D_2 are "diagonal" matrices, which are of the following structures:

If $m-k-l \geq 0$,

$$D_1 = \begin{array}{c} \\ k \\ l \\ m-k-l \end{array} \begin{array}{c} \begin{array}{cc} k & l \end{array} \\ \begin{pmatrix} 0 & 0 \\ 0 & C \\ 0 & 0 \end{pmatrix} \end{array}$$

$$D_1 = \begin{array}{c} \\ l \\ p-l \end{array} \begin{array}{c} \begin{array}{cc} k & l \end{array} \\ \begin{pmatrix} 0 & S \\ 0 & 0 \end{pmatrix} \end{array}$$

$$(0\ R) = \begin{array}{c} \\ k \\ l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & R_{11} & R_{12} \\ 0 & 0 & R_{22} \end{pmatrix} \end{array}$$

where

C = diag ($alpha(k+1),...,alpha(k+l)$)

S = diag ($beta(k+1),...,beta(k+l)$)

$C^2 + S^2 = I$

R is stored in $a(1:k+l, n-k-l+1:n)$ on exit.

If $m-k-l < 0$,

$$D_1 = \begin{array}{c} \\ k \\ m-k \end{array} \begin{array}{c} \begin{array}{ccc} k & m-l & k+l-m \end{array} \\ \begin{pmatrix} I & 0 & 0 \\ 0 & C & 0 \end{pmatrix} \end{array}$$

$$D_2 = \begin{array}{c} \\ m-k \\ k+l-m \\ p-l \end{array} \begin{array}{c} \begin{array}{ccc} k & m-k & k+l-m \end{array} \\ \begin{pmatrix} 0 & S & 0 \\ 0 & 0 & I \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$(0\ R) = \begin{array}{c} \\ k \\ m-k \\ k+l-m \end{array} \begin{array}{c} \begin{array}{cccc} n-k-l & k & m-k & k+l-m \end{array} \\ \begin{pmatrix} 0 & R_{11} & R_{12} & R_{13} \\ 0 & 0 & R_{22} & R_{23} \\ 0 & 0 & 0 & R_{33} \end{pmatrix} \end{array}$$

where

C = diag ($alpha(K+1),...,alpha(m)$),

S = diag ($beta(K+1),...,beta(m)$),

$C^2 + S^2 = I$

On exit, $\begin{pmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \end{pmatrix}$ is stored in $a(1{:}m, n-k-l+1{:}n)$ and R_{33} is stored in $b(m-k+1{:}l, n+m-k-l+1{:}n)$.

The computation of the orthogonal/unitary transformation matrices U, V or Q is optional. These matrices may either be formed explicitly, or they *may* be postmultiplied into input matrices U_1, V_1, or Q_1.

Input Parameters

jobu CHARACTER*1. Must be 'U', 'I', or 'N'.

	If *jobu* = 'U', *u* must contain an orthogonal/unitary matrix U_1 on entry. If *jobu* = 'I', *u* is initialized to the unit matrix. If *jobu* = 'N', *u* is not computed.
jobv	CHARACTER*1. Must be 'V', 'I', or 'N'. If *jobv* = 'V', *v* must contain an orthogonal/unitary matrix V_1 on entry. If *jobv* = 'I', *v* is initialized to the unit matrix. If *jobv* = 'N', *v* is not computed.
jobq	CHARACTER*1. Must be 'Q', 'I', or 'N'. If *jobq* = 'Q', *q* must contain an orthogonal/unitary matrix Q_1 on entry. If *jobq* = 'I', *q* is initialized to the unit matrix. If *jobq* = 'N', *q* is not computed.
m	INTEGER. The number of rows of the matrix *A* ($m \geq 0$).
p	INTEGER. The number of rows of the matrix *B* ($p \geq 0$).
n	INTEGER. The number of columns of the matrices *A* and *B* ($n \geq 0$).
k, *l*	INTEGER. Specify the subblocks in the input matrices *A* and *B*, whose GSVD is going to be computed by ?tgsja.
a,*b*,*u*,*v*,*q*,*work*	REAL for stgsja DOUBLE PRECISION for dtgsja COMPLEX for ctgsja DOUBLE COMPLEX for ztgsja. Arrays: *a*(*lda*,*) contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the *p*-by-*n* matrix *B*. The second dimension of *b* must be at least max(1, *n*). If *jobu* = 'U', *u*(*ldu*,*) must contain a matrix U_1 (usually the orthogonal/unitary matrix returned by ?ggsvp). The second dimension of *u* must be at least max(1, *m*). If *jobv* = 'V', *v*(*ldv*,*) must contain a matrix V_1 (usually the orthogonal/unitary matrix returned by ?ggsvp). The second dimension of *v* must be at least max(1, *p*).

If *jobq* = 'Q', *q*(*ldq*,*) must contain a matrix Q_1 (usually the orthogonal/unitary matrix returned by ?ggsvp).
The second dimension of *q* must be at least max(1, *n*).
work(*) is a workspace array.
The dimension of *work* must be at least max(1, 2*n*).

lda — INTEGER. The first dimension of *a*; at least max(1, *m*).

ldb — INTEGER. The first dimension of *b*; at least max(1, *p*).

ldu — INTEGER. The first dimension of the array *u* .

ldu ≥ max(1, *m*) if *jobu* = 'U'; *ldu* ≥ 1 otherwise.

ldv — INTEGER. The first dimension of the array *v* .

ldv ≥ max(1, *p*) if *jobv* = 'V'; *ldv* ≥ 1 otherwise.

ldq — INTEGER. The first dimension of the array *q* .

ldq ≥ max(1, *n*) if *jobq* = 'Q'; *ldq* ≥ 1 otherwise.

tola, *tolb* — REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
tola and *tolb* are the convergence criteria for the Jacobi-Kogbetliantz iteration procedure. Generally, they are the same as used in ?ggsvp:
tola = max(*m*, *n*)*|*A*|*MACHEPS,
tolb = max(*p*, *n*)*|*B*|*MACHEPS.

Output Parameters

a — On exit, *a*(*n*-*k*+1:*n*, 1:min(*k*+*l*, *m*)) contains the triangular matrix *R* or part of *R*.

b — On exit, if necessary, *b*(*m*-*k*+1: *l*, *n*+*m*-*k*-*l*+1: *n*)) contains a part of *R*.

alpha, *beta* — REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
Arrays, DIMENSION at least max(1, *n*). Contain the generalized singular value pairs of *A* and *B*:
alpha(1:*k*) = 1,
beta(1:*k*) = 0,
and if *m*-*k*-*l* ≥ 0,
alpha(*k*+1:*k*+*l*) = diag(*C*),

	beta(*k*+1:*k*+*l*) = diag(*S*), or if *m*-*k*-*l* < 0, *alpha*(*k*+1:*m*)= *C*, *alpha*(*m*+1:*k*+*l*)=0 *beta*(*K*+1:*M*) = *S*, *beta*(*m*+1:*k*+*l*) = 1. Furthermore, if *k*+*l* < *n*, *alpha*(*k*+*l*+1:*n*)= 0 and *beta*(*k*+*l*+1:*n*) = 0.
u	If *jobu* = 'I', *u* contains the orthogonal/unitary matrix *U*. If *jobu* = 'U', *u* contains the product U_1**U*. If *jobu* = 'N', *u* is not referenced.
v	If *jobv* = 'I', *v* contains the orthogonal/unitary matrix *U*. If *jobv* = 'V', *v* contains the product V_1**V*. If *jobv* = 'N', *v* is not referenced.
q	If *jobq* = 'I', *q* contains the orthogonal/unitary matrix *U*. If *jobq* = 'Q', *q* contains the product Q_1**Q*. If *jobq* = 'N', *q* is not referenced.
ncycle	INTEGER. The number of cycles required for convergence.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = 1, the procedure does not converge after MAXIT cycles.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine tgsja interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
b	Holds the matrix *B* of size (*p*,*n*).
u	Holds the matrix *U* of size (*m*,*m*).
v	Holds the matrix *V* of size (*p*,*p*).
q	Holds the matrix *Q* of size (*n*,*n*).
alpha	Holds the vector of length (*n*).

beta	Holds the vector of length (*n*).
jobu	If omitted, this argument is restored based on the presence of argument *u* as follows: *jobu* = 'U', if *u* is present, *jobu* = 'N', if *u* is omitted. If present, *jobu* must be equal to 'I' or 'U' and the argument *u* must also be present. Note that there will be an error condition if *jobu* is present and *u* omitted.
jobv	If omitted, this argument is restored based on the presence of argument *v* as follows: *jobv* = 'V', if *v* is present, *jobv* = 'N', if *v* is omitted. If present, *jobv* must be equal to 'I' or 'V' and the argument *v* must also be present. Note that there will be an error condition if *jobv* is present and *v* omitted.
jobq	If omitted, this argument is restored based on the presence of argument *q* as follows: *jobq* = 'Q', if *q* is present, *jobq* = 'N', if *q* is omitted. If present, *jobq* must be equal to 'I' or 'Q' and the argument *q* must also be present. Note that there will be an error condition if *jobq* is present and *q* omitted.

Driver Routines

Each of the LAPACK driver routines solves a complete problem. To arrive at the solution, driver routines typically call a sequence of appropriate computational routines.

Driver routines are described in the following sections :

Linear Least Squares (LLS) Problems

Generalized LLS Problems

Symmetric Eigenproblems

Nonsymmetric Eigenproblems

Linear Least Squares (LLS) Problems

This section describes LAPACK driver routines used for solving linear least-squares problems. Table 4-8 lists all such routines for Fortran-77 interface. Respective routine names in Fortran-95 interface are without the first symbol (see Routine Naming Conventions).

Table 4-8 Driver Routines for Solving LLS Problems

Routine Name	Operation performed
?gels	Uses QR or LQ factorization to solve a overdetermined or underdetermined linear system with full rank matrix.
?gelsy	Computes the minimum-norm solution to a linear least squares problem using a complete orthogonal factorization of A.
?gelss	Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A.
?gelsd	Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A and a divide and conquer method.

?gels

Uses QR or LQ factorization to solve a overdetermined or underdetermined linear system with full rank matrix.

Syntax

Fortran 77:

```
call sgels(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
call dgels(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
call cgels(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
call zgels(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
```

Fortran 95:

```
call gels(a, b [,trans] [,info])
```

Description

This routine solves overdetermined or underdetermined real/ complex linear systems involving an m-by-n matrix A, or its transpose/ conjugate-transpose, using a QR or LQ factorization of A. It is assumed that A has full rank.

The following options are provided:

1. If $trans$ = 'N' and $m \geq n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

$$\text{minimize } || b - A x ||_2$$

2. If $trans$ = 'N' and $m < n$: find the minimum norm solution of an underdetermined system $A*X = B$.

3. If $trans$ = 'T' or 'C' and $m \geq n$: find the minimum norm solution of an undetermined system $A_H*X = B$.

4. If $trans$ = 'T' or 'C' and $m < n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

$$\text{minimize } || b - A^H x ||_2$$

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m-by-$nrhs$ right hand side matrix B and the n-by-nrh solution matrix X.

Input Parameters

$trans$	CHARACTER*1. Must be 'N', 'T', or 'C'. If $trans$ = 'N', the linear system involves matrix A; If $trans$ = 'T', the linear system involves the transposed matrix A^T (for real flavors only); If $trans$ = 'C', the linear system involves the conjugate-transposed matrix A^H (for complex flavors only).
m	INTEGER. The number of rows of the matrix A ($m \geq 0$).
n	INTEGER. The number of columns of the matrix A ($n \geq 0$).

nrhs	INTEGER. The number of right-hand sides; the number of columns in *B* (*nrhs* ≥ 0).
a, *b*, *work*	REAL for sgels DOUBLE PRECISION for dgels COMPLEX for cgels DOUBLE COMPLEX for zgels. Arrays: *a*(*lda*,*) contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the matrix *B* of right hand side vectors, stored columnwise; *B* is *m*-by-*nrhs* if *trans* = 'N' , or *n*-by-*nrhs* if *trans* = 'T' or 'C'. The second dimension of *b* must be at least max(1, *nrhs*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
ldb	INTEGER. The first dimension of *b*; must be at least max(1, *m*, *n*).
lwork	INTEGER. The size of the *work* array; must be at least min(*m*, *n*)+max(1, *m*, *n*, *nrhs*). If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a	On exit, overwritten by the factorization data as follows: if *m* ≥ *n*, array *a* contains the details of the *QR* factorization of the matrix *A* as returned by ?geqrf; if *m* ≥ *n*, array *a* contains the details of the *LQ* factorization of the matrix *A* as returned by ?gelqf.
b	If *info* = 0, *b* overwritten by the solution vectors, stored columnwise:

if *trans* = 'N' and $m \geq n$, rows 1 to n of *b* contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of modulus of elements n+1 to m in that column;

if *trans* = 'N' and $m \geq n$, rows 1 to n of *b* contain the minimum norm solution vectors;
if *trans* = 'T' or 'C' and $m < n$, rows 1 to m of *b* contain the minimum norm solution vectors; if *trans* = 'T' or 'C' and $m < n$, rows 1 to m of *b* contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of modulus of elements m+1 to n in that column.

work(1) — If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, the *i*-th diagonal element of the triangular factor of *A* is zero, so that *A* does not have full rank; the least squares solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gels interface are the following:

a — Holds the matrix *A* of size (*m*,*n*).

b — Holds the matrix of size max(*m*,*n*)-by-*nrhs*.
If *trans* = 'N', then, on entry, the size of *b* is *m*-by-*nrhs*,
If *trans* = 'T', then, on entry, the size of *b* is *n*-by-*nrhs*,

trans — Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = min (*m*, *n*)+max(1, *m*, *n*, *nrhs*)**blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?gelsy

Computes the minimum-norm solution to a linear least squares problem using a complete orthogonal factorization of A.

Syntax

Fortran 77:

call sgelsy(*m*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*, *jpvt*, *rcond*, *rank*, *work*, *lwork*, *info*)

call dgelsy(*m*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*, *jpvt*, *rcond*, *rank*, *work*, *lwork*, *info*)

call cgelsy(*m*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*, *jpvt*, *rcond*, *rank*, *work*, *lwork*, *rwork*, *info*)

call zgelsy(*m*, *n*, *nrhs*, *a*, *lda*, *b*, *ldb*, *jpvt*, *rcond*, *rank*, *work*, *lwork*, *rwork*, *info*)

Fortran 95:

call gelsy(*a*, *b* [,*rank*] [,*jpvt*] [,*rcond*] [,*info*])

Description

This routine computes the minimum-norm solution to a real/complex linear least squares problem:

`minimize || b - A x ||`$_2$

using a complete orthogonal factorization of A. A is an m-by-n matrix which may be rank-deficient. Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m-by-$nrhs$ right hand side matrix B and the n-by-$nrhs$ solution matrix X.

The routine first computes a QR factorization with column pivoting:

$$AP = Q \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix}$$

with R_{11} defined as the largest leading submatrix whose estimated condition number is less than 1/$rcond$. The order of R_{11}, $rank$, is the effective rank of A. Then, R_{22} is considered to be negligible, and R_{12} is annihilated by orthogonal/unitary transformations from the right, arriving at the complete orthogonal factorization:

$$AP = Q \begin{pmatrix} T_{11} & 0 \\ 0 & 0 \end{pmatrix} Z$$

The minimum-norm solution is then

$$X = PZ^H \begin{pmatrix} T_{11}^{-1} & Q_1{}^H b \\ 0 & 0 \end{pmatrix}$$

where Q_1 consists of the first $rank$ columns of Q. This routine is basically identical to the original `?gelsx` except three differences:

- The call to the subroutine ?geqpf has been substituted by the call to the subroutine ?geqp3. This subroutine is a BLAS-3 version of the *QR* factorization with column pivoting.
- Matrix *B* (the right hand side) is updated with BLAS-3.
- The permutation of matrix *B* (the right hand side) is faster and more simple.

Input Parameters

m	INTEGER. The number of rows of the matrix *A* (*m* ≥ 0).
n	INTEGER. The number of columns of the matrix *A* (*n* ≥ 0).
nrhs	INTEGER. The number of right-hand sides; the number of columns in *B* (*nrhs* ≥ 0).
a, *b*, *work*	REAL for sgelsy DOUBLE PRECISION for dgelsy COMPLEX for cgelsy DOUBLE COMPLEX for zgelsy. Arrays: *a*(*lda*,*) contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the *m*-by-*nrhs* right hand side matrix *B*. The second dimension of *b* must be at least max(1, *nrhs*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
ldb	INTEGER. The first dimension of *b*; must be at least max(1, *m*, *n*).
jpvt	INTEGER. Array, DIMENSION at least max(1, *n*). On entry, if *jpvt*(*i*)≠ 0, the *i*-th column of *A* is permuted to the front of *A***P*, otherwise the *i*-th column of *A* is a free column.
rcond	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors.

rcond is used to determine the effective rank of *A*, which is defined as the order of the largest leading triangular submatrix R_{11} in the *QR* factorization with pivoting of *A*, whose estimated condition number < 1/*rcond*.

lwork — INTEGER. The size of the *work* array.
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

rwork — REAL for cgelsy DOUBLE PRECISION for zgelsy.
Workspace array, DIMENSION at least max(1, 2*n*). Used in complex flavors only.

Output Parameters

a — On exit, overwritten by the details of the complete orthogonal factorization of *A*.

b — Overwritten by the *n*-by-*nrhs* solution matrix *X*.

jpvt — On exit, if *jpvt*(*i*)= *k*, then the *i*th column of *AP* was the *k*-th column of *A*.

rank — INTEGER. The effective rank of *A*, that is, the order of the submatrix R_{11}. This is the same as the order of the submatrix T_{11} in the complete orthogonal factorization of *A*.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gelsy interface are the following:

a — Holds the matrix *A* of size (*m*,*n*).

b	Holds the matrix of size max(*m*,*n*)-by-*nrhs*. On entry, contains the *m*-by-*nrhs* right hand side matrix *B*, On exit, overwritten by the *n*-by-*nrhs* solution matrix *X*.
jpvt	Holds the vector of length (*n*). Default value for this element is *jpvt*(*i*) = 0.
rcond	Default value for this element is *rcond* = 100*EPSILON(1.0_WP).

Application Notes

For real flavors:

The unblocked strategy requires that:

lwork ≥ max(*mn*+3*n*+1, 2**mn* + *nrhs*),

where *mn* = min(*m*, *n*).

The block algorithm requires that:

lwork ≥ max(*mn*+2*n*+*nb**(*n*+1), 2**mn*+*nb***nrhs*),

where *nb* is an upper bound on the blocksize returned by ilaenv for the routines sgeqp3/dgeqp3, stzrzf/dtzrzf, stzrqf/dtzrqf, sormqr/dormqr, and sormrz/dormrz.

For complex flavors:

The unblocked strategy requires that:

lwork ≥ *mn* + max(2**mn*, *n*+1, *mn* + *nrhs*),

where *mn* = min(*m*, *n*).

The block algorithm requires that:

lwork < *mn* + max(2**mn*, *nb**(*n*+1), *mn*+*mn***nb*, *mn*+ *nb***nrhs*),

where *nb* is an upper bound on the blocksize returned by ilaenv for the routines cgeqp3/zgeqp3, ctzrzf/ztzrzf, ctzrqf/ztzrqf, cunmqr/zunmqr, and cunmrz/zunmrz.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?gelss

Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A.

Syntax

Fortran 77:

```
call sgelss(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, info)
call dgelss(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, info)
call cgelss(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, rwork, info)
call zgelss(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, rwork, info)
```

Fortran 95:

```
call gelss(a, b [,rank] [,s] [,rcond] [,info])
```

Description

This routine computes the minimum norm solution to a real linear least squares problem:

$$\text{minimize } || b - Ax ||_2$$

using the singular value decomposition (SVD) of A. A is an m-by-n matrix which may be rank-deficient. Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m-by-$nrhs$ right hand side matrix B and the n-by-$nrhs$ solution matrix X. The effective rank of A is determined by treating as zero those singular values which are less than $rcond$ times the largest singular value.

Input Parameters

m	INTEGER. The number of rows of the matrix *A* (*m* ≥ 0).
n	INTEGER. The number of columns of the matrix *A* (*n* ≥ 0).
nrhs	INTEGER. The number of right-hand sides; the number of columns in *B* (*nrhs* ≥ 0).
a, *b*, *work*	REAL for sgelss DOUBLE PRECISION for dgelss COMPLEX for cgelss DOUBLE COMPLEX for zgelss. Arrays: *a*(*lda*,*) contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the *m*-by-*nrhs* right hand side matrix *B*. The second dimension of *b* must be at least max(1, *nrhs*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
ldb	INTEGER. The first dimension of *b*; must be at least max(1, *m*, *n*).
rcond	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. *rcond* is used to determine the effective rank of *A*. Singular values *s*(*i*) ≤ *rcond* **s*(1) are treated as zero. If *rcond* <0, machine precision is used instead.
lwork	INTEGER. The size of the *work* array; *lwork*≥ 1. If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.
rwork	REAL for cgelss DOUBLE PRECISION for zgelss.

Workspace array used in complex flavors only. DIMENSION at least max(1, 5*min(*m*, *n*)).

Output Parameters

a — On exit, the first min(*m*, *n*) rows of *A* are overwritten with its right singular vectors, stored row-wise.

b — Overwritten by the *n*-by-*nrhs* solution matrix *X*.

If *m*≥*n* and *rank* = *n*, the residual sum-of-squares for the solution in the *i*-th column is given by the sum of squares of modulus of elements *n*+1:*m* in that column.

s — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Array, DIMENSION at least max(1, min(*m*, *n*)). The singular values of *A* in decreasing order. The condition number of *A* in the 2-norm is
k2(*A*) = *s*(1)/ *s*(min(*m*, *n*)) .

rank — INTEGER. The effective rank of *A*, that is, the number of singular values which are greater than *rcond* **s*(1).

work(1) — If *info* = 0, on exit, *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, then the algorithm for computing the SVD failed to converge; *i* indicates the number of off-diagonal elements of an intermediate bidiagonal form which did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gelss interface are the following:

a — Holds the matrix *A* of size (*m*,*n*).

b	Holds the matrix of size max(*m*,*n*)-by-*nrhs*. On entry, contains the *m*-by-*nrhs* right hand side matrix *B*, On exit, overwritten by the *n*-by-*nrhs* solution matrix *X*.
s	Holds the vector of length min(*m*,*n*).
rcond	Default value for this element is *rcond* = 100*EPSILON(1.0_WP).

Application Notes

For real flavors:

lwork ≥ 3*min(*m*, *n*)+ max(2*min(*m*, *n*), max(*m*, *n*), *nrhs*)

For complex flavors:

lwork ≥ 2*min(*m*, *n*)+ max(*m*, *n*, *nrhs*)

For good performance, *lwork* should generally be larger.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?gelsd

Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A and a divide and conquer method.

Syntax

Fortran 77:

```
call sgelsd(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, iwork, info)
call dgelsd(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, iwork, info)
call cgelsd(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, rwork, iwork, info)
call zgelsd(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, rwork, iwork, info)
```

Fortran 95:

```
call gelsd(a, b [,rank] [,s] [,rcond] [,info])
```

Description

This routine computes the minimum-norm solution to a real linear least squares problem:

$$\text{minimize } \| b - Ax \|_2$$

using the singular value decomposition (SVD) of A. A is an m-by-n matrix which may be rank-deficient.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m-by-$nrhs$ right hand side matrix B and the n-by-$nrhs$ solution matrix X.

The problem is solved in three steps:

1. Reduce the coefficient matrix A to bidiagonal form with Householder transformations, reducing the original problem into a "bidiagonal least squares problem" (BLS).
2. Solve the BLS using a divide and conquer approach.
3. Apply back all the Householder transformations to solve the original least squares problem.

The effective rank of *A* is determined by treating as zero those singular values which are less than *rcond* times the largest singular value.

The routine uses auxiliary routines ?lals0 and ?lalsa.

Input Parameters

m	INTEGER. The number of rows of the matrix *A* (*m* ≥ 0).
n	INTEGER. The number of columns of the matrix *A* (*n* ≥ 0).
nrhs	INTEGER. The number of right-hand sides; the number of columns in *B* (*nrhs* ≥ 0).
a, *b*, *work*	REAL for sgelsd DOUBLE PRECISION for dgelsd COMPLEX for cgelsd DOUBLE COMPLEX for zgelsd. Arrays: *a*(*lda*,*) contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the *m*-by-*nrhs* right hand side matrix *B*. The second dimension of *b* must be at least max(1, *nrhs*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
ldb	INTEGER. The first dimension of *b*; must be at least max(1, *m*, *n*).
rcond	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. *rcond* is used to determine the effective rank of *A*. Singular values *s*(*i*) ≤ *rcond* **s*(1) are treated as zero. If *rcond* ≤ 0, machine precision is used instead.
lwork	INTEGER. The size of the *work* array; *lwork* ≥ 1. If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the array *work* and the minimum sizes of the arrays *rwork* and *iwork*, and

returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* is issued by xerbla.
See Application Notes for the suggested value of *lwork*.

iwork — INTEGER. Workspace array. See Application Notes for the suggested dimension of *iwork*.

rwork — REAL for cgelsd
DOUBLE PRECISION for zgelsd.
Workspace array, used in complex flavors only. See Application Notes for the suggested dimension of *rwork*.

Output Parameters

a — On exit, *A* has been overwritten.

b — Overwritten by the *n*-by-*nrhs* solution matrix *X*.

If $m \geq n$ and $rank = n$, the residual sum-of-squares for the solution in the *i*-th column is given by the sum of squares of modulus of elements *n*+1:*m* in that column.

s — REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Array, DIMENSION at least max(1, min(*m*, *n*)). The singular values of *A* in decreasing order. The condition number of *A* in the 2-norm is
$k^2(A) = s(1) / s(\min(m, n))$.

rank — INTEGER. The effective rank of *A*, that is, the number of singular values which are greater than *rcond* **s*(1).

work(1) — If *info* = 0, on exit, *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

rwork(1) — If *info* = 0, on exit, *rwork*(1) returns the minimum size of the workspace array *iwork* required for optimum performance.

iwork(1) — If *info* = 0, on exit, *iwork*(1) returns the minimum size of the workspace array *iwork* required for optimum performance.

info — INTEGER.

If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, then the algorithm for computing the SVD failed to converge; *i* indicates the number of off-diagonal elements of an intermediate bidiagonal form that did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `gelsd` interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
b	Holds the matrix of size max(*m*,*n*)-by-*nrhs*. On entry, contains the *m*-by-*nrhs* right hand side matrix *B*, On exit, overwritten by the *n*-by-*nrhs* solution matrix *X*.
s	Holds the vector of length min(*m*,*n*).
rcond	Default value for this element is *rcond* = 100*EPSILON(1.0_WP).

Application Notes

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

The exact minimum amount of workspace needed depends on *m*, *n* and *nrhs*. The size *lwork* of the workspace array *work* must be as given below.

For real flavors:

If $m \geq n$,

$$lwork \geq 12n + 2n*smlsiz + 8n*nlvl + n*nrhs + (smlsiz+1)^2;$$

If $m < n$,

$$lwork \geq 12m + 2m*smlsiz + 8m*nlvl + m*nrhs + (smlsiz+1)^2;$$

For complex flavors:

If $m \geq n$,

lwork< 2*n* + *n***nrhs*;

If *m* < *n*,

lwork ≥ 2*m* + *m***nrhs*;

where *smlsiz* is returned by ilaenv and is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25), and

nlvl = INT($\log_2$(min(*m*, *n*)/(*smlsiz*+1))) + 1.

For good performance, *lwork* should generally be larger.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The dimension of the workspace array *iwork* must be at least

3*min(*m*, *n*)**nlvl* + 11*min(*m*, *n*).

The dimension of the workspace array *iwork* (for complex flavors) must be at least max(1, *lrwork*).

lrwork ≥ 10*n* + 2*n***smlsiz* + 8*n***nlvl* + 3**smlsiz***nrhs* + (*smlsiz*+1)^2 if *m* ≥ *n*, and

lrwork ≥ 10*m* + 2*m***smlsiz* + 8*m***nlvl* + 3**smlsiz***nrhs* + (*smlsiz*+1)^2 if *m* ≥ *n*.

Generalized LLS Problems

This section describes LAPACK driver routines used for solving generalized linear least-squares problems. Table 4-9 lists all such routines for Fortran-77 interface. Respective routine names in Fortran-95 interface are without the first symbol (see Routine Naming Conventions).

Table 4-9 Driver Routines for Solving Generalized LLS Problems

Routine Name	Operation performed
?gglse	Solves the linear equality-constrained least squares problem using a generalized RQ factorization.
?ggglm	Solves a general Gauss-Markov linear model problem using a generalized QR factorization.

?gglse

Solves the linear equality-constrained least squares problem using a generalized RQ factorization.

Syntax

Fortran 77:

```
call sgglse(m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info)
call dgglse(m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info)
call cgglse(m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info)
call zgglse(m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info)
```

Fortran 95:

```
call gglse(a, b, c, d, x [,info])
```

Description

This routine solves the linear equality-constrained least squares (LSE) problem:

`minimize || c - A x ||`^{2} subject to `B x = d`

where `A` is an `m`-by-`n` matrix, `B` is a `p`-by-`n` matrix, `c` is a given `m`-vector, and `d` is a given `p`-vector. It is assumed that $p \le n \le m+p$, and

$$\text{rank}(B) = p \quad \text{and} \quad \text{rank}\begin{pmatrix} A \\ B \end{pmatrix} = n .$$

These conditions ensure that the LSE problem has a unique solution, which is obtained using a generalized *RQ* factorization of the matrices (*B*, *A*) given by

B=(0 *R*)**Q*, *A*=*Z***T***Q*

Input Parameters

m	INTEGER. The number of rows of the matrix *A* (*m* ≥ 0).
n	INTEGER. The number of columns of the matrices *A* and *B* (*n* ≥ 0).
p	INTEGER. The number of rows of the matrix *B* (0 ≤ *p* ≤ *n* ≤ *m*+*p*).
a, *b*, *c*, *d*, *work*	REAL for sgglse DOUBLE PRECISION for dgglse COMPLEX for cgglse DOUBLE COMPLEX for zgglse. Arrays: *a*(*lda*,*) contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the *p*-by-*n* matrix *B*. The second dimension of *b* must be at least max(1, *n*). *c*(*), dimension at least max(1, *m*), contains the right hand side vector for the least squares part of the LSE problem. *d*(*), dimension at least max(1, *p*), contains the right hand side vector for the constrained equation. *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
ldb	INTEGER. The first dimension of *b*; at least max(1, *p*).
lwork	INTEGER. The size of the *work* array; *lwork* ≥ max(1, *m*+*n*+*p*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

x	REAL for sgglse DOUBLE PRECISION for dgglse COMPLEX for cgglse DOUBLE COMPLEX for zgglse. Array, DIMENSION at least max(1, *n*). On exit, contains the solution of the LSE problem.
a	On exit, the elements on and above the diagonal of the array contain the min(*m*, *n*)-by-*n* upper trapezoidal matrix *T*.
b	On exit, the upper triangle of the subarray *b*(1:*p*, *n*–*p*+1:*n*) contains the *p*-by-*p* upper triangular matrix *R*.
d	On exit, *d* is destroyed.
c	On exit, the residual sum-of-squares for the solution is given by the sum of squares of elements *n*-*p*+1 to *m* of vector *c*.
work(1)	If *info* = 0, on exit, *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = –*i*, the *i*-th parameter had an illegal value. If *info* = 1, the upper triangular factor *R* associated with *B* in the generalized RQ factorization of the pair (*B*, *A*) is singular, so that rank(*B*) < *P*; the least squares solution could not be computed. If *info* = 2, the (*n*–*p*)-by-(*n*–*p*) part of the upper trapezoidal factor *T* associated with *A* in the generalized RQ factorization of the pair (*B*, *A*) is singular, so that $\text{rank}\begin{pmatrix} A \\ B \end{pmatrix} < n$; the least squares solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `gglse` interface are the following:

a	Holds the matrix *A* of size (*m*,*n*).
b	Holds the matrix *B* of size (*p*,*n*).
c	Holds the vector of length (*m*).
d	Holds the vector of length (*p*).
x	Holds the vector of length (*n*).

Application Notes

For optimum performance, use

lwork ≥ *p* + min(*m*, *n*) + max(*m*, *n*)**nb*,

where *nb* is an upper bound for the optimal blocksizes for `?geqrf`, `?gerqf`, `?ormqr`/`?unmqr` and `?ormrq`/`?unmrq`.

You may set *lwork* to -1. The routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?ggglm

Solves a general Gauss-Markov linear model problem using a generalized QR factorization.

Syntax

Fortran 77:

```
call sggglm(n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info)
call dggglm(n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info)
call cggglm(n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info)
call zggglm(n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info)
```

Fortran 95:

```
call ggglm(a, b, d, x, y [,info])
```

Description

This routine solves a general Gauss-Markov linear model (GLM) problem:

$\text{minimize}_x \; || \; y \; ||_2$ subject to $d = Ax + By$

where A is an n-by-m matrix, B is an n-by-p matrix, and d is a given n-vector. It is assumed that $m \le n \le m+p$, and $\text{rank}(A) = m$ and $\text{rank}(A \; B) = n$.

Under these assumptions, the constrained equation is always consistent, and there is a unique solution x and a minimal 2-norm solution y, which is obtained using a generalized QR factorization of the matrices $(A, \; B)$ given by

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix}; \qquad B = Q * T * Z.$$

In particular, if matrix B is square nonsingular, then the problem GLM is equivalent to the following weighted linear least squares problem

$\text{minimize}_x \; || \; B^{-1}(d-Ax) ||_2$.

Input Parameters

n	INTEGER. The number of rows of the matrices *A* and *B* (*n* ≥ 0).
m	INTEGER. The number of columns in *A* (*m* ≥ 0).
p	INTEGER. The number of columns in *B* (*p* ≥ *n* – *m*).
a, *b*, *d*, *work*	REAL for sggglm DOUBLE PRECISION for dggglm COMPLEX for cggglm DOUBLE COMPLEX for zggglm. Arrays: *a*(*lda*,*) contains the *n*-by-*m* matrix *A*. The second dimension of *a* must be at least max(1, *m*). *b*(*ldb*,*) contains the *n*-by-*p* matrix *B*. The second dimension of *b* must be at least max(1, *p*). *d*(*), dimension at least max(1, *n*), contains the left hand side of the GLM equation. *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).
ldb	INTEGER. The first dimension of *b*; at least max(1, *n*).
lwork	INTEGER. The size of the *work* array; *lwork* ≥ max(1, *n*+*m*+*p*). If *lwork* = –1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

x, *y*	REAL for sggglm DOUBLE PRECISION for dggglm COMPLEX for cggglm DOUBLE COMPLEX for zggglm. Arrays *x*(*), *y*(*). DIMENSION at least max(1, *m*) for *x* and at least max(1, *p*) for *y*.

	On exit, *x* and *y* are the solutions of the GLM problem.
a	On exit, the upper triangular part of the array *a* contains the *m*-by-*m* upper triangular matrix *R*.
b	On exit, if $n \le p$, the upper triangle of the subarray *b*(1:*n*, *p*−*n*+1:*p*) contains the *n*-by-*n* upper triangular matrix *T*; if $n > p$, the elements on and above the (*n*−*p*)-th subdiagonal contain the *n*-by-*p* upper trapezoidal matrix *T*.
d	On exit, *d* is destroyed
work(1)	If *info* = 0, on exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = −*i*, the *i*-th parameter had an illegal value. If *info* = 1, the upper triangular factor *R* associated with *A* in the generalized QR factorization of the pair (*A*, *B*) is singular, so that rank(*A*) < *m*; the least squares solution could not be computed. If *info* = 2, the bottom (*n*−*m*)-by-(*n*−*m*) part of the upper trapezoidal factor *T* associated with *B* in the generalized QR factorization of the pair (*A*, *B*) is singular, so that rank(*A* *B*) < *n*; the least squares solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ggglm interface are the following:

a	Holds the matrix *A* of size (*n*,*m*).
b	Holds the matrix *B* of size (*n*,*p*).
d	Holds the vector of length (*n*).
x	Holds the vector of length (*m*).
y	Holds the vector of length (*p*).

Application Notes

For optimum performance, use

lwork ≥ *m* + min(*n*, *p*) + max(*n*, *p*)**nb*,

where *nb* is an upper bound for the optimal blocksizes for ?geqrf, ?gerqf, ?ormqr/?unmqr and ?ormrq/?unmrq.

You may set *lwork* to -1. The routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Symmetric Eigenproblems

This section describes LAPACK driver routines used for solving symmetric eigenvalue problems. See also computational routines computational routines that can be called to solve these problems. Table 4-10 lists all such driver routines for Fortran-77 interface. Respective routine names in Fortran-95 interface are without the first symbol (see Routine Naming Conventions).

Table 4-10 Driver Routines for Solving Symmetric Eigenproblems

Routine Name	Operation performed
?syev/?heev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix.
?syevd/?heevd	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian matrix using divide and conquer algorithm.
?syevx/?heevx	Computes selected eigenvalues and, optionally, eigenvectors of a symmetric / Hermitian matrix.
?syevr/?heevr	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix using the Relatively Robust Representations.
?spev/?hpev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix in packed storage.

Routine Name	Operation performed
?spevd/?hpevd	Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian matrix held in packed storage.
?spevx/?hpevx	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix in packed storage.
?sbev /?hbev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian band matrix.
?sbevd/?hbevd	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian band matrix using divide and conquer algorithm.
?sbevx/?hbevx	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian band matrix.
?stev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix.
?stevd	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric tridiagonal matrix using divide and conquer algorithm.
?stevx	Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.
?stevr	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix using the Relatively Robust Representations.

?syev

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix.

Syntax

Fortran 77:

```
call ssyev(jobz, uplo, n, a, lda, w, work, lwork, info)
call dsyev(jobz, uplo, n, a, lda, w, work, lwork, info)
```

Fortran 95:

```
call syev(a, w [,jobz] [,uplo] [,info])
```

Description

This routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A*.

Note that for most cases of real symmetric eigenvalue problems the default choice should be ?syevr function as its underlying algorithm is faster and uses less workspace.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then only eigenvalues are computed. If *jobz* = 'V', then eigenvalues and eigenvectors are computed.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *a* stores the upper triangular part of *A*. If *uplo* = 'L', *a* stores the lower triangular part of *A*.
n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
a, *work*	REAL for ssyev DOUBLE PRECISION for dsyev Arrays: *a*(*lda*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A*, as specified by *uplo*. The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of the array *a*. Must be at least max(1, *n*).
lwork	INTEGER. The dimension of the array *work*. Constraint: *lwork* ≥ max(1, 3*n*-1). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

See Application Notes for the suggested value of *lwork*.

Output Parameters

a	On exit, if *jobz* = 'V', then if *info* = 0, array *a* contains the orthonormal eigenvectors of the matrix *A*. If *jobz* = 'N', then on exit the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.
w	REAL for ssyev DOUBLE PRECISION for dsyev Array, DIMENSION at least max(1, *n*). If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order.
work(1)	On exit, if *lwork* > 0, then *work(1)* returns the required minimal size of *lwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine syev interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
w	Holds the vector of length (*n*).
job	Must be 'N' or 'V'. The default value is 'N'.
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For optimum performance use

lwork ≥ (*nb*+2)*n,

where *nb* is the blocksize for ?sytrd returned by ilaenv.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?heev

Computes all eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

Syntax

Fortran 77:

```
call cheev(jobz, uplo, n, a, lda, w, work, lwork, rwork, info)
call zheev(jobz, uplo, n, a, lda, w, work, lwork, rwork, info)
```

Fortran 95:

```
call heev(a, w [,jobz] [,uplo] [,info])
```

Description

This routine computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A*.

Note that for most cases of complex Hermitian eigenvalue problems the default choice should be ?heevr function as its underlying algorithm is faster and uses less workspace.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then only eigenvalues are computed. If *jobz* = 'V', then eigenvalues and eigenvectors are computed.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *a* stores the upper triangular part of *A*. If *uplo* = 'L', *a* stores the lower triangular part of *A*.
n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
a, *work*	COMPLEX for cheev DOUBLE COMPLEX for zheev Arrays: *a*(*lda*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A*, as specified by *uplo*. The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of the array *a*. Must be at least max(1, *n*).
lwork	INTEGER. The dimension of the array *work*. C onstraint: *lwork* ≥ max(1, 2*n*-1). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.
rwork	REAL for cheev DOUBLE PRECISION for zheev. Workspace array, DIMENSION at least max(1, 3*n*-2).

Output Parameters

a	On exit, if *jobz* = 'V', then if *info* = 0, array *a* contains the orthonormal eigenvectors of the matrix *A*. If *jobz* = 'N', then on exit the lower triangle

	(if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.
w	REAL for cheev DOUBLE PRECISION for zheev Array, DIMENSION at least max(1, *n*). If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order.
work(1)	On exit, if *lwork* > 0, then *work(1)* returns the required minimal size of *lwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine heev interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
w	Holds the vector of length (*n*).
job	Must be 'N' or 'V'. The default value is 'N'.
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For optimum performance use

lwork ≥ (*nb*+1)*n*,

where *nb* is the blocksize for ?hetrd returned by ilaenv.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?syevd

Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric matrix using divide and conquer algorithm.

Syntax

Fortran 77:

```
call ssyevd(jobz, uplo, n, a, lda, w, work, lwork, iwork, liwork, info)
call dsyevd(jobz, uplo, n, a, lda, w, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call syevd(a, w [,jobz] [,uplo] [,info])
```

Description

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric matrix *A*. In other words, it can compute the spectral factorization of *A* as: $A = Z\Lambda Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i, and *Z* is the orthogonal matrix whose columns are the eigenvectors z_i. Thus,

$Azi = \lambda_i z_i$ for $i = 1, 2, \ldots, n$.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the *QL* or *QR* algorithm.

Note that for most cases of real symmetric eigenvalue problems the default choice should be ?syevr function as its underlying algorithm is faster and uses less workspace. ?syevd requires more workspace but is faster in some cases, especially for large matrices.

Input Parameters

jobz — CHARACTER*1. Must be 'N' or 'V'.
If *jobz* = 'N', then only eigenvalues are computed.
If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

uplo — CHARACTER*1. Must be 'U' or 'L'.
If *uplo* = 'U', *a* stores the upper triangular part of *A*.
If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n — INTEGER. The order of the matrix *A* (*n* ≥ 0).

a — REAL for ssyevd
DOUBLE PRECISION for dsyevd
Array, DIMENSION (*lda*, *).
a(*lda*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A*, as specified by *uplo*.
The second dimension of *a* must be at least max(1, *n*).

lda — INTEGER. The first dimension of the array *a*.
Must be at least max(1, *n*).

work — REAL for ssyevd
DOUBLE PRECISION for dsyevd.
Workspace array, DIMENSION at least *lwork*.

lwork — INTEGER.
The dimension of the array *work*.
Constraints:

if *n* ≤ 1, then *lwork* ≥ 1;

if *jobz* = 'N' and *n* > 1, then *lwork* ≥ 2*n + 1;

if *jobz* = 'V' and *n* > 1, then *lwork* ≥ $2*n^2+ 6*n + 1$.
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the

work and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See Application Notes for details.

iwork — INTEGER.
Workspace array, its dimension max(1, *liwork*).

liwork — INTEGER.
The dimension of the array *iwork*.
Constraints:

if $n \le 1$, then *liwork* ≥ 1;

if *jobz* = 'N' and $n > 1$, then *liwork* ≥ 1;

if *jobz* = 'V' and $n > 1$, then *liwork* ≥ 5*n + 3.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See Application Notes for details.

Output Parameters

w — REAL for ssyevd
DOUBLE PRECISION for dsyevd
Array, DIMENSION at least max(1, *n*).
If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order. See also *info*.

a — If *jobz* = 'V', then on exit this array is overwritten by the orthogonal matrix *Z* which contains the eigenvectors of *A*.

work(1) — On exit, if *lwork* > 0, then *work(1)* returns the required minimal size of *lwork*.

iwork(1) — On exit, if *liwork* > 0, then *iwork(1)* returns the required minimal size of *liwork*.

info — INTEGER.
If *info* = 0, the execution is successful.

If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.
If *info* = *i*, and *jobz* = 'V', then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns *info*/(*n*+1) through mod(*info*,*n*+1).
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine syevd interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
w	Holds the vector of length (*n*).
jobz	Must be 'N' or 'V'. The default value is 'N'.
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T + E$ such that $||E||_2 = O(\varepsilon)\ ||T||_2$, where ε is the machine precision.

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If you choose the first option and set any of admissible *lwork* (or *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex analogue of this routine is ?heevd

?heevd

Computes all eigenvalues and (optionally) all eigenvectors of a complex Hermitian matrix using divide and conquer algorithm.

Syntax

Fortran 77:

```
call cheevd(jobz, uplo, n, a, lda, w, work, lwork, rwork, lrwork, iwork, liwork, info)
```

```
call zheevd(jobz, uplo, n, a, lda, w, work, lwork, rwork, lrwork, iwork, liwork, info)
```

Fortran 95:

```
call heevd(a, w [,job] [,uplo] [,info])
```

Description

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian matrix *A*. In other words, it can compute the spectral factorization of *A* as: $A = Z\Lambda Z^H$.

Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i, and *Z* is the (complex) unitary matrix whose columns are the eigenvectors z_i. Thus,

$Azi = \lambda_i zi$ for $i = 1, 2, \ldots, n$.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the *QL* or *QR* algorithm.

Note that for most cases of complex Hermetian eigenvalue problems the default choice should be ?heevr function as its underlying algorithm is faster and uses less workspace. ?heevd requires more workspace but is faster in some cases, especially for large matrices.

Input Parameters

jobz — CHARACTER*1. Must be 'N' or 'V'.
If *jobz* = 'N', then only eigenvalues are computed.
If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

uplo — CHARACTER*1. Must be 'U' or 'L'.
If *uplo* = 'U', *a* stores the upper triangular part of *A*.
If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n — INTEGER. The order of the matrix *A* (*n* ≥ 0).

a — COMPLEX for cheevd
DOUBLE COMPLEX for zheevd
Array, DIMENSION (*lda*, *).
a(*lda*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A*, as specified by *uplo*.
The second dimension of *a* must be at least max(1, *n*).

lda — INTEGER. The first dimension of the array *a*. Must be at least max(1, *n*).

work — COMPLEX for cheevd
DOUBLE COMPLEX for zheevd.
Workspace array, DIMENSION max(1, *lwork*).

lwork — INTEGER.
The dimension of the array *work*. Constraints:

if *n* ≤ 1, then *lwork* ≥ 1;

if *jobz* = 'N' and *n* > 1, then *lwork* ≥ *n*+1;

if *jobz* = 'V' and *n* > 1, then *lwork* ≥ n^2+2*n.
If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See Application Notes for details.

rwork — REAL for cheevd
DOUBLE PRECISION for zheevd

Workspace array, DIMENSION at least *lrwork*.

lrwork INTEGER.
The dimension of the array *rwork*. Constraints:

if $n \leq 1$, then *lrwork* ≥ 1;

if *job* = 'N' and $n > 1$, then *lrwork* $\geq n$;

if *job* = 'V' and $n > 1$, then *lrwork* $\geq 2*n^2+ 5*n + 1$.
If *lrwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See Application Notes for details.

iwork INTEGER. Workspace array, its dimension max(1, *liwork*).

liwork INTEGER.
The dimension of the array *iwork*. Constraints: if $n \leq 1$, then

liwork ≥ 1;

if *jobz* = 'N' and $n > 1$, then *liwork* ≥ 1;

if *jobz* = 'V' and $n > 1$, then *liwork* $\geq 5*n+3$.
If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See Application Notes for details.

Output Parameters

w REAL for cheevd
DOUBLE PRECISION for zheevd
Array, DIMENSION at least max(1, *n*).
If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order. See also *info*.

a If *jobz* = 'V', then on exit this array is overwritten by the unitary matrix *Z* which contains the eigenvectors of *A*.

work(1)	On exit, if *lwork* > 0, then the real part of *work*(1) returns the required minimal size of *lwork*.
rwork(1)	On exit, if *lrwork* > 0, then *rwork*(1) returns the required minimal size of *lrwork*.
iwork(1)	On exit, if *liwork* > 0, then *iwork*(1) returns the required minimal size of *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *i*, and *jobz* = 'N', then the algorithm failed to converge; *i* off-diagonal elements of an intermediate tridiagonal form did not converge to zero; if *info* = *i*, and *jobz* = 'V', then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns *info*/(*n*+1) through mod(*info*, *n*+1). If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine heevd interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
w	Holds the vector of length (*n*).
jobz	Must be 'N' or 'V'. The default value is 'N'.
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A + E$ such that $||E||_2 = O(\varepsilon)\ ||A||_2$, where ε is the machine precision.

If you are in doubt how much workspace to supply, use a generous value of *lwork* (*liwork* or *lrwork*) for the first run or set *lwork* = -1 (*liwork* = -1, *lrwork* = -1).

If you choose the first option and set any of admissible *lwork* (*liwork* or *lrwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*) on exit. Use this value (*work*(1), *iwork*(1), *rwork*(1)) for subsequent runs.

If you set *lwork* = -1 (*liwork* = -1, *lrwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*, *lrwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real analogue of this routine is ?syevd. See also ?hpevd for matrices held in packed storage, and ?hbevd for banded matrices.

?syevx

Computes selected eigenvalues and, optionally, eigenvectors of a symmetric matrix.

Syntax

Fortran 77:

```
call ssyevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z,
ldz, work, lwork, iwork, ifail, info)
```

```
call dsyevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z,
ldz, work, lwork, iwork, ifail, info)
```

Fortran 95:

```
call syevx(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol]
[,info])
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Note that for most cases of real symmetric eigenvalue problems the default choice should be ?syevr function as its underlying algorithm is faster and uses less workspace. ?syevx is faster for a few selected eigenvalues.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then only eigenvalues are computed. If *jobz* = 'V', then eigenvalues and eigenvectors are computed.
range	CHARACTER*1. Must be 'A', 'V', or 'I'. If *range* = 'A', all eigenvalues will be found. If *range* = 'V', all eigenvalues in the half-open interval (*vl*, *vu*] will be found. If *range* = 'I', the eigenvalues with indices *il* through *iu* will be found.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *a* stores the upper triangular part of *A*. If *uplo* = 'L', *a* stores the lower triangular part of *A*.
n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
a, *work*	REAL for ssyevx DOUBLE PRECISION for dsyevx. Arrays: *a*(*lda*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A*, as specified by *uplo*. The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of the array *a*. Must be at least max(1, *n*).
vl, *vu*	REAL for ssyevx DOUBLE PRECISION for dsyevx. If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues; *vl* ≤ *vu*. Not referenced if *range* = 'A'or 'I'.
il, *iu*	INTEGER.

If *range* = 'I', the indices of the smallest and largest eigenvalues to be returned.

Constraints: $1 \le il \le iu \le n$, if $n > 0$; $il = 1$ and $iu = 0$, if $n = 0$.

Not referenced if *range* = 'A' or 'V'.

abstol

REAL for ssyevx

DOUBLE PRECISION for dsyevx.

The absolute error tolerance for the eigenvalues. See Application Notes for more information.

ldz

INTEGER. The first dimension of the output array *z*; $ldz \ge 1$.

If *jobz* = 'V', then $ldz \ge \max(1, n)$.

lwork

INTEGER.

The dimension of the array *work*.

If $n \le 1$ then $lwork \ge 1$, otherwise $lwork = 8*n$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

iwork

INTEGER. Workspace array, DIMENSION at least max(1, $5n$).

Output Parameters

a

On exit, the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

m

INTEGER. The total number of eigenvalues found;

$0 \le m \le n$.

If *range* = 'A', $m = n$, and if *range* = 'I', $m = iu-il+1$.

w

REAL for ssyevx

DOUBLE PRECISION for dsyevx

Array, DIMENSION at least max(1, n). The first m elements contain the selected eigenvalues of the matrix *A* in ascending order.

z	REAL for ssyevx DOUBLE PRECISION for dsyevx. Array *z*(*ldz*,*) contains eigenvectors. The second dimension of *z* must be at least max(1, *m*). If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the i-th column of *z* holding the eigenvector associated with *w*(i). If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*. If *jobz* = 'N', then *z* is not referenced. Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.
work(1)	On exit, if *lwork* > 0, then *work*(1) returns the required minimal size of *lwork*.
ifail	INTEGER. Array, DIMENSION at least max(1, *n*). If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, then *ifail* contains the indices of the eigenvectors that failed to converge. If *jobz* = 'V', then *ifail* is not referenced.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine syevx interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).

w	Holds the vector of length (*n*).
a	Holds the matrix *A* of size (*m*, *n*).
ifail	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
vl	Default value for this element is *vl* = -HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted Note that there will be an error condition if *ifail* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

For optimum performance use *lwork* ≥ (*nb*+3)***n*, where *nb* is the maximum of the blocksize for ?sytrd and ?ormtr returned by ilaenv.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol* + ε * max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*|*T*| is used in its place, where|*T*| is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*slamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*slamch('S').

?heevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

Syntax

Fortran 77:

```
call cheevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z,
ldz, work, lwork, rwork, iwork, ifail, info)
```

```
call zheevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z,
ldz, work, lwork, rwork, iwork, ifail, info)
```

Fortran 95:

```
call heevx(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol]
[,info])
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Note that for most cases of complex Hermetian eigenvalue problems the default choice should be ?heevr function as its underlying algorithm is faster and uses less workspace. ?heevx is faster for a few selected eigenvalues.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then only eigenvalues are computed. If *jobz* = 'V', then eigenvalues and eigenvectors are computed.
range	CHARACTER*1. Must be 'A', 'V', or 'I'. If *range* = 'A', all eigenvalues will be found. If *range* = 'V', all eigenvalues in the half-open interval (*vl*, *vu*] will be found. If *range* = 'I', the eigenvalues with indices *il* through *iu* will be found.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *a* stores the upper triangular part of *A*. If *uplo* = 'L', *a* stores the lower triangular part of *A*.
n	INTEGER. The order of the matrix *A* (*n* < 0).
a, *work*	COMPLEX for cheevx DOUBLE COMPLEX for zheevx. Arrays: *a*(*lda*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A*, as specified by *uplo*. The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of the array *a*. Must be at least max(1, *n*).
vl, *vu*	REAL for cheevx DOUBLE PRECISION for zheevx. If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues; *vl* ≤ *vu*. Not referenced if *range* = 'A'or 'I'.
il, *iu*	INTEGER. If *range* = 'I', the indices of the smallest and largest eigenvalues to be returned. Constraints: 1 ≤ *il* ≤ *iu* ≤ *n*, if *n* > 0;*il* = 1 and *iu* = 0, if *n* = 0. Not referenced if *range* = 'A'or 'V'.

abstol REAL for cheevx
DOUBLE PRECISION for zheevx. The absolute error tolerance for the eigenvalues. See Application Notes for more information.

ldz INTEGER. The first dimension of the output array *z*; *ldz* ≥ 1.
If *jobz* = 'V', then *ldz* ≥max(1, *n*).

lwork INTEGER.
The dimension of the array *work*.
lwork ≥ 1 if *n*≤1; otherwise at least 2**n*.
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.
See Application Notes for the suggested value of *lwork*.

rwork REAL for cheevx
DOUBLE PRECISION for zheevx.
Workspace array, DIMENSION at least max(1, 7*n*).

iwork INTEGER. Workspace array, DIMENSION at least max(1, 5*n*).

Output Parameters

a On exit, the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

m INTEGER. The total number of eigenvalues found;0 ≤ *m* ≤ *n*.
If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu*-*il*+1.

w REAL for cheevx
DOUBLE PRECISION for zheevx
Array, DIMENSION at least max(1, *n*). The first *m* elements contain the selected eigenvalues of the matrix *A* in ascending order.

z COMPLEX for cheevx
DOUBLE COMPLEX for zheevx.
Array *z*(*ldz*,*) contains eigenvectors.
The second dimension of *z* must be at least max(1, *m*).

	If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the i-th column of *z* holding the eigenvector associated with *w*(i). If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*. If *jobz* = 'N', then *z* is not referenced. Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.
work(1)	On exit, if *lwork* > 0, then *work(1)* returns the required minimal size of *lwork*.
ifail	INTEGER. Array, DIMENSION at least max(1, *n*). If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, then *ifail* contains the indices of the eigenvectors that failed to converge. If *jobz* = 'V', then *ifail* is not referenced.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = –*i*, the *i*-th parameter had an illegal value. If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine heevx interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*).
ifail	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.

vl	Default value for this element is *vl* = -HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted Note that there will be an error condition if *ifail* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

For optimum performance use *lwork* ≥ (*nb*+1)***n*, where *nb* is the maximum of the blocksize for ?hetrd and ?unmtr returned by ilaenv.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol* + ε * max(|a|,|b|) , where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*|*T*| will be used in its place, where |*T*| is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*slamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*slamch('S').

?syevr

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix using the Relatively Robust Representations.

Syntax

Fortran 77:

```
call ssyevr(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z,
ldz, isuppz, work, lwork, iwork, liwork, info)
```

```
call dsyevr(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z,
ldz, isuppz, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call syevr(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol]
[,info])
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

The routine first reduces the matrix *A* to tridiagonal form *T* with a call to ?sytrd. Then, whenever possible, ?syevr calls ?stemr to compute the eigenspectrum using Relatively Robust Representations. ?stemr computes eigenvalues by the *dqds* algorithm, while orthogonal eigenvectors are computed from various "good" $L*D*L^T$ representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the each unreduced block of *T*,

a. Compute $T - \sigma * I = L * D * L^T$, so that L and D define all the wanted eigenvalues to high relative accuracy. This means that small relative changes in the entries of D and L cause only small relative changes in the eigenvalues and eigenvectors. The standard (unfactored) representation of the tridiagonal matrix T does not have this property in general;

b. Compute the eigenvalues to suitable accuracy. If the eigenvectors are desired, the algorithm attains full accuracy of the computed eigenvalues only right before the corresponding vectors have to be computed, see steps c) and d);

c. For each cluster of close eigenvalues, select a new shift close to the cluster, find a new factorization, and refine the shifted eigenvalues to suitable accuracy;

d. For each eigenvalue with a large enough relative separation compute the corresponding eigenvector by forming a rank revealing twisted factorization. Go back to c) for any clusters that remain.

The desired accuracy of the output can be specified by the input parameter *abstol*.

The routine ?syevr calls ?stemr when the full spectrum is requested on machines that conform to the IEEE-754 floating point standard. ?syevr calls ?stebz and ?stein on non-IEEE machines and when partial spectrum requests are made.

Note that ?syevr is preferable for most cases of real symmetric eigenvalue problems as its underlying algorithm is fast and uses less workspace.

Input Parameters

jobz — CHARACTER*1. Must be 'N' or 'V'.
If *jobz* = 'N', then only eigenvalues are computed.
If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

range — CHARACTER*1. Must be 'A' or 'V' or 'I'.
If *range* = 'A', the routine computes all eigenvalues.
If *range* = 'V', the routine computes eigenvalues *lambda*(*i*) in the half-open interval:
vl < *lambda*(*i*) ≤ *vu*.
If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.
For *range* = 'V' or 'I' and *iu*-*il* < *n*-1, sstebz/dstebz and sstein/dstein are called.

uplo — CHARACTER*1. Must be 'U' or 'L'.

	If *uplo* = 'U', *a* stores the upper triangular part of *A*. If *uplo* = 'L', *a* stores the lower triangular part of *A*.
n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
a, *work*	REAL for ssyevr DOUBLE PRECISION for dsyevr. Arrays: *a*(*lda*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A*, as specified by *uplo*. The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of the array *a*. Must be at least max(1, *n*).
vl, *vu*	REAL for ssyevr DOUBLE PRECISION for dsyevr. If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: *vl*< *vu*. If *range* = 'A' or 'I', *vl* and *vu* are not referenced.
il, *iu*	INTEGER. If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: 1 ≤ *il* ≤ *iu* ≤ *n*, if *n* > 0; *il*=1 and *iu*=0, if *n* = 0. If *range* = 'A' or 'V', *il* and *iu* are not referenced.
abstol	REAL for ssyevr DOUBLE PRECISION for dsyevr. The absolute error tolerance to which each eigenvalue/eigenvector is required. If *jobz* = 'V', the eigenvalues and eigenvectors output have residual norms bounded by *abstol*, and the dot products between different eigenvectors are bounded by *abstol*.

If *abstol* < *n* *eps*|*T*|, then *n* *eps*|*T*| will be used in its place, where eps is the machine precision, and |*T*| is the 1-norm of the matrix *T*. The eigenvalues are computed to an accuracy of eps*|*T*| irrespective of *abstol*.
If high relative accuracy is important, set *abstol* to ?lamch('S').

ldz — INTEGER. The leading dimension of the output array *z*.
Constraints:

ldz ≥ 1 if *jobz* = 'N';
ldz < max(1, *n*) if *jobz* = 'V'.

lwork — INTEGER.
The dimension of the array *work*.

Constraint: *lwork* ≥ max(1, 26*n*).
If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.
See Application Notes for the suggested value of *lwork*.

iwork — INTEGER. Workspace array, its dimension max(1, *liwork*).

liwork — INTEGER.

The dimension of the array *iwork*, *lwork* ≥ max(1, 10*n*).
If *liwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by xerbla.

Output Parameters

a — On exit, the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

m — INTEGER. The total number of eigenvalues found, 0 ≤ *m* ≤ *n*.
If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu*-*il*+1.

w, *z* — REAL for ssyevr

DOUBLE PRECISION for dsyevr.
Arrays:
w(*), DIMENSION at least max(1, *n*), contains the selected eigenvalues in ascending order, stored in *w*(1) to *w*(*m*);
z(*ldz*, *), the second dimension of *z* must be at least max(1, *m*).
If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).
If *jobz* = 'N', then *z* is not referenced. Note that you must ensure that at least max(1, *m*) columns are supplied in the array *z* ; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

isuppz — INTEGER.
Array, DIMENSION at least 2 *max(1, *m*).
The support of the eigenvectors in *z*, i.e., the indices indicating the nonzero elements in *z*. The *i*-th eigenvector is nonzero only in elements *isuppz*(2*i*-1) through *isuppz*(2*i*). Currently is not implemented, nor referenced.

work(1) — On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

iwork(1) — On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *liwork*.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, an internal error has occurred.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine syevr interface are the following:

a — Holds the matrix *A* of size (*n*, *n*).

w — Holds the vector of length (*n*).

z	Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.
isuppz	Holds the vector of length (2**m*), where the values (2**m*) are significant.
uplo	Must be 'U' or 'L'. The default value is 'U'.
vl	Default value for this element is *vl* = -HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted Note that there will be an error condition if *isuppz* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

For optimum performance use *lwork* ≥ (*nb*+6)**n*, where *nb* is the maximum of the blocksize for ?sytrd and ?ormtr returned by ilaenv.

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If you choose the first option and set any of admissible *lwork* (or *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Normal execution of ?stegr may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

?heevr

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix using the Relatively Robust Representations.

Syntax

Fortran 77:

```
call cheevr(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z,
ldz, isuppz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

```
call zheevr(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z,
ldz, isuppz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

Fortran 95:

```
call heevr(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol]
[,info])
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

THe routine first reduces the matrix *A* to tridiagonal form *T* with a call to ?HETRD. Then, whenever possible, ?heevr calls ?stegr to compute the eigenspectrum using Relatively Robust Representations. ?stegr computes eigenvalues by the *dqds* algorithm, while orthogonal eigenvectors are computed from various "good" $L*D*L^T$ representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For each unreduced block (submatrix) of *T*,

a. Compute $T - \sigma * I = L * D * L^T$, so that L and D define all the wanted eigenvalues to high relative accuracy. This means that small relative changes in the entries of D and L cause only small relative changes in the eigenvalues and eigenvectors. The standard (unfactored) representation of the tridiagonal matrix T does not have this property in general;

b. compute the eigenvalues to suitable accuracy. If the eigenvectors are desired, the algorithm attains full accuracy of the computed eigenvalues only right before the corresponding vectors have to be computed, see steps c) and d);

c. for each cluster of close eigenvalues, select a new shift close to the cluster, find a new factorization, and refine the shifted eigenvalues to suitable accuracy;

d. For each eigenvalue with a large enough relative separation compute the corresponding eigenvector by forming a rank revealing twisted factorization. Go back to the step c) for any clusters that remain.

The desired accuracy of the output can be specified by the input parameter *abstol*.

The routine ?heevr calls ?stemr when the full spectrum is requested on machines which conform to the IEEE-754 floating point standard, or ?stebz and ?stein on non-IEEE machines and when partial spectrum requests are made.

Note that the routine ?heevr is preferable for most cases of complex Hermitian eigenvalue problems as its underlying algorithm is fast and uses less workspace.

Input Parameters

jobz — CHARACTER*1. Must be 'N' or 'V'.
If *job* = 'N', then only eigenvalues are computed.
If *job* = 'V', then eigenvalues and eigenvectors are computed.

range — CHARACTER*1. Must be 'A' or 'V' or 'I'.
If *range* = 'A', the routine computes all eigenvalues.
If *range* = 'V', the routine computes eigenvalues *lambda*(*i*) in the half-open interval: $vl < lambda(i) \leq vu$.
If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.
For *range* = 'V'or 'I', sstebz/dstebz and cstein/zstein are called.

uplo — CHARACTER*1. Must be 'U' or 'L'.
If *uplo* = 'U', *a* stores the upper triangular part of *A*.

If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n
INTEGER. The order of the matrix *A* (*n* ≥ 0).

a, *work*
COMPLEX for cheevr
DOUBLE COMPLEX for zheevr.
Arrays:
a(*lda*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A*, as specified by *uplo*.
The second dimension of *a* must be at least max(1, *n*).
work is a workspace array, its dimension max(1, *lwork*).

lda
INTEGER. The first dimension of the array *a*.
Must be at least max(1, *n*).

vl, *vu*
REAL for cheevr
DOUBLE PRECISION for zheevr.
If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
Constraint: *vl*< *vu*.
If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu*
INTEGER.
If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: 1 ≤ *il* ≤ *iu* ≤ *n*, if *n* > 0; *il*=1 and *iu*=0 if *n* = 0.
If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol
REAL for cheevr
DOUBLE PRECISION for zheevr.
The absolute error tolerance to which each eigenvalue/eigenvector is required.
If *jobz* = 'V', the eigenvalues and eigenvectors output have residual norms bounded by *abstol*, and the dot products between different eigenvectors are bounded by *abstol*.
If *abstol* < *n* *eps*|*T*|, then *n* *eps*|*T*| will be used in its place, where eps is the machine precision, and |*T*| is the 1-norm of the matrix *T*. The eigenvalues are computed to an accuracy of eps*|*T*| irrespective of *abstol*.

If high relative accuracy is important, set *abstol* to ?lamch('S').

ldz INTEGER. The leading dimension of the output array *z*.
Constraints:

ldz ≥ 1 if *jobz* = 'N';

ldz ≥ max(1, *n*) if *jobz* = 'V'.

lwork INTEGER.
The dimension of the array *work*.

Constraint: *lwork* ≥ max(1, 2*n*).
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

rwork REAL for cheevr
DOUBLE PRECISION for zheevr.
Workspace array, DIMENSION max(1, *lwork*).

lrwork INTEGER.
The dimension of the array *rwork*;

lwork ≥ max(1, 24*n*).
If *lrwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla.

iwork INTEGER. Workspace array, its dimension max(1, *liwork*).

liwork INTEGER.
The dimension of the array *iwork*,

lwork ≥ max(1, 10*n*).
If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla.

Output Parameters

a	On exit, the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.
m	INTEGER. The total number of eigenvalues found, $0 \le m \le n$. If *range* = 'A', $m = n$, and if *range* = 'I', $m = iu-il+1$.
w	REAL for cheevr DOUBLE PRECISION for zheevr. Array, DIMENSION at least max(1, *n*), contains the selected eigenvalues in ascending order, stored in *w*(1) to *w*(*m*).
z	COMPLEX for cheevr DOUBLE COMPLEX for zheevr. Array *z*(*ldz*, *); the second dimension of *z* must be at least max(1, *m*). If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). If *jobz* = 'N', then *z* is not referenced. Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.
isuppz	INTEGER. Array, DIMENSION at least 2 *max(1, *m*). The support of the eigenvectors in *z*, i.e., the indices indicating the nonzero elements in *z*. The *i*-th eigenvector is nonzero only in elements *isuppz*(2*i*-1) through *isuppz*(2*i*).
work(1)	On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.
rwork(1)	On exit, if *info* = 0, then *rwork(1)* returns the required minimal size of *lrwork*.

iwork(1)	On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, an internal error has occurred.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine heevr interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.
isuppz	Holds the vector of length (2**n*), where the values (2**m*) are significant.
uplo	Must be 'U' or 'L'. The default value is 'U'.
vl	Default value for this element is *vl* = -HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted Note that there will be an error condition if *isuppz* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

For optimum performance use *lwork* ≥ (*nb*+1)***n*, where *nb* is the maximum of the blocksize for ?hetrd and ?unmtr returned by ilaenv.

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *lrwork*, or *liwork*) for the first run or set *lwork* = -1 (*lrwork* = -1, *liwork* = -1).

If you choose the first option and set any of admissible *lwork* (or *lrwork*, *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *rwork*, *iwork*) on exit. Use this value (*work*(1), *rwork*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *rwork*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*lrwork*, *liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Normal execution of ?stegr may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

?spev

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix in packed storage.

Syntax

Fortran 77:

```
call sspev(jobz, uplo, n, ap, w, z, ldz, work, info)
call dspev(jobz, uplo, n, ap, w, z, ldz, work, info)
```

Fortran 95:

```
call spev(a, w [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A* in packed storage.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *job* = 'N', then only eigenvalues are computed. If *job* = 'V', then eigenvalues and eigenvectors are computed.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *ap* stores the packed upper triangular part of *A*. If *uplo* = 'L', *ap* stores the packed lower triangular part of *A*.
n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
ap, *work*	REAL for sspev DOUBLE PRECISION for dspev Arrays: *ap(*)* contains the packed upper or lower triangle of symmetric matrix *A*, as specified by *uplo*. The dimension of *ap* must be at least max(1, *n**(*n*+1)/2).
work	(*) is a workspace array, DIMENSION at least max(1, 3*n*).
ldz	INTEGER. The leading dimension of the output array *z*. Constraints: if *jobz* = 'N', then *ldz* ≥ 1; if *jobz* = 'V', then *ldz* ≥ max(1, *n*).

Output Parameters

w, *z*	REAL for sspev DOUBLE PRECISION for dspev Arrays: *w*(*), DIMENSION at least max(1, *n*). If *info* = 0, *w* contains the eigenvalues of the matrix *A* in ascending order.

	z(ldz,*). The second dimension of z must be at least max(1, n). If jobz = 'V', then if info = 0, z contains the orthonormal eigenvectors of the matrix A, with the i-th column of z holding the eigenvector associated with w(i). If jobz = 'N', then z is not referenced.
ap	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of A.
info	INTEGER. If info = 0, the execution is successful. If info = -i, the i-th parameter had an illegal value. If info = i, then the algorithm failed to converge; i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine spev interface are the following:

a	Stands for argument ap in Fortan 77 interface. Holds the array A of size (n*(n+1)/2).
w	Holds the vector of length (n).
z	Holds the matrix Z of size (n, n).
uplo	Must be 'U' or 'L'. The default value is 'U'.
jobz	Restored based on the presence of the argument z as follows: jobz = 'V', if z is present, jobz = 'N', if z is omitted.

?hpev

Computes all eigenvalues and, optionally, eigenvectors of a Hermitian matrix in packed storage.

Syntax

Fortran 77:

```
call chpev(jobz, uplo, n, ap, w, z, ldz, work, rwork, info)
call zhpev(jobz, uplo, n, ap, w, z, ldz, work, rwork, info)
```

Fortran 95:

```
call hpev(a, w [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A* in packed storage.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *job* = 'N', then only eigenvalues are computed. If *job* = 'V', then eigenvalues and eigenvectors are computed.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *ap* stores the packed upper triangular part of *A*. If *uplo* = 'L', *ap* stores the packed lower triangular part of *A*.
n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
ap, *work*	COMPLEX for chpev DOUBLE COMPLEX for zhpev. Arrays: *ap(*)* contains the packed upper or lower triangle of Hermitian matrix A, as specified by *uplo*. The dimension of *ap* must be at least max(1, *n**(*n*+1)/2).

work (*) is a workspace array, DIMENSION at least max(1, 2n-1).

ldz INTEGER. The leading dimension of the output array *z*.
Constraints:
if *jobz* = 'N', then *ldz* ≥ 1;
if *jobz* = 'V', then *ldz* ≥ max(1, *n*) .

rwork REAL for chpev
DOUBLE PRECISION for zhpev.
Workspace array, DIMENSION at least max(1, 3n-2).

Output Parameters

w REAL for chpev
DOUBLE PRECISION for zhpev.
Array, DIMENSION at least max(1, *n*).
If *info* = 0, *w* contains the eigenvalues of the matrix *A* in ascending order.

z COMPLEX for chpev
DOUBLE COMPLEX for zhpev.
Array z(*ldz*,*).
The second dimension of *z* must be at least max(1, *n*).
If *jobz* = 'V', then if *info* = 0, *z* contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).
If *jobz* = 'N', then *z* is not referenced.

ap On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of A.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `hpev` interface are the following:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n*(n+1)/2*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = `'V'`, if *z* is present, *jobz* = `'N'`, if *z* is omitted.

?spevd

Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a real symmetric matrix held in packed storage.

Syntax

Fortran 77:

```
call sspevd(jobz, uplo, n, ap, w, z, ldz, work, lwork, iwork, liwork, info)
call dspevd(jobz, uplo, n, ap, w, z, ldz, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call spevd(a, w [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric matrix *A* (held in packed storage). In other words, it can compute the spectral factorization of *A* as:

$A = Z\Lambda Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i, and Z is the orthogonal matrix whose columns are the eigenvectors z_i. Thus,

$Az_i = \lambda_i z_i$ for $i = 1, 2, \ldots, n$.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then only eigenvalues are computed. If *jobz* = 'V', then eigenvalues and eigenvectors are computed.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *ap* stores the packed upper triangular part of *A*. If *uplo* = 'L', *ap* stores the packed lower triangular part of *A*.
n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
ap, *work*	REAL for sspevd DOUBLE PRECISION for dspevd Arrays: *ap(*)* contains the packed upper or lower triangle of symmetric matrix A, as specified by *uplo*. The dimension of *ap* must be at least max(1, *n**(*n*+1)/2) *work* is a workspace array, its dimension max(1, *lwork*).
ldz	INTEGER. The leading dimension of the output array *z*. Constraints: if *jobz* = 'N', then *ldz* ≥ 1; if *jobz* = 'V', then *ldz* ≥ max(1, *n*).
lwork	INTEGER. The dimension of the array *work*. Constraints: if *n* ≤ 1, then *lwork* ≥ 1; if *jobz* = 'N' and *n* > 1, then *lwork* ≥ 2**n*;

if *jobz* = 'V' and *n* > 1, then

lwork ≥ n^2+ 6**n* + 1.

If *lwork* = −1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See Application Notes for details.

iwork — INTEGER. Workspace array, its dimension max(1, *liwork*).

liwork — INTEGER.

The dimension of the array *iwork*.

Constraints:

if *n* ≤ 1, then *liwork* ≥ 1;

if *jobz* = 'N' and *n* > 1, then *liwork* ≥ 1;

if *jobz* = 'V' and *n* > 1, then *liwork* ≥ 5**n*+3.

If *liwork* = −1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See Application Notes for details.

Output Parameters

w, *z*

REAL for sspevd

DOUBLE PRECISION for dspevd

Arrays:

w(*), DIMENSION at least max(1, *n*).

If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order. See also *info*.

z(*ldz*,*).

The second dimension of *z* must be: at least 1 if *jobz* = 'N';at least max(1, *n*) if *jobz* = 'V'.

If *jobz* = 'V', then this array is overwritten by the orthogonal matrix *Z* which contains the eigenvectors of *A*.

If *jobz* = 'N', then *z* is not referenced.

ap	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of A.
work(1)	On exit, if *info* = 0, then *work*(1) returns the required *lwork*.
iwork(1)	On exit, if *info* = 0, then *iwork*(1) returns the required *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero. If *info* = –*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine spevd interface are the following:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n*(n+1)/2*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T + E$ such that $||E||_2 = O(\varepsilon)||T||_2$, where ε is the machine precision.

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = –1 (*liwork* = –1).

If you choose the first option and set any of admissible *lwork* (or *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = -1 (*liwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex analogue of this routine is ?hpevd.

See also ?syevd for matrices held in full storage, and ?sbevd for banded matrices.

?hpevd

Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a complex Hermitian matrix held in packed storage.

Syntax

Fortran 77:

```
call chpevd(jobz, uplo, n, ap, w, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

```
call zhpevd(jobz, uplo, n, ap, w, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

Fortran 95:

```
call hpevd(a, w [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian matrix A (held in packed storage). In other words, it can compute the spectral factorization of A as: $A = Z\Lambda Z^H$.

Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i, and Z is the (complex) unitary matrix whose columns are the eigenvectors z_i. Thus,

$Az_i = \lambda_i z_i$ for $i = 1, 2, \ldots, n$.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then only eigenvalues are computed. If *jobz* = 'V', then eigenvalues and eigenvectors are computed.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *ap* stores the packed upper triangular part of *A*. If *uplo* = 'L', *ap* stores the packed lower triangular part of *A*.
n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
ap, *work*	COMPLEX for chpevd DOUBLE COMPLEX for zhpevd Arrays: *ap(*)* contains the packed upper or lower triangle of Hermitian matrix A, as specified by *uplo*. The dimension of *ap* must be at least max(1, *n**(*n*+1)/2). *work* is a workspace array, its dimension max(1, *lwork*).
ldz	INTEGER. The leading dimension of the output array *z*. Constraints: if *jobz* = 'N', then *ldz* ≥ 1; if *jobz* = 'V', then *ldz* ≥ max(1, *n*).
lwork	INTEGER. The dimension of the array *work*. Constraints: if *n* ≤ 1, then *lwork* ≥ 1; if *jobz* = 'N' and *n* > 1, then *lwork* ≥ *n*;

if *jobz* = 'V' and *n* > 1, then *lwork* ≥ 2*n.
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See Application Notes for details.

rwork REAL for chpevd
DOUBLE PRECISION for zhpevd
Workspace array, its dimension max(1, *lrwork*).

lrwork INTEGER.
The dimension of the array *rwork*. Constraints:

if *n* ≤ 1, then *lrwork* ≥ 1;

if *jobz* = 'N' and *n* > 1, then *lrwork* ≥ *n*;

if *jobz* = 'V' and *n* > 1, then *lrwork* ≥ 2*n^2 + 5*n + 1.
If *lrwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See Application Notes for details.

iwork INTEGER. Workspace array, its dimension max(1, *liwork*).

liwork INTEGER.
The dimension of the array *iwork*.
Constraints:

if *n* ≤ 1, then *liwork* ≥ 1;

if *jobz* = 'N' and *n* > 1, then *liwork* ≥ 1;

if *jobz* = 'V' and *n* > 1, then *liwork* ≥ 5*n+3.
If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See Application Notes for details.

Output Parameters

w	REAL for chpevd DOUBLE PRECISION for zhpevd Array, DIMENSION at least max(1, *n*). If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order. See also *info*.
z	COMPLEX for chpevd DOUBLE COMPLEX for zhpevd Array, DIMENSION (*ldz*,*). The second dimension of *z* must be: at least 1 if *jobz* = 'N'; at least max(1, *n*) if *jobz* = 'V'. If *jobz* = 'V', then this array is overwritten by the unitary matrix *Z* which contains the eigenvectors of *A*. If *jobz* = 'N', then *z* is not referenced.
ap	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of A.
work(1)	On exit, if *info* = 0, then *work*(1) returns the required minimal size of *lwork*.
rwork(1)	On exit, if *info* = 0, then *rwork*(1) returns the required minimal size of *lrwork*.
iwork(1)	On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `hpevd` interface are the following:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n*(n+1)/2*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = `'V'`, if *z* is present, *jobz* = `'N'`, if *z* is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T + E$ such that $||E||_2 = O(\varepsilon)||T||_2$, where ε is the machine precision.

If you are in doubt how much workspace to supply, use a generous value of *lwork* (*liwork* or *lrwork*) for the first run or set *lwork* = −1 (*liwork* = −1, *lrwork* = −1).

If you choose the first option and set any of admissible *lwork* (*liwork* or *lrwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*) on exit. Use this value (*work*(1), *iwork*(1), *rwork*(1)) for subsequent runs.

If you set *lwork* = −1 (*liwork* = −1, *lrwork* = −1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*, *lrwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real analogue of this routine is `?spevd`.

See also `?heevd` for matrices held in full storage, and `?hbevd` for banded matrices.

?spevx

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix in packed storage.

Syntax

Fortran 77:

```
call sspevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
```

```
call dspevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
```

Fortran 95:

```
call spevx(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol] [,info])
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A* in packed storage. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *job* = 'N', then only eigenvalues are computed. If *job* = 'V', then eigenvalues and eigenvectors are computed.
range	CHARACTER*1. Must be 'A' or 'V' or 'I'. If *range* = 'A', the routine computes all eigenvalues. If *range* = 'V', the routine computes eigenvalues *lambda*(*i*) in the half-open interval: $vl < lambda(i) \le vu$. If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *ap* stores the packed upper triangular part of *A*.

If *uplo* = 'L', *ap* stores the packed lower triangular part of *A*.

n — INTEGER. The order of the matrix *A* (*n* ≥ 0).

ap, *work* — REAL for sspevx
DOUBLE PRECISION for dspevx
Arrays:
ap(*) contains the packed upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*.
The dimension of *ap* must be at least max(1, *n**(*n*+1)/2).
work(*) is a workspace array, DIMENSION at least max(1, 8*n*).

vl, *vu* — REAL for sspevx
DOUBLE PRECISION for dspevx
If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
Constraint: *vl*< *vu*.
If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu* — INTEGER.
If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: 1 ≤ *il* ≤ *iu* ≤ *n*, if *n* > 0; *il*=1 and *iu*=0 if *n* = 0.
If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol — REAL for sspevx
DOUBLE PRECISION for dspevx
The absolute error tolerance to which each eigenvalue is required. See Application notes for details on error tolerance.

ldz — INTEGER. The leading dimension of the output array *z*.
Constraints:
if *jobz* = 'N', then *ldz* ≥ 1;
if *jobz* = 'V', then *ldz* ≥ max(1, *n*).

iwork — INTEGER. Workspace array, DIMENSION at least max(1, 5*n*).

Output Parameters

ap	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of *A*.
m	INTEGER. The total number of eigenvalues found, $0 \le m \le n$. If *range* = 'A', $m = n$, and if *range* = 'I', $m = iu-il+1$.
w, *z*	REAL for sspevx DOUBLE PRECISION for dspevx Arrays: *w(*)*, DIMENSION at least max(1, *n*). If *info* = 0, contains the selected eigenvalues of the matrix *A* in ascending order. z(*ldz*,*). The second dimension of *z* must be at least max(1, *m*). If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*. If *jobz* = 'N', then *z* is not referenced. Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.
ifail	INTEGER. Array, DIMENSION at least max(1, *n*). If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices the eigenvectors that failed to converge. If *jobz* = 'N', then *ifail* is not referenced.
info	INTEGER. If *info* = 0, the execution is successful.

If *info* = *-i*, the *i*-th parameter had an illegal value.
If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine spevx interface are the following:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n*(n+1)/2*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.
ifail	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
vl	Default value for this element is *vl* = -HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted Note that there will be an error condition if *ifail* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol* + ε * max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*||*T*||$_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?hpevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix in packed storage.

Syntax

Fortran 77:

```
call chpevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz,
work, rwork, iwork, ifail, info)
```

```
call zhpevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz,
work, rwork, iwork, ifail, info)
```

Fortran 95:

```
call hpevx(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol]
[,info])
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A* in packed storage. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *job* = 'N', then only eigenvalues are computed.

If *job* = 'V', then eigenvalues and eigenvectors are computed.

range — CHARACTER*1. Must be 'A' or 'V' or 'I'.
If *range* = 'A', the routine computes all eigenvalues.
If *range* = 'V', the routine computes eigenvalues *lambda*(*i*) in the half-open interval: *vl*<*lambda*(*i*) ≤ *vu*.
If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.

uplo — CHARACTER*1. Must be 'U' or 'L'.
If *uplo* = 'U', *ap* stores the packed upper triangular part of *A*.
If *uplo* = 'L', *ap* stores the packed lower triangular part of *A*.

n — INTEGER. The order of the matrix *A* (*n* ≥ 0).

ap, *work* — COMPLEX for chpevx
DOUBLE COMPLEX for zhpevx
Arrays:
ap(*) contains the packed upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*.
The dimension of *ap* must be at least max(1, *n**(*n*+1)/2).
work(*) is a workspace array, DIMENSION at least max(1, 2*n*).

vl, *vu* — REAL for chpevx
DOUBLE PRECISION for zhpevx
If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
Constraint: *vl*< *vu*.
If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu* — INTEGER.
If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: 1 ≤ *il* ≤ *iu* ≤ *n*, if *n* > 0; *il*=1 and *iu*=0 if *n* = 0.
If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol — REAL for chpevx
DOUBLE PRECISION for zhpevx

The absolute error tolerance to which each eigenvalue is required. See Application notes for details on error tolerance.

ldz — INTEGER. The leading dimension of the output array *z*. Constraints:

if *jobz* = 'N', then *ldz* ≥ 1;

if *jobz* = 'V', then *ldz* ≥ max(1, *n*).

rwork — REAL for chpevx
DOUBLE PRECISION for zhpevx
Workspace array, DIMENSION at least max(1, 7*n*).

iwork — INTEGER. Workspace array, DIMENSION at least max(1, 5*n*).

Output Parameters

ap — On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of *A*.

m — INTEGER. The total number of eigenvalues found, 0 ≤ *m* ≤ *n*.
If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu*-*il*+1.

w — REAL for chpevx
DOUBLE PRECISION for zhpevx
Array, DIMENSION at least max(1, *n*).
If *info* = 0, contains the selected eigenvalues of the matrix *A* in ascending order.

z — COMPLEX for chpevx
DOUBLE COMPLEX for zhpevx
Array z(*ldz*,*).
The second dimension of *z* must be at least max(1, *m*).
If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).
If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

	If *jobz* = 'N', then *z* is not referenced. Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.
ifail	INTEGER. Array, DIMENSION at least max(1, *n*). If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices the eigenvectors that failed to converge. If *jobz* = 'N', then *ifail* is not referenced.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hpevx interface are the following:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n*(n+1)/2*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.
ifail	Holds the vector of length (*n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
vl	Default value for this element is *vl* = -HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.

jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted Note that there will be an error condition if *ifail* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol* + ε * max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then $\varepsilon * ||T||_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?sbev

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric band matrix.

Syntax

Fortran 77:

```
call ssbev(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, info)
call dsbev(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, info)
```

Fortran 95:

```
call sbev(a, w [,uplo] [,z] [,info])
```

Description

This routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric band matrix *A*.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then only eigenvalues are computed. If *jobz* = 'V', then eigenvalues and eigenvectors are computed.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *ab* stores the upper triangular part of *A*. If *uplo* = 'L', *ab* stores the lower triangular part of *A*.
n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
kd	INTEGER. The number of super- or sub-diagonals in *A* (*kd* ≥ 0).
ab, *work*	REAL for ssbev DOUBLE PRECISION for dsbev. Arrays: *ab* (*ldab*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A* (as specified by *uplo*) in band storage format. The second dimension of *ab* must be at least max(1, *n*). *work* (*) is a workspace array. The dimension of *work* must be at least max(1, 3*n*-2).
ldab	INTEGER. The leading dimension of *ab*; must be at least *kd* +1.
ldz	INTEGER. The leading dimension of the output array *z*. Constraints: if *jobz* = 'N', then *ldz* ≥ 1; if *jobz* = 'V', then *ldz* ≥ max(1, *n*) .

Output Parameters

w, *z*	REAL for ssbev DOUBLE PRECISION for dsbev

	Arrays: *w(*)*, DIMENSION at least max(1, *n*). If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order. z(*ldz*,*). The second dimension of *z* must be at least max(1, *n*). If *jobz* = 'V', then if *info* = 0, *z* contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). If *jobz* = 'N', then *z* is not referenced.
ab	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. If *uplo* = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix *T* are returned in rows *kd* and *kd*+1 of *ab*, and if *uplo* = 'L', the diagonal and first subdiagonal of *T* are returned in the first two rows of *ab*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine sbev interface are the following:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*kd+1*,*n*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted.

?hbev

Computes all eigenvalues and, optionally, eigenvectors of a Hermitian band matrix.

Syntax

Fortran 77:

```
call chbev(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, rwork, info)
call zhbev(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, rwork, info)
```

Fortran 95:

```
call hbev(a, w [,uplo] [,z] [,info])
```

Description

This routine computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix *A*.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then only eigenvalues are computed. If *jobz* = 'V', then eigenvalues and eigenvectors are computed.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *ab* stores the upper triangular part of *A*. If *uplo* = 'L', *ab* stores the lower triangular part of *A*.
n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
kd	INTEGER. The number of super- or sub-diagonals in *A* (*kd* ≥ 0).
ab, *work*	COMPLEX for chbev DOUBLE COMPLEX for zhbev. Arrays: *ab* (*ldab*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by *uplo*) in band storage format. The second dimension of *ab* must be at least max(1, *n*).

	work (*) is a workspace array. The dimension of *work* must be at least max(1, *n*).
ldab	INTEGER. The leading dimension of *ab*; must be at least *kd* +1.
ldz	INTEGER. The leading dimension of the output array *z*. Constraints: if *jobz* = 'N', then *ldz* ≥ 1; if *jobz* = 'V', then *ldz* ≥ max(1, *n*) .
rwork	REAL for chbev DOUBLE PRECISION for zhbev Workspace array, DIMENSION at least max(1, 3*n*-2).

Output Parameters

w	REAL for chbev DOUBLE PRECISION for zhbev Array, DIMENSION at least max(1, *n*). If *info* = 0, contains the eigenvalues in ascending order.
z	COMPLEX for chbev DOUBLE COMPLEX for zhbev. Array z(*ldz*,*). The second dimension of *z* must be at least max(1, *n*). If *jobz* = 'V', then if *info* = 0, *z* contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). If *jobz* = 'N', then *z* is not referenced.
ab	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. If *uplo* = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix *T* are returned in rows *kd* and *kd*+1 of *ab*, and if *uplo* = 'L', the diagonal and first subdiagonal of *T* are returned in the first two rows of *ab*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = *i*, then the algorithm failed to converge;

i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `hbev` interface are the following:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*kd+1*,*n*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = `'V'`, if *z* is present, *jobz* = `'N'`, if *z* is omitted.

?sbevd

Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric band matrix using divide and conquer algorithm.

Syntax

Fortran 77:

```
call ssbevd(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork, iwork, liwork, info)
call dsbevd(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call sbevd(a, w [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric band matrix A. In other words, it can compute the spectral factorization of A as:

$A = Z\Lambda Z^T$

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i, and Z is the orthogonal matrix whose columns are the eigenvectors z_i. Thus,

$Az_i = \lambda_i z_i$ for $i = 1, 2, \ldots, n$.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then only eigenvalues are computed. If *jobz* = 'V', then eigenvalues and eigenvectors are computed.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *ab* stores the upper triangular part of A. If *uplo* = 'L', *ab* stores the lower triangular part of A.
n	INTEGER. The order of the matrix A ($n \geq 0$).
kd	INTEGER. The number of super- or sub-diagonals in A ($kd \geq 0$).
ab, *work*	REAL for ssbevd DOUBLE PRECISION for dsbevd. Arrays: *ab* (*ldab*,*) is an array containing either upper or lower triangular part of the symmetric matrix A (as specified by *uplo*) in band storage format. The second dimension of *ab* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
ldab	INTEGER. The leading dimension of *ab*; must be at least *kd*+1.

ldz INTEGER. The leading dimension of the output array *z*.
Constraints:

if *jobz* = 'N', then *ldz* ≥ 1;

if *jobz* = 'V', then *ldz* ≥ max(1, *n*) .

lwork INTEGER.
The dimension of the array *work*.
Constraints:

if *n* ≤ 1, then *lwork* ≥ 1;

if *jobz* = 'N' and *n* > 1, then *lwork* ≥ 2*n*;

if *jobz* = 'V' and *n* > 1, then *lwork* ≥ 2**n*^{2} + 5**n* + 1.
If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See Application Notes for details.

iwork INTEGER. Workspace array, its dimension max(1, *liwork*).

liwork INTEGER.

The dimension of the array *iwork*. Constraints: if *n* ≤ 1, then *liwork* < 1; if *job* = 'N' and *n* > 1, then *liwork* < 1; if *job* = 'V' and *n* > 1, then *liwork* < 5**n*+3.
If *liwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See Application Notes for details.

Output Parameters

w, *z* REAL for ssbevd
DOUBLE PRECISION for dsbevd
Arrays:
w(*), DIMENSION at least max(1, *n*).

	If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order. See also *info*. z(*ldz*,*). The second dimension of *z* must be: at least 1 if *job* = 'N'; at least max(1, *n*) if *job* = 'V'. If *job* = 'V', then this array is overwritten by the orthogonal matrix *Z* which contains the eigenvectors of *A*. The *i*-th column of *Z* contains the eigenvector which corresponds to the eigenvalue *w(i)*. If *job* = 'N', then *z* is not referenced.
ab	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.
work(1)	On exit, if *lwork* > 0, then *work*(1) returns the required minimal size of *lwork*.
iwork(1)	On exit, if *liwork* > 0, then *iwork*(1) returns the required minimal size of *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine sbevd interface are the following:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*kd+1*,*n*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
jobz	Restored based on the presence of the argument *z* as follows:

jobz = 'V', if *z* is present,
jobz = 'N', if *z* is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T + E$ such that $||E||_2 = O(\varepsilon)||T||_2$, where ε is the machine precision.

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = –1 (*liwork* = –1).

If you choose the first option and set any of admissible *lwork* (or *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = –1 (*liwork* = –1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex analogue of this routine is ?hbevd.

See also ?syevd for matrices held in full storage, and ?spevd for matrices held in packed storage.

?hbevd

Computes all eigenvalues and (optionally) all eigenvectors of a complex Hermitian band matrix using divide and conquer algorithm.

Syntax

Fortran 77:

```
call chbevd(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

```
call zhbevd(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

Fortran 95:

```
call hbevd(a, w [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian band matrix A. In other words, it can compute the spectral factorization of A as: $A = Z\Lambda Z^H$.

Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i, and Z is the (complex) unitary matrix whose columns are the eigenvectors z_i. Thus,

$Az_i = \lambda_i z_i$ for $i = 1, 2, \ldots, n$.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then only eigenvalues are computed. If *jobz* = 'V', then eigenvalues and eigenvectors are computed.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *ab* stores the upper triangular part of A.

If *uplo* = 'L', *ab* stores the lower triangular part of *A*.

n — INTEGER. The order of the matrix *A* (*n* ≥ 0).

kd — INTEGER. The number of super- or sub-diagonals in *A* (*kd* ≥ 0).

ab, *work* — COMPLEX for chbevd
DOUBLE COMPLEX for zhbevd.
Arrays:
ab (*ldab*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by *uplo*) in band storage format.
The second dimension of *ab* must be at least max(1, *n*).
work (*) is a workspace array , its dimension max(1, *lwork*).

ldab — INTEGER. The leading dimension of *ab*; must be at least *kd*+1.

ldz — INTEGER. The leading dimension of the output array *z*.
Constraints:
if *jobz* = 'N', then *ldz* ≥ 1;
if *jobz* = 'V', then *ldz* ≥ max(1, *n*) .

lwork — INTEGER.
The dimension of the array *work*.
Constraints:
if *n* ≤ 1, then *lwork* ≥ 1;
if *jobz* = 'N' and *n* > 1, then *lwork* ≥ *n*;
if *jobz* = 'V' and *n* > 1, then *lwork* ≥ 2**n*^{2}.
If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla.
See Application Notes for details.

rwork — REAL for chbevd
DOUBLE PRECISION for zhbevd
Workspace array, DIMENSION at least *lrwork*.

lrwork INTEGER.
The dimension of the array *rwork*.
Constraints:
if $n \leq 1$, then *lrwork* ≥ 1;
if *jobz* = 'N' and $n > 1$, then *lrwork* $\geq n$;
if *jobz* = 'V' and $n > 1$, then *lrwork* $\geq 2*n^2 + 5*n + 1$.
If *lrwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See Application Notes for details.

iwork INTEGER. Workspace array, DIMENSION max(1, *liwork*).

liwork INTEGER.
The dimension of the array *iwork*.
Constraints:
if *jobz* = 'N' or $n \leq 1$, then *liwork* ≥ 1;
if *jobz* = 'V' and $n > 1$, then *liwork* $\geq 5*n+3$.
If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See Application Notes for details.

Output Parameters

w REAL for chbevd
DOUBLE PRECISION for zhbevd
Array, DIMENSION at least max(1, *n*).
If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order. See also *info*.

z COMPLEX for chbevd
DOUBLE COMPLEX for zhbevd
Array, DIMENSION (*ldz*,*).

	The second dimension of *z* must be: at least 1 if *jobz* = 'N'; at least max(1, *n*) if *jobz* = 'V'. If *jobz* = 'V', then this array is overwritten by the unitary matrix *Z* which contains the eigenvectors of *A*. The *i*-th column of *Z* contains the eigenvector which corresponds to the eigenvalue *w(i)*. If *jobz* = 'N', then *z* is not referenced.
ab	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.
work(1)	On exit, if *lwork* > 0, then the real part of *work*(1) returns the required minimal size of *lwork*.
rwork(1)	On exit, if *lrwork* > 0, then *rwork*(1) returns the required minimal size of *lrwork*.
iwork(1)	On exit, if *liwork* > 0, then *iwork*(1) returns the required minimal size of *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero. If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hbevd interface are the following:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*kd+1*,*n*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present,

jobz = 'N', if *z* is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T + E$ such that $||E||_2 = O(\varepsilon)||T||_2$, where ε is the machine precision.

If you are in doubt how much workspace to supply, use a generous value of *lwork* (*liwork* or *lrwork*) for the first run or set *lwork* = -1 (*liwork* = -1, *lrwork* = -1).

If you choose the first option and set any of admissible *lwork* (*liwork* or *lrwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*) on exit. Use this value (*work*(1), *iwork*(1), *rwork*(1)) for subsequent runs.

If you set *lwork* = -1 (*liwork* = -1, *lrwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*, *lrwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real analogue of this routine is ?sbevd.

See also ?heevd for matrices held in full storage, and ?hpevd for matrices held in packed storage.

?sbevx

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix.

Syntax

Fortran 77:

call ssbevx(*jobz*, *range*, *uplo*, *n*, *kd*, *ab*, *ldab*, *q*, *ldq*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *work*, *iwork*, *ifail*, *info*)

call dsbevx(*jobz*, *range*, *uplo*, *n*, *kd*, *ab*, *ldab*, *q*, *ldq*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *w*, *z*, *ldz*, *work*, *iwork*, *ifail*, *info*)

Fortran 95:

```
call sbevx(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,q]
[,abstol] [,info])
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then only eigenvalues are computed. If *jobz* = 'V', then eigenvalues and eigenvectors are computed.
range	CHARACTER*1. Must be 'A' or 'V' or 'I'. If *range* = 'A', the routine computes all eigenvalues. If *range* = 'V', the routine computes eigenvalues λ_i in the half-open interval: $vl < \lambda_i \leq vu$. If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *ab* stores the upper triangular part of *A*. If *uplo* = 'L', *ab* stores the lower triangular part of *A*.
n	INTEGER. The order of the matrix *A* ($n \geq 0$).
kd	INTEGER. The number of super- or sub-diagonals in *A* ($kd \geq 0$).
ab, *work*	REAL for ssbevx DOUBLE PRECISION for dsbevx. Arrays: *ab* (*ldab*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A* (as specified by *uplo*) in band storage format. The second dimension of *ab* must be at least max(1, *n*). *work* (*) is a workspace array. The dimension of *work* must be at least max(1, 7*n*).

ldab	INTEGER. The leading dimension of *ab*; must be at least *kd* +1.
vl, *vu*	REAL for ssbevx DOUBLE PRECISION for dsbevx. If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: *vl*< *vu*. If *range* = 'A' or 'I', *vl* and *vu* are not referenced.
il, *iu*	INTEGER. If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: 1 ≤ *il* ≤ *iu* ≤ *n*, if *n* > 0; *il*=1 and *iu*=0 if *n* = 0. If *range* = 'A' or 'V', *il* and *iu* are not referenced.
abstol	REAL for chpevx DOUBLE PRECISION for zhpevx The absolute error tolerance to which each eigenvalue is required. See Application notes for details on error tolerance.
ldq, *ldz*	INTEGER. The leading dimensions of the output arrays *q* and *z*, respectively. Constraints: *ldq* ≥ 1, *ldz* ≥ 1; If *jobz* = 'V', then *ldq* ≥ max(1, *n*) and *ldz* ≥ max(1, *n*).
iwork	INTEGER. Workspace array, DIMENSION at least max(1, 5*n*).

Output Parameters

q	REAL for ssbevx DOUBLE PRECISION for dsbevx. Array, DIMENSION (*ldz*,*n*). If *jobz* = 'V', the *n*-by-*n* orthogonal matrix is used in the reduction to tridiagonal form. If *jobz* = 'N', the array *q* is not referenced.
m	INTEGER. The total number of eigenvalues found, 0 ≤ *m* ≤ *n*. If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu*-*il*+1.

w, *z*
REAL for ssbevx
DOUBLE PRECISION for dsbevx
Arrays:
w(*), DIMENSION at least max(1, *n*). The first *m* elements of *w* contain the selected eigenvalues of the matrix *A* in ascending order.
z(*ldz*,*).
The second dimension of *z* must be at least max(1, *m*).
If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).
If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
If *jobz* = 'N', then *z* is not referenced.
Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

ab
On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.
If *uplo* = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix *T* are returned in rows *kd* and *kd*+1 of *ab*, and if *uplo* = 'L', the diagonal and first subdiagonal of *T* are returned in the first two rows of *ab*.

ifail
INTEGER.
Array, DIMENSION at least max(1, *n*).
If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices the eigenvectors that failed to converge.
If *jobz* = 'N', then *ifail* is not referenced.

info
INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `sbevx` interface are the following:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*kd+1*,*n*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.
ifail	Holds the vector of length (*n*).
q	Holds the matrix *Q* of size (*n*, *n*).
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
vl	Default value for this element is *vl* = -HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = `'V'`, if *z* is present, *jobz* = `'N'`, if *z* is omitted Note that there will be an error condition if either *ifail* or *q* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = `'V'`, if one of or both *vl* and *vu* are present, *range* = `'I'`, if one of or both *il* and *iu* are present, *range* = `'A'`, if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol* + ε * max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*||*T*||$_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?hbevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian band matrix.

Syntax

Fortran 77:

```
call chbevx(jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il, iu, abstol,
m, w, z, ldz, work, rwork, iwork, ifail, info)
```

```
call zhbevx(jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il, iu, abstol,
m, w, z, ldz, work, rwork, iwork, ifail, info)
```

Fortran 95:

```
call hbevx(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,q]
[,abstol] [,info])
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz CHARACTER*1. Must be 'N' or 'V'.
If *job* = 'N', then only eigenvalues are computed.

	If *job* = 'V', then eigenvalues and eigenvectors are computed.
range	CHARACTER*1. Must be 'A' or 'V' or 'I'. If *range* = 'A', the routine computes all eigenvalues. If *range* = 'V', the routine computes eigenvalues *lambda*(*i*) in the half-open interval: *vl* < *lambda*(*i*) ≤ *vu*. If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', *ab* stores the upper triangular part of *A*. If *uplo* = 'L', *ab* stores the lower triangular part of *A*.
n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
kd	INTEGER. The number of super- or sub-diagonals in *A* (*kd* ≥ 0).
ab, *work*	COMPLEX for chbevx DOUBLE COMPLEX for zhbevx. Arrays: *ab* (*ldab*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by *uplo*) in band storage format. The second dimension of *ab* must be at least max(1, *n*). *work* (*) is a workspace array. The dimension of *work* must be at least max(1, *n*).
ldab	INTEGER. The leading dimension of *ab*; must be at least *kd* +1.
vl, *vu*	REAL for chbevx DOUBLE PRECISION for zhbevx. If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: *vl* < *vu*. If *range* = 'A' or 'I', *vl* and *vu* are not referenced.
il, *iu*	INTEGER. If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: $1 \le il \le iu \le n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.
If $range$ = 'A' or 'V', il and iu are not referenced.

abstol — REAL for chbevx
DOUBLE PRECISION for zhbevx.
The absolute error tolerance to which each eigenvalue is required. See Application notes for details on error tolerance.

ldq, *ldz* — INTEGER. The leading dimensions of the output arrays q and z, respectively.
Constraints:

$ldq \ge 1$, $ldz \ge 1$;

If $jobz$ = 'V', then $ldq \ge \max(1, n)$ and $ldz \ge \max(1, n)$.

rwork — REAL for chbevx
DOUBLE PRECISION for zhbevx
Workspace array, DIMENSION at least max(1, $7n$).

iwork — INTEGER. Workspace array, DIMENSION at least max(1, $5n$).

Output Parameters

q — COMPLEX for chbevx DOUBLE COMPLEX for zhbevx.
Array, DIMENSION (ldz,n).
If $jobz$ = 'V', the n-by-n unitary matrix is used in the reduction to tridiagonal form.
If $jobz$ = 'N', the array q is not referenced.

m — INTEGER. The total number of eigenvalues found,

$0 \le m \le n$.
If $range$ = 'A', $m = n$, and if $range$ = 'I', $m = iu-il+1$.

w — REAL for chbevx
DOUBLE PRECISION for zhbevx
Array, DIMENSION at least max(1, n). The first m elements contain the selected eigenvalues of the matrix A in ascending order.

z — COMPLEX for chbevx
DOUBLE COMPLEX for zhbevx.
Array z(ldz,*).

	The second dimension of *z* must be at least max(1, *m*). If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*. If *jobz* = 'N', then *z* is not referenced. Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.
ab	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. If *uplo* = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix *T* are returned in rows *kd* and *kd*+1 of *ab*, and if *uplo* = 'L', the diagonal and first subdiagonal of *T* are returned in the first two rows of *ab*.
ifail	INTEGER. Array, DIMENSION at least max(1, *n*). If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge. If *jobz* = 'N', then *ifail* is not referenced.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = –*i*, the *i*-th parameter had an illegal value. If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hbevx interface are the following:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*kd+1*,*n*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.
ifail	Holds the vector of length (*n*).
q	Holds the matrix *Q* of size (*n*, *n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
vl	Default value for this element is *vl* = -HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted Note that there will be an error condition if either *ifail* or *q* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol* + ε * max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*||*T*||$_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?stev

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix.

Syntax

Fortran 77:

```
call sstev(jobz, n, d, e, z, ldz, work, info)
call dstev(jobz, n, d, e, z, ldz, work, info)
```

Fortran 95:

```
call stev(d, e [,z] [,info])
```

Description

This routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix *A*.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then only eigenvalues are computed. If *jobz* = 'V', then eigenvalues and eigenvectors are computed.
n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
d, *e*, *work*	REAL for sstev DOUBLE PRECISION for dstev. Arrays: *d*(*) contains the *n* diagonal elements of the tridiagonal matrix *A*. The dimension of *d* must be at least max(1, *n*). *e*(*) contains the *n*-1 subdiagonal elements of the tridiagonal matrix *A*. The dimension of *e* must be at least max(1, *n*-1). The *n*-th element of this array is used as workspace.

	work(*) is a workspace array. The dimension of *work* must be at least max(1, 2*n*-2). If *jobz* = 'N', *work* is not referenced.
ldz	INTEGER. The leading dimension of the output array *z*; *ldz* ≥ 1. If *jobz* = 'V' then *ldz* ≥ max(1, *n*).

Output Parameters

d	On exit, if *info* = 0, contains the eigenvalues of the matrix *A* in ascending order.
z	REAL for sstev DOUBLE PRECISION for dstev Array, DIMENSION (*ldz*, *). The second dimension of *z* must be at least max(1, *n*). If *jobz* = 'V', then if *info* = 0, *z* contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of *z* holding the eigenvector associated with the eigenvalue returned in *d*(*i*). If *job* = 'N', then *z* is not referenced.
e	On exit, this array is overwritten with intermediate results.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, then the algorithm failed to converge; *i* elements of *e* did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine stev interface are the following:

d	Holds the vector of length (*n*).
e	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*).
jobz	Restored based on the presence of the argument *z* as follows:

jobz = 'V', if *z* is present,
jobz = 'N', if *z* is omitted.

?stevd

Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric tridiagonal matrix using divide and conquer algorithm.

Syntax

Fortran 77:

```
call sstevd(jobz, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
call dstevd(jobz, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call stevd(d, e [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric tridiagonal matrix T. In other words, the routine can compute the spectral factorization of T as: $T = Z\Lambda Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i, and Z is the orthogonal matrix whose columns are the eigenvectors z_i. Thus,

$Tz_i = \lambda_i z_i$ for $i = 1, 2, \ldots, n$.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

There is no complex analogue of this routine.

Input Parameters

jobz — CHARACTER*1. Must be 'N' or 'V'.
If *jobz* = 'N', then only eigenvalues are computed.
If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

n — INTEGER. The order of the matrix T ($n \geq 0$).

d, *e*, *work* — REAL for sstevd
DOUBLE PRECISION for dstevd.
Arrays:
d(*) contains the n diagonal elements of the tridiagonal matrix T.
The dimension of *d* must be at least max(1, n).
e(*) contains the n-1 off-diagonal elements of T.
The dimension of *e* must be at least max(1, n-1). The n–th element of this array is used as workspace.
work(*) is a workspace array.
The dimension of *work* must be at least *lwork*.

ldz — INTEGER. The leading dimension of the output array *z*.
Constraints:
ldz ≥ 1 if *job* = 'N';
ldz < max(1, *n*) if *job* = 'V'.

lwork — INTEGER.
The dimension of the array *work*.
Constraints:
if *jobz* = 'N' or $n \leq 1$, then *lwork* ≥ 1;
if *jobz* = 'V' and $n > 1$, then *lwork* ≥ $n^2 + 4*n + 1$.
If *lwork* = −1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See Application Notes for details.

iwork — INTEGER. Workspace array, its dimension max(1, *liwork*).

liwork — INTEGER.
The dimension of the array *iwork*.
Constraints:
if *jobz* = 'N' or $n \leq 1$, then *liwork* ≥ 1;
if *jobz* = 'V' and $n > 1$, then *liwork* ≥ 5*n+3.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See Application Notes for details.

Output Parameters

d
On exit, if *info* = 0, contains the eigenvalues of the matrix *T* in ascending order.
See also *info*.

z
REAL for sstevd
DOUBLE PRECISION for dstevd
Array, DIMENSION (*ldz*, *).
The second dimension of *z* must be:
at least 1 if *jobz* = 'N';
at least max(1, *n*) if *jobz* = 'V'.
If *jobz* = 'V', then this array is overwritten by the orthogonal matrix *Z* which contains the eigenvectors of *T*.
If *jobz* = 'N', then *z* is not referenced.

e
On exit, this array is overwritten with intermediate results.

work(1)
On exit, if *lwork* > 0, then *work*(1) returns the required minimal size of *lwork*.

iwork(1)
On exit, if *liwork* > 0, then *iwork*(1) returns the required minimal size of *liwork*.

info
INTEGER.
If *info* = 0, the execution is successful.
If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `stevd` interface are the following:

d	Holds the vector of length (*n*).
e	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*).
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T + E$ such that $||E||_2 = O(\varepsilon)||T||_2$, where ε is the machine precision.

If λ_i is an exact eigenvalue, and m_i is the corresponding computed value, then

$$|\mu_i - \lambda_i| \le c(n)\varepsilon ||T||_2$$

where $c(n)$ is a modestly increasing function of n.

If z_i is the corresponding exact eigenvector, and w_i is the corresponding computed vector, then the angle $\theta(z_i, w_i)$ between them is bounded as follows:

$$\theta(z_i, w_i) \le c(n)\varepsilon ||T||_2 / \min_{i \neq j}|\lambda_i - \lambda_j|.$$

Thus the accuracy of a computed eigenvector depends on the gap between its eigenvalue and all the other eigenvalues.

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If you choose the first option and set any of admissible *lwork* (or *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = -1 (*liwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?stevx

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

Fortran 77:

```
call sstevx(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
call dstevx(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
```

Fortran 95:

```
call stevx(d, e, w [, z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol] [,info])
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz — CHARACTER*1. Must be 'N' or 'V'.
If *job* = 'N', then only eigenvalues are computed.
If *job* = 'V', then eigenvalues and eigenvectors are computed.

range — CHARACTER*1. Must be 'A' or 'V' or 'I'.
If *range* = 'A', the routine computes all eigenvalues.
If *range* = 'V', the routine computes eigenvalues *lambda*(*i*) in the half-open interval: $vl < lambda(i) \le vu$.
If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.

n — INTEGER. The order of the matrix *A* (*n* ≥ 0).

d, *e*, *work* — REAL for sstevx
DOUBLE PRECISION for dstevx.
Arrays:
d(*) contains the *n* diagonal elements of the tridiagonal matrix *A*.
The dimension of *d* must be at least max(1, *n*).
e(*) contains the *n*-1 subdiagonal elements of *A*.
The dimension of *e* must be at least max(1, *n*-1). The *n*-th element of this array is used as workspace.
work(*) is a workspace array.
The dimension of *work* must be at least max(1, 5*n*).

vl, *vu* — REAL for sstevx
DOUBLE PRECISION for dstevx.
If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
Constraint: *vl*< *vu*.
If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu* — INTEGER.
If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: 1 ≤ *il* ≤ *iu* ≤ *n*, if *n* > 0; *il*=1 and *iu*=0 if *n* = 0.
If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol — REAL for sstevx
DOUBLE PRECISION for dstevx. The absolute error tolerance to which each eigenvalue is required. See Application notes for details on error tolerance.

ldz — INTEGER. The leading dimensions of the output array *z*; *ldz* ≥ 1. If *jobz* = 'V', then *ldz* ≥ max(1, *n*).

iwork — INTEGER. Workspace array, DIMENSION at least max(1, 5*n*).

Output Parameters

m — INTEGER. The total number of eigenvalues found, 0 ≤ *m* ≤ *n*.

	If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu*-*il*+1.
w, *z*	REAL for sstevx DOUBLE PRECISION for dstevx. Arrays: *w(*)*, DIMENSION at least max(1, *n*). The first *m* elements of *w* contain the selected eigenvalues of the matrix *A* in ascending order. z(*ldz*,*). The second dimension of *z* must be at least max(1, *m*). If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*. If *jobz* = 'N', then *z* is not referenced. Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.
d, *e*	On exit, these arrays may be multiplied by a constant factor chosen to avoid overflow or underflow in computing the eigenvalues.
ifail	INTEGER. Array, DIMENSION at least max(1, *n*). If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge. If *jobz* = 'N', then *ifail* is not referenced.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `stevx` interface are the following:

d	Holds the vector of length (*n*).
e	Holds the vector of length (*n*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.
ifail	Holds the vector of length (*n*).
vl	Default value for this element is *vl* = -HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted Note that there will be an error condition if *ifail* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol* + ε * max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then $\varepsilon * ||A||_1$ will be used in its place. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?stevr

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix using the Relatively Robust Representations.

Syntax

Fortran 77:

```
call sstevr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work, lwork, iwork, liwork, info)
```

```
call dstevr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call stevr(d, e, w [, z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol] [,info])
```

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix *T*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Whenever possible, the routine calls ?stemr to compute the eigenspectrum using Relatively Robust Representations. ?stegr computes eigenvalues by the *dqds* algorithm, while orthogonal eigenvectors are computed from various "good" $L*D*L^T$ representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the i-th unreduced block of *T*,

a. Compute $T - \sigma_i = L_i*D_i*L_i^T$, such that $L_i*D_i*L_i^T$ is a relatively robust representation;

b. Compute the eigenvalues, λ_j, of $L_i*D_i*L_i^T$ to high relative accuracy by the *dqds* algorithm;

c. If there is a cluster of close eigenvalues, "choose" σ_i close to the cluster, and go to step (a);

d. Given the approximate eigenvalue λ_j of $L_i D_i L_i^T$, compute the corresponding eigenvector by forming a rank-revealing twisted factorization.

The desired accuracy of the output can be specified by the input parameter *abstol*.

The routine ?stevr calls ?stemr when the full spectrum is requested on machines which conform to the IEEE-754 floating point standard. ?stevr calls ?stebz and ?stein on non-IEEE machines and when partial spectrum requests are made.

Input Parameters

jobz — CHARACTER*1. Must be 'N' or 'V'.
If *jobz* = 'N', then only eigenvalues are computed.
If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

range — CHARACTER*1. Must be 'A' or 'V' or 'I'.
If *range* = 'A', the routine computes all eigenvalues.
If *range* = 'V', the routine computes eigenvalues *lambda*(*i*) in the half-open interval:
vl<*lambda*(*i*) ≤ *vu*.
If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.
For *range* = 'V'or 'I' and *iu*-*il* < *n*–1, sstebz/dstebz and sstein/dstein are called.

n — INTEGER. The order of the matrix *T* (*n* ≥ 0).

d, *e*, *work* — REAL for sstevr
DOUBLE PRECISION for dstevr.
Arrays:
d(*) contains the *n* diagonal elements of the tridiagonal matrix *T*.
The dimension of *d* must be at least max(1, *n*).
e(*) contains the *n*-1 subdiagonal elements of *A*.
The dimension of *e* must be at least max(1, *n*-1). The *n*-th element of this array is used as workspace.
work is a workspace array, its dimension max(1, *lwork*).

vl, *vu*	REAL for sstevr DOUBLE PRECISION for dstevr. If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: *vl*< *vu*. If *range* = 'A' or 'I', *vl* and *vu* are not referenced.
il, *iu*	INTEGER. If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: 1 ≤ *il* ≤ *iu*≤ *n*, if *n* > 0; *il*=1 and *iu*=0 if *n* = 0. If *range* = 'A' or 'V', *il* and *iu* are not referenced.
abstol	REAL for ssyevr DOUBLE PRECISION for dsyevr. The absolute error tolerance to which each eigenvalue/eigenvector is required. If *jobz* = 'V', the eigenvalues and eigenvectors output have residual norms bounded by *abstol*, and the dot products between different eigenvectors are bounded by *abstol*. If *abstol* < *n* *eps*|*T*|, then *n* *eps*|*T*| will be used in its place, where eps is the machine precision, and |*T*| is the 1-norm of the matrix *T*. The eigenvalues are computed to an accuracy of eps*|*T*| irrespective of *abstol*. If high relative accuracy is important, set *abstol* to ?lamch('S').
ldz	INTEGER. The leading dimension of the output array *z*. Constraints: *ldz* ≥ 1 if *jobz* = 'N'; *ldz* ≥ max(1, *n*) if *jobz* = 'V'.
lwork	INTEGER. The dimension of the array *work*. Constraint: *lwork* ≥ max(1, 20**n*). If *lwork* = −1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the

work and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See Application Notes for details.

iwork INTEGER.
Workspace array, its dimension max(1, *liwork*).

liwork INTEGER.
The dimension of the array *iwork*,
lwork ≥ max(1, 10*n).
If *liwork* = -1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See Application Notes for details.

Output Parameters

m INTEGER. The total number of eigenvalues found,
0 ≤ *m* ≤ *n*. If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu*-*il*+1.

w, *z* REAL for sstevr
DOUBLE PRECISION for dstevr.
Arrays:
w(*), DIMENSION at least max(1, *n*).
The first *m* elements of *w* contain the selected eigenvalues of the matrix *T* in ascending order.
z(*ldz*,*).
The second dimension of *z* must be at least max(1, *m*).
If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).
If *jobz* = 'N', then *z* is not referenced.
Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

d, *e*	On exit, these arrays may be multiplied by a constant factor chosen to avoid overflow or underflow in computing the eigenvalues.
isuppz	INTEGER. Array, DIMENSION at least 2 *max(1, *m*). The support of the eigenvectors in *z*, i.e., the indices indicating the nonzero elements in *z*. The *i*-th eigenvector is nonzero only in elements *isuppz*(2*i*-1) through *isuppz*(2*i*). Implemented only for *range* = 'A' or 'I' and *iu*-*il* = *n*-1.
work(1)	On exit, if *info* = 0, then *work*(1) returns the required minimal size of *lwork*.
iwork(1)	On exit, if *info* = 0, then *iwork*(1) returns the required minimal size of *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, an internal error has occurred.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine stevr interface are the following:

d	Holds the vector of length (*n*).
e	Holds the vector of length (*n*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.
isuppz	Holds the vector of length (2**n*), where the values (2**m*) are significant.
vl	Default value for this element is *vl* = -HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.

iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted Note that there will be an error condition if *ifail* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

Normal execution of the routine ?stegr may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = –1 (*liwork* = –1).

If you choose the first option and set any of admissible *lwork* (or *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = –1 (*liwork* = –1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Nonsymmetric Eigenproblems

This section describes LAPACK driver routines used for solving nonsymmetric eigenproblems. See also computational routines that can be called to solve these problems.

Table 4-11 lists all such driver routines for Fortran-77 interface. Respective routine names in Fortran-95 interface are without the first symbol (see Routine Naming Conventions).

Table 4-11 Driver Routines for Solving Nonsymmetric Eigenproblems

Routine Name	Operation performed
?gees	Computes the eigenvalues and Schur factorization of a general matrix, and orders the factorization so that selected eigenvalues are at the top left of the Schur form.
?geesx	Computes the eigenvalues and Schur factorization of a general matrix, orders the factorization and computes reciprocal condition numbers.
?geev	Computes the eigenvalues and left and right eigenvectors of a general matrix.
?geevx	Computes the eigenvalues and left and right eigenvectors of a general matrix, with preliminary matrix balancing, and computes reciprocal condition numbers for the eigenvalues and right eigenvectors.

?gees

Computes the eigenvalues and Schur factorization of a general matrix, and orders the factorization so that selected eigenvalues are at the top left of the Schur form.

Syntax

Fortran 77:

```
call sgees(jobvs, sort, select, n, a, lda, sdim, wr, wi, vs, ldvs, work, lwork, bwork, info)
call dgees(jobvs, sort, select, n, a, lda, sdim, wr, wi, vs, ldvs, work, lwork, bwork, info)
call cgees(jobvs, sort, select, n, a, lda, sdim, w, vs, ldvs, work, lwork, rwork, bwork, info)
call zgees(jobvs, sort, select, n, a, lda, sdim, w, vs, ldvs, work, lwork, rwork, bwork, info)
```

Fortran 95:

```
call gees(a, wr, wi [,vs] [,select] [,sdim] [,info])
call gees(a, w [,vs] [,select] [,sdim] [,info])
```

Description

This routine computes for an *n*-by-*n* real/complex nonsymmetric matrix *A*, the eigenvalues, the real Schur form *T*, and, optionally, the matrix of Schur vectors *Z*. This gives the Schur factorization $A = Z*T*Z^H$.

Optionally, it also orders the eigenvalues on the diagonal of the real-Schur/Schur form so that selected eigenvalues are at the top left. The leading columns of *Z* then form an orthonormal basis for the invariant subspace corresponding to the selected eigenvalues.

A real matrix is in real-Schur form if it is upper quasi-triangular with 1-by-1 and 2-by-2 blocks. 2-by-2 blocks will be standardized in the form

$$\begin{pmatrix} a & b \\ c & a \end{pmatrix}$$

where $b*c < 0$. The eigenvalues of such a block are $a \pm \sqrt{bc}$

A complex matrix is in Schur form if it is upper triangular.

Input Parameters

jobvs	CHARACTER*1. Must be 'N' or 'V'. If *jobvs* = 'N', then Schur vectors are not computed. If *jobvs* = 'V', then Schur vectors are computed.
sort	CHARACTER*1. Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the Schur form. If *sort* = 'N', then eigenvalues are not ordered. If *sort* = 'S', eigenvalues are ordered (see *select*).
select	LOGICAL FUNCTION of two REAL arguments for real flavors.

	LOGICAL FUNCTION of one COMPLEX argument for complex flavors. *select* must be declared EXTERNAL in the calling subroutine. If *sort* = 'S', *select* is used to select eigenvalues to sort to the top left of the Schur form. If *sort* = 'N', *select* is not referenced. For real flavors: An eigenvalue *wr*(j)+sqrt(-1)**wi*(j) is selected if *select*(*wr*(j), *wi*(j)) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected. Note that a selected complex eigenvalue may no longer satisfy *select*(*wr*(j), *wi*(j))= .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case *info* may be set to *n*+2 (see *info* below). For complex flavors: An eigenvalue *w*(j) is selected if *select*(*w*(j)) is true.
n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
a, *work*	REAL for sgees DOUBLE PRECISION for dgees COMPLEX for cgees DOUBLE COMPLEX for zgees. Arrays: *a*(*lda*,*) is an array containing the *n*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of the array *a*. Must be at least max(1, *n*).
ldvs	INTEGER. The leading dimension of the output array *vs*. Constraints: *ldvs* ≥ 1; *ldvs* ≥ max(1, *n*) if *jobvs* = 'V'.
lwork	INTEGER. The dimension of the array *work*.

Constraint:

lwork ≥ max(1, 3*n*) for real flavors;

lwork ≥ max(1, 2*n*) for complex flavors.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

rwork — REAL for cgees
DOUBLE PRECISION for zgees
Workspace array, DIMENSION at least max(1, *n*). Used in complex flavors only.

bwork — LOGICAL. Workspace array, DIMENSION at least max(1, *n*). Not referenced if *sort* = 'N'.

Output Parameters

a — On exit, this array is overwritten by the real-Schur/Schur form *T*.

sdim — INTEGER.
If *sort* = 'N', *sdim*= 0.
If *sort* = 'S', *sdim* is equal to the number of eigenvalues (after sorting) for which *select* is true.
Note that for real flavors complex conjugate pairs for which *select* is true for either eigenvalue count as 2.

wr, *wi* — REAL for sgees
DOUBLE PRECISION for dgees
Arrays, DIMENSION at least max (1, *n*) each. Contain the real and imaginary parts, respectively, of the computed eigenvalues, in the same order that they appear on the diagonal of the output real-Schur form *T*. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w — COMPLEX for cgees
DOUBLE COMPLEX for zgees.
Array, DIMENSION at least max(1, *n*). Contains the computed eigenvalues. The eigenvalues are stored in the same order as they appear on the diagonal of the output Schur form *T*.

vs	REAL for sgees DOUBLE PRECISION for dgees COMPLEX for cgees DOUBLE COMPLEX for zgees. Array *vs*(*ldvs*,*);the second dimension of *vs* must be at least max(1, *n*). If *jobvs* = 'V', *vs* contains the orthogonal/unitary matrix *Z* of Schur vectors. If *jobvs* = 'N', *vs* is not referenced.
work(1)	On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = *i*, and *i* ≤ *n*: the *QR* algorithm failed to compute all the eigenvalues; elements 1:*ilo*-1 and *i*+1:*n* of *wr* and *wi* (for real flavors) or *w* (for complex flavors) contain those eigenvalues which have converged; if *jobvs* = 'V', *vs* contains the matrix which reduces *A* to its partially converged Schur form; *i* = *n*+1: the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned); *i* = *n*+2: after reordering, round-off changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy *select* = .TRUE.. This could also be caused by underflow due to scaling.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gees interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
wr	Holds the vector of length (*n*). Used in real flavors only.
wi	Holds the vector of length (*n*). Used in real flavors only.
w	Holds the vector of length (*n*). Used in complex flavors only.
vs	Holds the matrix *VS* of size (*n*, *n*).
jobvs	Restored based on the presence of the argument *vs* as follows: *jobvs* = 'V', if *vs* is present, *jobvs* = 'N', if *vs* is omitted.
sort	Restored based on the presence of the argument *select* as follows: *sort* = 'S', if *select* is present, *sort* = 'N', if *select* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?geesx

Computes the eigenvalues and Schur factorization of a general matrix, orders the factorization and computes reciprocal condition numbers.

Syntax

Fortran 77:

```
call sgeesx(jobvs, sort, select, sense, n, a, lda, sdim, wr, wi, vs, ldvs,
rconde, rcondv, work, lwork, iwork, liwork, bwork, info)

call dgeesx(jobvs, sort, select, sense, n, a, lda, sdim, wr, wi, vs, ldvs,
rconde, rcondv, work, lwork, iwork, liwork, bwork, info)

call cgeesx(jobvs, sort, select, sense, n, a, lda, sdim, w, vs, ldvs, rconde,
rcondv, work, lwork, rwork, bwork, info)

call zgeesx(jobvs, sort, select, sense, n, a, lda, sdim, w, vs, ldvs, rconde,
rcondv, work, lwork, rwork, bwork, info)
```

Fortran 95:

```
call geesx(a, wr, wi [,vs] [,select] [,sdim] [,rconde] [,rcondev] [,info])

call geesx(a, w [,vs] [,select] [,sdim] [,rconde] [,rcondev] [,info])
```

Description

This routine computes for an n-by-n real/complex nonsymmetric matrix A, the eigenvalues, the real-Schur/Schur form T, and, optionally, the matrix of Schur vectors Z. This gives the Schur factorization $A = Z*T*Z^H$.

Optionally, it also orders the eigenvalues on the diagonal of the real-Schur/Schur form so that selected eigenvalues are at the top left; computes a reciprocal condition number for the average of the selected eigenvalues (*rconde*); and computes a reciprocal condition number for the right invariant subspace corresponding to the selected eigenvalues (*rcondv*). The leading columns of Z form an orthonormal basis for this invariant subspace.

For further explanation of the reciprocal condition numbers *rconde* and *rcondv*, see [LUG], Section 4.10 (where these quantities are called s and sep respectively).

A real matrix is in real-Schur form if it is upper quasi-triangular with 1-by-1 and 2-by-2 blocks. 2-by-2 blocks will be standardized in the form

$$\begin{pmatrix} a & b \\ c & a \end{pmatrix}$$

where *b*c* < 0. The eigenvalues of such a block are $a \pm \sqrt{bc}$

A complex matrix is in Schur form if it is upper triangular.

Input Parameters

jobvs — CHARACTER*1. Must be 'N' or 'V'.
If *jobvs* = 'N', then Schur vectors are not computed.
If *jobvs* = 'V', then Schur vectors are computed.

sort — CHARACTER*1. Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the Schur form.
If *sort* = 'N', then eigenvalues are not ordered.
If *sort* = 'S', eigenvalues are ordered (see *select*).

select — LOGICAL FUNCTION of two REAL arguments for real flavors.
LOGICAL FUNCTION of one COMPLEX argument for complex flavors.
select must be declared EXTERNAL in the calling subroutine.
If *sort* = 'S', *select* is used to select eigenvalues to sort to the top left of the Schur form.
If *sort* = 'N', *select* is not referenced.
For real flavors:
An eigenvalue *wr*(j)+sqrt(-1)**wi*(j) is selected if *select*(*wr*(j), *wi*(j)) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.
Note that a selected complex eigenvalue may no longer satisfy *select*(*wr*(j), *wi*(j)) = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case *info* may be set to *n*+2 (see *info* below).
For complex flavors:

	An eigenvalue *w*(j) is selected if *select*(*w*(j)) is true.
sense	CHARACTER*1. Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed. If *sense* = 'N', none are computed; If *sense* = 'E', computed for average of selected eigenvalues only; If *sense* = 'V', computed for selected right invariant subspace only; If *sense* = 'B', computed for both. If *sense* is 'E', 'V', or 'B', then *sort* must equal 'S'.
n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
a, *work*	REAL for sgeesx DOUBLE PRECISION for dgeesx COMPLEX for cgeesx DOUBLE COMPLEX for zgeesx. Arrays: *a*(*lda*,*) is an array containing the *n*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of the array *a*. Must be at least max(1, *n*).
ldvs	INTEGER. The leading dimension of the output array *vs*. Constraints: *ldvs* ≥ 1; *ldvs* ≥ max(1, *n*) if *jobvs* = 'V'.
lwork	INTEGER. The dimension of the array *work*. Constraint: *lwork* ≥ max(1, 3*n*) for real flavors; *lwork* ≥ max(1, 2*n*) for complex flavors. Also, if *sense* = 'E', 'V', or 'B', then *lwork* ≥ *n*+2**sdim**(*n*−*sdim*) for real flavors; *lwork* ≥ 2**sdim**(*n*−*sdim*) for complex flavors; where *sdim* is the number of selected eigenvalues computed by this routine.

Note that 2*sdim*(n-sdim) ≤ n*n/2. Note also that an error is only returned if lwork<max(1, 2*n), but if sense = 'E', or 'V', or 'B' this may not be large enough.
For good performance, lwork must generally be larger.
If lwork = -1, then a workspace query is assumed; the routine only calculates upper bound on the optimal size of the array work, returns this value as the first entry of the work array, and no error message related to lwork is issued by xerbla.

iwork INTEGER.
Workspace array, DIMENSION (*liwork*). Used in real flavors only. Not referenced if *sense* = 'N' or 'E'.

liwork INTEGER.
The dimension of the array *iwork*. Used in real flavors only.
Constraint:

liwork ≥ 1;

if *sense* = 'V' or 'B', *liwork* ≥ *sdim**(*n*-*sdim*).

rwork REAL for cgeesx
DOUBLE PRECISION for zgeesx
Workspace array, DIMENSION at least max(1, *n*). Used in complex flavors only.

bwork LOGICAL. Workspace array, DIMENSION at least max(1, *n*). Not referenced if *sort* = 'N'.

Output Parameters

a On exit, this array is overwritten by the real-Schur/Schur form *T*.

sdim INTEGER.
If *sort* = 'N', *sdim*= 0.
If *sort* = 'S', *sdim* is equal to the number of eigenvalues (after sorting) for which *select* is true.
Note that for real flavors complex conjugate pairs for which *select* is true for either eigenvalue count as 2.

wr, *wi* REAL for sgeesx
DOUBLE PRECISION for dgeesx

	Arrays, DIMENSION at least max (1, *n*) each. Contain the real and imaginary parts, respectively, of the computed eigenvalues, in the same order that they appear on the diagonal of the output real-Schur form *T*. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.
w	COMPLEX for cgeesx DOUBLE COMPLEX for zgeesx. Array, DIMENSION at least max(1, *n*). Contains the computed eigenvalues. The eigenvalues are stored in the same order as they appear on the diagonal of the output Schur form *T*.
vs	REAL for sgeesx DOUBLE PRECISION for dgeesx COMPLEX for cgeesx DOUBLE COMPLEX for zgeesx. Array *vs*(*ldvs*,*);the second dimension of *vs* must be at least max(1, *n*). If *jobvs* = 'V', *vs* contains the orthogonal/unitary matrix *Z* of Schur vectors. If *jobvs* = 'N', *vs* is not referenced.
rconde, *rcondv*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If *sense* = 'E' or 'B', *rconde* contains the reciprocal condition number for the average of the selected eigenvalues. If *sense* = 'N' or 'V', *rconde* is not referenced. If *sense* = 'V' or 'B', *rcondv* contains the reciprocal condition number for the selected right invariant subspace. If *sense* = 'N' or 'E', *rcondv* is not referenced.
work(1)	On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = –*i*, the *i*th parameter had an illegal value. If *info* = *i*, and *i* ≤ *n*:

the *QR* algorithm failed to compute all the eigenvalues; elements 1:*ilo*-1 and *i*+1:*n* of *wr* and *wi* (for real flavors) or *w* (for complex flavors) contain those eigenvalues which have converged; if *jobvs* = 'V', *vs* contains the transformation which reduces *A* to its partially converged Schur form;
i = *n*+1:
the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned);
i = *n*+2:
after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy *select* = .TRUE.. This could also be caused by underflow due to scaling.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine geesx interface are the following:

a Holds the matrix *A* of size (*n*, *n*).

wr Holds the vector of length (*n*). Used in real flavors only.

wi Holds the vector of length (*n*). Used in real flavors only.

w Holds the vector of length (*n*). Used in complex flavors only.

vs Holds the matrix *VS* of size (*n*, *n*).

jobvs Restored based on the presence of the argument *vs* as follows:
jobvs = 'V', if *vs* is present,
jobvs = 'N', if *vs* is omitted.

sort Restored based on the presence of the argument *select* as follows:
sort = 'S', if *select* is present,
sort = 'N', if *select* is omitted.

sense Restored based on the presence of arguments *rconde* and rcondv as follows:
sense = 'B', if both *rconde* and *rcondv* are present,
sense = 'E', if *rconde* is present and *rcondv* omitted,

sense = 'V', if *rconde* is omitted and *rcondv* present,
sense = 'N', if both *rconde* and *rcondv* are omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = –1 (*liwork* = –1).

If you choose the first option and set any of admissible *lwork* (or *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?geev

Computes the eigenvalues and left and right eigenvectors of a general matrix.

Syntax

Fortran 77:

```
call sgeev(jobvl, jobvr, n, a, lda, wr, wi, vl, ldvl, vr, ldvr, work, lwork, info)
call dgeev(jobvl, jobvr, n, a, lda, wr, wi, vl, ldvl, vr, ldvr, work, lwork, info)
call cgeev(jobvl, jobvr, n, a, lda, w, vl, ldvl, vr, ldvr, work, lwork, rwork, info)
call zgeev(jobvl, jobvr, n, a, lda, w, vl, ldvl, vr, ldvr, work, lwork, rwork, info)
```

Fortran 95:

```
call geev(a, wr, wi [,vl] [,vr] [,info])
call geev(a, w [,vl] [,vr] [,info])
```

Description

This routine computes for an n-by-n real/complex nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors. The right eigenvector $v(j)$ of A satisfies

$$A*v(j) = \lambda(j)*v(j)$$

where $\lambda(j)$ is its eigenvalue.

The left eigenvector $u(j)$ of A satisfies

$$u(j)^H*A = \lambda(j)*u(j)^H$$

where $u(j)^H$ denotes the conjugate transpose of $u(j)$. The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Input Parameters

jobvl	CHARACTER*1. Must be 'N' or 'V'. If *jobvl* = 'N', then left eigenvectors of *A* are not computed. If *jobvl* = 'V', then left eigenvectors of *A* are computed.
jobvr	CHARACTER*1. Must be 'N' or 'V'. If *jobvr* = 'N', then right eigenvectors of *A* are not computed. If *jobvr* = 'V', then right eigenvectors of *A* are computed.
n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
a, *work*	REAL for sgeev DOUBLE PRECISION for dgeev COMPLEX for cgeev DOUBLE COMPLEX for zgeev. Arrays: *a*(*lda*,*) is an array containing the *n*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).

lda	INTEGER. The first dimension of the array *a*. Must be at least max(1, *n*).
ldvl, *ldvr*	INTEGER. The leading dimensions of the output arrays *vl* and *vr*, respectively. Constraints: *ldvl* ≥ 1; *ldvr* ≥ 1. If *jobvl* = 'V', *ldvl* ≥ max(1, *n*); If *jobvr* = 'V', *ldvr* ≥ max(1, *n*).
lwork	INTEGER. The dimension of the array *work*. Constraint: *lwork* ≥ max(1, 3*n*), and if *jobvl* = 'V' or *jobvr* = 'V', *lwork* < max(1, 4*n*) (for real flavors); *lwork* < max(1, 2*n*) (for complex flavors). For good performance, *lwork* must generally be larger. If *lwork* = –1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.
rwork	REAL for cgeev DOUBLE PRECISION for zgeev Workspace array, DIMENSION at least max(1, 2*n*). Used in complex flavors only.

Output Parameters

a	On exit, this array is overwritten by intermediate results.
wr, *wi*	REAL for sgeev DOUBLE PRECISION for dgeev Arrays, DIMENSION at least max (1, *n*) each. Contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.
w	COMPLEX for cgeev DOUBLE COMPLEX for zgeev.

Array, DIMENSION at least max(1, *n*).
Contains the computed eigenvalues.

vl, *vr*

REAL for sgeev
DOUBLE PRECISION for dgeev
COMPLEX for cgeev
DOUBLE COMPLEX for zgeev.
Arrays:
vl(*ldvl*,*);the second dimension of *vl* must be at least max(1, *n*).
If *jobvl* = 'V', the left eigenvectors *u*(j) are stored one after another in the columns of *vl*, in the same order as their eigenvalues.
If *jobvl* = 'N', *vl* is not referenced.
For real flavors:
If the j-th eigenvalue is real, then *u*(j) = *vl*(:,j), the j-th column of *vl*.
If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then *u*(j) = *vl*(:,j) + *i***vl*(:,j+1) and *u*(j+1) = *vl*(:,j)– *i***vl*(:,j+1), where *i* = sqrt(–1).
For complex flavors:
u(j) = *vl*(:,j), the j-th column of *vl*.
vr(*ldvr*,*); the second dimension of *vr* must be at least max(1, *n*).
If *jobvr* = 'V', the right eigenvectors *v*(j) are stored one after another in the columns of *vr*, in the same order as their eigenvalues.
If *jobvr* = 'N', *vr* is not referenced.
For real flavors:
If the j-th eigenvalue is real, then *v*(j) = *vr*(:,j), the j-th column of *vr*.
If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then *v*(j) = *vr*(:,j) + *i***vr*(:,j+1) and *v*(j+1) = *vr*(:,j) – *i***vr*(:,j+1), where *i* = sqrt(–1).
For complex flavors:
v(j) = *vr*(:,j), the j-th column of *vr*.

work(1)	On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = *i*, the *QR* algorithm failed to compute all the eigenvalues, and no eigenvectors have been computed; elements *i*+1:*n* of *wr* and *wi* (for real flavors) or *w* (for complex flavors) contain those eigenvalues which have converged.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine geev interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
wr	Holds the vector of length (*n*). Used in real flavors only.
wi	Holds the vector of length (*n*). Used in real flavors only.
w	Holds the vector of length (*n*). Used in complex flavors only.
vl	Holds the matrix *VL* of size (*n*, *n*).
vr	Holds the matrix *VR* of size (*n*, *n*).
jobvl	Restored based on the presence of the argument *vl* as follows: *jobvl* = 'V', if *vl* is present, *jobvl* = 'N', if *vl* is omitted.
jobvr	Restored based on the presence of the argument *vr* as follows: *jobvr* = 'V', if *vr* is present, *jobvr* = 'N', if *vr* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?geevx

Computes the eigenvalues and left and right eigenvectors of a general matrix, with preliminary matrix balancing, and computes reciprocal condition numbers for the eigenvalues and right eigenvectors.

Syntax

Fortran 77:

```
call sgeevx(balanc, jobvl, jobvr, sense, n, a, lda, wr, wi, vl, ldvl, vr,
ldvr, ilo, ihi, scale, abnrm, rconde, rcondv, work, lwork, iwork, info)
```

```
call dgeevx(balanc, jobvl, jobvr, sense, n, a, lda, wr, wi, vl, ldvl, vr,
ldvr, ilo, ihi, scale, abnrm, rconde, rcondv, work, lwork, iwork, info)
```

```
call cgeevx(balanc, jobvl, jobvr, sense, n, a, lda, w, vl, ldvl, vr, ldvr,
ilo, ihi, scale, abnrm, rconde, rcondv, work, lwork, rwork, info)
```

```
call zgeevx(balanc, jobvl, jobvr, sense, n, a, lda, w, vl, ldvl, vr, ldvr,
ilo, ihi, scale, abnrm, rconde, rcondv, work, lwork, rwork, info)
```

Fortran 95:

```
call geevx(a, wr, wi [,vl] [,vr] [,balanc] [,ilo] [,ihi] [,scale] [,abnrm] [,
rconde] [,rcondv] [,info])
```

```
call geevx(a, w [,vl] [,vr] [,balanc] [,ilo] [,ihi] [,scale] [,abnrm] [,rconde]
[, rcondv] [,info])
```

Description

This routine computes for an *n*-by-*n* real/complex nonsymmetric matrix *A*, the eigenvalues and, optionally, the left and/or right eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (*ilo*, *ihi*, *scale*, and *abnrm*), reciprocal condition numbers for the eigenvalues (*rconde*), and reciprocal condition numbers for the right eigenvectors (*rcondv*).

The right eigenvector *v*(j) of *A* satisfies

$$A*v(j) = \lambda(j)*v(j)$$

where $\lambda(j)$ is its eigenvalue.

The left eigenvector *u*(j) of *A* satisfies

$$u(j)^H*A = \lambda(j)*u(j)^H$$

where $u(j)^H$ denotes the conjugate transpose of *u*(j). The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Balancing a matrix means permuting the rows and columns to make it more nearly upper triangular, and applying a diagonal similarity transformation *D***A**inv(*D*), where *D* is a diagonal matrix, to make its rows and columns closer in norm and the condition numbers of its eigenvalues and eigenvectors smaller. The computed reciprocal condition numbers correspond to the balanced matrix. Permuting rows and columns will not change the condition numbers in exact arithmetic) but diagonal scaling will. For further explanation of balancing, see [LUG], Section 4.10.

Input Parameters

balanc — CHARACTER*1. Must be 'N', 'P', 'S', or 'B'. Indicates how the input matrix should be diagonally scaled and/or permuted to improve the conditioning of its eigenvalues.
If *balanc* = 'N', do not diagonally scale or permute;
If *balanc* = 'P', perform permutations to make the matrix more nearly upper triangular. Do not diagonally scale;
If *balanc* = 'S', diagonally scale the matrix, i.e. replace *A* by *D***A**inv(*D*), where *D* is a diagonal matrix chosen to make the rows and columns of *A* more equal in norm. Do not permute;
If *balanc* = 'B', both diagonally scale and permute *A*.

Computed reciprocal condition numbers will be for the matrix after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.

jobvl — CHARACTER*1. Must be 'N' or 'V'.
If *jobvl* = 'N', left eigenvectors of *A* are not computed;
If *jobvl* = 'V', left eigenvectors of *A* are computed.
If *sense* = 'E' or 'B', then *jobvl* must be 'V'.

jobvr — CHARACTER*1. Must be 'N' or 'V'.
If *jobvr* = 'N', right eigenvectors of *A* are not computed;
If *jobvr* = 'V', right eigenvectors of *A* are computed.
If *sense* = 'E' or 'B', then *jobvr* must be 'V'.

sense — CHARACTER*1. Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed.
If *sense* = 'N', none are computed;
If *sense* = 'E', computed for eigenvalues only;
If *sense* = 'V', computed for right eigenvectors only;
If *sense* = 'B', computed for eigenvalues and right eigenvectors.
If *sense* is 'E' or 'B', both left and right eigenvectors must also be computed (*jobvl* = 'V' and *jobvr* = 'V').

n — INTEGER. The order of the matrix *A* (*n* ≥ 0).

a, *work* — REAL for sgeevx
DOUBLE PRECISION for dgeevx
COMPLEX for cgeevx
DOUBLE COMPLEX for zgeevx.
Arrays:
a(*lda*,*) is an array containing the *n*-by-*n* matrix *A*.
The second dimension of *a* must be at least max(1, *n*).
work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The first dimension of the array *a*. Must be at least max(1, *n*).

ldvl, *ldvr* — INTEGER. The leading dimensions of the output arrays *vl* and *vr*, respectively.
Constraints:
ldvl ≥ 1; *ldvr* ≥ 1.

If *jobvl* = 'V', *ldvl* ≥ max(1, *n*);

If *jobvr* = 'V', *ldvr* ≥ max(1, *n*).

lwork INTEGER.
The dimension of the array *work*.
For real flavors:

If *sense* = 'N' or 'E', *lwork* ≥ max(1, 2*n*), and if *jobvl* = 'V' or *jobvr* = 'V', *lwork* ≥ 3*n*;

If *sense* = 'V' or 'B', *lwork* ≥ *n**(*n*+6).
For good performance, *lwork* must generally be larger.
For complex flavors:

If *sense* = 'N'or 'E', *lwork* ≥ max(1, 2*n*);

If *sense* = 'V' or 'B', *lwork* ≥ n^2+2*n*. For good performance, *lwork* must generally be larger.
If *lwork* = –1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

rwork REAL for cgeevx
DOUBLE PRECISION for zgeevx
Workspace array, DIMENSION at least max(1, 2*n*). Used in complex flavors only.

iwork INTEGER.
Workspace array, DIMENSION at least max(1, 2*n*-2). Used in real flavors only. Not referenced if *sense* = 'N' or 'E'.

Output Parameters

a On exit, this array is overwritten.
If *jobvl* = 'V' or *jobvr* = 'V', it contains the real-Schur/Schur form of the balanced version of the input matrix *A*.

wr, *wi* REAL for sgeevx
DOUBLE PRECISION for dgeevx

Arrays, DIMENSION at least max (1, *n*) each. Contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w

COMPLEX for cgeevx
DOUBLE COMPLEX for zgeevx.
Array, DIMENSION at least max(1, *n*). Contains the computed eigenvalues.

vl, *vr*

REAL for sgeevx
DOUBLE PRECISION for dgeevx
COMPLEX for cgeevx
DOUBLE COMPLEX for zgeevx.
Arrays:
vl(*ldvl*,*); the second dimension of *vl* must be at least max(1, *n*).
If *jobvl* = 'V', the left eigenvectors *u*(j) are stored one after another in the columns of *vl*, in the same order as their eigenvalues.
If *jobvl* = 'N', *vl* is not referenced.
For real flavors:
If the j-th eigenvalue is real, then *u*(j) = *vl*(:,j), the j-th column of *vl*.
If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then *u*(j) = *vl*(:,j) + *i***vl*(:,j+1) and (j+1) = *vl*(:,j) – *i***vl*(:,j+1), where *i* = sqrt(–1).
For complex flavors:
u(j) = *vl*(:,j), the j-th column of *vl*.
vr(*ldvr*,*); the second dimension of *vr* must be at least max(1, *n*).
If *jobvr* = 'V', the right eigenvectors *v*(j) are stored one after another in the columns of *vr*, in the same order as their eigenvalues.
If *jobvr* = 'N', *vr* is not referenced.
For real flavors:
If the j-th eigenvalue is real, then *v*(j) = *vr*(:,j), the j-th column of *vr*.

	If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then `v(j) = vr(:,j) + i*vr(:,j+1)` and `v(j+1) = vr(:,j) – i*vr(:,j+1)`, where `i = sqrt(-1)` . For complex flavors: `v(j) = vr(:,j)`, the j-th column of *vr*.
ilo, *ihi*	INTEGER. *ilo* and *ihi* are integer values determined when *A* was balanced. The balanced `A(i,j) = 0` if `i > j` and `j = 1,..., ilo-1` or `i = ihi+1,..., n`. If *balanc* = 'N' or 'S', *ilo* = 1 and *ihi* = *n*.
scale	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Array, DIMENSION at least max(1, *n*). Details of the permutations and scaling factors applied when balancing *A*. If *P*(j) is the index of the row and column interchanged with row and column j, and *D*(j) is the scaling factor applied to row and column j, then *scale*(j) = *P*(j), for `j = 1,...,ilo-1` = *D*(j), for `j = ilo,...,ihi` = *P*(j) for `j = ihi+1,..., n`. The order in which the interchanges are made is *n* to *ihi*+1, then 1 to *ilo*-1.
abnrm	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. The one-norm of the balanced matrix (the maximum of the sum of absolute values of elements of any column).
rconde, *rcondv*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1, *n*) each. *rconde*(j) is the reciprocal condition number of the j-th eigenvalue. *rcondv*(j) is the reciprocal condition number of the j-th right eigenvector.
work(1)	On exit, if `info = 0`, then *work(1)* returns the required minimal size of *lwork*.
info	INTEGER.

If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* = *i*, the *QR* algorithm failed to compute all the eigenvalues, and no eigenvectors or condition numbers have been computed; elements 1:*ilo*-1 and *i*+1:*n* of *wr* and *wi* (for real flavors) or *w* (for complex flavors) contain eigenvalues which have converged.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine geevx interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
wr	Holds the vector of length (*n*). Used in real flavors only.
wi	Holds the vector of length (*n*). Used in real flavors only.
w	Holds the vector of length (*n*). Used in complex flavors only.
vl	Holds the matrix *VL* of size (*n*, *n*).
vr	Holds the matrix *VR* of size (*n*, *n*).
scale	Holds the vector of length (*n*).
rconde	Holds the vector of length (*n*).
rcondv	Holds the vector of length (*n*).
balanc	Must be 'N', 'B', 'P' or 'S'. The default value is 'N'.
jobvl	Restored based on the presence of the argument *vl* as follows: *jobvl* = 'V', if *vl* is present, *jobvl* = 'N', if *vl* is omitted.
jobvr	Restored based on the presence of the argument *vr* as follows: *jobvr* = 'V', if *vr* is present, *jobvr* = 'N', if *vr* is omitted.
sense	Restored based on the presence of arguments *rconde* and rcondv as follows: *sense* = 'B', if both *rconde* and *rcondv* are present, *sense* = 'E', if *rconde* is present and *rcondv* omitted, *sense* = 'V', if *rconde* is omitted and *rcondv* present,

sense = 'N', if both *rconde* and *rcondv* are omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Singular Value Decomposition

This section describes LAPACK driver routines used for solving singular value problems. See also computational routines computational routines that can be called to solve these problems. Table 4-12 lists all such driver routines for Fortran-77 interface. Respective routine names in Fortran-95 interface are without the first symbol (see Routine Naming Conventions).

Table 4-12 Driver Routines for Singular Value Decomposition

Routine Name	Operation performed
?gesvd	Computes the singular value decomposition of a general rectangular matrix.
?gesdd	Computes the singular value decomposition of a general rectangular matrix using a divide and conquer method.
?ggsvd	Computes the generalized singular value decomposition of a pair of general rectangular matrices.

?gesvd

Computes the singular value decomposition of a general rectangular matrix.

Syntax

Fortran 77:

```
call sgesvd(jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, info)
call dgesvd(jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, info)
call cgesvd(jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, info)
call zgesvd(jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, info)
```

Fortran 95:

```
call gesvd(a, s [,u] [,vt] [,ww] [,job] [,info])
```

Description

This routine computes the singular value decomposition (SVD) of a real/complex m-by-n matrix A, optionally computing the left and/or right singular vectors. The SVD is written

$A = U*\Sigma V^H$

where Σ is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal/unitary matrix, and V is an n-by-n orthogonal/unitary matrix. The diagonal elements of Σ are the singular values of A; they are real and non-negative, and are returned in descending order. The first min(m, n) columns of U and V are the left and right singular vectors of A.

Note that the routine returns V^H, not V.

Input Parameters

jobu	CHARACTER*1. Must be 'A', 'S', 'O', or 'N'. Specifies options for computing all or part of the matrix U. If *jobu* = 'A', all m columns of U are returned in the array *u*;

	if *jobu* = 'S', the first min(*m*, *n*) columns of *U* (the left singular vectors) are returned in the array *u*; if *jobu* = 'O', the first min(*m*, *n*) columns of *U* (the left singular vectors) are overwritten on the array *a*; if *jobu* = 'N', no columns of *U* (no left singular vectors) are computed.
jobvt	CHARACTER*1. Must be 'A', 'S', 'O', or 'N'. Specifies options for computing all or part of the matrix V^H. If *jobvt* = 'A', all *n* rows of V^H are returned in the array *vt*; if *jobvt* = 'S', the first min(*m*,*n*) rows of V^H (the right singular vectors) are returned in the array *vt*; if *jobvt* = 'O', the first min(*m*,*n*) rows of V^H (the right singular vectors) are overwritten on the array *a*; if *jobvt* = 'N', no rows of V^H (no right singular vectors) are computed. *jobvt* and *jobu* cannot both be 'O'.
m	INTEGER. The number of rows of the matrix *A* (*m* ≥ 0).
n	INTEGER. The number of columns in *A* (*n* ≥ 0).
a, *work*	REAL for sgesvd DOUBLE PRECISION for dgesvd COMPLEX for cgesvd DOUBLE COMPLEX for zgesvd. Arrays: *a*(*lda*,*) is an array containing the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of the array *a*. Must be at least max(1, *m*).
ldu, *ldvt*	INTEGER. The leading dimensions of the output arrays *u* and *vt*, respectively. Constraints: *ldu* ≥ 1; *ldvt* < 1. If *jobu* = 'S' or 'A', *ldu* ≥ *m*; If *jobvt* = 'A', *ldvt*≥ *n*;

If *jobvt* = 'S', *ldvt* ≥ min(*m*, *n*).

lwork INTEGER.

The dimension of the array *work*; *lwork* ≥ 1.
Constraints:

lwork ≥ max(3*min(*m*, *n*)+max(*m*, *n*), 5*min(*m*,*n*)) (for real flavors);

lwork ≥ 2*min(*m*, *n*)+max(*m*, *n*) (for complex flavors).
For good performance, *lwork* must generally be larger.
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for details.

rwork REAL for cgesvd
DOUBLE PRECISION for zgesvd
Workspace array, DIMENSION at least max(1, 5*min(*m*, *n*)).
Used in complex flavors only.

Output Parameters

a On exit,
If *jobu* = 'O', *a* is overwritten with the first min(*m*,*n*) columns of *U* (the left singular vectors, stored columnwise);
If *jobvt* = 'O', *a* is overwritten with the first min(*m*, *n*) rows of V^H (the right singular vectors, stored rowwise);

If *jobu*≠'O' and *jobvt*≠'O', the contents of *a* are destroyed.

s REAL for single precision flavors DOUBLE PRECISION for double precision flavors.
Array, DIMENSION at least max(1, min(*m*,*n*)). Contains the singular values of *A* sorted so that *s*(i) < *s*(i+1).

u, *vt* REAL for sgesvd
DOUBLE PRECISION for dgesvd
COMPLEX for cgesvd
DOUBLE COMPLEX for zgesvd.
Arrays:

	u(*ldu*,*); the second dimension of *u* must be at least max(1, *m*) if *jobu* = 'A', and at least max(1, min(*m*, *n*)) if *jobu* = 'S'. If *jobu* = 'A', *u* contains the *m*-by-*m* orthogonal/unitary matrix *U*. If *jobu* = 'S', *u* contains the first min(*m*, *n*) columns of *U* (the left singular vectors, stored columnwise). If *jobu* = 'N' or 'O', *u* is not referenced. *vt*(*ldvt*,*); the second dimension of *vt* must be at least max(1, *n*). If *jobvt* = 'A', *vt* contains the *n*-by-*n* orthogonal/unitary matrix V_H. If *jobvt* = 'S', *vt* contains the first min(*m*, *n*) rows of V^H (the right singular vectors, stored rowwise). If *jobvt* = 'N'or 'O', *vt* is not referenced.
work	On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*. For real flavors: If *info* > 0, *work*(2:min(*m*,*n*)) contains the unconverged superdiagonal elements of an upper bidiagonal matrix *B* whose diagonal is in *s* (not necessarily sorted). *B* satisfies *A* = *u* * *B* * *vt*, so it has the same singular values as *A*, and singular vectors related by *u* and *vt*.
rwork	On exit (for complex flavors), if *info* > 0, *rwork*(1:min(*m*,*n*)-1) contains the unconverged superdiagonal elements of an upper bidiagonal matrix *B* whose diagonal is in *s* (not necessarily sorted). *B* satisfies *A* = *u* * *B* * *vt*, so it has the same singular values as *A*, and singular vectors related by *u* and *vt*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, then if ?bdsqr did not converge, *i* specifies how many superdiagonals of the intermediate bidiagonal form *B* did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `gesvd` interface are the following:

a Holds the matrix *A* of size (*m*, *n*).

s Holds the vector of length min(*m*, *n*).

u Holds the matrix *U* of size (*m*,min(*m*, *n*)).

vt Holds the matrix *VT* of size (min(*m*, *n*),*n*).

ww Holds the vector of length (min(*m*, *n*)-1).

ww Holds the vector of length min(*m*, *n*)-1. *ww* contains the unconverged superdiagonal elements of an upper bidiagonal matrix *B* whose diagonal is in *s* (not necessarily sorted). *B* satisfies *A = U * B * VT*, so it has the same singular values as *A*, and singular vectors related by *U* and *VT*.

job Must be either `'N'`, or `'U'`, or `'V'`. The default value is `'N'`.
If *job* = `'U'`, and *u* is not present, then *u* is returned in the array *a*.
If *job* = `'V'`, and *vt* is not present, then *vt* is returned in the array *a*.

jobu Restored based on the presence of the argument *u*, value of *job* and sizes of arrays *u* and *a* as follows:
jobu = `'A'`, if *u* is present and the number of columns in *u* is equal to the number of rows in *a*,
jobu = `'S'`, if *u* is present and the number of columns in *u* is not equal to the number of rows in *a*,
jobu = `'O'`, if *u* is not present and *job* is equal to `'U'`,
jobu = `'N'`, if *u* is not present and *job* is not equal to `'U'`.

jobvt Restored based on the presence of the argument *vt*, value of *job* and sizes of arrays *vt* and *a* as follows:
jobvt = `'A'`, if *vt* is present and the number of columns in *vt* is equal to the number of rows in *a*,
jobvt = `'S'`, if *vt* is present and the number of columns in *vt* is not equal to the number of rows in *a*,
jobvt = `'O'`, if *vt* is not present and *job* is equal to `'V'`,
jobvt = `'N'`, if *vt* is not present and *job* is not equal to `'V'`,

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?gesdd

Computes the singular value decomposition of a general rectangular matrix using a divide and conquer method.

Syntax

Fortran 77:

```
call sgesdd(jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, iwork, info)
call dgesdd(jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, iwork, info)
call cgesdd(jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, iwork, info)
call zgesdd(jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, iwork, info)
```

Fortran 95:

```
call gesdd(a, s [,u] [,vt] [,jobz] [,info])
```

Description

This routine computes the singular value decomposition (SVD) of a real/complex *m*-by-*n* matrix *A*, optionally computing the left and/or right singular vectors.

If singular vectors are desired, it uses a divide and conquer algorithm. The SVD is written

$A = U*\Sigma V^H$,

where Σ is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal/unitary matrix, and V is an n-by-n orthogonal/unitary matrix. The diagonal elements of Σ are the singular values of A; they are real and non-negative, and are returned in descending order. The first min(m, n) columns of U and V are the left and right singular vectors of A.

Note that the routine returns V^H, not V.

Input Parameters

jobz	CHARACTER*1. Must be 'A', 'S', 'O', or 'N'. Specifies options for computing all or part of the matrix U. If *jobz* = 'A', all m columns of U and all n rows of V^T are returned in the arrays *u* and *vt*; if *jobz* = 'S', the first min(m, n) columns of U and the first min(m, n) rows of V^T are returned in the arrays *u* and *vt*; if *jobz* = 'O', then if $m \geq n$, the first n columns of U are overwritten in the array *a* and all rows of V^T are returned in the array *vt*; if $m \geq n$, all columns of U are returned in the array *u* and the first m rows of V^T are overwritten in the array *a*; if *jobz* = 'N', no columns of U or rows of V^T are computed.
m	INTEGER. The number of rows of the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
a, *work*	REAL for sgesdd DOUBLE PRECISION for dgesdd COMPLEX for cgesdd DOUBLE COMPLEX for zgesdd. Arrays: *a*(*lda*,*) is an array containing the m-by-n matrix A. The second dimension of *a* must be at least max(1, n). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of the array *a*. Must be at least max(1, m).

ldu, *ldvt*	INTEGER. The leading dimensions of the output arrays *u* and *vt*, respectively. Constraints: *ldu* ≥ 1; *ldvt* ≥ 1. If *jobz* = 'S' or 'A', or *jobz* = 'O' and *m* < *n*, then *ldu* ≥ *m*; If *jobz* = 'A' or *jobz* = 'O' and *m* < *n*, then *ldvt* ≥ *n*; If *jobz* = 'S', *ldvt* ≥ min(*m*, *n*).
lwork	INTEGER. The dimension of the array *work*; *lwork* ≥ 1. If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the *work*(1), and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.
rwork	REAL for cgesdd DOUBLE PRECISION for zgesdd Workspace array, DIMENSION at least max(1, 5*min(*m*,*n*)) if *jobz* = 'N'. Otherwise, the dimension of *rwork* must be at least max(1, 5*(min(*m*,*n*))^2 + 7*min(*m*,*n*)). This array is used in complex flavors only.
iwork	INTEGER. Workspace array, DIMENSION at least max(1, 8 *min(*m*, *n*)).

Output Parameters

a	On exit: If *jobz* = 'O', then if *m*≥ *n*, *a* is overwritten with the first *n* columns of *U* (the left singular vectors, stored columnwise). If *m* < *n*, *a* is overwritten with the first *m* rows of V^T (the right singular vectors, stored rowwise); If *jobz*≠'O', the contents of *a* are destroyed.
s	REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least max(1, min(*m*,*n*)). Contains the singular values of *A* sorted so that *s*(i) ≥ *s*(i+1).

u, *vt*

REAL for sgesdd
DOUBLE PRECISION for dgesdd
COMPLEX for cgesdd
DOUBLE COMPLEX for zgesdd.
Arrays:
u(*ldu*,*); the second dimension of *u* must be at least max(1, *m*) if *jobz* = 'A' or *jobz* = 'O' and *m* < *n*.
If *jobz* = 'S', the second dimension of *u* must be at least max(1, min(*m*, *n*)).
If *jobz* = 'A'or *jobz* = 'O' and *m* < *n*, *u* contains the *m*-by-*m* orthogonal/unitary matrix *U*.
If *jobz* = 'S', *u* contains the first min(*m*, *n*) columns of *U* (the left singular vectors, stored columnwise).
If *jobz* = 'O' and *m* < *n*, or *jobz* = 'N', *u* is not referenced.
vt(*ldvt*,*); the second dimension of *vt* must be at least max(1, *n*).
If *jobz* = 'A'or *jobz* = 'O' and *m*≥*n*, *vt* contains the *n*-by-*n* orthogonal/unitary matrix V^T.
If *jobz* = 'S', *vt* contains the first min(*m*, *n*) rows of V^T (the right singular vectors, stored rowwise).
If *jobz* = 'O' and *m* < *n*, or *jobz* = 'N', *vt* is not referenced.

work(1)

On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info

INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, then ?bdsdc did not converge, updating process failed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gesdd interface are the following:

a	Holds the matrix *A* of size (*m*, *n*).
s	Holds the vector of length min(*m*, *n*).
u	Holds the matrix *U* of size (*m*,min(*m*, *n*)).
vt	Holds the matrix *VT* of size (min(*m*, *n*),*n*).
job	Must be 'N', 'A', 'S', or 'O'. The default value is 'N'.

Application Notes

For real flavors:

If *jobz* = 'N', *lwork* ≥ 3*min(*m*, *n*) + max (max(*m*,*n*), 6*min(*m*, *n*));

If *jobz* = 'O', *lwork* ≥ 3*(min(*m*, *n*))^2 +

max (max(*m*, *n*), 5*(min(*m*, *n*))^2 + 4*min(*m*, *n*));

If *jobz* = 'S' or 'A', *lwork* < 3*(min(*m*, *n*))^2 +

max (max(*m*, *n*), 4*(min(*m*, *n*))^2 + 4*min(*m*, *n*)).

For complex flavors:

If *jobz* = 'N', *lwork* ≥ 2*min(*m*, *n*) + max(*m*, *n*);

If *jobz* = 'O', *lwork* ≥ 2*(min(*m*, *n*))^2 + max(*m*, *n*) + 2*min(*m*, *n*);

If *jobz* = 'S' or 'A', *lwork* ≥ (min(*m*, *n*))^2 + max(*m*, *n*) + 2*min(*m*, *n*);

For good performance, *lwork* should generally be larger.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?ggsvd

Computes the generalized singular value decomposition of a pair of general rectangular matrices.

Syntax

Fortran 77:

```
call sggsvd(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u,
ldu, v, ldv, q, ldq, work, iwork, info)
```

```
call dggsvd(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u,
ldu, v, ldv, q, ldq, work, iwork, info)
```

```
call cggsvd(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u,
ldu, v, ldv, q, ldq, work, rwork, iwork, info)
```

```
call zggsvd(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u,
ldu, v, ldv, q, ldq, work, rwork, iwork, info)
```

Fortran 95:

```
call ggsvd(a, b, alpha, beta [, k] [,l] [,u] [,v] [,q] [,iwork] [,info])
```

Description

This routine computes the generalized singular value decomposition (GSVD) of an m-by-n real/complex matrix A and p-by-n real/complex matrix B:

$U^H*A*Q = D_1*(0\ R), V^H*B*Q = D_2*(0\ R),$

where U, V and Q are orthogonal/unitary matrices.

Let $k+l$ = the effective numerical rank of the matrix $(A^H, B^H)^H$, then R is a $(k+l)$-by-$(k+l)$ nonsingular upper triangular matrix, D_1 and D_2 are m-by-$(k+l)$ and p-by-$(k+l)$ "diagonal" matrices and of the following structures, respectively:

If $m-k-l \geq 0$,

$$D_1 = \begin{array}{c} \\ k \\ l \\ m-k-l \end{array} \begin{array}{c} \begin{array}{cc} k & l \end{array} \\ \begin{pmatrix} I & 0 \\ 0 & C \\ 0 & 0 \end{pmatrix} \end{array}$$

$$D_2 = \begin{array}{c} \\ l \\ p-l \end{array} \begin{array}{c} \begin{array}{cc} k & l \end{array} \\ \begin{pmatrix} 0 & S \\ 0 & 0 \end{pmatrix} \end{array}$$

$$(0 \quad R) = \begin{array}{c} \\ k \\ l \end{array} \begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & R_{11} & R_{12} \\ 0 & 0 & R_{22} \end{pmatrix} \end{array},$$

where

C = diag($alpha(K+1)$,..., $alpha(K+l)$)

S = diag($beta(K+1)$,...,$beta(K+l)$)

$C^2 + S^2 = I$

R is stored in $a(1:k+l, n-k-l+1:n)$ on exit.

If $m-k-l < 0$,

$$D_1 = \begin{array}{c} \\ k \\ m-k \end{array} \begin{array}{c} \begin{array}{ccc} k & m-k & k+l-m \end{array} \\ \begin{pmatrix} I & 0 & 0 \\ 0 & C & 0 \end{pmatrix} \end{array}$$

$$D_2 = \begin{array}{c} \\ m-k \\ k+l-m \\ p-l \end{array} \begin{array}{c} \begin{array}{ccc} k & m-k & k+l-m \end{array} \\ \begin{pmatrix} 0 & S & 0 \\ & 0 & I \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$(0 \quad R) = \begin{array}{c} \\ k \\ m-k \\ k+l-m \end{array} \begin{array}{c} \begin{array}{cccc} n-k-l & k & m-k & k+l-m \end{array} \\ \begin{pmatrix} 0 & R_{11} & R_{12} & R_{13} \\ 0 & 0 & R_{22} & R_{23} \\ 0 & 0 & 0 & R_{33} \end{pmatrix} \end{array}$$

where

C = diag($alpha(K+1)$,..., $alpha(m)$),

S = diag($beta(K+1)$,...,$beta(m)$),

C_2 + S_2 = I

On exit, $\begin{pmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \end{pmatrix}$ is stored in $a(1{:}m, n{-}k{-}l{+}1{:}n)$ and R_{33} is stored in $b(m{-}k{+}1{:}l, n{+}m{-}k{-}l{+}1{:}n)$.

The routine computes C, S, R, and optionally the orthogonal/unitary transformation matrices U, V and Q.

In particular, if B is an n-by-n nonsingular matrix, then the GSVD of A and B implicitly gives the SVD of AB^{-1}:

$A*B^{-1} = U(*D_1 \ D_2^{-1})*V^H$.

If $(A^H, B^H)^H$ has orthonormal columns, then the GSVD of A and B is also equal to the CS decomposition of A and B. Furthermore, the GSVD can be used to derive the solution of the eigenvalue problem:

$A^H*A \ \lambda x = \lambda \ B^H*B \ x$.

Input Parameters

jobu	CHARACTER*1. Must be 'U' or 'N'. If *jobu* = 'U', orthogonal/unitary matrix U is computed. If *jobu* = 'N', U is not computed.
jobv	CHARACTER*1. Must be 'V' or 'N'. If *jobv* = 'V', orthogonal/unitary matrix V is computed. If *jobv* = 'N', V is not computed.
jobq	CHARACTER*1. Must be 'Q' or 'N'. If *jobq* = 'Q', orthogonal/unitary matrix Q is computed. If *jobq* = 'N', Q is not computed.
m	INTEGER. The number of rows of the matrix A ($m \geq 0$).
n	INTEGER. The number of columns of the matrices A and B ($n \geq 0$).
p	INTEGER. The number of rows of the matrix B ($p \geq 0$).
a, *b*, *work*	REAL for sggsvd DOUBLE PRECISION for dggsvd COMPLEX for cggsvd DOUBLE COMPLEX for zggsvd. Arrays: *a*(*lda*,*) contains the m-by-n matrix A. The second dimension of *a* must be at least max(1, n). *b*(*ldb*,*) contains the p-by-n matrix B. The second dimension of *b* must be at least max(1, n). *work*(*) is a workspace array. The dimension of *work* must be at least max(3n, m, p)+n.

lda INTEGER. The first dimension of *a*; at least max(1, *m*).

ldb INTEGER. The first dimension of *b*; at least max(1, *p*).

ldu INTEGER. The first dimension of the array *u* .
ldu ≥ max(1, *m*) if *jobu* = 'U'; *ldu* ≥ 1 otherwise.

ldv INTEGER. The first dimension of the array *v* .
ldv ≥ max(1, *p*) if *jobv* = 'V'; *ldv* ≥ 1 otherwise.

ldq INTEGER. The first dimension of the array *q* .
ldq ≥ max(1, *n*) if *jobq* = 'Q'; *ldq* ≥ 1 otherwise.

iwork INTEGER.
Workspace array, DIMENSION at least max(1, *n*).

rwork REAL for cggsvd DOUBLE PRECISION for zggsvd.
Workspace array, DIMENSION at least max(1, 2*n*). Used in complex flavors only.

Output Parameters

k, *l* INTEGER. On exit, *k* and *l* specify the dimension of the subblocks. The sum *k+l* is equal to the effective numerical rank of $(A^H, B^H)^H$.

a On exit, *a* contains the triangular matrix *R* or part of *R*.

b On exit, *b* contains part of the triangular matrix R if *m-k-l* < 0.

alpha, *beta* REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
Arrays, DIMENSION at least max(1, *n*) each.
Contain the generalized singular value pairs of *A* and *B*:
alpha(1:*k*) = 1,
beta(1:*k*) = 0,
and if *m-k-l* ≥ 0,
alpha(*k*+1:*k*+*l*) = *C*,
beta(*k*+1:*k*+*l*) = *S*,
or if *m-k-l* < 0,
alpha(*k*+1:*m*)= *C*, *alpha*(*m*+1:*k*+*l*)=0
beta(*k*+1:*m*) = *S*, *beta*(*m*+1:*k*+*l*) = 1
and

	alpha(*k*+*l*+1:*n*) = 0 *beta*(*k*+*l*+1:*n*) = 0.
u, *v*, *q*	REAL for sggsvd DOUBLE PRECISION for dggsvd COMPLEX for cggsvd DOUBLE COMPLEX for zggsvd. Arrays: *u*(*ldu*,*); the second dimension of *u* must be at least max(1, *m*). If *jobu* = 'U', *u* contains the *m*-by-*m* orthogonal/unitary matrix *U*. If *jobu* = 'N', *u* is not referenced. *v*(*ldv*,*); the second dimension of *v* must be at least max(1, *p*). If *jobv* = 'V', *v* contains the *p*-by-*p* orthogonal/unitary matrix *V*. If *jobv* = 'N', *v* is not referenced. *q*(*ldq*,*); the second dimension of *q* must be at least max(1, *n*). If *jobq* = 'Q', *q* contains the *n*-by-*n* orthogonal/unitary matrix *Q*. If *jobq* = 'N', *q* is not referenced.
iwork	On exit, *iwork* stores the sorting information.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = 1, the Jacobi-type procedure failed to converge. For further details, see subroutine ?tgsja.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ggsvd interface are the following:

a	Holds the matrix *A* of size (*m*, *n*).
b	Holds the matrix *B* of size (*p*, *n*).

alpha	Holds the vector of length (*n*).
beta	Holds the vector of length (*n*).
u	Holds the matrix *U* of size (*m*, *m*).
v	Holds the matrix *V* of size (*p*, *p*).
q	Holds the matrix *Q* of size (*n*, *n*).
iwork	Holds the vector of length (*n*).
jobu	Restored based on the presence of the argument *u* as follows: *jobu* = 'U', if *u* is present, *jobu* = 'N', if *u* is omitted.
jobv	Restored based on the presence of the argument *v* as follows: *jobz* = 'V', if *v* is present, *jobz* = 'N', if *v* is omitted.
jobq	Restored based on the presence of the argument *q* as follows: *jobz* = 'Q', if *q* is present, *jobz* = 'N', if *q* is omitted.

Generalized Symmetric Definite Eigenproblems

This section describes LAPACK driver routines used for solving generalized symmetric definite eigenproblems. See also computational routines that can be called to solve these problems. Table 4-13 lists all such driver routines for Fortran-77 interface. Respective routine names in Fortran-95 interface are without the first symbol (see Routine Naming Conventions).

Table 4-13 Driver Routines for Solving Generalized Symmetric Definite Eigenproblems

Routine Name	Operation performed
?sygv/?hegv	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem.
?sygvd/?hegvd	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem. If eigenvectors are desired, it uses a divide and conquer method.
?sygvx/?hegvx	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem.
?spgv/?hpgv	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with matrices in packed storage.

Routine Name	Operation performed
?spgvd/?hpgvd	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with matrices in packed storage. If eigenvectors are desired, it uses a divide and conquer method.
?spgvx/?hpgvx	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with matrices in packed storage.
?sbgv/?hbgv	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with banded matrices.
?sbgvd/?hbgvd	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.
?sbgvx/?hbgvx	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian definite eigenproblem with banded matrices.

?sygv

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.

Syntax

Fortran 77:

```
call ssygv(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, info)
call dsygv(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, info)
```

Fortran 95:

```
call sygv(a, b, w [,itype] [,jobz] [,uplo] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$Ax = \lambda Bx$, $ABx = \lambda x$, or $B\,Ax = \lambda x$.

Here A and B are assumed to be symmetric and B is also positive definite.

Input Parameters

itype	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if *itype* = 1, the problem type is *A*x = lambda*B*x*; if *itype* = 2, the problem type is *A*B*x* = *lambda*x*; if *itype* = 3, the problem type is *B*A*x = lambda*x*.
jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then compute eigenvalues only. If *jobz* = 'V', then compute eigenvalues and eigenvectors.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', arrays *a* and *b* store the upper triangles of *A* and *B*; If *uplo* = 'L', arrays *a* and *b* store the lower triangles of *A* and *B*.
n	INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).
a, *b*, *work*	REAL for ssygv DOUBLE PRECISION for dsygv. Arrays: *a*(*lda*,*) contains the upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the upper or lower triangle of the symmetric positive definite matrix *B*, as specified by *uplo*. The second dimension of *b* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).
ldb	INTEGER. The first dimension of *b*; at least max(1, *n*).

lwork INTEGER.
The dimension of the array *work*;
lwork ≥ max(1, 3*n*-1).
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

Output Parameters

a On exit, if *jobz* = 'V', then if *info* = 0, *a* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:
if *itype* = 1 or 2, $Z^T*B*Z = I$;
if *itype* = 3, $Z^T*\mathrm{inv}(B)*Z = I$;
If *jobz* = 'N', then on exit the upper triangle (if *uplo* = 'U') or the lower triangle (if *uplo* = 'L') of *A*, including the diagonal, is destroyed.

b On exit, if *info* ≤ *n*, the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization $B = U^T*U$ or $B = L*L^T$.

w REAL for ssygv
DOUBLE PRECISION for dsygv.
Array, DIMENSION at least max(1, *n*).
If *info* = 0, contains the eigenvalues in ascending order.

work(1) On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th argument had an illegal value.
If *info* > 0, spotrf/dpotrf and ssyev/dsyev returned an error code:
If *info* = *i* ≤ *n*, ssyev/dsyev failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `sygv` interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *n*).
w	Holds the vector of length (*n*).
itype	Must be 1, 2, or 3. The default value is 1.
jobz	Must be `'N'` or `'V'`. The default value is `'N'`.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.

Application Notes

For optimum performance use *lwork* ≥ (*nb*+2)**n*, where *nb* is the blocksize for `ssytrd`/`dsytrd` returned by `ilaenv`.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hegv

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem.

Syntax

Fortran 77:

```
call chegv(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, rwork, info)
call zhegv(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, rwork, info)
```

Fortran 95:

```
call hegv(a, b, w [,itype] [,jobz] [,uplo] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$Ax = \lambda Bx$, $ABx = \lambda x$, or $BAx = \lambda x$.

Here A and B are assumed to be Hermitian and B is also positive definite.

Input Parameters

itype	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if *itype* = 1, the problem type is *A*x = lambda*B*x*; if *itype* = 2, the problem type is *A*B*x = lambda*x*; if *itype* = 3, the problem type is *B*A*x = lambda*x*.
jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then compute eigenvalues only. If *jobz* = 'V', then compute eigenvalues and eigenvectors.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', arrays *a* and *b* store the upper triangles of *A* and *B*; If *uplo* = 'L', arrays *a* and *b* store the lower triangles of *A* and *B*.

n	INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).
a, *b*, *work*	COMPLEX for chegv DOUBLE COMPLEX for zhegv. Arrays: *a*(*lda*,*) contains the upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the upper or lower triangle of the Hermitian positive definite matrix *B*, as specified by *uplo*. The second dimension of *b* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).
ldb	INTEGER. The first dimension of *b*; at least max(1, *n*).
lwork	INTEGER. The dimension of the array *work*; $lwork \geq \max(1, 2n-1)$. If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.
rwork	REAL for chegv DOUBLE PRECISION for zhegv. Workspace array, DIMENSION at least max(1, 3*n*-2).

Output Parameters

a	On exit, if *jobz* = 'V', then if *info* = 0, *a* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows: if *itype* = 1 or 2, $Z^H*B*Z = I$; if *itype* = 3, $Z^H*\text{inv}(B)*Z = I$; If *jobz* = 'N', then on exit the upper triangle (if *uplo* = 'U') or the lower triangle (if *uplo* = 'L') of *A*, including the diagonal, is destroyed.

b	On exit, if *info* ≤ *n*, the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization $B = U^H*U$ or $B = L*L^H$.
w	REAL for chegv DOUBLE PRECISION for zhegv. Array, DIMENSION at least max(1, *n*). If *info* = 0, contains the eigenvalues in ascending order.
work(1)	On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th argument had an illegal value. If *info* > 0, cpotrf/zpotrf and cheev/zheev returned an error code: If *info* = *i* ≤ *n*, cheev/zheev failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero; If *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hegv interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *n*).
w	Holds the vector of length (*n*).
itype	Must be 1, 2, or 3. The default value is 1.
jobz	Must be 'N' or 'V'. The default value is 'N'.
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For optimum performance use *lwork* ≥ (*nb*+1)**n*, where *nb* is the blocksize for `chetrd`/`zhetrd` returned by `ilaenv`.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?sygvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

Fortran 77:

```
call ssygvd(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, iwork, liwork, info)
```

```
call dsygvd(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call sygvd(a, b, w [,itype] [,jobz] [,uplo] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$Ax = \lambda Bx$, $ABx = \lambda x$, or $BAx = \lambda x$.

Here *A* and *B* are assumed to be symmetric and *B* is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

itype	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if *itype* = 1, the problem type is *A*x = lambda*B*x*; if *itype* = 2, the problem type is *A*B*x* = *lambda*x*; if *itype* = 3, the problem type is *B*A*x = lambda*x*.
jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then compute eigenvalues only. If *jobz* = 'V', then compute eigenvalues and eigenvectors.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', arrays *a* and *b* store the upper triangles of *A* and *B*; If *uplo* = 'L', arrays *a* and *b* store the lower triangles of *A* and *B*.
n	INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).
a, *b*, *work*	REAL for ssygvd DOUBLE PRECISION for dsygvd. Arrays: *a*(*lda*,*) contains the upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the upper or lower triangle of the symmetric positive definite matrix *B*, as specified by *uplo*. The second dimension of *b* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).
ldb	INTEGER. The first dimension of *b*; at least max(1, *n*).

lwork INTEGER.
The dimension of the array *work*.
Constraints:
If $n \leq 1$, *lwork* ≥ 1;
If *jobz* = 'N' and $n>1$, *lwork* $< 2n+1$;
If *jobz* = 'V' and $n>1$, *lwork* $< 2n^2+6n+1$.
If *lwork* = −1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See Application Notes for details.

iwork INTEGER.
Workspace array, its dimension max(1, *lwork*).

liwork INTEGER.
The dimension of the array *iwork*.
Constraints:
If $n \leq 1$, *liwork* ≥ 1;
If *jobz* = 'N' and $n>1$, *liwork* ≥ 1;
If *jobz* = 'V' and $n>1$, *liwork* $\geq 5n+3$.
If *liwork* = −1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See Application Notes for details.

Output Parameters

a On exit, if *jobz* = 'V', then if *info* = 0, *a* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:
if *itype* = 1 or 2, $Z^T*B*Z = I$;
if *itype* = 3, $Z^T*\text{inv}(B)*Z = I$;
If *jobz* = 'N', then on exit the upper triangle (if *uplo* = 'U') or the lower triangle (if *uplo* = 'L') of *A*, including the diagonal, is destroyed.

b	On exit, if *info* ≤ *n*, the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization $B = U^T*U$ or $B = L*L^T$.
w	REAL for ssygvd DOUBLE PRECISION for dsygvd. Array, DIMENSION at least max(1, *n*). If *info* = 0, contains the eigenvalues in ascending order.
work(1)	On exit, if *info* = 0, then *work*(1) returns the required minimal size of *lwork*.
iwork(1)	On exit, if *info* = 0, then *iwork*(1) returns the required minimal size of *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th argument had an illegal value. If *info* > 0= *i* ≤ *n*, and *jobz* = 'N', then the algorithm failed to converge; *i* off-diagonal elements of an intermediate tridiagonal form did not converge to zero; if *jobz* = 'V', then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns *info*/(*n*+1) through mod(*info*,*n*+1); If *info* > 0= *n* + *i*, for 1 ≤ *i* ≤ *n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine sygvd interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *n*).
w	Holds the vector of length (*n*).
itype	Must be 1, 2, or 3. The default value is 1.

jobz	Must be 'N' or 'V'. The default value is 'N'.
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = –1 (*liwork* = –1).

If you choose the first option and set any of admissible *lwork* (or *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = –1 (*liwork* = –1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hegvd

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

Fortran 77:

```
call chegvd(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, rwork,
lrwork, iwork, liwork, info)
```

```
call zhegvd(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, rwork,
lrwork, iwork, liwork, info)
```

Fortran 95:

```
call hegvd(a, b, w [,itype] [,jobz] [,uplo] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$Ax = \lambda Bx$, $ABx = \lambda x$, or $BAx = \lambda x$.

Here A and B are assumed to be Hermitian and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

itype	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if *itype* = 1, the problem type is *A*x = lambda*B*x*; if *itype* = 2, the problem type is *A*B*x = lambda*x*; if *itype* = 3, the problem type is *B*A*x = lambda*x*.
jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then compute eigenvalues only. If *jobz* = 'V', then compute eigenvalues and eigenvectors.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', arrays *a* and *b* store the upper triangles of *A* and *B*; If *uplo* = 'L', arrays *a* and *b* store the lower triangles of *A* and *B*.
n	INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).
a, *b*, *work*	COMPLEX for chegvd DOUBLE COMPLEX for zhegvd. Arrays: *a*(*lda*,*) contains the upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the upper or lower triangle of the Hermitian positive definite matrix *B*, as specified by *uplo*. The second dimension of *b* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).
ldb	INTEGER. The first dimension of *b*; at least max(1, *n*).

lwork INTEGER.
The dimension of the array *work*.
Constraints:
If $n \leq 1$, *lwork* ≥ 1;
If *jobz* = 'N' and $n>1$, *lwork* $\geq n+1$;
If *jobz* = 'V' and $n>1$, *lwork* $\geq n^2+2n$.
If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See Application Notes for details.

rwork REAL for chegvd
DOUBLE PRECISION for zhegvd.
Workspace array, DIMENSION max(1, *lrwork*).

lrwork INTEGER.
The dimension of the array *rwork*.
Constraints:
If $n \leq 1$, *lrwork* ≥ 1;
If *jobz* = 'N' and $n>1$, *lrwork* $\geq n$;
If *jobz* = 'V' and $n>1$, *lrwork* $< 2n^2+5n+1$.
If *lrwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See Application Notes for details.

iwork INTEGER.
Workspace array, DIMENSION max(1, *liwork*).

liwork INTEGER.
The dimension of the array *iwork*.
Constraints:
If $n \leq 1$, *liwork* ≥ 1;
If *jobz* = 'N' and $n>1$, *liwork* ≥ 1;
If *jobz* = 'V' and $n>1$, *liwork* $\geq 5n+3$.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See Application Notes for details.

Output Parameters

a — On exit, if *jobz* = 'V', then if *info* = 0, *a* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:
if *itype* = 1 or 2, $Z^H * B * Z = I$;
if *itype* = 3, $Z^H * \mathrm{inv}(B) * Z = I$;
If *jobz* = 'N', then on exit the upper triangle (if *uplo* = 'U') or the lower triangle (if *uplo* = 'L') of *A*, including the diagonal, is destroyed.

b — On exit, if *info* ≤ *n*, the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$.

w — REAL for chegvd
DOUBLE PRECISION for zhegvd.
Array, DIMENSION at least max(1, *n*).
If *info* = 0, contains the eigenvalues in ascending order.

work(1) — On exit, if *info* = 0, then *work*(1) returns the required minimal size of *lwork*.

rwork(1) — On exit, if *info* = 0, then *rwork*(1) returns the required minimal size of *lrwork*.

iwork(1) — On exit, if *info* = 0, then *iwork*(1) returns the required minimal size of *liwork*.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th argument had an illegal value.
If *info* = *i*, and *jobz* = 'N', then the algorithm failed to converge; *i* off-diagonal elements of an intermediate tridiagonal form did not converge to zero;

if *info* = *i*, and *jobz* = 'V', then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns *info*/(*n*+1) through mod(*info*, *n*+1).

If *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hegvd interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *n*).
w	Holds the vector of length (*n*).
itype	Must be 1, 2, or 3. The default value is 1.
jobz	Must be 'N' or 'V'. The default value is 'N'.
uplo	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* (*liwork* or *lrwork*) for the first run or set *lwork* = −1 (*liwork* = −1, *lrwork* = −1).

If you choose the first option and set any of admissible *lwork* (*liwork* or *lrwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*) on exit. Use this value (*work*(1), *iwork*(1), *rwork*(1)) for subsequent runs.

If you set *lwork* = −1 (*liwork* = −1, *lrwork* = −1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*, *lrwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?sygvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.

Syntax

Fortran 77:

```
call ssygvx(itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il, iu,
abstol, m, w, z, ldz, work, lwork, iwork, ifail, info)
```

```
call dsygvx(itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il, iu,
abstol, m, w, z, ldz, work, lwork, iwork, ifail, info)
```

Fortran 95:

```
call sygvx(a, b, w [,itype] [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail]
[,abstol] [,info])
```

Description

This routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$Ax = \lambda Bx$, $ABx = \lambda x$, or $BAx = \lambda x$.

Here A and B are assumed to be symmetric and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

itype	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if *itype* = 1, the problem type is *A*x = lambda*B*x*; if *itype* = 2, the problem type is *A*B*x* = *lambda*x*; if *itype* = 3, the problem type is *B*A*x = lambda*x*.
jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then compute eigenvalues only. If *jobz* = 'V', then compute eigenvalues and eigenvectors.
range	CHARACTER*1. Must be 'A' or 'V' or 'I'.

	If *range* = 'A', the routine computes all eigenvalues. If *range* = 'V', the routine computes eigenvalues *lambda*(i) in the half-open interval: *vl*<*lambda*(i)≤ *vu*. If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', arrays *a* and *b* store the upper triangles of *A* and *B*; If *uplo* = 'L', arrays *a* and *b* store the lower triangles of *A* and *B*.
n	INTEGER. The order of the matrices *A* and *B* (*n* ≥ 0).
a, *b*, *work*	REAL for ssygvx DOUBLE PRECISION for dsygvx. Arrays: *a*(*lda*,*) contains the upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*. The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) contains the upper or lower triangle of the symmetric positive definite matrix *B*, as specified by *uplo*. The second dimension of *b* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).
ldb	INTEGER. The first dimension of *b*; at least max(1, *n*).
vl, *vu*	REAL for ssygvx DOUBLE PRECISION for dsygvx. If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: *vl*< *vu*. If *range* = 'A' or 'I', *vl* and *vu* are not referenced.
il, *iu*	INTEGER. If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: 1 ≤ *il* ≤ *iu* ≤ *n*, if *n* > 0; *il*=1 and *iu*=0 if *n* = 0.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol — REAL for ssygvx
DOUBLE PRECISION for dsygvx. The absolute error tolerance for the eigenvalues. See Application Notes for more information.

ldz — INTEGER. The leading dimension of the output array *z*. Constraints:
ldz ≥ 1; if *jobz* = 'V', *ldz* ≥ max(1, *n*).

lwork — INTEGER.
The dimension of the array *work*;
lwork < max(1, 8*n*).
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.
See Application Notes for the suggested value of *lwork*.

iwork — INTEGER.
Workspace array, DIMENSION at least max(1, 5*n*).

Output Parameters

a — On exit, the upper triangle (if *uplo* = 'U') or the lower triangle (if *uplo* = 'L') of *A*, including the diagonal, is overwritten.

b — On exit, if *info* ≤ *n*, the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization $B = U^T*U$ or $B = L*L^T$.

m — INTEGER. The total number of eigenvalues found,
0 ≤ *m* ≤ *n*. If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu*-*il*+1.

w, *z* — REAL for ssygvx
DOUBLE PRECISION for dsygvx.
Arrays:
w(*), DIMENSION at least max(1, *n*).
The first *m* elements of *w* contain the selected eigenvalues in ascending order.

	z(*ldz*,*). The second dimension of *z* must be at least max(1, *m*). If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized as follows: if *itype* = 1 or 2, $Z^T*B*Z = I$; if *itype* = 3, $Z^T*\mathrm{inv}(B)*Z = I$; If *jobz* = 'N', then *z* is not referenced. If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*. Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.
work(1)	On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.
ifail	INTEGER. Array, DIMENSION at least max(1, *n*). If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge. If *jobz* = 'N', then *ifail* is not referenced.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th argument had an illegal value. If *info* > 0, spotrf/dpotrf and ssyevx/dsyevx returned an error code: If *info* = *i* ≤ *n*, ssyevx/dsyevx failed to converge, and *i* eigenvectors failed to converge. Their indices are stored in the array *ifail*; If *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `sygvx` interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *n*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.
ifail	Holds the vector of length (*n*).
itype	Must be 1, 2, or 3. The default value is 1.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
vl	Default value for this element is *vl* = -HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted. Note that there will be an error condition if *ifail* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval `[a,b]` of width less than or equal to *abstol* + ε * max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*||*T*||$_1$ is be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

For optimum performance use *lwork* ≥ (*nb*+3)**n*, where *nb* is the blocksize for ssytrd/dsytrd returned by ilaenv.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hegvx

Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem.

Syntax

Fortran 77:

```
call chegvx(itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il, iu,
abstol, m, w, z, ldz, work, lwork, rwork, iwork, ifail, info)
```

```
call zhegvx(itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il, iu,
abstol, m, w, z, ldz, work, lwork, rwork, iwork, ifail, info)
```

Fortran 95:

```
call hegvx(a, b, w [,itype] [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail]
[,abstol] [,info])
```

Description

This routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$Ax = \lambda Bx$, $ABx = \lambda x$, or $BAx = \lambda x$.

Here A and B are assumed to be Hermitian and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

itype	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if *itype* = 1, the problem type is *A*x = lambda*B*x*; if *itype* = 2, the problem type is *A*B*x = lambda*x*; if *itype* = 3, the problem type is *B*A*x = lambda*x*.
jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then compute eigenvalues only. If *jobz* = 'V', then compute eigenvalues and eigenvectors.
range	CHARACTER*1. Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.
If *range* = 'V', the routine computes eigenvalues *lambda*(*i*) in the half-open interval:

vl<*lambda*(*i*)≤ *vu*.

If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.

uplo — CHARACTER*1. Must be 'U' or 'L'.
If *uplo* = 'U', arrays *a* and *b* store the upper triangles of *A* and *B*;
If *uplo* = 'L', arrays *a* and *b* store the lower triangles of *A* and *B*.

n — INTEGER. The order of the matrices *A* and *B* (*n* ≥ 0).

a, *b*, *work* — COMPLEX for chegvx
DOUBLE COMPLEX for zhegvx.
Arrays:
a(*lda*,*) contains the upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*.
The second dimension of *a* must be at least max(1, *n*).
b(*ldb*,*) contains the upper or lower triangle of the Hermitian positive definite matrix *B*, as specified by *uplo*.
The second dimension of *b* must be at least max(1, *n*).
work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The first dimension of *a*; at least max(1, *n*).

ldb — INTEGER. The first dimension of *b*; at least max(1, *n*).

vl, *vu* — REAL for chegvx
DOUBLE PRECISION for zhegvx.
If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
Constraint: *vl*< *vu*.
If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu* — INTEGER.
If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: 1 ≤ *il* ≤ *iu* ≤ *n*, if *n* > 0; *il*=1 and *iu*=0 if *n* = 0.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol — REAL for chegvx
DOUBLE PRECISION for zhegvx.
The absolute error tolerance for the eigenvalues. See Application Notes for more information.

ldz — INTEGER. The leading dimension of the output array *z*.
Constraints:

ldz ≥ 1; if *jobz* = 'V', *ldz* ≥ max(1, *n*).

lwork — INTEGER.

The dimension of the array *work*; *lwork* ≥ max(1, 2*n*). If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla. See Application Notes for the suggested value of *lwork*.

rwork — REAL for chegvx
DOUBLE PRECISION for zhegvx.
Workspace array, DIMENSION at least max(1, 7*n*).

iwork — INTEGER.
Workspace array, DIMENSION at least max(1, 5*n*).

Output Parameters

a — On exit, the upper triangle (if *uplo* = 'U') or the lower triangle (if *uplo* = 'L') of *A*, including the diagonal, is overwritten.

b — On exit, if *info* ≤ *n*, the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization $B = U^H*U$ or $B = L*L^H$.

m — INTEGER. The total number of eigenvalues found,

$0 \le m \le n$. If *range* = 'A', $m = n$, and if *range* = 'I', $m = iu-il+1$.

w — REAL for chegvx
DOUBLE PRECISION for zhegvx.
Array, DIMENSION at least max(1, *n*).

The first *m* elements of *w* contain the selected eigenvalues in ascending order.

z — COMPLEX for chegvx
DOUBLE COMPLEX for zhegvx.
Array z(*ldz*,*). The second dimension of *z* must be at least max(1, *m*).
If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).
The eigenvectors are normalized as follows:
if *itype* = 1 or 2, $Z^H*B*Z = I$;
if *itype* = 3, $Z^H*\mathrm{inv}(B)*Z = I$;
If *jobz* = 'N', then *z* is not referenced.
If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

work(1) — On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

ifail — INTEGER.
Array, DIMENSION at least max(1, *n*).
If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.
If *jobz* = 'N', then *ifail* is not referenced.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = –*i*, the *i*th argument had an illegal value.
If *info* > 0, cpotrf/zpotrf and cheevx/zheevx returned an error code:

If *info* = *i* ≤ *n*, cheevx/zheevx failed to converge, and *i* eigenvectors failed to converge. Their indices are stored in the array *ifail*;

If *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hegvx interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *n*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.
ifail	Holds the vector of length (*n*).
itype	Must be 1, 2, or 3. The default value is 1.
uplo	Must be 'U' or 'L'. The default value is 'U'.
vl	Default value for this element is *vl* = -HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted. Note that there will be an error condition if *ifail* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present,

Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol* + ε * max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*||*T*||$_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

For optimum performance use *lwork* ≥ (*nb*+1)**n*, where *nb* is the blocksize for chetrd/zhetrd returned by ilaenv.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?spgv

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage.

Syntax

Fortran 77:

```
call sspgv(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, info)
call dspgv(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, info)
```

Fortran 95:

```
call spgv(a, b, w [,itype] [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$Ax = \lambda Bx$, $ABx = \lambda x$, or $BAx = \lambda x$.

Here A and B are assumed to be symmetric, stored in packed format, and B is also positive definite.

Input Parameters

itype	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if *itype* = 1, the problem type is *A*x = lambda*B*x*; if *itype* = 2, the problem type is *A*B*x* = *lambda*x*; if *itype* = 3, the problem type is *B*A*x = lambda*x*.
jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then compute eigenvalues only. If *jobz* = 'V', then compute eigenvalues and eigenvectors.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', arrays *ap* and *bp* store the upper triangles of *A* and *B*;

If *uplo* = 'L', arrays *ap* and *bp* store the lower triangles of *A* and *B*.

n — INTEGER. The order of the matrices *A* and *B* (*n* ≥ 0).

ap, *bp*, *work* — REAL for sspgv
DOUBLE PRECISION for dspgv.
Arrays:
ap(*) contains the packed upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*.
The dimension of *ap* must be at least max(1, *n**(*n*+1)/2).
bp(*) contains the packed upper or lower triangle of the symmetric matrix *B*, as specified by *uplo*.
The dimension of *bp* must be at least max(1, *n**(*n*+1)/2).
work(*) is a workspace array, DIMENSION at least max(1, 3*n*).

ldz — INTEGER. The leading dimension of the output array *z*; *ldz* ≥ 1. If *jobz* = 'V', *ldz* ≥ max(1, *n*).

Output Parameters

ap — On exit, the contents of *ap* are overwritten.

bp — On exit, contains the triangular factor *U* or *L* from the Cholesky factorization $B = U^T*U$ or $B = L*L^T$, in the same storage format as *B*.

w, *z* — REAL for sspgv
DOUBLE PRECISION for dspgv.
Arrays:
w(*), DIMENSION at least max(1, *n*).
If *info* = 0, contains the eigenvalues in ascending order.
z(*ldz*,*).
The second dimension of *z* must be at least max(1, *n*).
If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:
if *itype* = 1 or 2, $Z^T*B*Z = I$;
if *itype* = 3, $Z^T*\mathrm{inv}(B)*Z = I$;
If *jobz* = 'N', then *z* is not referenced.

info — INTEGER.

If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th argument had an illegal value.
If *info* > 0, spptrf/dpptrf and sspev/dspev returned an error code:

If *info* = *i* ≤ *n*, sspev/dspev failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine spgv interface are the following:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n*(n+1)/2*).
b	Stands for argument *bp* in Fortan 77 interface. Holds the array *B* of size (*n*(n+1)/2*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*).
itype	Must be 1, 2, or 3. The default value is 1.
uplo	Must be 'U' or 'L'. The default value is 'U'.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted.

?hpgv

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem with matrices in packed storage.

Syntax

Fortran 77:

```
call chpgv(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, rwork, info)
call zhpgv(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, rwork, info)
```

Fortran 95:

```
call hpgv(a, b, w [,itype] [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$Ax = \lambda Bx$, $ABx = \lambda x$, or $BAx = \lambda x$.

Here A and B are assumed to be Hermitian, stored in packed format, and B is also positive definite.

Input Parameters

itype — INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved:
if *itype* = 1, the problem type is *A*x = lambda*B*x*;
if *itype* = 2, the problem type is *A*B*x = lambda*x*;
if *itype* = 3, the problem type is *B*A*x = lambda*x*.

jobz — CHARACTER*1. Must be 'N' or 'V'.
If *jobz* = 'N', then compute eigenvalues only.
If *jobz* = 'V', then compute eigenvalues and eigenvectors.

uplo — CHARACTER*1. Must be 'U' or 'L'.
If *uplo* = 'U', arrays *ap* and *bp* store the upper triangles of *A* and *B*;

If *uplo* = 'L', arrays *ap* and *bp* store the lower triangles of *A* and *B*.

n — INTEGER. The order of the matrices *A* and *B* (*n* ≥ 0).

ap, *bp*, *work* — COMPLEX for chpgv
DOUBLE COMPLEX for zhpgv.
Arrays:
ap(*) contains the packed upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*.
The dimension of *ap* must be at least max(1, *n**(*n*+1)/2).
bp(*) contains the packed upper or lower triangle of the Hermitian matrix *B*, as specified by *uplo*.
The dimension of *bp* must be at least max(1, *n**(*n*+1)/2).
work(*) is a workspace array, DIMENSION at least max(1, 2*n*-1).

ldz — INTEGER. The leading dimension of the output array *z*; *ldz* ≥ 1. If *jobz* = 'V', *ldz* ≥ max(1, *n*).

rwork — REAL for chpgv
DOUBLE PRECISION for zhpgv.
Workspace array, DIMENSION at least max(1, 3*n*-2).

Output Parameters

ap — On exit, the contents of *ap* are overwritten.

bp — On exit, contains the triangular factor *U* or *L* from the Cholesky factorization $B = U^H*U$ or $B = L*L^H$, in the same storage format as *B*.

w — REAL for chpgv
DOUBLE PRECISION for zhpgv.
Array, DIMENSION at least max(1, *n*).
If *info* = 0, contains the eigenvalues in ascending order.

z — COMPLEX for chpgv
DOUBLE COMPLEX for zhpgv.
Array z(*ldz*,*).
The second dimension of *z* must be at least max(1, *n*).
If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:

if *itype* = 1 or 2, $Z^H*B*Z = I$;
if *itype* = 3, $Z^H*\text{inv}(B)*Z = I$;
If *jobz* = 'N', then *z* is not referenced.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th argument had an illegal value.
If *info* > 0, cpptrf/zpptrf and chpev/zhpev returned an error code:

If *info* = *i* ≤ *n*, chpev/zhpev failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hpgv interface are the following:

a Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n**(*n*+1)/2).

b Stands for argument *bp* in Fortan 77 interface. Holds the array *B* of size (*n**(*n*+1)/2).

w Holds the vector of length (*n*).

z Holds the matrix *Z* of size (*n*, *n*).

itype Must be 1, 2, or 3. The default value is 1.

uplo Must be 'U' or 'L'. The default value is 'U'.

jobz Restored based on the presence of the argument *z* as follows:
jobz = 'V', if *z* is present,
jobz = 'N', if *z* is omitted.

?spgvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

Fortran 77:

```
call sspgvd(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork, iwork, liwork, info)
call dspgvd(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call spgvd(a, b, w [,itype] [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$Ax = \lambda Bx$, $ABx = \lambda x$, or $BAx = \lambda x$.

Here A and B are assumed to be symmetric, stored in packed format, and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

itype	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if *itype* = 1, the problem type is *A*x = lambda*B*x*; if *itype* = 2, the problem type is *A*B*x* = *lambda*x*; if *itype* = 3, the problem type is *B*A*x = lambda*x*.
jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then compute eigenvalues only. If *jobz* = 'V', then compute eigenvalues and eigenvectors.

uplo — CHARACTER*1. Must be 'U' or 'L'.
If *uplo* = 'U', arrays *ap* and *bp* store the upper triangles of *A* and *B*;
If *uplo* = 'L', arrays *ap* and *bp* store the lower triangles of *A* and *B*.

n — INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

ap, *bp*, *work* — REAL for sspgvd
DOUBLE PRECISION for dspgvd.
Arrays:
ap(*) contains the packed upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*.
The dimension of *ap* must be at least max(1, n*(n+1)/2).
bp(*) contains the packed upper or lower triangle of the symmetric matrix *B*, as specified by *uplo*.
The dimension of *bp* must be at least max(1, n*(n+1)/2).
work is a workspace array, its dimension max(1, *lwork*).

ldz — INTEGER. The leading dimension of the output array *z*; *ldz* ≥ 1. If *jobz* = 'V', *ldz* $\geq$ max(1, *n*).

lwork — INTEGER.
The dimension of the array *work*.
Constraints:

If $n \leq 1$, *lwork* ≥ 1;

If *jobz* = 'N' and $n>1$, *lwork* $\geq 2n$;

If *jobz* = 'V' and $n>1$, *lwork* $\geq 2n^2+6n+1$.

If *lwork* = −1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla. See Application Notes for details.

iwork — INTEGER.
Workspace array, its dimension max(1, *lwork*).

liwork — INTEGER.
The dimension of the array *iwork*.
Constraints:

If $n \leq 1$, $liwork \geq 1$;

If $jobz$ = 'N' and $n>1$, $liwork \geq 1$;

If $jobz$ = 'V' and $n>1$, $liwork \geq 5n+3$.

If $liwork$ = −1, then a workspace query is assumed; the routine only calculates the required sizes of the $work$ and $iwork$ arrays, returns these values as the first entries of the $work$ and $iwork$ arrays, and no error message related to $lwork$ or $liwork$ is issued by xerbla. See Application Notes for details.

Output Parameters

ap

On exit, the contents of ap are overwritten.

bp

On exit, contains the triangular factor U or L from the Cholesky factorization $B = U^T*U$ or $B = L*L^T$, in the same storage format as B.

w, *z*

REAL for sspgv

DOUBLE PRECISION for dspgv.

Arrays:

w(*), DIMENSION at least max(1, n).

If $info$ = 0, contains the eigenvalues in ascending order.

z(ldz,*).

The second dimension of z must be at least max(1, n).

If $jobz$ = 'V', then if $info$ = 0, z contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows:

if $itype$ = 1 or 2, Z^T*B*Z = I;

if $itype$ = 3, Z^T*inv(B)*Z = I;

If $jobz$ = 'N', then z is not referenced.

work(1)

On exit, if $info$ = 0, then $work$(1) returns the required minimal size of $lwork$.

iwork(1)

On exit, if $info$ = 0, then $iwork$(1) returns the required minimal size of $liwork$.

info

INTEGER.

If $info$ = 0, the execution is successful.

If $info$ = −i, the i-th argument had an illegal value.

If *info* > 0, spptrf/dpptrf and sspevd/dspevd returned an error code:

If *info* = *i* ≤ *n*, sspevd/dspevd failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine spgvd interface are the following:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n*(n+1)/2*).
b	Stands for argument *bp* in Fortan 77 interface. Holds the array *B* of size (*n*(n+1)/2*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*).
itype	Must be 1, 2, or 3. The default value is 1.
uplo	Must be 'U' or 'L'. The default value is 'U'.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = –1 (*liwork* = –1).

If you choose the first option and set any of admissible *lwork* (or *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = -1 (*liwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Application Notes

?hpgvd

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem with matrices in packed storage. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

Fortran 77:

```
call chpgvd(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork, rwork,
lrwork, iwork, liwork, info)
```

```
call zhpgvd(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork, rwork,
lrwork, iwork, liwork, info)
```

Fortran 95:

```
call hpgvd(a, b, w [,itype] [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$Ax = \lambda Bx$, $ABx = \lambda x$, or $BAx = \lambda x$.

Here *A* and *B* are assumed to be Hermitian, stored in packed format, and *B* is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

itype	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if *itype* = 1, the problem type is *A*x = lambda*B*x*; if *itype* = 2, the problem type is *A*B*x = lambda*x*; if *itype* = 3, the problem type is *B*A*x = lambda*x*.
jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then compute eigenvalues only. If *jobz* = 'V', then compute eigenvalues and eigenvectors.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', arrays *ap* and *bp* store the upper triangles of *A* and *B*; If *uplo* = 'L', arrays *ap* and *bp* store the lower triangles of *A* and *B*.
n	INTEGER. The order of the matrices *A* and *B* (*n* ≥ 0).
ap, *bp*, *work*	COMPLEX for chpgvd DOUBLE COMPLEX for zhpgvd. Arrays: *ap(*)* contains the packed upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*. The dimension of *ap* must be at least max(1, *n**(*n*+1)/2). *bp(*)* contains the packed upper or lower triangle of the Hermitian matrix *B*, as specified by *uplo*. The dimension of *bp* must be at least max(1, *n**(*n*+1)/2). *work* is a workspace array, its dimension max(1, *lwork*).
ldz	INTEGER. The leading dimension of the output array *z*; *ldz* ≥ 1. If *jobz* = 'V', *ldz* ≥ max(1, *n*).
lwork	INTEGER. The dimension of the array *work*. Constraints: If *n* ≤ 1, *lwork* ≥ 1;

If *jobz* = 'N' and *n*>1, *lwork* ≥ *n*;

If *jobz* = 'V' and *n*>1, *lwork* ≥ 2*n*.

If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See Application Notes for details.

rwork — REAL for chpgvd
DOUBLE PRECISION for zhpgvd.
Workspace array, its dimension max(1, *lrwork*).

lrwork — INTEGER.
The dimension of the array *rwork*.
Constraints:

If *n* ≤ 1, *lrwork* ≥ 1;

If *jobz* = 'N' and *n*>1, *lrwork* ≥ *n*;

If *jobz* = 'V' and *n*>1, *lrwork* ≥ $2n^2+5n+1$.

If *lrwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See Application Notes for details.

iwork — INTEGER.
Workspace array, its dimension max(1, *liwork*).

liwork — INTEGER.
The dimension of the array *iwork*.
Constraints:

If *n* ≤ 1, *liwork* ≥ 1;

If *jobz* = 'N' and *n*>1, *liwork* ≥ 1;

If *jobz* = 'V' and *n*>1, *liwork* ≥ 5*n*+3.

If *liwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries

of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See Application Notes for details.

Output Parameters

ap — On exit, the contents of *ap* are overwritten.

bp — On exit, contains the triangular factor *U* or *L* from the Cholesky factorization $B = U^H*U$ or $B = L*L^H$, in the same storage format as *B*.

w — REAL for chpgvd
DOUBLE PRECISION for zhpgvd.
Array, DIMENSION at least max(1, *n*).
If *info* = 0, contains the eigenvalues in ascending order.

z — COMPLEX for chpgvd
DOUBLE COMPLEX for zhpgvd.
Array z(*ldz*,*).
The second dimension of *z* must be at least max(1, *n*).
If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:
if *itype* = 1 or 2, $Z^H*B*Z = I$;
if *itype* = 3, $Z^H*\mathrm{inv}(B)*Z = I$;
If *jobz* = 'N', then *z* is not referenced.

work(1) — On exit, if *info* = 0, then *work*(1) returns the required minimal size of *lwork*.

rwork(1) — On exit, if *info* = 0, then *rwork*(1) returns the required minimal size of *lrwork*.

iwork(1) — On exit, if *info* = 0, then *iwork*(1) returns the required minimal size of *liwork*.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th argument had an illegal value.
If *info* > 0, cpptrf/zpptrf and chpevd/zhpevd returned an error code:

If *info* = *i* ≤ *n*, chpevd/zhpevd failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hpgvd interface are the following:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n*(n+1)/2*).
b	Stands for argument *bp* in Fortan 77 interface. Holds the array *B* of size (*n*(n+1)/2*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*).
itype	Must be 1, 2, or 3. The default value is 1.
uplo	Must be 'U' or 'L'. The default value is 'U'.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* (*liwork* or *lrwork*) for the first run or set *lwork* = -1 (*liwork* = -1, *lrwork* = -1).

If you choose the first option and set any of admissible *lwork* (*liwork* or *lrwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*) on exit. Use this value (*work*(1), *iwork*(1), *rwork*(1)) for subsequent runs.

If you set *lwork* = -1 (*liwork* = -1, *lrwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*, *lrwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?spgvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage.

Syntax

Fortran 77:

```
call sspgvx(itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m,
w, z, ldz, work, iwork, ifail, info)
```

```
call dspgvx(itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m,
w, z, ldz, work, iwork, ifail, info)
```

Fortran 95:

```
call spgvx(a, b, w [,itype] [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail]
[,abstol] [,info])
```

Description

This routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$Ax = \lambda Bx$, $ABx = \lambda x$, or $BAx = \lambda x$.

Here *A* and *B* are assumed to be symmetric, stored in packed format, and *B* is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

itype	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if *itype* = 1, the problem type is *A*x = lambda*B*x*; if *itype* = 2, the problem type is *A*B*x* = *lambda*x*; if *itype* = 3, the problem type is *B*A*x = lambda*x*.
jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then compute eigenvalues only. If *jobz* = 'V', then compute eigenvalues and eigenvectors.
range	CHARACTER*1. Must be 'A' or 'V' or 'I'. If *range* = 'A', the routine computes all eigenvalues. If *range* = 'V', the routine computes eigenvalues *lambda(i)* in the half-open interval: *vl<lambda(i)≤ vu*. If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', arrays *ap* and *bp* store the upper triangles of *A* and *B*; If *uplo* = 'L', arrays *ap* and *bp* store the lower triangles of *A* and *B*.
n	INTEGER. The order of the matrices *A* and *B* (*n* ≥ 0).
ap, *bp*, *work*	REAL for sspgvx DOUBLE PRECISION for dspgvx. Arrays: *ap(*)* contains the packed upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*. The dimension of *ap* must be at least max(1, *n**(*n*+1)/2). *bp(*)* contains the packed upper or lower triangle of the symmetric matrix *B*, as specified by *uplo*. The dimension of *bp* must be at least max(1, *n**(*n*+1)/2). *work(*)* is a workspace array, DIMENSION at least max(1, 8*n*).
vl, *vu*	REAL for sspgvx DOUBLE PRECISION for dspgvx.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
Constraint: *vl*< *vu*.
If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu*
INTEGER.
If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: 1 ≤ *il* ≤ *iu* ≤ *n*, if *n* > 0; *il*=1 and *iu*=0 if *n* = 0.
If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol
REAL for sspgvx
DOUBLE PRECISION for dspgvx.
The absolute error tolerance for the eigenvalues. See Application Notes for more information.

ldz
INTEGER. The leading dimension of the output array *z*.
Constraints:
ldz ≥ 1; if *jobz* = 'V', *ldz* ≥ max(1, *n*).

iwork
INTEGER.
Workspace array, DIMENSION at least max(1, 5*n*).

Output Parameters

ap
On exit, the contents of *ap* are overwritten.

bp
On exit, contains the triangular factor *U* or *L* from the Cholesky factorization $B = U^T*U$ or $B = L*L^T$, in the same storage format as *B*.

m
INTEGER. The total number of eigenvalues found,
0 ≤ *m* ≤ *n*. If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu*-*il*+1.

w, *z*
REAL for sspgvx
DOUBLE PRECISION for dspgvx.
Arrays:
w(*), DIMENSION at least max(1, *n*).
If *info* = 0, contains the eigenvalues in ascending order.
z(*ldz*,*).
The second dimension of *z* must be at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized as follows:
if *itype* = 1 or 2, $Z^T*B*Z = I$;
if *itype* = 3, $Z^T*\mathrm{inv}(B)*Z = I$;
If *jobz* = 'N', then *z* is not referenced.
If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

ifail — INTEGER.
Array, DIMENSION at least max(1, *n*).
If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.
If *jobz* = 'N', then *ifail* is not referenced.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th argument had an illegal value.
If *info* > 0, spptrf/dpptrf and sspevx/dspevx returned an error code:

If *info* = *i* ≤ *n*, sspevx/dspevx failed to converge, and *i* eigenvectors failed to converge. Their indices are stored in the array *ifail*;

If *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `spgvx` interface are the following:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n*(n+1)/2*).
b	Stands for argument *bp* in Fortan 77 interface. Holds the array *B* of size (*n*(n+1)/2*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.
ifail	Holds the vector of length (*n*).
itype	Must be 1, 2, or 3. The default value is 1.
uplo	Must be 'U' or 'L'. The default value is 'U'.
vl	Default value for this element is *vl* = -HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted. Note that there will be an error condition if *ifail* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol* + ε * max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*||*T*||$_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?hpgvx

Computes selected eigenvalues and, optionally, eigenvectors of a generalized Hermitian definite eigenproblem with matrices in packed storage.

Syntax

Fortran 77:

```
call chpgvx(itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m,
w, z, ldz, work, rwork, iwork, ifail, info)
```

```
call zhpgvx(itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m,
w, z, ldz, work, rwork, iwork, ifail, info)
```

Fortran 95:

```
call hpgvx(a, b, w [,itype] [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail]
[,abstol] [,info])
```

Description

This routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

$Ax = \lambda Bx$, $ABx = \lambda x$, or $BAx = \lambda x$.

Here *A* and *B* are assumed to be Hermitian, stored in packed format, and *B* is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

itype — INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved:
if *itype* = 1, the problem type is *A*x = lambda*B*x*;
if *itype* = 2, the problem type is *A*B*x = lambda*x*;
if *itype* = 3, the problem type is *B*A*x = lambda*x*.

jobz — CHARACTER*1. Must be 'N' or 'V'.
If *jobz* = 'N', then compute eigenvalues only.
If *jobz* = 'V', then compute eigenvalues and eigenvectors.

range — CHARACTER*1. Must be 'A' or 'V' or 'I'.
If *range* = 'A', the routine computes all eigenvalues.
If *range* = 'V', the routine computes eigenvalues *lambda(i)* in the half-open interval:

vl< *lambda(i)* ≤ *vu*.

If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.

uplo — CHARACTER*1. Must be 'U' or 'L'.
If *uplo* = 'U', arrays *ap* and *bp* store the upper triangles of *A* and *B*;
If *uplo* = 'L', arrays *ap* and *bp* store the lower triangles of *A* and *B*.

n — INTEGER. The order of the matrices *A* and *B* (*n* ≥ 0).

ap, *bp*, *work* — COMPLEX for chpgvx
DOUBLE COMPLEX for zhpgvx.
Arrays:
ap(*) contains the packed upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*.
The dimension of *ap* must be at least max(1, *n**(*n*+1)/2).
bp(*) contains the packed upper or lower triangle of the Hermitian matrix *B*, as specified by *uplo*.
The dimension of *bp* must be at least max(1, *n**(*n*+1)/2).
work(*) is a workspace array, DIMENSION at least max(1, 2*n*).

vl, *vu* — REAL for chpgvx
DOUBLE PRECISION for zhpgvx.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
Constraint: *vl*< *vu*.
If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu*
INTEGER.
If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: 1 ≤ *il* ≤ *iu* ≤ *n*, if *n* > 0; *il*=1 and *iu*=0 if *n* = 0.
If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol
REAL for chpgvx
DOUBLE PRECISION for zhpgvx.
The absolute error tolerance for the eigenvalues.
See Application Notes for more information.

ldz
INTEGER. The leading dimension of the output array *z*; *ldz* ≥ 1. If *jobz* = 'V', *ldz* ≥ max(1, *n*).

rwork
REAL for chpgvx
DOUBLE PRECISION for zhpgvx.
Workspace array, DIMENSION at least max(1, 7*n*).

iwork
INTEGER.
Workspace array, DIMENSION at least max(1, 5*n*).

Output Parameters

ap
On exit, the contents of *ap* are overwritten.

bp
On exit, contains the triangular factor *U* or *L* from the Cholesky factorization $B = U^H*U$ or $B = L*L^H$, in the same storage format as *B*.

m
INTEGER. The total number of eigenvalues found, 0 ≤ *m* ≤ *n*. If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu*-*il*+1.

w
REAL for chpgvx
DOUBLE PRECISION for zhpgvx.
Array, DIMENSION at least max(1, *n*).
If *info* = 0, contains the eigenvalues in ascending order.

z

COMPLEX for chpgvx
DOUBLE COMPLEX for zhpgvx.
Array z(*ldz*,*).
The second dimension of *z* must be at least max(1, *n*).
If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).
The eigenvectors are normalized as follows:
if *itype* = 1 or 2, Z^H**B***Z* = I;
if *itype* = 3, Z^H*inv(*B*)**Z* = I;
If *jobz* = 'N', then *z* is not referenced.
If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

ifail

INTEGER.
Array, DIMENSION at least max(1, *n*).
If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.
If *jobz* = 'N', then *ifail* is not referenced.

info

INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th argument had an illegal value.
If *info* > 0, cpptrf/zpptrf and chpevx/zhpevx returned an error code:

If *info* = *i* ≤ *n*, chpevx/zhpevx failed to converge, and *i* eigenvectors failed to converge. Their indices are stored in the array *ifail*;

If *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `hpgvx` interface are the following:

a	Stands for argument *ap* in Fortan 77 interface. Holds the array *A* of size (*n*(n+1)/2*).
b	Stands for argument *bp* in Fortan 77 interface. Holds the array *B* of size (*n*(n+1)/2*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*), where the values *n* and *m* are significant.
ifail	Holds the vector of length (*n*).
itype	Must be 1, 2, or 3. The default value is 1.
uplo	Must be `'U'` or `'L'`. The default value is `'U'`.
vl	Default value for this element is *vl* = -HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = `'V'`, if *z* is present, *jobz* = `'N'`, if *z* is omitted. Note that there will be an error condition if *ifail* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = `'V'`, if one of or both *vl* and *vu* are present, *range* = `'I'`, if one of or both *il* and *iu* are present, *range* = `'A'`, if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol* + ε * max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*||*T*||$_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?sbgv

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices.

Syntax

Fortran 77:

```
call ssbgv(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, info)
call dsbgv(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, info)
```

Fortran 95:

```
call sbgv(a, b, w [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form *Ax* = λ*Bx*. Here *A* and *B* are assumed to be symmetric and banded, and *B* is also positive definite.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then compute eigenvalues only. If *jobz* = 'V', then compute eigenvalues and eigenvectors.
uplo	CHARACTER*1. Must be 'U' or 'L'.

	If *uplo* = 'U', arrays *ab* and *bb* store the upper triangles of *A* and *B*; If *uplo* = 'L', arrays *ab* and *bb* store the lower triangles of *A* and *B*.
n	INTEGER. The order of the matrices *A* and *B* (*n* ≥ 0).
ka	INTEGER. The number of super- or sub-diagonals in *A* (*ka* ≥ 0).
kb	INTEGER. The number of super- or sub-diagonals in *B* (*kb* ≥ 0).
ab, *bb*, *work*	REAL for ssbgv DOUBLE PRECISION for dsbgv Arrays: *ab* (*ldab*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A* (as specified by *uplo*) in band storage format. The second dimension of the array *ab* must be at least max(1, *n*). *bb*(*ldbb*,*) is an array containing either upper or lower triangular part of the symmetric matrix *B* (as specified by *uplo*) in band storage format. The second dimension of the array *bb* must be at least max(1, *n*). *work(*)* is a workspace array, dimension at least max(1, 3*n*)
ldab	INTEGER. The first dimension of the array *ab*; must be at least *ka*+1.
ldbb	INTEGER. The first dimension of the array *bb*; must be at least *kb*+1.
ldz	INTEGER. The leading dimension of the output array *z*; *ldz* ≥ 1. If *jobz* = 'V', *ldz* ≥ max(1, *n*).

Output Parameters

ab	On exit, the contents of *ab* are overwritten.

bb — On exit, contains the factor *S* from the split Cholesky factorization $B = S^T*S$, as returned by spbstf/dpbstf.

w, *z* — REAL for ssbgv
DOUBLE PRECISION for dsbgv
Arrays:
w(*), DIMENSION at least max(1, *n*).
If *info* = 0, contains the eigenvalues in ascending order.
z(*ldz*,*).
The second dimension of *z* must be at least max(1, *n*).
If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that $Z^T*B*Z = I$.
If *jobz* = 'N', then *z* is not referenced.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th argument had an illegal value.
If *info* > 0, and

if $i \le n$, the algorithm failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if *info* = *n* + *i*, for $1 \le i \le n$, then spbstf/dpbstf returned *info* = *i* and *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine sbgv interface are the following:

a — Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*ka+1*,*n*).

b — Stands for argument *bb* in Fortan 77 interface. Holds the array *B* of size (*kb+1*,*n*).

w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted.

?hbgv

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem with banded matrices.

Syntax

Fortran 77:

```
call chbgv(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, rwork, info)
call zhbgv(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, rwork, info)
```

Fortran 95:

```
call hbgv(a, b, w [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form $Ax = \lambda Bx$. Here A and B are assumed to be Hermitian and banded, and B is also positive definite.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then compute eigenvalues only. If *jobz* = 'V', then compute eigenvalues and eigenvectors.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', arrays *ab* and *bb* store the upper triangles of *A* and *B*;

If *uplo* = 'L', arrays *ab* and *bb* store the lower triangles of *A* and *B*.

n — INTEGER. The order of the matrices *A* and *B* (*n* ≥ 0).

ka — INTEGER. The number of super- or sub-diagonals in *A* (*ka* ≥ 0).

kb — INTEGER. The number of super- or sub-diagonals in *B* (*kb* ≥ 0).

ab, *bb*, *work* — COMPLEX for chbgv
DOUBLE COMPLEX for zhbgv
Arrays:
ab (*ldab*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by *uplo*) in band storage format.
The second dimension of the array *ab* must be at least max(1, *n*).
bb(*ldbb*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *B* (as specified by *uplo*) in band storage format.
The second dimension of the array *bb* must be at least max(1, *n*).
work(*) is a workspace array, dimension at least max(1, *n*).

ldab — INTEGER. The first dimension of the array *ab*; must be at least *ka*+1.

ldbb — INTEGER. The first dimension of the array *bb*; must be at least *kb*+1.

ldz — INTEGER. The leading dimension of the output array *z*; *ldz* ≥ 1. If *jobz* = 'V', *ldz* ≥ max(1, *n*).

rwork — REAL for chbgv
DOUBLE PRECISION for zhbgv.
Workspace array, DIMENSION at least max(1, 3*n*).

Output Parameters

ab — On exit, the contents of *ab* are overwritten.

bb	On exit, contains the factor *S* from the split Cholesky factorization *B* = S^H**S*, as returned by cpbstf/zpbstf.
w	REAL for chbgv DOUBLE PRECISION for zhbgv. Array, DIMENSION at least max(1, *n*). If *info* = 0, contains the eigenvalues in ascending order.
z	COMPLEX for chbgv DOUBLE COMPLEX for zhbgv Array z(*ldz*,*). The second dimension of *z* must be at least max(1, *n*). If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that Z^H**B***Z* = I. If *jobz* = 'N', then *z* is not referenced.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th argument had an illegal value. If *info* > 0, and if *i* ≤ *n*, the algorithm failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero; if *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then cpbstf/zpbstf returned *info* = *i* and *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hbgv interface are the following:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*ka+1*,*n*).

b	Stands for argument *bb* in Fortan 77 interface. Holds the array *B* of size (*kb+1*,*n*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted.

?sbgvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

Fortran 77:

```
call ssbgvd(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, lwork, iwork, liwork, info)
```

```
call dsbgvd(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call sbgvd(a, b, w [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form $Ax = \lambda Bx$. Here A and B are assumed to be symmetric and banded, and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'.

	If *jobz* = 'N', then compute eigenvalues only. If *jobz* = 'V', then compute eigenvalues and eigenvectors.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', arrays *ab* and *bb* store the upper triangles of *A* and *B*; If *uplo* = 'L', arrays *ab* and *bb* store the lower triangles of *A* and *B*.
n	INTEGER. The order of the matrices *A* and *B* (*n* ≥ 0).
ka	INTEGER. The number of super- or sub-diagonals in *A* (*ka* ≥ 0).
kb	INTEGER. The number of super- or sub-diagonals in *B* (*kb* ≥ 0).
ab, *bb*, *work*	REAL for ssbgvd DOUBLE PRECISION for dsbgvd Arrays: *ab* (*ldab*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A* (as specified by *uplo*) in band storage format. The second dimension of the array *ab* must be at least max(1, *n*). *bb*(*ldbb*,*) is an array containing either upper or lower triangular part of the symmetric matrix *B* (as specified by *uplo*) in band storage format. The second dimension of the array *bb* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
ldab	INTEGER. The first dimension of the array *ab*; must be at least *ka*+1.
ldbb	INTEGER. The first dimension of the array *bb*; must be at least *kb*+1.
ldz	INTEGER. The leading dimension of the output array *z*; *ldz* ≥ 1. If *jobz* = 'V', *ldz* ≥ max(1, *n*).
lwork	INTEGER. The dimension of the array *work*. Constraints:

If $n \le 1$, $lwork \ge 1$;

If $jobz$ = 'N' and $n>1$, $lwork \ge 3n$;

If $jobz$ = 'V' and $n>1$, $lwork \ge 2n^2+5n+1$.

If $lwork$ = -1, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ and $iwork$ arrays, returns these values as the first entries of the $work$ and $iwork$ arrays, and no error message related to $lwork$ or $liwork$ is issued by xerbla. See Application Notes for details.

iwork INTEGER.
Workspace array, its dimension max(1, $liwork$).

liwork INTEGER.
The dimension of the array $iwork$.
Constraints:

If $n \le 1$, $liwork \ge 1$;

If $jobz$ = 'N' and $n>1$, $liwork \ge 1$;

If $jobz$ = 'V' and $n>1$, $liwork \ge 5n+3$.

If $liwork$ = -1, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ and $iwork$ arrays, returns these values as the first entries of the $work$ and $iwork$ arrays, and no error message related to $lwork$ or $liwork$ is issued by xerbla. See Application Notes for details.

Output Parameters

ab On exit, the contents of ab are overwritten.

bb On exit, contains the factor S from the split Cholesky factorization $B = S^T*S$, as returned by spbstf/dpbstf.

w, *z* REAL for ssbgvd
DOUBLE PRECISION for dsbgvd
Arrays:
w(*), DIMENSION at least max(1, n).
If $info$ = 0, contains the eigenvalues in ascending order.
z(ldz,*).
The second dimension of z must be at least max(1, n).

	If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that $Z^T*B*Z = I$. If *jobz* = 'N', then *z* is not referenced.
work(1)	On exit, if *info* = 0, then *work*(1) returns the required minimal size of *lwork*.
iwork(1)	On exit, if *info* = 0, then *iwork*(1) returns the required minimal size of *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th argument had an illegal value. If *info* > 0, and if *i* ≤ *n*, the algorithm failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero; if *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then spbstf/dpbstf returned *info* = *i* and *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine sbgvd interface are the following:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*ka+1*,*n*).
b	Stands for argument *bb* in Fortan 77 interface. Holds the array *B* of size (*kb+1*,*n*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
jobz	Restored based on the presence of the argument *z* as follows:

jobz = 'V', if *z* is present,
jobz = 'N', if *z* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = –1 (*liwork* = –1).

If you choose the first option and set any of admissible *lwork* (or *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = –1 (*liwork* = –1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hbgvd

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

Fortran 77:

```
call chbgvd(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, lwork,
rwork, lrwork, iwork, liwork, info)
```

```
call zhbgvd(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, lwork,
rwork, lrwork, iwork, liwork, info)
```

Fortran 95:

```
call hbgvd(a, b, w [,uplo] [,z] [,info])
```

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form $Ax = \lambda Bx$. Here A and B are assumed to be Hermitian and banded, and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then compute eigenvalues only. If *jobz* = 'V', then compute eigenvalues and eigenvectors.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', arrays *ab* and *bb* store the upper triangles of *A* and *B*; If *uplo* = 'L', arrays *ab* and *bb* store the lower triangles of *A* and *B*.
n	INTEGER. The order of the matrices *A* and *B* (*n* ≥ 0).
ka	INTEGER. The number of super- or sub-diagonals in *A* (*ka* ≥ 0).
kb	INTEGER. The number of super- or sub-diagonals in *B* (*kb* ≥ 0).
ab, *bb*, *work*	COMPLEX for chbgvd DOUBLE COMPLEX for zhbgvd Arrays: *ab* (*ldab*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by *uplo*) in band storage format. The second dimension of the array *ab* must be at least max(1, *n*). *bb*(*ldbb*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *B* (as specified by *uplo*) in band storage format. The second dimension of the array *bb* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).

ldab INTEGER. The first dimension of the array *ab*; must be at least *ka*+1.

ldbb INTEGER. The first dimension of the array *bb*; must be at least *kb*+1.

ldz INTEGER. The leading dimension of the output array *z*; *ldz* ≥ 1. If *jobz* = 'V', *ldz* ≥ max(1, *n*).

lwork INTEGER.
The dimension of the array *work*.
Constraints:

If *n* ≤ 1, *lwork* ≥ 1;

If *jobz* = 'N' and *n*>1, *lwork* ≥ *n*;
If *jobz* = 'V' and *n*>1, *lwork* < $2n^2$.
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See Application Notes for details.

rwork REAL for chbgvd
DOUBLE PRECISION for zhbgvd.
Workspace array, DIMENSION max(1, *lrwork*).

lrwork INTEGER.
The dimension of the array *rwork*.
Constraints:

If *n* ≤ 1, *lrwork* ≥ 1;

If *jobz* = 'N' and *n*>1, *lrwork* ≥ *n*;

If *jobz* = 'V' and *n*>1, *lrwork* ≥ $2n^2$+5*n* +1.
If *lrwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See Application Notes for details.

iwork INTEGER.
Workspace array, DIMENSION max(1, *liwork*).

liwork INTEGER.
The dimension of the array *iwork*.
Constraints:
If $n \leq 1$, $liwork < 1$;
If *jobz* = 'N' and $n>1$, $liwork \geq 1$;
If *jobz* = 'V' and $n>1$, $liwork \geq 5n+3$.
If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by xerbla. See Application Notes for details.

Output Parameters

ab On exit, the contents of *ab* are overwritten.

bb On exit, contains the factor *S* from the split Cholesky factorization $B = S^H*S$, as returned by cpbstf/zpbstf.

w REAL for chbgvd
DOUBLE PRECISION for zhbgvd.
Array, DIMENSION at least max(1, *n*) .
If *info* = 0, contains the eigenvalues in ascending order.

z COMPLEX for chbgvd
DOUBLE COMPLEX for zhbgvd
Array z(*ldz*,*) .
The second dimension of *z* must be at least max(1, *n*).
If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors , with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that $Z^H*B*Z = I$.
If *jobz* = 'N', then *z* is not referenced.

work(1) On exit, if *info* = 0, then *work*(1) returns the required minimal size of *lwork*.

rwork(1) On exit, if *info* = 0, then *rwork*(1) returns the required minimal size of *lrwork*.

iwork(1)	On exit, if *info* = 0, then *iwork*(1) returns the required minimal size of *liwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th argument had an illegal value. If *info* > 0, and if *i* ≤ *n*, the algorithm failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero; if *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then cpbstf/zpbstf returned *info* = *i* and *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hbgvd interface are the following:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*ka+1*,*n*).
b	Stands for argument *bb* in Fortan 77 interface. Holds the array *B* of size (*kb+1*,*n*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* (*liwork* or *lrwork*) for the first run or set *lwork* = -1 (*liwork* = -1, *lrwork* = -1).

If you choose the first option and set any of admissible *lwork* (*liwork* or *lrwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*) on exit. Use this value (*work*(1), *iwork*(1), *rwork*(1)) for subsequent runs.

If you set *lwork* = -1 (*liwork* = -1, *lrwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*, *lrwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?sbgvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices.

Syntax

Fortran 77:

```
call ssbgvx(jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q, ldq, vl, vu,
il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
```

```
call dsbgvx(jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q, ldq, vl, vu,
il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
```

Fortran 95:

```
call sbgvx(a, b, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,q]
[,abstol] [,info])
```

Description

This routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form $Ax = \lambda Bx$. Here A and B are assumed to be symmetric and banded, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either all eigenvalues, a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz — CHARACTER*1. Must be 'N' or 'V'.
If *jobz* = 'N', then compute eigenvalues only.
If *jobz* = 'V', then compute eigenvalues and eigenvectors.

range — CHARACTER*1. Must be 'A' or 'V' or 'I'.
If *range* = 'A', the routine computes all eigenvalues.
If *range* = 'V', the routine computes eigenvalues *lambda*(*i*) in the half-open interval:
$vl < lambda(i) \leq vu$.
If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.

uplo — CHARACTER*1. Must be 'U' or 'L'.
If *uplo* = 'U', arrays *ab* and *bb* store the upper triangles of *A* and *B*;
If *uplo* = 'L', arrays *ab* and *bb* store the lower triangles of *A* and *B*.

n — INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

ka — INTEGER. The number of super- or sub-diagonals in *A* ($ka \geq 0$).

kb — INTEGER. The number of super- or sub-diagonals in *B* ($kb \geq 0$).

ab, *bb*, *work* — REAL for ssbgvx
DOUBLE PRECISION for dsbgvx
Arrays:
ab (*ldab*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A* (as specified by *uplo*) in band storage format.
The second dimension of the array *ab* must be at least max(1, *n*).
bb(*ldbb*,*) is an array containing either upper or lower triangular part of the symmetric matrix *B* (as specified by *uplo*) in band storage format.
The second dimension of the array *bb* must be at least max(1, *n*).
work(*) is a workspace array, DIMENSION (7**n*).

ldab	INTEGER. The first dimension of the array *ab*; must be at least *ka*+1.
ldbb	INTEGER. The first dimension of the array *bb*; must be at least *kb*+1.
vl, *vu*	REAL for ssbgvx DOUBLE PRECISION for dsbgvx. If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: *vl*< *vu*. If *range* = 'A' or 'I', *vl* and *vu* are not referenced.
il, *iu*	INTEGER. If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: 1 ≤ *il* ≤ *iu* ≤ *n*, if *n* > 0; *il*=1 and *iu*=0 if *n* = 0. If *range* = 'A' or 'V', *il* and *iu* are not referenced.
abstol	REAL for ssbgvx DOUBLE PRECISION for dsbgvx. The absolute error tolerance for the eigenvalues. See Application Notes for more information.
ldz	INTEGER. The leading dimension of the output array *z*; *ldz* ≥ 1. If *jobz* = 'V', *ldz* ≥ max(1, *n*).
ldq	INTEGER. The leading dimension of the output array *q*; *ldq* < 1. If *jobz* = 'V', *ldq* < max(1, *n*).
iwork	INTEGER. Workspace array, DIMENSION (5**n*).

Output Parameters

ab	On exit, the contents of *ab* are overwritten.
bb	On exit, contains the factor *S* from the split Cholesky factorization $B = S^T*S$, as returned by spbstf/dpbstf.
m	INTEGER. The total number of eigenvalues found, 0 ≤ *m* ≤ *n*. If *range* = 'A', *m* = *n*, and if *range* = 'I',

	m = *iu*-*il*+1.
w, *z*, *q*	REAL for ssbgvx DOUBLE PRECISION for dsbgvx Arrays: *w*(*), DIMENSION at least max(1, *n*) . If *info* = 0, contains the eigenvalues in ascending order. z(*ldz*,*) . The second dimension of *z* must be at least max(1, *n*). If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors , with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that $Z^T*B*Z = I$. If *jobz* = 'N', then *z* is not referenced. q(*ldq*,*) . The second dimension of *q* must be at least max(1, *n*). If *jobz* = 'V', then *q* contains the *n*-by-*n* matrix used in the reduction of *Ax* = *lambda*Bx* to standard form, that is, *Cx* = *lambda*x* and consequently *C* to tridiagonal form. If *jobz* = 'N', then *q* is not referenced.
ifail	INTEGER. Array, DIMENSION (*m*). If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge. If *jobz* = 'N', then *ifail* is not referenced.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th argument had an illegal value. If *info* > 0, and if *i* ≤ *n*, the algorithm failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero; if *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then spbstf/dpbstf returned *info* = *i* and *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine sbgvx interface are the following:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array *A* of size (*ka+1*,*n*).
b	Stands for argument *bb* in Fortan 77 interface. Holds the array *B* of size (*kb+1*,*n*).
w	Holds the vector of length (*n*).
z	Holds the matrix *Z* of size (*n*, *n*).
ifail	Holds the vector of length (*n*).
q	Holds the matrix *Q* of size (*n*, *n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
vl	Default value for this element is *vl* = -HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted. Note that there will be an error condition if *ifail* or *q* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol* + ε * max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*||*T*||$_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

?hbgvx

Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem with banded matrices.

Syntax

Fortran 77:

```
call chbgvx(jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q, ldq, vl, vu,
il, iu, abstol, m, w, z, ldz, work, rwork, iwork, ifail, info)
```

```
call zhbgvx(jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q, ldq, vl, vu,
il, iu, abstol, m, w, z, ldz, work, rwork, iwork, ifail, info)
```

Fortran 95:

```
call hbgvx(a, b, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,q]
[,abstol] [,info])
```

Description

This routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite banded eigenproblem, of the form $Ax = \lambda Bx$. Here *A* and *B* are assumed to be Hermitian and banded, and *B* is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either all eigenvalues, a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz	CHARACTER*1. Must be 'N' or 'V'. If *jobz* = 'N', then compute eigenvalues only. If *jobz* = 'V', then compute eigenvalues and eigenvectors.
range	CHARACTER*1. Must be 'A' or 'V' or 'I'. If *range* = 'A', the routine computes all eigenvalues. If *range* = 'V', the routine computes eigenvalues *lambda*(*i*) in the half-open interval: *vl*< *lambda*(*i*) ≤ *vu*. If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.
uplo	CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', arrays *ab* and *bb* store the upper triangles of *A* and *B*; If *uplo* = 'L', arrays *ab* and *bb* store the lower triangles of *A* and *B*.
n	INTEGER. The order of the matrices *A* and *B* (*n* ≥ 0).
ka	INTEGER. The number of super- or sub-diagonals in *A* (*ka* ≥ 0).
kb	INTEGER. The number of super- or sub-diagonals in *B* (*kb* ≥ 0).
ab, *bb*, *work*	COMPLEX for chbgvx DOUBLE COMPLEX for zhbgvx Arrays: *ab* (*ldab*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by *uplo*) in band storage format. The second dimension of the array *ab* must be at least max(1, *n*). *bb*(*ldbb*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *B* (as specified by *uplo*) in band storage format. The second dimension of the array *bb* must be at least max(1, *n*).

work(*) is a workspace array, DIMENSION at least max(1, *n*).

ldab — INTEGER. The first dimension of the array *ab*; must be at least *ka*+1.

ldbb — INTEGER. The first dimension of the array *bb*; must be at least *kb*+1.

vl, *vu* — REAL for chbgvx
DOUBLE PRECISION for zhbgvx.
If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
Constraint: *vl*< *vu*.
If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu* — INTEGER.
If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: 1 ≤ *il* ≤ *iu* ≤ *n*, if *n* > 0; *il*=1 and *iu*=0 if *n* = 0.
If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol — REAL for chbgvx
DOUBLE PRECISION for zhbgvx.
The absolute error tolerance for the eigenvalues. See Application Notes for more information.

ldz — INTEGER. The leading dimension of the output array *z*; *ldz* ≥ 1. If *jobz* = 'V', *ldz* ≥ max(1, *n*).

ldq — INTEGER. The leading dimension of the output array *q*; *ldq* ≥ 1. If *jobz* = 'V', *ldq* ≥ max(1, *n*).

rwork — REAL for chbgvx
DOUBLE PRECISION for zhbgvx.
Workspace array, DIMENSION at least max(1, 7*n*).

iwork — INTEGER.
Workspace array, DIMENSION at least max(1, 5*n*).

Output Parameters

ab — On exit, the contents of *ab* are overwritten.

bb — On exit, contains the factor *S* from the split Cholesky factorization $B = S^H * S$, as returned by cpbstf/zpbstf.

m — INTEGER. The total number of eigenvalues found, $0 \le m \le n$. If *range* = 'A', $m = n$, and if *range* = 'I', $m = iu - il + 1$.

w — REAL for chbgvx
DOUBLE PRECISION for zhbgvx.
Array *w*(*), DIMENSION at least max(1, *n*).
If *info* = 0, contains the eigenvalues in ascending order.

z, *q* — COMPLEX for chbgvx
DOUBLE COMPLEX for zhbgvx
Arrays:
z(*ldz*,*).
The second dimension of *z* must be at least max(1, *n*).
If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that $Z^H * B * Z = I$.
If *jobz* = 'N', then *z* is not referenced.
q(*ldq*,*).
The second dimension of *q* must be at least max(1, *n*).
If *jobz* = 'V', then *q* contains the *n*-by-*n* matrix used in the reduction of $Ax = \lambda Bx$ to standard form, that is, $Cx = \lambda x$ and consequently *C* to tridiagonal form.
If *jobz* = 'N', then *q* is not referenced.

ifail — INTEGER.
Array, DIMENSION at least max(1, *n*).
If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.
If *jobz* = 'N', then *ifail* is not referenced.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th argument had an illegal value.
If *info* > 0, and

if $i \leq n$, the algorithm failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if $info = n + i$, for $1 \leq i \leq n$, then cpbstf/zpbstf returned $info = i$ and B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine hbgvx interface are the following:

a	Stands for argument *ab* in Fortan 77 interface. Holds the array A of size (*ka+1*,*n*).
b	Stands for argument *bb* in Fortan 77 interface. Holds the array B of size (*kb+1*,*n*).
w	Holds the vector of length (*n*).
z	Holds the matrix Z of size (*n*, *n*).
ifail	Holds the vector of length (*n*).
q	Holds the matrix Q of size (*n*, *n*).
uplo	Must be 'U' or 'L'. The default value is 'U'.
vl	Default value for this element is *vl* = -HUGE(*vl*).
vu	Default value for this element is *vu* = HUGE(*vl*).
il	Default value for this argument is *il* = 1.
iu	Default value for this argument is *iu* = *n*.
abstol	Default value for this element is *abstol* = 0.0_WP.
jobz	Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted. Note that there will be an error condition if *ifail* or *q* is present and *z* is omitted.
range	Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows:

range = 'V', if one of or both *vl* and *vu* are present,
range = 'I', if one of or both *il* and *iu* are present,
range = 'A', if none of *vl*, *vu*, *il*, *iu* is present,
Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol* + ε * max(|a|,|b|), where ε is the machine precision.

If *abstol* is less than or equal to zero, then ε*||*T*||$_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to 2*?lamch('S').

Generalized Nonsymmetric Eigenproblems

This section describes LAPACK driver routines used for solving generalized nonsymmetric eigenproblems. See also computational routines computational routines that can be called to solve these problems. Table 4-14 lists all such driver routines for Fortran-77 interface. Respective routine names in Fortran-95 interface are without the first symbol (see Routine Naming Conventions).

Table 4-14 Driver Routines for Solving Generalized Nonsymmetric Eigenproblems

Routine Name	Operation performed
?gges	Computes the generalized eigenvalues, Schur form, and the left and/or right Schur vectors for a pair of nonsymmetric matrices.
?ggesx	Computes the generalized eigenvalues, Schur form, and, optionally, the left and/or right matrices of Schur vectors.
?ggev	Computes the generalized eigenvalues, and the left and/or right generalized eigenvectors for a pair of nonsymmetric matrices.
?ggevx	Computes the generalized eigenvalues, and, optionally, the left and/or right generalized eigenvectors.

?gges

Computes the generalized eigenvalues, Schur form, and the left and/or right Schur vectors for a pair of nonsymmetric matrices.

Syntax

Fortran 77:

```
call sgges(jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alphar,
alphai, beta, vsl, ldvsl, vsr, ldvsr, work, lwork, bwork, info)

call dgges(jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alphar,
alphai, beta, vsl, ldvsl, vsr, ldvsr, work, lwork, bwork, info)

call cgges(jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alpha, beta,
vsl, ldvsl, vsr, ldvsr, work, lwork, rwork, bwork, info)

call zgges(jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alpha, beta,
vsl, ldvsl, vsr, ldvsr, work, lwork, rwork, bwork, info)
```

Fortran 95:

```
call gges(a, b, alphar, alphai, beta [,vsl] [,vsr] [,select] [,sdim] [,info])

call gges(a, b, alpha, beta [, vsl] [,vsr] [,select] [,sdim] [,info])
```

Description

This routine computes for a pair of n-by-n real/complex nonsymmetric matrices (A,B), the generalized eigenvalues, the generalized real/complex Schur form (S,T), optionally, the left and/or right matrices of Schur vectors (vsl and vsr). This gives the generalized Schur factorization

$$(A,B) = (vsl*S*vsr^H, vsl*T*vsr^H)$$

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix S and the upper triangular matrix T. The leading columns of vsl and vsr then form an orthonormal/unitary basis for the corresponding left and right eigenspaces (deflating subspaces).

(If only the generalized eigenvalues are needed, use the driver ?ggev instead, which is faster.)

A generalized eigenvalue for a pair of matrices (*A*,*B*) is a scalar *w* or a ratio *alpha* / *beta* = *w*, such that *A* - *w***B* is singular. It is usually represented as the pair (*alpha*, *beta*), as there is a reasonable interpretation for *beta*=0 or for both being zero. A pair of matrices (*S*,*T*) is in generalized real Schur form if *T* is upper triangular with non-negative diagonal and *S* is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of *S* will be "standardized" by making the corresponding elements of *T* have the form:

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$$

and the pair of corresponding 2-by-2 blocks in *S* and *T* will have a complex conjugate pair of generalized eigenvalues. A pair of matrices (*S*,*T*) is in generalized complex Schur form if *S* and *T* are upper triangular and, in addition, the diagonal of *T* are non-negative real numbers.

Input Parameters

jobvsl	CHARACTER*1. Must be 'N' or 'V'. If *jobvsl* = 'N', then the left Schur vectors are not computed. If *jobvsl* = 'V', then the left Schur vectors are computed.
jobvsr	CHARACTER*1. Must be 'N' or 'V'. If *jobvsr* = 'N', then the right Schur vectors are not computed. If *jobvsr* = 'V', then the right Schur vectors are computed.
sort	CHARACTER*1. Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form. If *sort* = 'N', then eigenvalues are not ordered. If *sort* = 'S', eigenvalues are ordered (see *selctg*).
selctg	LOGICAL FUNCTION of three REAL arguments for real flavors. LOGICAL FUNCTION of two COMPLEX arguments for complex flavors. *selctg* must be declared EXTERNAL in the calling subroutine.

If *sort* = 'S', *selctg* is used to select eigenvalues to sort to the top left of the Schur form.
If *sort* = 'N', *selctg* is not referenced.
For real flavors:
An eigenvalue (*alphar*(j) + *alphai*(j))/*beta*(j) is selected if *selctg*(*alphar*(j), *alphai*(j), *beta*(j)) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.
Note that in the ill-conditioned case, a selected complex eigenvalue may no longer satisfy *selctg*(*alphar*(j), *alphai*(j), *beta*(j)) = .TRUE. after ordering. In this case *info* is set to *n*+2 .
For complex flavors:
An eigenvalue *alpha*(j) / *beta*(j) is selected if *selctg*(*alpha*(j), *beta*(j)) is true.
Note that a selected complex eigenvalue may no longer satisfy *selctg*(*alpha*(j), *beta*(j)) = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case *info* is set to *n*+2 (see *info* below).

n — INTEGER. The order of the matrices *A*, *B*, *vsl*, and *vsr* (*n* ≥ 0).

a, *b*, *work* — REAL for sgges
DOUBLE PRECISION for dgges
COMPLEX for cgges
DOUBLE COMPLEX for zgges.
Arrays:
a(*lda*,*) is an array containing the *n*-by-*n* matrix *A* (first of the pair of matrices).
The second dimension of *a* must be at least max(1, *n*).
b(*ldb*,*) is an array containing the *n*-by-*n* matrix *B* (second of the pair of matrices).
The second dimension of *b* must be at least max(1, *n*).
work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The first dimension of the array *a*. Must be at least max(1, *n*).

ldb	INTEGER. The first dimension of the array *b*. Must be at least max(1, *n*).
ldvsl, *ldvsr*	INTEGER. The first dimensions of the output matrices *vsl* and *vsr*, respectively. Constraints: *ldvsl* ≥ 1. If *jobvsl* = 'V', *ldvsl* ≥ max(1, *n*). *ldvsr* ≥ 1. If *jobvsr* = 'V', *ldvsr* ≥ max(1, *n*).
lwork	INTEGER. The dimension of the array *work*. *lwork* ≥ max(1, 8*n*+16) for real flavors; *lwork* ≥ max(1, 2*n*) for complex flavors. For good performance, *lwork* must generally be larger. If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.
rwork	REAL for cgges DOUBLE PRECISION for zgges Workspace array, DIMENSION at least max(1, 8*n*). This array is used in complex flavors only.
bwork	LOGICAL. Workspace array, DIMENSION at least max(1, *n*). Not referenced if *sort* = 'N'.

Output Parameters

a	On exit, this array has been overwritten by its generalized Schur form *S*.
b	On exit, this array has been overwritten by its generalized Schur form *T*.
sdim	INTEGER. If *sort* = 'N', *sdim*= 0. If *sort* = 'S', *sdim* is equal to the number of eigenvalues (after sorting) for which *selctg* is true. Note that for real flavors complex conjugate pairs for which *selctg* is true for either eigenvalue count as 2.
alphar, *alphai*	REAL for sgges;

DOUBLE PRECISION for dgges.
Arrays, DIMENSION at least max(1, *n*) each. Contain values that form generalized eigenvalues in real flavors.
See *beta*.

alpha
COMPLEX for cgges;
DOUBLE COMPLEX for zgges.
Array, DIMENSION at least max(1, *n*). Contain values that form generalized eigenvalues in complex flavors. See *beta*.

beta
REAL for sgges
DOUBLE PRECISION for dgges
COMPLEX for cgges
DOUBLE COMPLEX for zgges.
Array, DIMENSION at least max(1, *n*).
For real flavors:
On exit, (*alphar*(j) + *alphai*(j)*i)/*beta*(j), j=1,..., *n*, will be the generalized eigenvalues.
alphar(j) + *alphai*(j)*i and *beta*(j), j=1,..., *n* are the diagonals of the complex Schur form (*S*,*T*) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (*A*,*B*) were further reduced to triangular form using complex unitary transformations. If *alphai*(j) is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with *alphai*(j+1) negative.
For complex flavors:
On exit, *alpha*(j)/*beta*(j), j=1,..., *n*, will be the generalized eigenvalues. *alpha*(j), j=1,...,*n*, and *beta*(j), j=1,..., *n* are the diagonals of the complex Schur form (*S*,*T*) output by cgges/zgges. The *beta*(j) will be non-negative real.
See also Application Notes below.

vsl, *vsr*
REAL for sgges
DOUBLE PRECISION for dgges
COMPLEX for cgges
DOUBLE COMPLEX for zgges.
Arrays:
vsl(*ldvsl*,*), the second dimension of *vsl* must be at least max(1, *n*).

	If *jobvsl* = 'V', this array will contain the left Schur vectors. If *jobvsl* = 'N', *vsl* is not referenced. *vsr*(*ldvsr*,*), the second dimension of *vsr* must be at least max(1, *n*). If *jobvsr* = 'V', this array will contain the right Schur vectors. If *jobvsr* = 'N', *vsr* is not referenced.
work(1)	On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value. If *info* = *i*, and *i* ≤ *n*: the *QZ* iteration failed. (*A*, *B*) is not in Schur form, but *alphar*(j), *alphai*(j) (for real flavors), or *alpha*(j) (for complex flavors), and *beta*(j), j=*info*+1, . . . , *n* should be correct. *i* > *n*: errors that usually indicate LAPACK problems: *i* = *n*+1: other than *QZ* iteration failed in ?hgeqz; *i* = *n*+2: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the generalized Schur form no longer satisfy *selctg* = .TRUE.. This could also be caused due to scaling; *i* = *n*+3: reordering failed in ?tgsen.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine gges interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *n*).
alphar	Holds the vector of length (*n*). Used in real flavors only.
alphai	Holds the vector of length (*n*). Used in real flavors only.

alpha	Holds the vector of length (*n*). Used in complex flavors only.
beta	Holds the vector of length (*n*).
vsl	Holds the matrix *VSL* of size (*n*, *n*).
vsr	Holds the matrix *VSR* of size (*n*, *n*).
jobvsl	Restored based on the presence of the argument *vsl* as follows: *jobvsl* = 'V', if *vsl* is present, *jobvsl* = 'N', if *vsl* is omitted.
jobvsr	Restored based on the presence of the argument *vsr* as follows: *jobvsr* = 'V', if *vsr* is present, *jobvsr* = 'N', if *vsr* is omitted.
sort	Restored based on the presence of the argument *select* as follows: *sort* = 'S', if *select* is present, *sort* = 'N', if *select* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The quotients *alphar*(j)/*beta*(j) and *alphai*(j)/*beta*(j) may easily over- or underflow, and *beta*(j) may even be zero. Thus, you should avoid simply computing the ratio. However, *alphar* and *alphai* will be always less than and usually comparable with norm(*A*) in magnitude, and *beta* always less than and usually comparable with norm(*B*).

?ggesx

Computes the generalized eigenvalues, Schur form, and, optionally, the left and/or right matrices of Schur vectors.

Syntax

Fortran 77:

```
call sggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb, sdim,
alphar, alphai, beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv, work, lwork,
iwork, liwork, bwork, info)

call dggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb, sdim,
alphar, alphai, beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv, work, lwork,
iwork, liwork, bwork, info)

call cggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb, sdim,
alpha, beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv, work, lwork, rwork,
iwork, liwork, bwork, info)

call zggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb, sdim,
alpha, beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv, work, lwork, rwork,
iwork, liwork, bwork, info)
```

Fortran 95:

```
call ggesx(a, b, alphar, alphai, beta [,vsl] [,vsr] [,select] [,sdim] [,rconde]
[, rcondv] [,info])

call ggesx(a, b, alpha, beta [, vsl] [,vsr] [,select] [,sdim] [,rconde]
[,rcondv] [, info])
```

Description

This routine computes for a pair of n-by-n real/complex nonsymmetric matrices (A,B), the generalized eigenvalues, the generalized real/complex Schur form (S,T), optionally, the left and/or right matrices of Schur vectors (vsl and vsr). This gives the generalized Schur factorization

$$(A,B) = (vsl*S*vsr^{H}, vsl*T*vsr^{H})$$

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix *S* and the upper triangular matrix *T*; computes a reciprocal condition number for the average of the selected eigenvalues (*rconde*); and computes a reciprocal condition number for the right and left deflating subspaces corresponding to the selected eigenvalues (*rcondv*). The leading columns of *vsl* and *vsr* then form an orthonormal/unitary basis for the corresponding left and right eigenspaces (deflating subspaces).

A generalized eigenvalue for a pair of matrices (*A*,*B*) is a scalar *w* or a ratio *alpha / beta = w*, such that *A - w*B* is singular. It is usually represented as the pair (*alpha*, *beta*), as there is a reasonable interpretation for *beta*=0 or for both being zero. A pair of matrices (*S*,*T*) is in generalized real Schur form if *T* is upper triangular with non-negative diagonal and *S* is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of *S* will be "standardized" by making the corresponding elements of *T* have the form:

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$$

and the pair of corresponding 2-by-2 blocks in *S* and *T* will have a complex conjugate pair of generalized eigenvalues. A pair of matrices (*S*,*T*) is in generalized complex Schur form if *S* and *T* are upper triangular and, in addition, the diagonal of *T* are non-negative real numbers.

Input Parameters

jobvsl — CHARACTER*1. Must be 'N' or 'V'.
If *jobvsl* = 'N', then the left Schur vectors are not computed.
If *jobvsl* = 'V', then the left Schur vectors are computed.

jobvsr — CHARACTER*1. Must be 'N' or 'V'.
If *jobvsr* = 'N', then the right Schur vectors are not computed.
If *jobvsr* = 'V', then the right Schur vectors are computed.

sort CHARACTER*1. Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form.
If *sort* = 'N', then eigenvalues are not ordered.
If *sort* = 'S', eigenvalues are ordered (see *selctg*).

selctg LOGICAL FUNCTION of three REAL arguments for real flavors.
LOGICAL FUNCTION of two COMPLEX arguments for complex flavors.
selctg must be declared EXTERNAL in the calling subroutine.
If *sort* = 'S', *selctg* is used to select eigenvalues to sort to the top left of the Schur form.
If *sort* = 'N', *selctg* is not referenced.
For real flavors:
An eigenvalue (*alphar*(j) + *alphai*(j))/*beta*(j) is selected if *selctg*(*alphar*(j), *alphai*(j), *beta*(j)) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.
Note that in the ill-conditioned case, a selected complex eigenvalue may no longer satisfy *selctg*(*alphar*(j), *alphai*(j), *beta*(j)) = .TRUE. after ordering. In this case *info* is set to *n*+2.
For complex flavors:
An eigenvalue *alpha*(j) / *beta*(j) is selected if *selctg*(*alpha*(j), *beta*(j)) is true.
Note that a selected complex eigenvalue may no longer satisfy *selctg*(*alpha*(j), *beta*(j)) = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case *info* is set to *n*+2 (see *info* below).

sense CHARACTER*1. Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed.
If *sense* = 'N', none are computed;
If *sense* = 'E', computed for average of selected eigenvalues only;
If *sense* = 'V', computed for selected deflating subspaces only;
If *sense* = 'B', computed for both.

If *sense* is 'E', 'V', or 'B', then *sort* must equal 'S'.

n INTEGER. The order of the matrices *A*, *B*, *vsl*, and *vsr* (*n* ≥ 0).

a, *b*, *work* REAL for sggesx
DOUBLE PRECISION for dggesx
COMPLEX for cggesx
DOUBLE COMPLEX for zggesx.
Arrays:
a(*lda*,*) is an array containing the *n*-by-*n* matrix *A* (first of the pair of matrices).
The second dimension of *a* must be at least max(1, *n*).
b(*ldb*,*) is an array containing the *n*-by-*n* matrix *B* (second of the pair of matrices).
The second dimension of *b* must be at least max(1, *n*).
work is a workspace array, its dimension max(1, *lwork*).

lda INTEGER. The first dimension of the array *a*.
Must be at least max(1, *n*).

ldb INTEGER. The first dimension of the array *b*.
Must be at least max(1, *n*).

ldvsl, *ldvsr* INTEGER. The first dimensions of the output matrices *vsl* and *vsr*, respectively. Constraints:

ldvsl ≥ 1. If *jobvsl* = 'V', *ldvsl* ≥ max(1, *n*).

ldvsr ≥ 1. If *jobvsr* = 'V', *ldvsr* ≥ max(1, *n*).

lwork INTEGER.
The dimension of the array *work*.
For real flavors:

If *n*=0 then *lwork*≥1.

If *n*>0 and *sense* = 'N', then *lwork* ≥ max(8*n, 6*n+16).

If *n*>0 and *sense* = 'E', 'V', or 'B', then *lwork* ≥ max(8*n, 6*n+16, 2*sdim*(n-sdim));
For complex flavors:

If *n*=0 then *lwork*≥1.

If *n*>0 and *sense* = 'N', then *lwork* ≥ max(1, 2*n);

If $n>0$ and *sense* = 'E', 'V', or 'B', then *lwork* ≥ max(1, 2**n*, 2**sdim**(*n*-*sdim*)).

Note that 2**sdim**(*n*-*sdim*) ≤ *n***n*/2.

An error is only returned if *lwork* < max(8**n*, 6**n*+16) for real flavors, and *lwork* < max(1, 2**n*) for complex flavors, but if *sense* = 'E', 'V', or 'B', this may not be large enough.

If *lwork*=-1, then a workspace query is assumed; the routine only calculates the bound on the optimal size of the *work* array and the minimum size of the *iwork* array, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla.

rwork

REAL for cggesx

DOUBLE PRECISION for zggesx

Workspace array, DIMENSION at least max(1, 8*n*).

This array is used in complex flavors only.

iwork

INTEGER.

Workspace array, DIMENSION max(1, *liwork*).

liwork

INTEGER.

The dimension of the array *iwork*.

If *sense* = 'N', or *n*=0, then *liwork*≥1,

otherwise *liwork* ≥ (*n*+6) for real flavors, and *liwork* ≥ (*n*+2) for complex flavors.

If *liwork*=-1, then a workspace query is assumed; the routine only calculates the bound on the optimal size of the *work* array and the minimum size of the *iwork* array, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla.

bwork

LOGICAL.

Workspace array, DIMENSION at least max(1, *n*).

Not referenced if *sort* = 'N'.

Output Parameters

a — On exit, this array has been overwritten by its generalized Schur form *S*.

b — On exit, this array has been overwritten by its generalized Schur form *T*.

sdim — INTEGER.
If *sort* = 'N', *sdim*= 0.
If *sort* = 'S', *sdim* is equal to the number of eigenvalues (after sorting) for which *selctg* is true.
Note that for real flavors complex conjugate pairs for which *selctg* is true for either eigenvalue count as 2.

alphar, *alphai* — REAL for sggesx;
DOUBLE PRECISION for dggesx.
Arrays, DIMENSION at least max(1, *n*) each. Contain values that form generalized eigenvalues in real flavors.
See *beta*.

alpha — COMPLEX for cggesx;
DOUBLE COMPLEX for zggesx.
Array, DIMENSION at least max(1, *n*). Contain values that form generalized eigenvalues in complex flavors. See *beta*.

beta — REAL for sggesx
DOUBLE PRECISION for dggesx
COMPLEX for cggesx
DOUBLE COMPLEX for zggesx.
Array, DIMENSION at least max(1, *n*).
For real flavors:
On exit, (*alphar*(j) + *alphai*(j)*i)/*beta*(j), j=1,..., *n* will be the generalized eigenvalues.
alphar(j) + *alphai*(j)*i and *beta*(j), j=1,..., *n* are the diagonals of the complex Schur form (*S*,*T*) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (*A*,*B*) were further reduced to triangular form using complex unitary transformations. If *alphai*(j) is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with *alphai*(j+1) negative.
For complex flavors:

	On exit, *alpha*(j)/*beta*(j), j=1,..., *n* will be the generalized eigenvalues. *alpha*(j), j=1,..., *n*, and *beta*(j), j=1,...,*n* are the diagonals of the complex Schur form (*S*,*T*) output by cggesx/zggesx. The *beta*(j) will be non-negative real. See also Application Notes below.
vsl, *vsr*	REAL for sggesx DOUBLE PRECISION for dggesx COMPLEX for cggesx DOUBLE COMPLEX for zggesx. Arrays: *vsl*(*ldvsl*,*), the second dimension of *vsl* must be at least max(1, *n*). If *jobvsl* = 'V', this array will contain the left Schur vectors. If *jobvsl* = 'N', *vsl* is not referenced. *vsr*(*ldvsr*,*), the second dimension of *vsr* must be at least max(1, *n*). If *jobvsr* = 'V', this array will contain the right Schur vectors. If *jobvsr* = 'N', *vsr* is not referenced.
rconde, *rcondv*	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION (2) each If *sense* = 'E' or 'B', *rconde*(1) and *rconde*(2) contain the reciprocal condition numbers for the average of the selected eigenvalues. Not referenced if *sense* = 'N' or 'V'. If *sense* = 'V' or 'B', *rcondv*(1) and *rcondv*(2) contain the reciprocal condition numbers for the selected deflating subspaces. Not referenced if *sense* = 'N' or 'E'.
work(1)	On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.
iwork(1)	On exit, if *info* = 0, then *iwork(1)* returns the required minimal size of *liwork*.
info	INTEGER. If *info* = 0, the execution is successful.

If *info* = *-i*, the *i*th parameter had an illegal value.
If *info* = *i*, and

i ≤ *n*:
the *QZ* iteration failed. (*A*, *B*) is not in Schur form, but *alphar*(j), *alphai*(j) (for real flavors), or *alpha*(j) (for complex flavors), and *beta*(j), j=*info*+1,..., *n* should be correct.
i > *n*: errors that usually indicate LAPACK problems:
i = *n*+1: other than *QZ* iteration failed in ?hgeqz;
i = *n*+2: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the generalized Schur form no longer satisfy *selctg* = .TRUE.. This could also be caused due to scaling;
i = *n*+3: reordering failed in ?tgsen.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ggesx interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *n*).
alphar	Holds the vector of length (*n*). Used in real flavors only.
alphai	Holds the vector of length (*n*). Used in real flavors only.
alpha	Holds the vector of length (*n*). Used in complex flavors only.
beta	Holds the vector of length (*n*).
vsl	Holds the matrix *VSL* of size (*n*, *n*).
vsr	Holds the matrix *VSR* of size (*n*, *n*).
rconde	Holds the vector of length (2).
rcondv	Holds the vector of length (2).
jobvsl	Restored based on the presence of the argument *vsl* as follows: *jobvsl* = 'V', if *vsl* is present, *jobvsl* = 'N', if *vsl* is omitted.
jobvsr	Restored based on the presence of the argument *vsr* as follows:

	jobvsr = 'V', if *vsr* is present, *jobvsr* = 'N', if *vsr* is omitted.
sort	Restored based on the presence of the argument *select* as follows: *sort* = 'S', if *select* is present, *sort* = 'N', if *select* is omitted.
sense	Restored based on the presence of arguments *rconde* and rcondv as follows: *sense* = 'B', if both *rconde* and *rcondv* are present, *sense* = 'E', if *rconde* is present and *rcondv* omitted, *sense* = 'V', if *rconde* is omitted and *rcondv* present, *sense* = 'N', if both *rconde* and *rcondv* are omitted.

Note that there will be an error condition if *rconde* or *rcondv* are present and *select* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = –1 (*liwork* = –1).

If you choose the first option and set any of admissible *lwork* (or *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The quotients *alphar*(j)/*beta*(j) and *alphai*(j)/*beta*(j) may easily over- or underflow, and *beta*(j) may even be zero. Thus, you should avoid simply computing the ratio. However, *alphar* and *alphai* will be always less than and usually comparable with norm(*A*) in magnitude, and *beta* always less than and usually comparable with norm(*B*).

?ggev

Computes the generalized eigenvalues, and the left and/or right generalized eigenvectors for a pair of nonsymmetric matrices.

Syntax

Fortran 77:

```
call sggev(jobvl, jobvr, n, a, lda, b, ldb, alphar, alphai, beta, vl, ldvl, vr, ldvr, work, lwork, info)
call dggev(jobvl, jobvr, n, a, lda, b, ldb, alphar, alphai, beta, vl, ldvl, vr, ldvr, work, lwork, info)
call cggev(jobvl, jobvr, n, a, lda, b, ldb, alpha, beta, vl, ldvl, vr, ldvr, work, lwork, rwork, info)
call zggev(jobvl, jobvr, n, a, lda, b, ldb, alpha, beta, vl, ldvl, vr, ldvr, work, lwork, rwork, info)
```

Fortran 95:

```
call ggev(a, b, alphar, alphai, beta [,vl] [,vr] [,info])
call ggev(a, b, alpha, beta [, vl] [,vr] [,info])
```

Description

This routine computes for a pair of n-by-n real/complex nonsymmetric matrices (A,B), the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors.

A generalized eigenvalue for a pair of matrices (A,B) is a scalar λ or a ratio $alpha / beta = \lambda$, such that $A - \lambda * B$ is singular. It is usually represented as the pair $(alpha, beta)$, as there is a reasonable interpretation for $beta = 0$ and even for both being zero. The right generalized eigenvector $v(j)$ corresponding to the generalized eigenvalue $l(j)$ of (A,B) satisfies

$$A*v*(j) = \lambda(j)*B*v*(j).$$

The left generalized eigenvector $u(j)$ corresponding to the generalized eigenvalue $\lambda(j)$ of (A,B) satisfies

$$u(j)^H*A = \lambda(j)*u*(j)^H*B$$

where $u(j)^H$ denotes the conjugate transpose of $u(j)$.

Input Parameters

jobvl	CHARACTER*1. Must be 'N' or 'V'. If *jobvl* = 'N', the left generalized eigenvectors are not computed; If *jobvl* = 'V', the left generalized eigenvectors are computed.
jobvr	CHARACTER*1. Must be 'N' or 'V'. If *jobvr* = 'N', the right generalized eigenvectors are not computed; If *jobvr* = 'V', the right generalized eigenvectors are computed.
n	INTEGER. The order of the matrices *A*, *B*, *vl*, and *vr* (*n* ≥ 0).
a, *b*, *work*	REAL for sggev DOUBLE PRECISION for dggev COMPLEX for cggev DOUBLE COMPLEX for zggev. Arrays: *a*(*lda*,*) is an array containing the *n*-by-*n* matrix *A* (first of the pair of matrices). The second dimension of *a* must be at least max(1, *n*). *b*(*ldb*,*) is an array containing the *n*-by-*n* matrix *B* (second of the pair of matrices). The second dimension of *b* must be at least max(1, *n*). *work* is a workspace array, its dimension max(1, *lwork*).
lda	INTEGER. The first dimension of the array *a*. Must be at least max(1, *n*).
ldb	INTEGER. The first dimension of the array *b*. Must be at least max(1, *n*).
ldvl, *ldvr*	INTEGER. The first dimensions of the output matrices *vl* and *vr*, respectively. Constraints: *ldvl* ≥ 1. If *jobvl* = 'V', *ldvl* ≥ max(1, *n*). *ldvr* ≥ 1. If *jobvr* = 'V', *ldvr* ≥ max(1, *n*).

lwork INTEGER.
The dimension of the array *work*.
lwork ≥ max(1, 8*n*+16) for real flavors;
lwork ≥ max(1, 2*n*) for complex flavors.
For good performance, *lwork* must generally be larger.
If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

rwork REAL for cggev
DOUBLE PRECISION for zggev
Workspace array, DIMENSION at least max(1, 8*n*).
This array is used in complex flavors only.

Output Parameters

a, *b* On exit, these arrays have been overwritten.

alphar, *alphai* REAL for sggev;
DOUBLE PRECISION for dggev.
Arrays, DIMENSION at least max(1, *n*) each. Contain values that form generalized eigenvalues in real flavors.
See *beta*.

alpha COMPLEX for cggev;
DOUBLE COMPLEX for zggev.
Array, DIMENSION at least max(1, *n*). Contain values that form generalized eigenvalues in complex flavors. See *beta*.

beta REAL for sggev
DOUBLE PRECISION for dggev
COMPLEX for cggev
DOUBLE COMPLEX for zggev.
Array, DIMENSION at least max(1, *n*).
For real flavors:
On exit, (*alphar*(j)+ *alphai*(j)*i)/*beta*(j), j=1,..., *n*, are the generalized eigenvalues.
If *alphai*(j) is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with *alphai*(j+1) negative.

For complex flavors:
On exit, *alpha*(j)/*beta*(j), j=1,..., *n*, are the generalized eigenvalues.
See also Application Notes below.

vl, *vr*

REAL for sggev
DOUBLE PRECISION for dggev
COMPLEX for cggev
DOUBLE COMPLEX for zggev.
Arrays:
vl(*ldvl*,*); the second dimension of *vl* must be at least max(1, *n*).
If *jobvl* = 'V', the left generalized eigenvectors *u*(j) are stored one after another in the columns of *vl*, in the same order as their eigenvalues. Each eigenvector is scaled so the largest component has abs(Re) + abs(Im) = 1.
If *jobvl* = 'N', *vl* is not referenced.
For real flavors:
If the j-th eigenvalue is real, then *u*(j) = *vl*(:,j), the j-th column of *vl*.
If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then *u*(j) = *vl*(:,j) + *i**vl*(:,j+1) and *u*(j+1) = *vl*(:,j) – *i**vl*(:,j+1), where *i* = sqrt(–1).
For complex flavors:
u(j) = *vl*(:,j), the j-th column of *vl*.
vr(*ldvr*,*); the second dimension of *vr* must be at least max(1, *n*).
If *jobvr* = 'V', the right generalized eigenvectors *v*(j) are stored one after another in the columns of *vr*, in the same order as their eigenvalues. Each eigenvector is scaled so the largest component has abs(Re) + abs(Im) = 1.
If *jobvr* = 'N', *vr* is not referenced.
For real flavors:
If the j-th eigenvalue is real, then *v*(j) = *vr*(:,j), the j-th column of *vr*.
If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then *v*(j) = *vr*(:,j) + *i**vr*(:,j+1) and *v*(j+1) = *vr*(:,j) – *i**vr*(:,j+1).

	For complex flavors: `v(j) = vr(:,j)`, the j-th column of `vr`.
`work(1)`	On exit, if `info = 0`, then `work(1)` returns the required minimal size of `lwork`.
`info`	INTEGER. If `info = 0`, the execution is successful. If `info = -i`, the `i`-th parameter had an illegal value. If `info = i`, and `i ≤ n`: the `QZ` iteration failed. No eigenvectors have been calculated, but `alphar`(j), `alphai`(j) (for real flavors), or `alpha`(j) (for complex flavors), and `beta`(j), `j=info+1,..., n` should be correct. `i > n`: errors that usually indicate LAPACK problems: `i = n+1`: other than `QZ` iteration failed in `?hgeqz`; `i = n+2`: error return from `?tgevc`.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine `ggev` interface are the following:

`a`	Holds the matrix `A` of size (`n`, `n`).
`b`	Holds the matrix `B` of size (`n`, `n`).
`alphar`	Holds the vector of length (`n`). Used in real flavors only.
`alphai`	Holds the vector of length (`n`). Used in real flavors only.
`alpha`	Holds the vector of length (`n`). Used in complex flavors only.
`beta`	Holds the vector of length (`n`).
`vl`	Holds the matrix `VL` of size (`n`, `n`).
`vr`	Holds the matrix `VR` of size (`n`, `n`).
`jobvl`	Restored based on the presence of the argument `vl` as follows: `jobvl = 'V'`, if `vl` is present, `jobvl = 'N'`, if `vl` is omitted.
`jobvr`	Restored based on the presence of the argument `vr` as follows:

jobvr = 'V', if *vr* is present,
jobvr = 'N', if *vr* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The quotients *alphar*(j)/*beta*(j) and *alphai*(j)/*beta*(j) may easily over- or underflow, and *beta*(j) may even be zero. Thus, you should avoid simply computing the ratio. However, *alphar* and *alphai* (for real flavors) or *alpha* (for complex flavors) will be always less than and usually comparable with norm(*A*) in magnitude, and *beta* always less than and usually comparable with norm(*B*).

?ggevx

Computes the generalized eigenvalues, and, optionally, the left and/or right generalized eigenvectors.

Syntax

Fortran 77:

```
call sggevx(balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alphar, alphai,
beta, vl, ldvl, vr, ldvr, ilo, ihi, lscale, rscale, abnrm, bbnrm, rconde,
rcondv, work, lwork, iwork, bwork, info)
```

```
call dggevx(balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alphar, alphai,
beta, vl, ldvl, vr, ldvr, ilo, ihi, lscale, rscale, abnrm, bbnrm, rconde,
rcondv, work, lwork, iwork, bwork, info)
```

```
call cggevx(balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alpha, beta, vl,
ldvl, vr, ldvr, ilo, ihi, lscale, rscale, abnrm, bbnrm, rconde, rcondv, work,
lwork, rwork, iwork, bwork, info)
```

```
call zggevx(balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alpha, beta, vl,
ldvl, vr, ldvr, ilo, ihi, lscale, rscale, abnrm, bbnrm, rconde, rcondv, work,
lwork, rwork, iwork, bwork, info)
```

Fortran 95:

```
call ggevx(a, b, alphar, alphai, beta [,vl] [,vr] [,balanc] [,ilo] [,ihi] [,
lscale] [,rscale] [,abnrm] [,bbnrm] [,rconde] [,rcondv] [,info])
```

```
call ggevx(a, b, alpha, beta [, vl] [,vr] [,balanc] [,ilo] [,ihi] [,lscale]
[, rscale] [,abnrm] [,bbnrm] [,rconde] [,rcondv] [,info])
```

Description

This routine computes for a pair of n-by-n real/complex nonsymmetric matrices (A,B), the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (*ilo*, *ihi*, *lscale*, *rscale*, *abnrm*, and *bbnrm*), reciprocal condition numbers for the eigenvalues (*rconde*), and reciprocal condition numbers for the right eigenvectors (*rcondv*).

A generalized eigenvalue for a pair of matrices (*A*,*B*) is a scalar λ or a ratio *alpha* / *beta* = λ, such that *A* – λ**B* is singular. It is usually represented as the pair (*alpha*, *beta*), as there is a reasonable interpretation for *beta*=0 and even for both being zero. The right generalized eigenvector *v*(j) corresponding to the generalized eigenvalue λ(j) of (*A*,*B*) satisfies

*A*v**(j) = λ(j)**B*v**(j).

The left generalized eigenvector *u*(j) corresponding to the generalized eigenvalue λ(j) of (*A*,*B*) satisfies

u(j)^{H}**A* = *l*(j)**u**(j)^{H}**B*

where *u*(j)^{H} denotes the conjugate transpose of *u*(j).

Input Parameters

balanc	CHARACTER*1. Must be 'N', 'P', 'S', or 'B'. Specifies the balance option to be performed. If *balanc* = 'N', do not diagonally scale or permute; If *balanc* = 'P', permute only; If *balanc* = 'S', scale only; If *balanc* = 'B', both permute and scale. Computed reciprocal condition numbers will be for the matrices after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.
jobvl	CHARACTER*1. Must be 'N' or 'V'. If *jobvl* = 'N', the left generalized eigenvectors are not computed; If *jobvl* = 'V', the left generalized eigenvectors are computed.
jobvr	CHARACTER*1. Must be 'N' or 'V'. If *jobvr* = 'N', the right generalized eigenvectors are not computed; If *jobvr* = 'V', the right generalized eigenvectors are computed.
sense	CHARACTER*1. Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed. If *sense* = 'N', none are computed;

If *sense* = 'E', computed for eigenvalues only;
If *sense* = 'V', computed for eigenvectors only;
If *sense* = 'B', computed for eigenvalues and eigenvectors.

n — INTEGER. The order of the matrices *A*, *B*, *vl*, and *vr* ($n \geq 0$).

a, *b*, *work* — REAL for sggevx
DOUBLE PRECISION for dggevx
COMPLEX for cggevx
DOUBLE COMPLEX for zggevx.
Arrays:
a(*lda*,*) is an array containing the *n*-by-*n* matrix *A* (first of the pair of matrices).
The second dimension of *a* must be at least max(1, *n*).
b(*ldb*,*) is an array containing the *n*-by-*n* matrix *B* (second of the pair of matrices).
The second dimension of *b* must be at least max(1, *n*).
work is a workspace array, its dimension max(1, *lwork*).

lda — INTEGER. The first dimension of the array *a*.
Must be at least max(1, *n*).

ldb — INTEGER. The first dimension of the array *b*.
Must be at least max(1, *n*).

ldvl, *ldvr* — INTEGER. The first dimensions of the output matrices *vl* and *vr*, respectively.
Constraints:

ldvl ≥ 1. If *jobvl* = 'V', *ldvl* ≥ max(1, *n*).

ldvr ≥ 1. If *jobvr* = 'V', *ldvr* ≥ max(1, *n*).

lwork — INTEGER.

The dimension of the array *work*. *lwork* ≥ max(1, 2**n*);
For real flavors:
If *balanc* = 'S', or 'B', or *jobvl* = 'V', or *jobvr* = 'V', then *lwork* ≥ max(1, 6**n*);

if *sense* = 'E', or 'B', then *lwork* ≥ max(1, 10**n*);

if *sense* = 'V', or 'B', *lwork* ≥ $(2n^2 + 8*n+16)$.
For complex flavors:

if *sense* = 'E', *lwork* ≥ max(1, 4*n);

if *sense* = 'V', or 'B', *lwork* ≥max(1, $2*n^2+ 2*n$).
If *lwork* = −1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

rwork — REAL for cggevx
DOUBLE PRECISION for zggevx
Workspace array, DIMENSION at least max(1, 6*n) if *balanc* = 'S', or 'B', and at least max(1, 2*n) otherwise.
This array is used in complex flavors only.

iwork — INTEGER.
Workspace array, DIMENSION at least (*n*+6) for real flavors and at least (*n*+2) for complex flavors.
Not referenced if *sense* = 'E'.

bwork — LOGICAL. Workspace array, DIMENSION at least max(1, *n*).
Not referenced if *sense* = 'N'.

Output Parameters

a, *b* — On exit, these arrays have been overwritten.
If *jobvl* = 'V' or *jobvr* = 'V' or both, then *a* contains the first part of the real Schur form of the "balanced" versions of the input *A* and *B*, and *b* contains its second part.

alphar, *alphai* — REAL for sggevx;
DOUBLE PRECISION for dggevx.
Arrays, DIMENSION at least max(1, *n*) each. Contain values that form generalized eigenvalues in real flavors.
See *beta*.

alpha — COMPLEX for cggevx;
DOUBLE COMPLEX for zggevx.
Array, DIMENSION at least max(1, *n*). Contain values that form generalized eigenvalues in complex flavors. See *beta*.

beta — REAL for sggevx
DOUBLE PRECISION for dggevx
COMPLEX for cggevx

DOUBLE COMPLEX for zggevx.
Array, DIMENSION at least max(1, *n*).
For real flavors:
On exit, (*alphar*(j) + *alphai*(j)*i)/*beta*(j), j=1,..., *n*, will be the generalized eigenvalues.
If *alphai*(j) is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with *alphai*(j+1) negative.
For complex flavors:
On exit, *alpha*(j)/*beta*(j), j=1,..., *n*, will be the generalized eigenvalues.
See also Application Notes below.

vl, *vr*

REAL for sggevx
DOUBLE PRECISION for dggevx
COMPLEX for cggevx
DOUBLE COMPLEX for zggevx.
Arrays:
vl(*ldvl*,*); the second dimension of *vl* must be at least max(1, *n*).
If *jobvl* = 'V', the left generalized eigenvectors *u*(j) are stored one after another in the columns of *vl*, in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component have abs(Re) + abs(Im) = 1.
If *jobvl* = 'N', *vl* is not referenced.
For real flavors:
If the j-th eigenvalue is real, then *u*(j) = *vl*(:,j), the j-th column of *vl*.
If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then *u*(j) = *vl*(:,j) + *i**vl*(:,j+1) and *u*(j+1) = *vl*(:,j) – *i**vl*(:,j+1), where *i* = sqrt(–1).
For complex flavors:
u(j) = *vl*(:,j), the j-th column of *vl*.
vr(*ldvr*,*); the second dimension of *vr* must be at least max(1, *n*).

If *jobvr* = 'V', the right generalized eigenvectors *v*(j) are stored one after another in the columns of *vr*, in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component have abs(Re) + abs(Im) = 1.
If *jobvr* = 'N', *vr* is not referenced.
For real flavors:
If the j-th eigenvalue is real, then *v*(j) = *vr*(:,j), the j-th column of *vr*.
If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then *v*(j) = *vr*(:,j) + *i***vr*(:,j+1) and *v*(j+1) = *vr*(:,j) – *i***vr*(:,j+1).
For complex flavors:
v(j) = *vr*(:,j), the j-th column of *vr*.

ilo, *ihi*

INTEGER. *ilo* and *ihi* are integer values such that on exit *A*(i,j) = 0 and *B*(i,j) = 0 if i > j and j = 1,..., *ilo*–1 or i = *ihi*+1,..., *n*.
If *balanc* = 'N' or 'S', *ilo* = 1 and *ihi* = *n*.

lscale, *rscale*

REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
Arrays, DIMENSION at least max(1, *n*) each.
lscale contains details of the permutations and scaling factors applied to the left side of *A* and *B*.
If *PL*(j) is the index of the row interchanged with row j, and *DL*(j) is the scaling factor applied to row j, then
lscale(j) = *PL*(j), for j = 1,..., *ilo*–1
= *DL*(j), for j = *ilo*,...,*ihi*
= *PL*(j) for j = *ihi*+1,..., *n*.
The order in which the interchanges are made is *n* to *ihi*+1, then 1 to *ilo*-1.
rscale contains details of the permutations and scaling factors applied to the right side of *A* and *B*.
If *PR*(j) is the index of the column interchanged with column j, and *DR*(j) is the scaling factor applied to column j, then
rscale(j) = *PR*(j), for j = 1,..., *ilo*–1
= *DR*(j), for j = *ilo*,...,*ihi*
= *PR*(j) for j = *ihi*+1,..., *n*.
The order in which the interchanges are made is *n* to *ihi*+1, then 1 to *ilo*-1.

abnrm, *bbnrm*
REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
The one-norms of the balanced matrices *A* and *B*, respectively.

rconde, *rcondv*
REAL for single precision flavors DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least max(1, *n*) each.
If *sense* = 'E', or 'B', *rconde* contains the reciprocal condition numbers of the eigenvalues, stored in consecutive elements of the array. For a complex conjugate pair of eigenvalues two consecutive elements of *rconde* are set to the same value. Thus *rconde*(j), *rcondv*(j), and the j-th columns of *vl* and *vr* all correspond to the same eigenpair (but not in general the j-th eigenpair, unless all eigenpairs are selected).
If *sense* = 'N', or 'V', *rconde* is not referenced.
If *sense* = 'V', or 'B', *rcondv* contains the estimated reciprocal condition numbers of the eigenvectors, stored in consecutive elements of the array. For a complex eigenvector two consecutive elements of *rcondv* are set to the same value.
If the eigenvalues cannot be reordered to compute *rcondv*(j), *rcondv*(j) is set to 0; this can only occur when the true value would be very small anyway.
If *sense* = 'N', or 'E', *rcondv* is not referenced.

work(1)
On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info
INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, and

i ≤ *n*:
the *QZ* iteration failed. No eigenvectors have been calculated, but *alphar*(j), *alphai*(j) (for real flavors), or *alpha*(j) (for complex flavors), and *beta*(j), j=*info*+1, ..., *n* should be correct.
i > *n*: errors that usually indicate LAPACK problems:

i = *n*+1: other than *QZ* iteration failed in ?hgeqz;
i = *n*+2: error return from ?tgevc.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their Fortran 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see Fortran-95 Interface Conventions.

Specific details for the routine ggevx interface are the following:

a	Holds the matrix *A* of size (*n*, *n*).
b	Holds the matrix *B* of size (*n*, *n*).
alphar	Holds the vector of length (*n*). Used in real flavors only.
alphai	Holds the vector of length (*n*). Used in real flavors only.
alpha	Holds the vector of length (*n*). Used in complex flavors only.
beta	Holds the vector of length (*n*).
vl	Holds the matrix *VL* of size (*n*, *n*).
vr	Holds the matrix *VR* of size (*n*, *n*).
lscale	Holds the vector of length (*n*).
rscale	Holds the vector of length (*n*).
rconde	Holds the vector of length (*n*).
rcondv	Holds the vector of length (*n*).
balanc	Must be 'N', 'B', or 'P'. The default value is 'N'.
jobvl	Restored based on the presence of the argument *vl* as follows: *jobvl* = 'V', if *vl* is present, *jobvl* = 'N', if *vl* is omitted.
jobvr	Restored based on the presence of the argument *vr* as follows: *jobvr* = 'V', if *vr* is present, *jobvr* = 'N', if *vr* is omitted.
sense	Restored based on the presence of arguments *rconde* and rcondv as follows: *sense* = 'B', if both *rconde* and *rcondv* are present, *sense* = 'E', if *rconde* is present and *rcondv* omitted, *sense* = 'V', if *rconde* is omitted and *rcondv* present, *sense* = 'N', if both *rconde* and *rcondv* are omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = –1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = –1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The quotients *alphar*(j)/*beta*(j) and *alphai*(j)/*beta*(j) may easily over- or underflow, and *beta*(j) may even be zero. Thus, you should avoid simply computing the ratio. However, *alphar* and *alphai* (for real flavors) or *alpha* (for complex flavors) will be always less than and usually comparable with norm(*A*) in magnitude, and *beta* always less than and usually comparable with norm(*B*).

LAPACK Auxiliary and Utility Routines

5

This chapter describes the Intel® Math Kernel Library implementation of LAPACK auxiliary and utility routines. The library includes auxiliary routines for both real and complex data.

Auxiliary Routines

Routine naming conventions, mathematical notation, and matrix storage schemes used for LAPACK auxiliary routines are the same as for the driver and computational routines described in previous chapters.

The table below summarizes information about the available LAPACK auxiliary routines.

Table 5-1 LAPACK Auxiliary Routines

Routine Name	Data Types	Description
?lacgv	c, z	Conjugates a complex vector.
?lacrm	c, z	Multiplies a complex matrix by a square real matrix.
?lacrt	c, z	Performs a linear transformation of a pair of complex vectors.
?laesy	c, z	Computes the eigenvalues and eigenvectors of a 2-by-2 complex symmetric matrix.
?rot	c, z	Applies a plane rotation with real cosine and complex sine to a pair of complex vectors.
?spmv	c, z	Computes a matrix-vector product for complex vectors using a complex symmetric packed matrix
?spr	c, z	Performs the symmetrical rank-1 update of a complex symmetric packed matrix.
?symv	c, z	Computes a matrix-vector product for a complex symmetric matrix.
?syr	c, z	Performs the symmetric rank-1 update of a complex symmetric matrix.

Routine Name	Data Types	Description
i?max1	c, z	Finds the index of the vector element whose real part has maximum absolute value.
?sum1	sc, dz	Forms the 1-norm of the complex vector using the true absolute value.
?gbtf2	s, d, c, z	Computes the LU factorization of a general band matrix using the unblocked version of the algorithm.
?gebd2	s, d, c, z	Reduces a general matrix to bidiagonal form using an unblocked algorithm.
?gehd2	s, d, c, z	Reduces a general square matrix to upper Hessenberg form using an unblocked algorithm.
?gelq2	s, d, c, z	Computes the LQ factorization of a general rectangular matrix using an unblocked algorithm.
?geql2	s, d, c, z	Computes the QL factorization of a general rectangular matrix using an unblocked algorithm.
?geqr2	s, d, c, z	Computes the QR factorization of a general rectangular matrix using an unblocked algorithm.
?gerq2	s, d, c, z	Computes the RQ factorization of a general rectangular matrix using an unblocked algorithm.
?gesc2	s, d, c, z	Solves a system of linear equations using the LU factorization with complete pivoting computed by ?getc2.
?getc2	s, d, c, z	Computes the LU factorization with complete pivoting of the general n-by-n matrix.
?getf2	s, d, c, z	Computes the LU factorization of a general *m*-by-*n* matrix using partial pivoting with row interchanges (unblocked algorithm).
?gtts2	s, d, c, z	Solves a system of linear equations with a tridiagonal matrix using the LU factorization computed by ?gttrf.
?isnan	s, d,	Tests input for NaN.

Routine Name	Data Types	Description
?laisnan	s, d,	Tests input for NaN by comparing itwo arguments for inequality.
?labrd	s, d, c, z	Reduces the first *nb* rows and columns of a general matrix to a bidiagonal form.
?lacn2	s, d, c, z	Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.
?lacon	s, d, c, z	Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.
?lacpy	s, d, c, z	Copies all or part of one two-dimensional array to another.
?ladiv	s, d, c, z	Performs complex division in real arithmetic, avoiding unnecessary overflow.
?lae2	s, d	Computes the eigenvalues of a 2-by-2 symmetric matrix.
?laebz	s, d	Computes the number of eigenvalues of a real symmetric tridiagonal matrix which are less than or equal to a given value, and performs other tasks required by the routine ?stebz.
?laed0	s, d, c, z	Used by ?stedc. Computes all eigenvalues and corresponding eigenvectors of an unreduced symmetric tridiagonal matrix using the divide and conquer method.
?laed1	s, d	Used by sstedc/dstedc. Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is tridiagonal.
?laed2	s, d	Used by sstedc/dstedc. Merges eigenvalues and deflates secular equation. Used when the original matrix is tridiagonal.
?laed3	s, d	Used by sstedc/dstedc. Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is tridiagonal.

Routine Name	Data Types	Description
?laed4	s, d	Used by sstedc/dstedc. Finds a single root of the secular equation.
?laed5	s, d	Used by sstedc/dstedc. Solves the 2-by-2 secular equation.
?laed6	s, d	Used by sstedc/dstedc. Computes one Newton step in solution of the secular equation.
?laed7	s, d, c, z	Used by ?stedc. Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is dense.
?laed8	s, d, c, z	Used by ?stedc. Merges eigenvalues and deflates secular equation. Used when the original matrix is dense.
?laed9	s, d	Used by sstedc/dstedc. Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is dense.
?laeda	s, d	Used by ?stedc. Computes the Z vector determining the rank-one modification of the diagonal matrix. Used when the original matrix is dense.
?laein	s, d, c, z	Computes a specified right or left eigenvector of an upper Hessenberg matrix by inverse iteration.
?laev2	s, d, c, z	Computes the eigenvalues and eigenvectors of a 2-by-2 symmetric/Hermitian matrix.
?laexc	s, d	Swaps adjacent diagonal blocks of a real upper quasi-triangular matrix in Schur canonical form, by an orthogonal similarity transformation.
?lag2	s, d	Computes the eigenvalues of a 2-by-2 generalized eigenvalue problem, with scaling as necessary to avoid over-/underflow.
?lags2	s, d	Computes 2-by-2 orthogonal matrices U, V, and Q, and applies them to matrices A and B such that the rows of the transformed A and B are parallel.

Routine Name	Data Types	Description
?lagtf	s, d	Computes an LU factorization of a matrix $T-\lambda I$, where T is a general tridiagonal matrix, and λ a scalar, using partial pivoting with row interchanges.
?lagtm	s, d, c, z	Performs a matrix-matrix product of the form $C = \alpha Ab+\beta C$, where A is a tridiagonal matrix, B and C are rectangular matrices, and α and β are scalars, which may be 0, 1, or -1.
?lagts	s, d	Solves the system of equations $(T-\lambda I)x = y$ or $(T-\lambda I)^T x = y$,where T is a general tridiagonal matrix and λ a scalar, using the LU factorization computed by ?lagtf.
?lagv2	s, d	Computes the Generalized Schur factorization of a real 2-by-2 matrix pencil (A,B) where B is upper triangular.
?lahqr	s, d, c, z	Computes the eigenvalues and Schur factorization of an upper Hessenberg matrix, using the double-shift/single-shift QR algorithm.
?lahrd	s, d, c, z	Reduces the first nb columns of a general rectangular matrix A so that elements below the k-th subdiagonal are zero, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A.
?lahr2	s, d, c, z	Reduces the specified number of first columns of a general rectangular matrix A so that elements below thespecified subdiagonal are zero, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A.
?laic1	s, d, c, z	Applies one step of incremental condition estimation.
?laln2	s, d	Solves a 1-by-1 or 2-by-2 linear system of equations of the specified form.
?lals0	s, d, c, z	Applies back multiplying factors in solving the least squares problem using divide and conquer SVD approach. Used by ?gelsd.

Routine Name	Data Types	Description
?lalsa	s, d, c, z	Computes the SVD of the coefficient matrix in compact form. Used by ?gelsd.
?lalsd	s, d, c, z	Uses the singular value decomposition of A to solve the least squares problem.
?lamrg	s, d	Creates a permutation list to merge the entries of two independently sorted sets into a single set sorted in ascending order.
?laneg	s, d	Computes the Sturm count.
?langb	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of general band matrix.
?lange	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general rectangular matrix.
?langt	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general tridiagonal matrix.
?lanhs	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of an upper Hessenberg matrix.
?lansb	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric band matrix.
?lanhb	c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hermitian band matrix.
?lansp	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric matrix supplied in packed form.

Routine Name	Data Types	Description
?lanhp	c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix supplied in packed form.
?lanst/?lanht	s, d/c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real symmetric or complex Hermitian tridiagonal matrix.
?lansy	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix.
?lanhe	c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix.
?lantb	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular band matrix.
?lantp	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular matrix supplied in packed form.
?lantr	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular matrix.
?lanv2	s, d	Computes the Schur factorization of a real 2-by-2 nonsymmetric matrix in standard form.
?lapll	s, d, c, z	Measures the linear dependence of two vectors.
?lapmt	s, d, c, z	Performs a forward or backward permutation of the columns of a matrix.
?lapy2	s, d	Returns sqrt(x^2+y^2).
?lapy3	s, d	Returns sqrt($x^2+y^2+z^2$).

Routine Name	Data Types	Description
?laqgb	s, d, c, z	Scales a general band matrix, using row and column scaling factors computed by ?gbequ.
?laqge	s, d, c, z	Scales a general rectangular matrix, using row and column scaling factors computed by ?geequ.
?laqhb	c, z	Scales a Hermetian band matrix, using scaling factors computed by ?pbequ.
?laqp2	s, d, c, z	Computes a QR factorization with column pivoting of the matrix block.
?laqps	s, d, c, z	Computes a step of QR factorization with column pivoting of a real m-by-n matrix *A* by using BLAS level 3.
?laqr0	s, d, c, z	Computes the eigenvalues of a Hessenberg matrix, and optionally the marixes from the Schur decomposition.
?laqr1	s, d, c, z	Sets a scalar multiple of the first column of the product of 2-by-2 or 3-by-3 matrix *H* and specified shifts.
?laqr2	s, d, c, z	Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggresive early deflation).
?laqr3	s, d, c, z	Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggresive early deflation).
?laqr4	s, d, c, z	Computes the eigenvalues of a Hessenberg matrix, and optionally the marices from the Schur decomposition.
?laqr5	s, d, c, z	Performs a single small-bulge multi-shift QR sweep.
?laqsb	s, d, c, z	Scales a symmetric/Hermitian band matrix, using scaling factors computed by ?pbequ.

Routine Name	Data Types	Description
?laqsp	s, d, c, z	Scales a symmetric/Hermitian matrix in packed storage, using scaling factors computed by ?ppequ.
?laqsy	s, d, c, z	Scales a symmetric/Hermitian matrix, using scaling factors computed by ?poequ.
?laqtr	s, d	Solves a real quasi-triangular system of equations, or a complex quasi-triangular system of special form, in real arithmetic.
?lar1v	s, d, c, z	Computes the (scaled) r-th column of the inverse of the submatrix in rows $b1$ through bn of the tridiagonal matrix $ldL^T - \sigma I$.
?lar2v	s, d, c, z	Applies a vector of plane rotations with real cosines and real/complex sines from both sides to a sequence of 2-by-2 symmetric/Hermitian matrices.
?larf	s, d, c, z	Applies an elementary reflector to a general rectangular matrix.
?larfb	s, d, c, z	Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.
?larfg	s, d, c, z	Generates an elementary reflector (Householder matrix).
?larft	s, d, c, z	Forms the triangular factor T of a block reflector $H = I - vtv^H$
?larfx	s, d, c, z	Applies an elementary reflector to a general rectangular matrix, with loop unrolling when the reflector has order ≤ 10.
?largv	s, d, c, z	Generates a vector of plane rotations with real cosines and real/complex sines.
?larnv	s, d, c, z	Returns a vector of random numbers from a uniform or normal distribution.
?larra	s, d	Computes the splitting points with the specified threshold.

Routine Name	Data Types	Description
?larrb	s, d	Provides limited bisection to locate eigenvalues for more accuracy.
?larrc	s, d	Computes the number of eigenvalues of the symmetric tridiagonal matrix.
?larrd	s, d	Computes the eigenvalues of a symmetric tridiagonal matrix to suitable accuracy.
?larre	s, d	Given the tridiagonal matrix T, sets small off-diagonal elements to zero and for each unreduced block T_i, finds base representations and eigenvalues.
?larrf	s, d	Finds a new relatively robust representation such that at least one of the eigenvalues is relatively isolated.
?larrj	s, d	Performs refinement of the initial estimates of the eigenvalues of the matrix T.
?larrk	s, d	Computes one eigenvalue of a symmetric tridiagonal matrix T to suitable accuracy.
?larrr	s, d	Performs tests to decide whether the symmetric tridiagonal matrix T warrants expensive computations which guarantee high relative accuracy in the eigenvalues.
?larrv	s, d, c, z	Computes the eigenvectors of the tridiagonal matrix $T = L\,D\,L^T$ given L, D and the eigenvalues of $L\,D\,L^T$.
?lartg	s, d, c, z	Generates a plane rotation with real cosine and real/complex sine.
?lartv	s, d, c, z	Applies a vector of plane rotations with real cosines and real/complex sines to the elements of a pair of vectors.
?laruv	s, d	Returns a vector of n random real numbers from a uniform distribution.
?larz	s, d, c, z	Applies an elementary reflector (as returned by ?tzrzf) to a general matrix.

Routine Name	Data Types	Description
?larzb	s, d, c, z	Applies a block reflector or its transpose/conjugate-transpose to a general matrix.
?larzt	s, d, c, z	Forms the triangular factor T of a block reflector $H = I - vtv^H$.
?las2	s, d	Computes singular values of a 2-by-2 triangular matrix.
?lascl	s, d, c, z	Multiplies a general rectangular matrix by a real scalar defined as c_{to}/c_{from}.
?lasd0	s, d	Computes the singular values of a real upper bidiagonal n-by-m matrix B with diagonal d and off-diagonal e. Used by ?bdsdc.
?lasd1	s, d	Computes the SVD of an upper bidiagonal matrix B of the specified size. Used by ?bdsdc.
?lasd2	s, d	Merges the two sets of singular values together into a single sorted set. Used by ?bdsdc.
?lasd3	s, d	Finds all square roots of the roots of the secular equation, as defined by the values in D and Z, and then updates the singular vectors by matrix multiplication. Used by ?bdsdc.
?lasd4	s, d	Computes the square root of the i-th updated eigenvalue of a positive symmetric rank-one modification to a positive diagonal matrix. Used by ?bdsdc.
?lasd5	s, d	Computes the square root of the i-th eigenvalue of a positive symmetric rank-one modification of a 2-by-2 diagonal matrix.Used by ?bdsdc.
?lasd6	s, d	Computes the SVD of an updated upper bidiagonal matrix obtained by merging two smaller ones by appending a row. Used by ?bdsdc.
?lasd7	s, d	Merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. Used by ?bdsdc.

Routine Name	Data Types	Description
?lasd8	s, d	Finds the square roots of the roots of the secular equation, and stores, for each element in D, the distance to its two nearest poles. Used by ?bdsdc.
?lasd9	s, d	Finds the square roots of the roots of the secular equation, and stores, for each element in D, the distance to its two nearest poles. Used by ?bdsdc.
?lasda	s, d	Computes the singular value decomposition (SVD) of a real upper bidiagonal matrix with diagonal *d* and off-diagonal *e*. Used by ?bdsdc.
?lasdq	s, d	Computes the SVD of a real bidiagonal matrix with diagonal *d* and off-diagonal *e*. Used by ?bdsdc.
?lasdt	s, d	Creates a tree of subproblems for bidiagonal divide and conquer. Used by ?bdsdc.
?laset	s, d, c, z	Initializes the off-diagonal elements and the diagonal elements of a matrix to given values.
?lasq1	s, d	Computes the singular values of a real square bidiagonal matrix. Used by ?bdsqr.
?lasq2	s, d	Computes all the eigenvalues of the symmetric positive definite tridiagonal matrix associated with the *qd* Array *Z* to high relative accuracy. Used by ?bdsqr and ?stegr.
?lasq3	s, d	Checks for deflation, computes a shift and calls *dqds*. Used by ?bdsqr.
?lasq4	s, d	Computes an approximation to the smallest eigenvalue using values of *d* from the previous transform. Used by ?bdsqr.
?lasq5	s, d	Computes one *dqds* transform in ping-pong form. Used by ?bdsqr and ?stegr.
?lasq6	s, d	Computes one *dqd* transform in ping-pong form. Used by ?bdsqr and ?stegr.

Routine Name	Data Types	Description
?lasr	s, d, c, z	Applies a sequence of plane rotations to a general rectangular matrix.
?lasrt	s, d	Sorts numbers in increasing or decreasing order.
?lassq	s, d, c, z	Updates a sum of squares represented in scaled form.
?lasv2	s, d	Computes the singular value decomposition of a 2-by-2 triangular matrix.
?laswp	s, d, c, z	Performs a series of row interchanges on a general rectangular matrix.
?lasy2	s, d	Solves the Sylvester matrix equation where the matrices are of order 1 or 2.
?lasyf	s, d, c, z	Computes a partial factorization of a real/complex symmetric matrix, using the diagonal pivoting method.
?lahef	c, z	Computes a partial factorization of a complex Hermitian indefinite matrix, using the diagonal pivoting method.
?latbs	s, d, c, z	Solves a triangular banded system of equations.
?latdf	s, d, c, z	Uses the LU factorization of the n-by-n matrix computed by ?getc2 and computes a contribution to the reciprocal Dif-estimate.
?latps	s, d, c, z	Solves a triangular system of equations with the matrix held in packed storage.
?latrd	s, d, c, z	Reduces the first *nb* rows and columns of a symmetric/Hermitian matrix *A* to real tridiagonal form by an orthogonal/unitary similarity transformation.
?latrs	s, d, c, z	Solves a triangular system of equations with the scale factor set to prevent overflow.
?latrz	s, d, c, z	Factors an upper trapezoidal matrix by means of orthogonal/unitary transformations.

Routine Name	Data Types	Description
?lauu2	s, d, c, z	Computes the product UU^H or L^HL, where U and L are upper or lower triangular matrices (unblocked algorithm).
?lauum	s, d, c, z	Computes the product UU^H or L^HL, where U and L are upper or lower triangular matrices (blocked algorithm).
?lazq3	s, d,	Checks for deflation, computes a shift and calls *dqds*
?lazq4	s, d,	Computes an approximation to the smallest eigenvalue using values of d from the previous transform.
?org2l/?ung2l	s, d/c, z	Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by ?geqlf (unblocked algorithm).
?org2r/?ung2r	s, d/c, z	Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by ?geqrf (unblocked algorithm).
?orgl2/?ungl2	s, d/c, z	Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by ?gelqf (unblocked algorithm).
?orgr2/?ungr2	s, d/c, z	Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by ?gerqf (unblocked algorithm).
?orm2l/?unm2l	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by ?geqlf (unblocked algorithm).
?orm2r/?unm2r	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by ?geqrf (unblocked algorithm).
?orml2/?unml2	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a LQ factorization determined by ?gelqf (unblocked algorithm).

Routine Name	Data Types	Description
?ormr2/?unmr2	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a RQ factorization determined by ?gerqf (unblocked algorithm).
?ormr3/?unmr3	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a RZ factorization determined by ?tzrzf (unblocked algorithm).
?pbtf2	s, d, c, z	Computes the Cholesky factorization of a symmetric/ Hermitian positive definite band matrix (unblocked algorithm).
?potf2	s, d, c, z	Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix (unblocked algorithm).
?ptts2	s, d, c, z	Solves a tridiagonal system of the form $AX=B$ using the $L D L^H$ factorization computed by ?pttrf.
?rscl	s, d, cs, zd	Multiplies a vector by the reciprocal of a real scalar.
?sygs2/?hegs2	s, d/c, z	Reduces a symmetric/Hermitian definite generalized eigenproblem to standard form, using the factorization results obtained from ?potrf (unblocked algorithm).
?sytd2/?hetd2	s, d/c, z	Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (unblocked algorithm).
?sytf2	s, d, c, z	Computes the factorization of a real/complex symmetric indefinite matrix, using the diagonal pivoting method (unblocked algorithm).
?hetf2	c, z	Computes the factorization of a complex Hermitian matrix, using the diagonal pivoting method (unblocked algorithm).
?tgex2	s, d, c, z	Swaps adjacent diagonal blocks in an upper (quasi) triangular matrix pair by an orthogonal/unitary equivalence transformation.

Routine Name	Data Types	Description
?tgsy2	s, d, c, z	Solves the generalized Sylvester equation (unblocked algorithm).
?trti2	s, d, c, z	Computes the inverse of a triangular matrix (unblocked algorithm).
clag2z	c → z	Converts a complex single precision matrix to a complex double precision matrix.
dlag2s	d → s	Converts a double precision matrix to a single precision matrix.
slag2d	s → d	Converts a single precision matrix to a double precision matrix.
zlag2c	z → c	Converts a complex double precision matrix to a complex single precision matrix.

?lacgv

Conjugates a complex vector.

Syntax

```
call clacgv( n, x, incx )
call zlacgv( n, x, incx )
```

Description

This routine conjugates a complex vector *x* of length *n* and increment *incx* (see "Vector Arguments in BLAS" in Appendix B).

Input Parameters

n — INTEGER. The length of the vector *x* (*n* ≥ 0).

x — COMPLEX for clacgv
COMPLEX*16 for zlacgv.
Array, dimension (1+(*n*-1)* |*incx*|).
Contains the vector of length *n* to be conjugated.

incx INTEGER. The spacing between successive elements of *x*.

Output Parameters

x On exit, overwritten with conjg(*x*).

?lacrm

Multiplies a complex matrix by a square real matrix.

Syntax

```
call clacrm( m, n, a, lda, b, ldb, c, ldc, rwork )
call zlacrm( m, n, a, lda, b, ldb, c, ldc, rwork )
```

Description

This routine performs a simple matrix-matrix multiplication of the form

*C = A*B*,

where *A* is *m*-by-*n* and complex, *B* is *n*-by-*n* and real, *C* is *m*-by-*n* and complex.

Input Parameters

m INTEGER. The number of rows of the matrix *A* and of the matrix *C* (*m* ≥ 0).

n INTEGER. The number of columns and rows of the matrix *B* and the number of columns of the matrix *C* (*n* ≥ 0).

a COMPLEX for clacrm
COMPLEX*16 for zlacrm
Array, DIMENSION (*lda*, *n*). Contains the *m*-by-*n* matrix *A*.

lda INTEGER. The leading dimension of the array *a*, *lda* ≥ max(1, *m*).

b REAL for clacrm
DOUBLE PRECISION for zlacrm

	Array, DIMENSION (*ldb*, *n*). Contains the *n*-by-*n* matrix *B*.
ldb	INTEGER. The leading dimension of the array *b*, *ldb* ≥ max(1, *n*).
ldc	INTEGER. The leading dimension of the output array *c*, *ldc* ≥ max(1, *n*).
rwork	REAL for clacrm DOUBLE PRECISION for zlacrm Workspace array, DIMENSION (2**m***n*).

Output Parameters

c	COMPLEX for clacrm COMPLEX*16 for zlacrm Array, DIMENSION (*ldc*, *n*). Contains the *m*-by-*n* matrix *C*.

?lacrt

Performs a linear transformation of a pair of complex vectors.

Syntax

```
call clacrt( n, cx, incx, cy, incy, c, s )
call zlacrt( n, cx, incx, cy, incy, c, s )
```

Description

This routine performs the following transformation

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \Rightarrow \begin{pmatrix} x \\ y \end{pmatrix},$$

where *c*, *s* are complex scalars and *x*, *y* are complex vectors.

Input Parameters

n	INTEGER. The number of elements in the vectors *cx* and *cy* ($n \geq 0$).
cx, *cy*	COMPLEX for clacrt COMPLEX*16 for zlacrt Arrays, dimension (n). Contain input vectors x and y, respectively.
incx	INTEGER. The increment between successive elements of *cx*.
incy	INTEGER. The increment between successive elements of *cy*.
c, *s*	COMPLEX for clacrt COMPLEX*16 for zlacrt Complex scalars that define the transform matrix

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}$$

Output Parameters

cx	On exit, overwritten with $c*x + s*y$.
cy	On exit, overwritten with $-s*x + c*y$.

?laesy

Computes the eigenvalues and eigenvectors of a 2-by-2 complex symmetric matrix, and checks that the norm of the matrix of eigenvectors is larger than a threshold value.

Syntax

```
call claesy( a, b, c, rt1, rt2, evscal, cs1, sn1 )
call zlaesy( a, b, c, rt1, rt2, evscal, cs1, sn1 )
```

Description

This routine performs the eigendecomposition of a 2-by-2 symmetric matrix

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix},$$

provided the norm of the matrix of eigenvectors is larger than some threshold value.

rt1 is the eigenvalue of larger absolute value, and *rt2* of smaller absolute value. If the eigenvectors are computed, then on return (*cs1*, *sn1*) is the unit eigenvector for *rt1*, hence

$$\begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} \cdot \begin{bmatrix} a & b \\ b & c \end{bmatrix} \cdot \begin{bmatrix} cs1 & -sn1 \\ sn1 & cs1 \end{bmatrix} = \begin{bmatrix} rt1 & 0 \\ 0 & rt2 \end{bmatrix}$$

Input Parameters

a, *b*, *c*
: `COMPLEX` for `claesy`
`COMPLEX*16` for `zlaesy`
Elements of the input matrix.

Output Parameters

rt1, *rt2*
: `COMPLEX` for `claesy`
`COMPLEX*16` for `zlaesy`
Eigenvalues of larger and smaller modulus, respectively.

evscal
: `COMPLEX` for `claesy`
`COMPLEX*16` for `zlaesy`
The complex value by which the eigenvector matrix was scaled to make it orthonormal. If *evscal* is zero, the eigenvectors were not computed. This means one of two things: the 2-by-2 matrix could not be diagonalized, or the norm of the matrix of eigenvectors before scaling was larger than the threshold value `thresh` (set to 0.1E0).

cs1, *sn1*	COMPLEX for claesy COMPLEX*16 for zlaesy If *evscal* is not zero, then (*cs1*, *sn1*) is the unit right eigenvector for rt1.

?rot

Applies a plane rotation with real cosine and complex sine to a pair of complex vectors.

Syntax

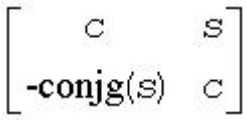
```
call crot( n, cx, incx, cy, incy, c, s )
call zrot( n, cx, incx, cy, incy, c, s )
```

Description

This routine applies a plane rotation, where the cosine (*c*) is real and the sine (*s*) is complex, and the vectors *cx* and *cy* are complex. This routine has its real equivalents in BLAS (see ?rot in Chapter 2).

Input Parameters

n	INTEGER. The number of elements in the vectors *cx* and *cy*.
cx, *cy*	COMPLEX for crot COMPLEX*16 for zrot Arrays of dimension (*n*), contain input vectors *x* and *y*, respectively.
incx	INTEGER. The increment between successive elements of *cx*.
incy	INTEGER. The increment between successive elements of *cy*.
c	REAL for crot DOUBLE PRECISION for zrot
s	COMPLEX for crot COMPLEX*16 for zrot Values that define a rotation



where `c*c + s*conjg(s) = 1.0`.

Output Parameters

cx	On exit, overwritten with `c*x + s*y`.
cy	On exit, overwritten with `-conjg(s)*x + c*y`.

?spmv

Computes a matrix-vector product for complex vectors using a complex symmetric packed matrix.

Syntax

```
call cspmv( uplo, n, alpha, ap, x, incx, beta, y, incy )
call zspmv( uplo, n, alpha, ap, x, incx, beta, y, incy )
```

Description

These routines perform a matrix-vector operation defined as

`y := alpha*a*x + beta*y`,

where:

alpha and *beta* are complex scalars,

x and *y* are *n*-element complex vectors

a is an *n*-by-*n* complex symmetric matrix, supplied in packed form.

These routines have their real equivalents in BLAS (see ?spmv in Chapter 2).

Input Parameters

uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix *a* is supplied in the packed array *ap*, as follows:

If *uplo* = 'U' or 'u', the upper triangular part of the matrix *a* is supplied in the array *ap*.
If *uplo* = 'L' or 'l', the lower triangular part of the matrix *a* is supplied in the array *ap* .

n — INTEGER.
Specifies the order of the matrix *a*.
The value of *n* must be at least zero.

alpha, *beta* — COMPLEX for cspmv
COMPLEX*16 for zspmv
Specify complex scalars *alpha* and *beta*. When *beta* is supplied as zero, then *y* need not be set on input.

ap — COMPLEX for cspmv
COMPLEX*16 for zspmv
Array, DIMENSION at least ((*n**(*n* + 1))/2). Before entry, with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *A*(1, 1), *ap*(2) and *ap*(3) contain *A*(1, 2) and *A*(2, 2) respectively, and so on. Before entry, with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, 1), *ap*(2) and *ap*(3) contain *a*(2, 1) and *a*(3, 1) respectively, and so on.

x — COMPLEX for cspmv
COMPLEX*16 for zspmv
Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array x must contain the *n*-element vector *x*.

incx — INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

y — COMPLEX for cspmv
COMPLEX*16 for zspmv
Array, DIMENSION at least (1 + (*n* - 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy	INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y	Overwritten by the updated vector *y*.

?spr

Performs the symmetrical rank-1 update of a complex symmetric packed matrix.

Syntax

```
call cspr( uplo, n, alpha, x, incx, ap )
call zspr( uplo, n, alpha, x, incx, ap )
```

Description

The ?spr routines perform a matrix-vector operation defined as

a:= *alpha***x**conjg(*x*') + *a*,

where:

alpha is a complex scalar

x is an *n*-element complex vector

a is an *n*-by-*n* complex symmetric matrix, supplied in packed form.

These routines have their real equivalents in BLAS (see ?spr in Chapter 2).

Input Parameters

uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix *a* is supplied in the packed array *ap*, as follows: If *uplo* = 'U' or 'u', the upper triangular part of the matrix *a* is supplied in the array *ap*. If *uplo* = 'L' or 'l', the lower triangular part of the matrix *a* is supplied in the array *ap* .
n	INTEGER.

Specifies the order of the matrix *a*.
The value of *n* must be at least zero.

alpha
COMPLEX for cspr
COMPLEX*16 for zspr
Specifies the scalar *alpha*.

x
COMPLEX for cspr
COMPLEX*16 for zspr
Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx
INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

ap
COMPLEX for cspr
COMPLEX*16 for zspr
Array, DIMENSION at least ((*n**(*n* + 1))/2). Before entry, with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains A(1,1), *ap*(2) and *ap*(3) contain A(1, 2) and A(2,2) respectively, and so on.
Before entry, with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1,1), *ap*(2) and *ap*(3) contain *a*(2,1) and *a*(3,1) respectively, and so on.
Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.

Output Parameters

ap
With *uplo* = 'U' or 'u', overwritten by the upper triangular part of the updated matrix.
With *uplo* = 'L' or 'l', overwritten by the lower triangular part of the updated matrix.

?symv

Computes a matrix-vector product for a complex symmetric matrix.

Syntax

```
call csymv( uplo, n, alpha, a, lda, x, incx, beta, y, incy )
call zsymv( uplo, n, alpha, a, lda, x, incx, beta, y, incy )
```

Description

These routines perform the matrix-vector operation defined as

y := *alpha***a***x* + *beta***y*,

where:

alpha and *beta* are complex scalars

x and *y* are *n*-element complex vectors

a is an *n*-by-*n* symmetric complex matrix.

These routines have their real equivalents in BLAS (see ?symv in Chapter 2).

Input Parameters

uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the array *a* is to be referenced, as follows: If *uplo* = 'U' or 'u', the upper triangular part of the array *a* is to be referenced. If *uplo* = 'L' or 'l', the lower triangular part of the array *a* is to be referenced.
n	INTEGER. Specifies the order of the matrix *a*. The value of *n* must be at least zero.
alpha, *beta*	COMPLEX for csymv COMPLEX*16 for zsymv Specify the scalars *alpha* and *beta*. When *beta* is supplied as zero, then *y* need not be set on input.
a	COMPLEX for csymv COMPLEX*16 for zsymv

Array, DIMENSION (*lda*, *n*). Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *a* is not referenced. Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *a* is not referenced.

lda — INTEGER. Specifies the first dimension of *A* as declared in the calling (sub)program. The value of *lda* must be at least max(1,*n*).

x — COMPLEX for csymv
COMPLEX*16 for zsymv
Array, DIMENSION at least (1 + (*n* – 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx — INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

y — COMPLEX for csymv
COMPLEX*16 for zsymv
Array, DIMENSION at least (1 + (*n* – 1)*abs(*incy*)). Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy — INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y — Overwritten by the updated vector *y*.

?syr

Performs the symmetric rank-1 update of a complex symmetric matrix.

Syntax

```
call csyr( uplo, n, alpha, x, incx, a, lda )
call zsyr( uplo, n, alpha, x, incx, a, lda )
```

Description

These routines perform the symmetric rank 1 operation defined as

`a := alpha*x*x' + a`,

where:

alpha is a complex scalar

x is an *n*-element complex vector

a is an *n*-by-*n* complex symmetric matrix.

These routines have their real equivalents in BLAS (see ?syr in Chapter 2).

Input Parameters

uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the array *a* is to be referenced, as follows: If *uplo* = 'U' or 'u', the upper triangular part of the array *a* is to be referenced. If *uplo* = 'L' or 'l', the lower triangular part of the array *a* is to be referenced.
n	INTEGER. Specifies the order of the matrix *a*. The value of *n* must be at least zero.
alpha	COMPLEX for csyr COMPLEX*16 for zsyr Specifies the scalar *alpha*.
x	COMPLEX for csyr COMPLEX*16 for zsyr Array, DIMENSION at least (1 + (*n* - 1)*abs(*incx*)). Before entry, the incremented array *x* must contain the *n*-element vector x.

incx INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

a COMPLEX for csyr
COMPLEX*16 for zsyr
Array, DIMENSION (*lda*, *n*). Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *a* is not referenced.
Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *a* is not referenced.

lda INTEGER. Specifies the first dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1,*n*).

Output Parameters

a With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated matrix.
With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated matrix.

i?max1

Finds the index of the vector element whose real part has maximum absolute value.

Syntax

```
index = icmax1( n, cx, incx )
index = izmax1( n, cx, incx )
```

Description

Given a complex vector *cx*, the i?max1 functions return the index of the vector element whose real part has maximum absolute value. These functions are based on the BLAS functions icamax/izamax, but using the absolute value of the real part. They are designed for use with clacon/zlacon.

Input Parameters

n	INTEGER. Specifies the number of elements in the vector *cx*.
cx	COMPLEX for icmax1 COMPLEX*16 for izmax1 Array, DIMENSION at least (1+(*n*-1)*abs(*incx*)). Contains the input vector.
incx	INTEGER. Specifies the spacing between successive elements of *cx*.

Output Parameters

index	INTEGER. Contains the index of the vector element whose real part has maximum absolute value.

?sum1

Forms the 1-norm of the complex vector using the true absolute value.

Syntax

```
res = scsum1( n, cx, incx )
res = dzsum1( n, cx, incx )
```

Description

Given a complex vector *cx*, scsum1/dzsum1 functions take the sum of the absolute values of vector elements and return a single/DOUBLE PRECISION result, respectively. These functions are based on scasum/dzasum from Level 1 BLAS, but use the true absolute value and were designed for use with clacon/zlacon.

Input Parameters

n	INTEGER. Specifies the number of elements in the vector *cx*.
cx	COMPLEX for scsum1 COMPLEX*16 for dzsum1 Array, DIMENSION at least (1+(*n*-1)*abs(*incx*)). Contains the input vector whose elements will be summed.
incx	INTEGER. Specifies the spacing between successive elements of *cx* (*incx* > 0).

Output Parameters

res	REAL for scsum1 DOUBLE PRECISION for dzsum1 Contains the sum of absolute values.

?gbtf2

Computes the LU factorization of a general band matrix using the unblocked version of the algorithm.

Syntax

```
call sgbtf2( m, n, kl, ku, ab, ldab, ipiv, info )
call dgbtf2( m, n, kl, ku, ab, ldab, ipiv, info )
call cgbtf2( m, n, kl, ku, ab, ldab, ipiv, info )
call zgbtf2( m, n, kl, ku, ab, ldab, ipiv, info )
```

Description

The routine forms the *LU* factorization of a general real/complex *m*-by-*n* band matrix *A* with *kl* sub-diagonals and *ku* super-diagonals. The routine uses partial pivoting with row interchanges and implements the unblocked version of the algorithm, calling Level 2 BLAS. See also ?gbtrf.

Input Parameters

m	INTEGER. The number of rows of the matrix *A* (*m* ≥ 0).

n	INTEGER. The number of columns in *A* (*n* ≥ 0).
kl	INTEGER. The number of sub-diagonals within the band of *A* (*kl* ≥ 0).
ku	INTEGER. The number of super-diagonals within the band of *A* (*ku* ≥ 0).
ab	REAL for sgbtf2 DOUBLE PRECISION for dgbtf2 COMPLEX for cgbtf2 COMPLEX*16 for zgbtf2. Array, DIMENSION (*ldab*,*). The array *ab* contains the matrix *A* in band storage (see Matrix Arguments). The second dimension of *ab* must be at least max(1, *n*).
ldab	INTEGER. The first dimension of the array *ab*. (*ldab* ≥ 2*kl* + *ku* +1)

Output Parameters

ab	Overwritten by details of the factorization. The diagonal and *kl* + *ku* super-diagonals of *U* are stored in the first 1 + *kl* + *ku* rows of *ab*. The multipliers used during the factorization are stored in the next *kl* rows.
ipiv	INTEGER. Array, DIMENSION at least max(1,min(*m*,*n*)). The pivot indices: row *i* was interchanged with row *ipiv*(*i*).
info	INTEGER. If *info* =0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i*, u_{ii} is 0. The factorization has been completed, but *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

?gebd2

Reduces a general matrix to bidiagonal form using an unblocked algorithm.

Syntax

```
call sgebd2( m, n, a, lda, d, e, tauq, taup, work, info )
call dgebd2( m, n, a, lda, d, e, tauq, taup, work, info )
call cgebd2( m, n, a, lda, d, e, tauq, taup, work, info )
call zgebd2( m, n, a, lda, d, e, tauq, taup, work, info )
```

Description

The routine reduces a general m-by-n matrix A to upper or lower bidiagonal form B by an orthogonal (unitary) transformation: $Q'*A*P = B$

If $m \geq n$, B is upper bidiagonal; if $m < n$, B is lower bidiagonal.

The routine does not form the matrices Q and P explicitly, but represents them as products of elementary reflectors. if $m \geq n$,

$Q = H(1)H(2)...H(n)$ and $P = G(1)G(2)...G(n-1)$

if $m < n$,

$Q = H(1)H(2)...H(m-1)$ and $P = G(1)G(2)...G(m)$

Each H(i) and G(i) has the form

$H(i) = I - tauq*v*v'$ and $G(i) = I - taup*u*u'$

where *tauq* and *taup* are scalars (real for sgebd2/dgebd2, complex for cgebd2/zgebd2), and v and u are vectors (real for sgebd2/dgebd2, complex for cgebd2/zgebd2).

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
a, *work*	REAL for sgebd2 DOUBLE PRECISION for dgebd2 COMPLEX for cgebd2

COMPLEX*16 for zgebd2.
Arrays:
a(*lda*,*) contains the *m*-by-*n* general matrix *A* to be reduced. The second dimension of *a* must be at least max(1, *n*).
work(*) is a workspace array, the dimension of *work* must be at least max(1, *m*, *n*).

lda — INTEGER. The first dimension of *a*; at least max(1, *m*).

Output Parameters

a — if *m* ≥ *n*, the diagonal and first super-diagonal of *a* are overwritten with the upper bidiagonal matrix *B*. Elements below the diagonal, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors, and elements above the first superdiagonal, with the array *taup*, represent the orthogonal/unitary matrix *p* as a product of elementary reflectors.
if *m* < *n*, the diagonal and first sub-diagonal of *a* are overwritten by the lower bidiagonal matrix *B*. Elements below the first subdiagonal, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors, and elements above the diagonal, with the array *taup*, represent the orthogonal/unitary matrix *p* as a product of elementary reflectors.

d — REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
Array, DIMENSION at least max(1, min(*m*, *n*)).
Contains the diagonal elements of the bidiagonal matrix *B*: *d*(i) = *a*(i, i).

e — REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors. Array, DIMENSION at least max(1, min(*m*, *n*) - 1).
Contains the off-diagonal elements of the bidiagonal matrix *B*:
if *m* ≥ *n*, *e*(i) = *a*(i, i+1) for i = 1,2,..., *n*-1;
if *m* < *n*, *e*(i) = *a*(i+1, i) for i = 1,2,..., *m*-1.

tauq, *taup*	REAL for sgebd2 DOUBLE PRECISION for dgebd2 COMPLEX for cgebd2 COMPLEX*16 for zgebd2. Arrays, DIMENSION at least max (1, min(*m*, *n*)). Contain scalar factors of the elementary reflectors which represent orthogonal/unitary matrices *Q* and *p*, respectively.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

?gehd2

Reduces a general square matrix to upper Hessenberg form using an unblocked algorithm.

Syntax

```
call sgehd2( n, ilo, ihi, a, lda, tau, work, info )
call dgehd2( n, ilo, ihi, a, lda, tau, work, info )
call cgehd2( n, ilo, ihi, a, lda, tau, work, info )
call zgehd2( n, ilo, ihi, a, lda, tau, work, info )
```

Description

The routine reduces a real/complex general matrix *A* to upper Hessenberg form *H* by an orthogonal or unitary similarity transformation *Q'*A*Q = H*.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of elementary reflectors.

Input Parameters

n	INTEGER The order of the matrix *A* (*n* ≥ 0).
ilo, *ihi*	INTEGER. It is assumed that *A* is already upper triangular in rows and columns 1:*ilo* -1 and *ihi*+1:*n*. If *A* has been output by ?gebal, then

	ilo and *ihi* must contain the values returned by that routine. Otherwise they should be set to *ilo* = 1 and *ihi* = *n*. Constraint: 1 ≤ *ilo* ≤ *ihi* ≤ max(1, *n*).
a, *work*	REAL for sgehd2 DOUBLE PRECISION for dgehd2 COMPLEX for cgehd2 COMPLEX*16 for zgehd2. Arrays: *a* (*lda*,*) contains the *n*-by-*n* matrix *A* to be reduced. The second dimension of *a* must be at least max(1, *n*). *work* (*n*) is a workspace array.
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).

Output Parameters

a	On exit, the upper triangle and the first subdiagonal of *A* are overwritten with the upper Hessenberg matrix *H* and the elements below the first subdiagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors. See Application Notes below.
tau	REAL for sgehd2 DOUBLE PRECISION for dgehd2 COMPLEX for cgehd2 COMPLEX*16 for zgehd2. Array, DIMENSION at least max (1, *n*-1). Contains the scalar factors of elementary reflectors. See Application Notes below.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*th parameter had an illegal value.

Application Notes

The matrix *Q* is represented as a product of (*ihi* - *ilo*) elementary reflectors

Q = *H*(*ilo*) *H*(*ilo* +1) ... *H*(*ihi* -1)

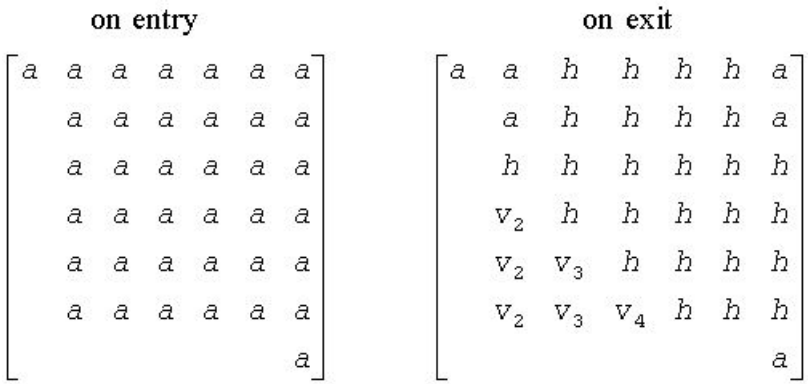
Each *H*(i) has the form

H(i) = *I* - *tau***v***v*'

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1:i) = 0, *v*(i+1) = 1 and *v*(*ihi*+1:*n*) = 0.

On exit, *v*(i+2:*ihi*) is stored in *a*(i+2:*ihi*, i) and *tau* in *tau*(i).

The contents of *a* are illustrated by the following example, with *n* = 7, *ilo* = 2 and *ihi* = 6:



where *a* denotes an element of the original matrix *A*, *h* denotes a modified element of the upper Hessenberg matrix *H*, and v_i denotes an element of the vector defining *H*(i).

?gelq2

Computes the LQ factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgelq2( m, n, a, lda, tau, work, info )
call dgelq2( m, n, a, lda, tau, work, info )
call cgelq2( m, n, a, lda, tau, work, info )
call zgelq2( m, n, a, lda, tau, work, info )
```

Description

The routine computes an *LQ* factorization of a real/complex *m*-by-*n* matrix *A* as *A* = *L***Q*.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) elementary reflectors :

Q = *H*(k) ... *H*(2) *H*(1) (or *Q* = *H*(k)' ... *H*(2)' *H*(1)' for complex flavors), where *k* = min(*m*, *n*)

Each *H*(i) has the form

H(i) = *I* - *tau***v***v*'

where *tau* is a real/complex scalar stored in *tau*(i), and *v* is a real/complex vector with *v*(1:i-1) = 0 and *v*(i) = 1.

On exit, *v*(i+1:*n*) is stored in *a*(i, i+1:*n*).

Input Parameters

m	INTEGER. The number of rows in the matrix *A* (*m* ≥ 0).
n	INTEGER. The number of columns in *A* (*n* ≥ 0).
a, *work*	REAL for sgelq2 DOUBLE PRECISION for dgelq2 COMPLEX for cgelq2 COMPLEX*16 for zgelq2. Arrays: *a*(*lda*,*) contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *work*(*m*) is a workspace array.
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).

Output Parameters

a	Overwritten by the factorization data as follows: on exit, the elements on and below the diagonal of the array *a* contain the *m*-by-min(*n*,*m*) lower trapezoidal matrix *L* (*L* is lower triangular if *n* ≥ *m*); the elements above the diagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of min(*n*,*m*) elementary reflectors.
tau	REAL for sgelq2 DOUBLE PRECISION for dgelq2 COMPLEX for cgelq2

COMPLEX*16 for zgelq2.
Array, DIMENSION at least max(1, min(*m*, *n*)).
Contains scalar factors of the elementary reflectors.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*th parameter had an illegal value.

?geql2

Computes the QL factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgeql2( m, n, a, lda, tau, work, info )
call dgeql2( m, n, a, lda, tau, work, info )
call cgeql2( m, n, a, lda, tau, work, info )
call zgeql2( m, n, a, lda, tau, work, info )
```

Description

The routine computes a *QL* factorization of a real/complex *m*-by-*n* matrix *A* as *A* = *Q***L*.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) elementary reflectors :

Q = *H*(k) ... *H*(2) *H*(1), where *k* = min(*m*, *n*)

Each *H*(i) has the form

H(i) = *I* - *tau***v***v*'

where *tau* is a real/complex scalar stored in *tau*(i), and *v* is a real/complex vector with *v*(*m*-k+i+1:*m*) = 0 and *v*(*m*-k+i) = 1.

On exit, *v*(1:*m*-k+i-1) is stored in *a*(1:*m*-k+i-1, *n*-k+i).

Input Parameters

m INTEGER. The number of rows in the matrix *A* (*m* ≥ 0).

n INTEGER. The number of columns in *A* (*n* ≥ 0).

a, *work*	REAL for sgeql2 DOUBLE PRECISION for dgeql2 COMPLEX for cgeql2 COMPLEX*16 for zgeql2. Arrays: *a*(*lda*,*) contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *work*(*m*) is a workspace array.
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).

Output Parameters

a	Overwritten by the factorization data as follows: on exit, if $m \geq n$, the lower triangle of the subarray *a*(*m*-*n*+1:*m*, 1:*n*) contains the *n*-by-*n* lower triangular matrix *L*; if $m < n$, the elements on and below the (*n*-*m*)th superdiagonal contain the *m*-by-*n* lower trapezoidal matrix *L*; the remaining elements, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.
tau	REAL for sgeql2 DOUBLE PRECISION for dgeql2 COMPLEX for cgeql2 COMPLEX*16 for zgeql2. Array, DIMENSION at least max(1, min(*m*, *n*)). Contains scalar factors of the elementary reflectors.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

?geqr2

Computes the QR factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgeqr2( m, n, a, lda, tau, work, info )
call dgeqr2( m, n, a, lda, tau, work, info )
call cgeqr2( m, n, a, lda, tau, work, info )
call zgeqr2( m, n, a, lda, tau, work, info )
```

Description

The routine computes a *QR* factorization of a real/complex *m*-by-*n* matrix *A* as *A = Q*R*.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) elementary reflectors :

Q = H(1)*H*(2) ... *H*(k), where *k = min*(*m*, *n*)

Each *H*(i) has the form

H(i) = *I* - *tau*v*v'*

where *tau* is a real/complex scalar stored in *tau*(i), and *v* is a real/complex vector with *v*(1:i-1) = 0 and *v*(i) = 1.

On exit, *v*(i+1:*m*) is stored in *a*(i+1:*m*, i).

Input Parameters

m	INTEGER. The number of rows in the matrix *A* (*m* ≥ 0).
n	INTEGER. The number of columns in *A* (*n* ≥ 0).
a, *work*	REAL for sgeqr2 DOUBLE PRECISION for dgeqr2 COMPLEX for cgeqr2 COMPLEX*16 for zgeqr2. Arrays: *a*(*lda*,*) contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*).

	work(*n*) is a workspace array.
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).

Output Parameters

a	Overwritten by the factorization data as follows: on exit, the elements on and above the diagonal of the array *a* contain the min(*n*,*m*)-by-*n* upper trapezoidal matrix *R* (*R* is upper triangular if *m* ≥ *n*); the elements below the diagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.
tau	REAL for sgeqr2 DOUBLE PRECISION for dgeqr2 COMPLEX for cgeqr2 COMPLEX*16 for zgeqr2. Array, DIMENSION at least max(1, min(*m*, *n*)). Contains scalar factors of the elementary reflectors.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

?gerq2

Computes the RQ factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgerq2( m, n, a, lda, tau, work, info )
call dgerq2( m, n, a, lda, tau, work, info )
call cgerq2( m, n, a, lda, tau, work, info )
call zgerq2( m, n, a, lda, tau, work, info )
```

Description

The routine computes a *RQ* factorization of a real/complex *m*-by-*n* matrix *A* as *A* = *R***Q*.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of min(*m*, *n*) elementary reflectors :

Q = *H*(1)*H*(2) ... *H*(k), where *k* = min(*m*, *n*)

Each *H*(i) has the form

H(i) = *I* - *tau***v***v*'

where *tau* is a real/complex scalar stored in *tau*(i), and *v* is a real/complex vector with *v*(*n*-k+i+1:*n*) = 0 and *v*(*n*-k+i) = 1.

On exit, *v*(1:*n*-k+i-1) is stored in *a*(*m*-k+i, 1:*n*-k+i-1).

Input Parameters

m	INTEGER. The number of rows in the matrix *A* (*m* ≥ 0).
n	INTEGER. The number of columns in *A* (*n* ≥ 0).
a, *work*	REAL for sgerq2 DOUBLE PRECISION for dgerq2 COMPLEX for cgerq2 COMPLEX*16 for zgerq2. Arrays: *a*(*lda*,*) contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1, *n*). *work*(*m*) is a workspace array.
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).

Output Parameters

a	Overwritten by the factorization data as follows: on exit, if *m* ≤ *n*, the upper triangle of the subarray *a*(1:*m*, *n*-*m*+1:*n*) contains the *m*-by-*m* upper triangular matrix *R*; if *m* > *n*, the elements on and above the (*m*-*n*)-th subdiagonal contain the *m*-by-*n* upper trapezoidal matrix *R*; the remaining elements, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.
tau	REAL for sgerq2 DOUBLE PRECISION for dgerq2

	COMPLEX for cgerq2 COMPLEX*16 for zgerq2. Array, DIMENSION at least max(1, min(*m*, *n*)). Contains scalar factors of the elementary reflectors.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

?gesc2

Solves a system of linear equations using the LU factorization with complete pivoting computed by ?getc2*.*

Syntax

```
call sgesc2( n, a, lda, rhs, ipiv, jpiv, scale )
call dgesc2( n, a, lda, rhs, ipiv, jpiv, scale )
call cgesc2( n, a, lda, rhs, ipiv, jpiv, scale )
call zgesc2( n, a, lda, rhs, ipiv, jpiv, scale )
```

Description

This routine solves a system of linear equations

*A*X = scale*RHS*

with a general *n*-by-*n* matrix *A* using the *LU* factorization with complete pivoting computed by ?getc2.

Input Parameters

n	INTEGER. The order of the matrix *A*.
a, *rhs*	REAL for sgesc2 DOUBLE PRECISION for dgesc2 COMPLEX for cgesc2 COMPLEX*16 for zgesc2. Arrays: *a*(*lda*,*) contains the *LU* part of the factorization of the *n*-by-*n* matrix *A* computed by ?getc2:

	*A = P*L*U*Q*. The second dimension of *a* must be at least max(1, *n*); *rhs*(*n*) contains on entry the right hand side vector for the system of equations.
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).
ipiv	INTEGER. Array, DIMENSION at least max(1,*n*). The pivot indices: for 1 ≤ i ≤ *n*, row i of the matrix has been interchanged with row *ipiv*(i).
jpiv	INTEGER. Array, DIMENSION at least max(1,*n*). The pivot indices: for 1 ≤ j ≤ *n*, column j of the matrix has been interchanged with column *jpiv*(j).

Output Parameters

rhs	On exit, overwritten with the solution vector *X*.
scale	REAL for sgesc2/cgesc2 DOUBLE PRECISION for dgesc2/zgesc2 Contains the scale factor. *scale* is chosen in the range 0 ≤ *scale* ≤ 1 to prevent overflow in the solution.

?getc2

Computes the LU factorization with complete pivoting of the general n-by-n matrix.

Syntax

```
call sgetc2( n, a, lda, ipiv, jpiv, info )
call dgetc2( n, a, lda, ipiv, jpiv, info )
call cgetc2( n, a, lda, ipiv, jpiv, info )
call zgetc2( n, a, lda, ipiv, jpiv, info )
```

Description

This routine computes an *LU* factorization with complete pivoting of the *n*-by-*n* matrix *A*. The factorization has the form *A = P*L*U*Q*, where *p* and *Q* are permutation matrices, *L* is lower triangular with unit diagonal elements and *U* is upper triangular.

The LU factorization computed by this routine is used by ?latdf to compute a contribution to the reciprocal Dif-estimate.

Input Parameters

n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
a	REAL for sgetc2 DOUBLE PRECISION for dgetc2 COMPLEX for cgetc2 COMPLEX*16 for zgetc2. Array *a*(*lda*,*) contains the *n*-by-*n* matrix *A* to be factored. The second dimension of *a* must be at least max(1, *n*);
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).

Output Parameters

a	On exit, the factors *L* and *U* from the factorization *A = P*L*U*Q*; the unit diagonal elements of *L* are not stored. If *U*(*k*, *k*) appears to be less than smin, *U*(*k*, *k*) is given the value of smin, i.e., giving a nonsingular perturbed system.
ipiv	INTEGER. Array, DIMENSION at least max(1,*n*). The pivot indices: for 1 ≤ i ≤ *n*, row i of the matrix has been interchanged with row *ipiv*(i).
jpiv	INTEGER. Array, DIMENSION at least max(1,*n*). The pivot indices: for 1 ≤ j ≤ *n*, column j of the matrix has been interchanged with column *jpiv*(j).
info	INTEGER. If *info* = 0, the execution is successful.

If *info* = *k* >0, *U*(*k*, *k*) is likely to produce overflow if we try to solve for *x* in *A***x* = *b*. So *U* is perturbed to avoid the overflow.

?getf2

Computes the LU factorization of a general m-by-n matrix using partial pivoting with row interchanges (unblocked algorithm).

Syntax

```
call sgetf2( m, n, a, lda, ipiv, info )
call dgetf2( m, n, a, lda, ipiv, info )
call cgetf2( m, n, a, lda, ipiv, info )
call zgetf2( m, n, a, lda, ipiv, info )
```

Description

The routine computes the *LU* factorization of a general *m*-by-*n* matrix *A* using partial pivoting with row interchanges. The factorization has the form

A = *P***L***U*

where *p* is a permutation matrix, *L* is lower triangular with unit diagonal elements (lower trapezoidal if *m* > *n*) and *U* is upper triangular (upper trapezoidal if *m* < *n*).

Input Parameters

m	INTEGER. The number of rows in the matrix *A* (*m* ≥ 0).
n	INTEGER. The number of columns in *A* (*n* ≥ 0).
a	REAL for sgetf2 DOUBLE PRECISION for dgetf2 COMPLEX for cgetf2 COMPLEX*16 for zgetf2. Array, DIMENSION (*lda*,*). Contains the matrix *A* to be factored. The second dimension of *a* must be at least max(1, *n*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).

Output Parameters

a	Overwritten by *L* and *U*. The unit diagonal elements of *L* are not stored.
ipiv	INTEGER. Array, DIMENSION at least max(1,min(*m*,*n*)). The pivot indices: for $1 \le i \le n$, row *i* was interchanged with row *ipiv*(*i*).
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i* >0, u_{ii} is 0. The factorization has been completed, but *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

?gtts2

Solves a system of linear equations with a tridiagonal matrix using the LU factorization computed by ?gttrf.

Syntax

```
call sgtts2( itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb )
call dgtts2( itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb )
call cgtts2( itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb )
call zgtts2( itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb )
```

Description

This routine solves for *X* one of the following systems of linear equations with multiple right hand sides:

*A*X = B*, A^T**X = B*, or A^H**X = B* (for complex matrices only), with a tridiagonal matrix A using the *LU* factorization computed by ?gttrf.

Input Parameters

itrans	INTEGER. Must be 0, 1, or 2.

	Indicates the form of the equations being solved: If *itrans* = 0, then *A*X* = *B* (no transpose). If *itrans* = 1, then $A^T*X = B$ (transpose). If *itrans* = 2, then $A^H*X = B$ (conjugate transpose).
n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
nrhs	INTEGER. The number of right-hand sides, i.e., the number of columns in *B* (*nrhs* ≥ 0).
dl,*d*,*du*,*du2*,*b*	REAL for sgtts2 DOUBLE PRECISION for dgtts2 COMPLEX for cgtts2 COMPLEX*16 for zgtts2. Arrays: *dl*(*n* - 1), *d*(*n*), *du*(*n* - 1), *du2*(*n* - 2), *b*(*ldb*,*nrhs*). The array *dl* contains the (*n* - 1) multipliers that define the matrix *L* from the *LU* factorization of *A*. The array *d* contains the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*. The array *du* contains the (*n* - 1) elements of the first super-diagonal of *U*. The array *du2* contains the (*n* - 2) elements of the second super-diagonal of *U*. The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.
ldb	INTEGER. The leading dimension of *b*; must be *ldb* ≥ max(1, *n*).
ipiv	INTEGER. Array, DIMENSION (*n*). The pivot indices array, as returned by ?gttrf.

Output Parameters

b	Overwritten by the solution matrix *X*.

?isnan

Tests input for NaN.

Syntax

```
val = sisnan( sin )
val = disnan( din )
```

Description

This logical routine returns `.TRUE.` if its argument is `NaN`, and `.FALSE.` otherwise.

Input Parameters

sin	`REAL` for `sisnan` Input to test for `NaN`.
din	`DOUBLE PRECISION` for `disnan` Input to test for `NaN`.

Output Parameters

val	`Logical`. Result of the test.

?laisnan

Tests input for NaN.

Syntax

```
val = slaisnan( sin1, sin2 )
val = dlaisnan( din1, din2 )
```

Description

This logical routine checks for `NaN`s (`NaN` stands for 'Not A Number') by comparing its two arguments for inequality. `NaN` is the only floating-point value where `NaN` ≠ `NaN` returns `.TRUE.` To check for `NaN`s, pass the same variable as both arguments.

This routine is not for general use. It exists solely to avoid over-optimization in `?isnan`.

Input Parameters

sin1, *sin2*	REAL for sisnan Two numbers to compare for inequality.
din2, *din2*	DOUBLE PRECISION for disnan Two numbers to compare for inequality.

Output Parameters

val	Logical. Result of the comparison.

?labrd

Reduces the first nb rows and columns of a general matrix to a bidiagonal form.

Syntax

```
call slabrd( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
call dlabrd( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
call clabrd( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
call zlabrd( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
```

Description

The routine reduces the first *nb* rows and columns of a general *m*-by-*n* matrix *A* to upper or lower bidiagonal form by an orthogonal/unitary transformation *Q*'**A***P*, and returns the matrices *X* and *Y* which are needed to apply the transformation to the unreduced part of *A*.

if *m* ≥ *n*, *A* is reduced to upper bidiagonal form; if *m* < *n*, to lower bidiagonal form.

The matrices *Q* and *P* are represented as products of elementary reflectors: *Q* = *H*(1) *H*(2) ... *H*(*nb*) and *P* = *G*(1) *G*(2) ... *G*(*nb*)

Each *H*(i) and *G*(i) has the form

H(i) = *I* - *tauq***v***v*' and *G*(i) = *I* - *taup***u***u*'

where *tauq* and *taup* are scalars, and *v* and *u* are vectors.

The elements of the vectors *v* and *u* together form the *m*-by-*nb* matrix *V* and the *nb*-by-*n* matrix *U'* which are needed, with *X* and *Y*, to apply the transformation to the unreduced part of the matrix, using a block update of the form: *A := A - V*Y' - X*U'*.

This is an auxiliary routine called by ?gebrd.

Input Parameters

m	INTEGER. The number of rows in the matrix *A* (*m* ≥ 0).
n	INTEGER. The number of columns in *A* (*n* ≥ 0).
nb	INTEGER. The number of leading rows and columns of *A* to be reduced.
a	REAL for slabrd DOUBLE PRECISION for dlabrd COMPLEX for clabrd COMPLEX*16 for zlabrd. Array *a*(*lda*,*) contains the matrix *A* to be reduced. The second dimension of *a* must be at least max(1, *n*).
lda	INTEGER. The first dimension of *a*; at least max(1, *m*).
ldx	INTEGER. The first dimension of the output array *x*; must beat least max(1, *m*).
ldy	INTEGER. The first dimension of the output array *y*; must beat least max(1, *n*).

Output Parameters

a	On exit, the first *nb* rows and columns of the matrix are overwritten; the rest of the array is unchanged. if *m* ≥ *n*, elements on and below the diagonal in the first *nb* columns, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors; and elements above the diagonal in the first *nb* rows, with the array *taup*, represent the orthogonal/unitary matrix *p* as a product of elementary reflectors. if *m* < *n*, elements below the diagonal in the first *nb* columns, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary

reflectors, and elements on and above the diagonal in the first *nb* rows, with the array *taup*, represent the orthogonal/unitary matrix *p* as a product of elementary reflectors.

d, *e*
REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors. Arrays, DIMENSION (*nb*) each. The array *d* contains the diagonal elements of the first *nb* rows and columns of the reduced matrix:
d(i) = *a*(i,i).
The array *e* contains the off-diagonal elements of the first *nb* rows and columns of the reduced matrix.

tauq, *taup*
REAL for slabrd
DOUBLE PRECISION for dlabrd
COMPLEX for clabrd
COMPLEX*16 for zlabrd.
Arrays, DIMENSION (*nb*) each. Contain scalar factors of the elementary reflectors which represent the orthogonal/unitary matrices *Q* and *P*, respectively.

x, *y*
REAL for slabrd
DOUBLE PRECISION for dlabrd
COMPLEX for clabrd
COMPLEX*16 for zlabrd.
Arrays, dimension *x*(*ldx*, *nb*), *y*(*ldy*, *nb*).
The array *x* contains the *m*-by-*nb* matrix *X* required to update the unreduced part of *A*.
The array *y* contains the *n*-by-*nb* matrix *Y* required to update the unreduced part of *A*.

Application Notes

if *m* ≥ *n*, then for the elementary reflectors *H*(i) and *G*(i),

v(1:i-1) = 0, *v*(i) = 1, and *v*(i:*m*) is stored on exit in *a*(i:*m*, i); *u*(1:i) = 0, *u*(i+1) = 1, and *u*(i+1:*n*) is stored on exit in *a*(i, i+1:*n*);

tauq is stored in *tauq*(i) and *taup* in *taup*(i).

if *m* < *n*,

`v(1:i) = 0`, `v(i+1) = 1`, and `v(i+1:m)` is stored on exit in `a(i+2:m, i)`; `u(1:i-1) = 0`, `u(i) = 1`, and `u(i:n)` is stored on exit in `a(i, i+1:n)`; *tauq* is stored in *tauq*(i) and *taup* in *taup*(i).

The contents of *a* on exit are illustrated by the following examples with `nb = 2`:

$m = 6,\ n = 5\ (m > n)$

$$\begin{bmatrix} 1 & 1 & u_1 & u_1 & u_1 \\ v_1 & 1 & 1 & u_2 & u_2 \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \end{bmatrix}$$

$m = 5,\ n = 6\ (m < n)$

$$\begin{bmatrix} 1 & u_1 & u_1 & u_1 & u_1 & u_1 \\ 1 & 1 & u_2 & u_2 & u_2 & u_2 \\ v_1 & 1 & a & a & a & a \\ v_1 & v_2 & a & a & a & a \\ v_1 & v_2 & a & a & a & a \end{bmatrix}$$

where *a* denotes an element of the original matrix which is unchanged, v_i denotes an element of the vector defining *H*(i), and u_i an element of the vector defining *G*(i).

?lacn2

Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.

Syntax

```
call slacn2( n, v, x, isgn, est, kase, isave )
call dlacn2( n, v, x, isgn, est, kase, isave )
call clacn2( n, v, x, est, kase, isave )
call zlacn2( n, v, x, est, kase, isave )
```

Description

This routine estimates the 1-norm of a square, real or complex matrix *A*. Reverse communication is used for evaluating matrix-vector products.

Input Parameters

n	INTEGER. The order of the matrix *A* ($n \geq 1$).
v, *x*	REAL for slacn2 DOUBLE PRECISION for dlacn2 COMPLEX for clacn2 COMPLEX*16 for zlacn2. Arrays, DIMENSION (*n*) each. *v* is a workspace array. *x* is used as input after an intermediate return.
isgn	INTEGER. Workspace array, DIMENSION (*n*), used with real flavors only.
est	REAL for slacn2/clacn2 DOUBLE PRECISION for dlacn2/zlacn2 On entry with *kase* set to 1 or 2, and *isave*(1) = 1, *est* must be unchanged from the previous call to the routine.
kase	INTEGER. On the initial call to the routine, *kase* must be set to 0.
isave	INTEGER. Array, DIMENSION (3). Contains variables from the previous call to the routine.

Output Parameters

est	An estimate (a lower bound) for norm(*A*).
kase	On an intermediate return, *kase* is set to 1 or 2, indicating whether *x* should be overwritten by *A***x* or *A*'**x*. On the final return, *kase* is set to 0.
v	On the final return, *v* = *A***w*, where *est* = norm(*v*)/norm(*w*) (*w* is not returned).
x	On an intermediate return, *x* should be overwritten by *A***x* , if *kase* = 1,

	*A'*x* , if *kase* = 2, (*A'* is the conjugate transpose of *A*), and the routine must be re-called with all the other parameters unchanged.
isave	This parameter is used to save variables between calls to the routine.

?lacon

Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.

Syntax

```
call slacon( n, v, x, isgn, est, kase )
call dlacon( n, v, x, isgn, est, kase )
call clacon( n, v, x, est, kase )
call zlacon( n, v, x, est, kase )
```

Description

This routine estimates the 1-norm of a square, real/complex matrix *A*. Reverse communication is used for evaluating matrix-vector products.

Input Parameters

n	INTEGER. The order of the matrix *A* (*n* ≥ 1).
v, *x*	REAL for slacon DOUBLE PRECISION for dlacon COMPLEX for clacon COMPLEX*16 for zlacon. Arrays, DIMENSION (*n*) each. *v* is a workspace array. *x* is used as input after an intermediate return.
isgn	INTEGER. Workspace array, DIMENSION (*n*), used with real flavors only.
est	REAL for slacon/clacon

DOUBLE PRECISION for dlacon/zlacon
An estimate that with *kase*=1 or 2 should be unchanged from the previous call to ?lacon.

kase — INTEGER.
On the initial call to ?lacon, *kase* should be 0.

Output Parameters

est — REAL for slacon/clacon
DOUBLE PRECISION for dlacon/zlacon
An estimate (a lower bound) for norm(*A*).

kase — On an intermediate return, *kase* will be 1 or 2, indicating whether *x* should be overwritten by *A***x* or *A*'**x*. On the final return from ?lacon, *kase* will again be 0.

v — On the final return, *v* = *A***w*, where *est* = norm(*v*)/norm(*w*) (*w* is not returned).

x — On an intermediate return, *x* should be overwritten by *A***x* , if *kase* = 1,
A'**x* , if *kase* = 2,
(where for complex flavors *A*' is the conjugate transpose of *A*), and ?lacon must be re-called with all the other parameters unchanged.

?lacpy

Copies all or part of one two-dimensional array to another.

Syntax

```
call slacpy( uplo, m, n, a, lda, b, ldb )
call dlacpy( uplo, m, n, a, lda, b, ldb )
call clacpy( uplo, m, n, a, lda, b, ldb )
call zlacpy( uplo, m, n, a, lda, b, ldb )
```

Description

This routine copies all or part of a two-dimensional matrix *A* to another matrix *B*.

Input Parameters

uplo	CHARACTER*1. Specifies the part of the matrix *A* to be copied to *B*. If *uplo* = 'U', the upper triangular part of *A* is copied. If *uplo* = 'L', the lower triangular part of *A* is copied. Otherwise, all of the matrix *A* is copied.
m	INTEGER. The number of rows in the matrix *A* (*m* ≥ 0).
n	INTEGER. The number of columns in *A* (*n* ≥ 0).
a	REAL for slacpy DOUBLE PRECISION for dlacpy COMPLEX for clacpy COMPLEX*16 for zlacpy. Array *a*(*lda*,*), contains the *m*-by-*n* matrix *A*. The second dimension of *a* must be at least max(1,*n*). If *uplo* = 'U', only the upper triangle or trapezoid is accessed; if *uplo* = 'L', only the lower triangle or trapezoid is accessed.
lda	INTEGER. The first dimension of *a*; *lda* ≥ max(1, *m*).
ldb	INTEGER. The first dimension of the output array *b*; *ldb* ≥ max(1, *m*).

Output Parameters

b	REAL for slacpy DOUBLE PRECISION for dlacpy COMPLEX for clacpy COMPLEX*16 for zlacpy. Array *b*(*ldb*,*), contains the *m*-by-*n* matrix *B*. The second dimension of *b* must be at least max(1,*n*). On exit, *B* = *A* in the locations specified by *uplo*.

?ladiv

Performs complex division in real arithmetic, avoiding unnecessary overflow.

Syntax

```
call sladiv( a, b, c, d, p, q )
call dladiv( a, b, c, d, p, q )
res = cladiv( x, y )
res = zladiv( x, y )
```

Description

The routines sladiv/dladiv perform complex division in real arithmetic as

$$p + iq = \frac{a + ib}{c + id}$$

Complex functions cladiv/zladiv compute the result as

res = *x*/*y*,

where *x* and *y* are complex. The computation of *x* / *y* will not overflow on an intermediary step unless the results overflows.

Input Parameters

a, *b*, *c*, *d*	REAL for sladiv DOUBLE PRECISION for dladiv The scalars *a*, *b*, *c*, and *d* in the above expression (for real flavors only).
x, *y*	COMPLEX for cladiv COMPLEX*16 for zladiv The complex scalars *x* and *y* (for complex flavors only).

Output Parameters

p, *q*	REAL for sladiv

	DOUBLE PRECISION for dladiv The scalars *p* and *q* in the above expression (for real flavors only).
res	COMPLEX for cladiv DOUBLE COMPLEX for zladiv Contains the result of division *x* / *y*.

?lae2

Computes the eigenvalues of a 2-by-2 symmetric matrix.

Syntax

```
call slae2( a, b, c, rt1, rt2 )
call dlae2( a, b, c, rt1, rt2 )
```

Description

The routines sla2/dlae2 compute the eigenvalues of a 2-by-2 symmetric matrix

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix}$$

On return, *rt1* is the eigenvalue of larger absolute value, and *rt1* is the eigenvalue of smaller absolute value.

Input Parameters

a, *b*, *c*	REAL for slae2 DOUBLE PRECISION for dlae2 The elements *a*, *b*, and *c* of the 2-by-2 matrix above.

Output Parameters

rt1, *rt2*	REAL for slae2 DOUBLE PRECISION for dlae2

The computed eigenvalues of larger and smaller absolute value, respectively.

Application Notes

rt1 is accurate to a few ulps barring over/underflow. *rt2* may be inaccurate if there is massive cancellation in the determinant *a***c*-*b***b*; higher precision or correctly rounded or correctly truncated arithmetic would be needed to compute *rt2* accurately in all cases.

Overflow is possible only if *rt1* is within a factor of 5 of overflow. Underflow is harmless if the input data is 0 or exceeds

underflow_threshold / macheps.

?laebz

Computes the number of eigenvalues of a real symmetric tridiagonal matrix which are less than or equal to a given value, and performs other tasks required by the routine ?stebz.

Syntax

call slaebz(*ijob*, *nitmax*, *n*, *mmax*, *minp*, *nbmin*, *abstol*, *reltol*, *pivmin*, *d*, *e*, *e2*, *nval*, *ab*, *c*, *mout*, *nab*, *work*, *iwork*, *info*)

call dlaebz(*ijob*, *nitmax*, *n*, *mmax*, *minp*, *nbmin*, *abstol*, *reltol*, *pivmin*, *d*, *e*, *e2*, *nval*, *ab*, *c*, *mout*, *nab*, *work*, *iwork*, *info*)

Description

The routine ?laebz contains the iteration loops which compute and use the function *n*(*w*), which is the count of eigenvalues of a symmetric tridiagonal matrix *T* less than or equal to its argument *w*. It performs a choice of two types of loops:

ijob =1, followed by

ijob =2: It takes as input a list of intervals and returns a list of sufficiently small intervals whose union contains the same eigenvalues as the union of the original intervals. The input intervals are (*ab*(j,1),*ab*(j,2)], j=1,...,*minp*. The output interval (*ab*(j,1),*ab*(j,2)] will contain eigenvalues *nab*(j,1)+1,...,*nab*(j,2), where 1 ≤ j ≤ *mout*.

ijob	=3: It performs a binary search in each input interval *(ab(j,1),ab(j,2)]* for a point *w(j)* such that *n(w(j))=nval(j)*, and uses *c(j)* as the starting point of the search. If such a *w(j)* is found, then on output *ab(j,1)=ab(j,2)=w*. If no such *w(j)* is found, then on output *(ab(j,1),ab(j,2)]* will be a small interval containing the point where *n(w)* jumps through *nval(j)*, unless that point lies outside the initial interval.

Note that the intervals are in all cases half-open intervals, that is, of the form *(a,b]*, which includes *b* but not *a* .

To avoid underflow, the matrix should be scaled so that its largest element is no greater than *overflow***(1/2) * *overflow***(1/4) in absolute value. To assure the most accurate computation of small eigenvalues, the matrix should be scaled to be not much smaller than that, either.

NOTE. In general, the arguments are not checked for unreasonable values.

Input Parameters

ijob	INTEGER. Specifies what is to be done: = 1: Compute *nab* for the initial intervals. = 2: Perform bisection iteration to find eigenvalues of *T*. = 3: Perform bisection iteration to invert *n(w)*, i.e., to find a point which has a specified number of eigenvalues of *T* to its left. Other values will cause *?laebz* to return with *info=-1*.
nitmax	INTEGER. The maximum number of "levels" of bisection to be performed, i.e., an interval of width *W* will not be made smaller than *2**(-nitmax)*W*. If not all intervals have converged after *nitmax* iterations, then *info* is set to the number of non-converged intervals.
n	INTEGER. The dimension *n* of the tridiagonal matrix *T*. It must be at least 1.
mmax	INTEGER. The maximum number of intervals. If more than *mmax* intervals are generated, then *?laebz* will quit with *info=mmax+1*.

minp — INTEGER. The initial number of intervals. It may not be greater than *mmax*.

nbmin — INTEGER. The smallest number of intervals that should be processed using a vector loop. If zero, then only the scalar loop will be used.

abstol — REAL for slaebz
DOUBLE PRECISION for dlaebz.
The minimum (absolute) width of an interval. When an interval is narrower than *abstol*, or than *reltol* times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, i.e., converged. This must be at least zero.

reltol — REAL for slaebz
DOUBLE PRECISION for dlaebz.
The minimum relative width of an interval. When an interval is narrower than *abstol*, or than *reltol* times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, i.e., converged. Note: this should always be at least *radix*machine epsilon*.

pivmin — REAL for slaebz
DOUBLE PRECISION for dlaebz.
The minimum absolute value of a "pivot" in the Sturm sequence loop. This value **must** be at least (max |e(j)**2|**safe_min*) and at least *safe_min*, where *safe_min* is at least the smallest number that can divide one without overflow.

d, *e*, *e2* — REAL for slaebz
DOUBLE PRECISION for dlaebz.
Arrays, dimension (*n*) each. The array *d* contains the diagonal elements of the tridiagonal matrix *T*.
The array *e* contains the off-diagonal elements of the tridiagonal matrix *T* in positions 1 through *n*-1. *e*(*n*)vis arbitrary.
The array *e2* contains the squares of the off-diagonal elements of the tridiagonal matrix *T*. *e2*(*n*) is ignored.

nval — INTEGER.
Array, dimension (*minp*).
If *ijob*=1 or 2, not referenced.

If *ijob*=3, the desired values of *n*(*w*).

ab — REAL for slaebz
DOUBLE PRECISION for dlaebz.
Array, dimension (*mmax*,2) The endpoints of the intervals. *ab*(j,1) is *a*(*j*), the left endpoint of the *j*-th interval, and *ab*(j,2) is *b*(*j*), the right endpoint of the *j*-th interval.

c — REAL for slaebz
DOUBLE PRECISION for dlaebz.
Array, dimension (*mmax*)
If *ijob*=1, ignored.
If *ijob*=2, workspace.
If *ijob*=3, then on input *c*(*j*) should be initialized to the first search point in the binary search.

nab — INTEGER.
Array, dimension (*mmax*,2)
If *ijob*=2, then on input, *nab*(i,j) should be set. It must satisfy the condition:

$n(ab(i,1)) \le nab(i,1) \le nab(i,2) \le n(ab(i,2))$,

which means that in interval i only eigenvalues *nab*(i,1)+1,...,*nab*(i,2) are considered. Usually, *nab*(i,j)=*n*(*ab*(i,j)), from a previous call to ?laebz with *ijob*=1.
If *ijob*=3, normally, *nab* should be set to some distinctive value(s) before ?laebz is called.

work — REAL for slaebz
DOUBLE PRECISION for dlaebz.
Workspace array, dimension (*mmax*).

iwork — INTEGER.
Workspace array, dimension (*mmax*).

Output Parameters

nval — The elements of *nval* will be reordered to correspond with the intervals in *ab*. Thus, *nval*(*j*) on output will not, in general be the same as *nval*(*j*) on input, but it will correspond with the interval (*ab*(j,1),*ab*(j,2)] on output.

ab The input intervals will, in general, be modified, split, and reordered by the calculation.

mout INTEGER.
If *ijob*=1, the number of eigenvalues in the intervals.
If *ijob*=2 or 3, the number of intervals output.
If *ijob*=3, *mout* will equal *minp*.

nab If *ijob*=1, then on output *nab*(i,j) will be set to *N*(*ab*(i,j)).
If *ijob*=2, then on output, *nab*(i,j) will contain max(*na*(*k*, min(*nb*(*k*), *N*(*ab*(i,j)))), where *k* is the index of the input interval that the output interval (*ab*(j,1),*ab*(j,2)] came from, and *na*(*k*) and *nb*(*k*) are the input values of *nab*(*k*,1) and *nab*(*k*,2).
If *ijob*=3, then on output, *nab*(i,j) contains *N*(*ab*(i,j)), unless *N*(*w*) > *nval*(i) for all search points *w*, in which case *nab*(i,1) will not be modified, i.e., the output value will be the same as the input value (modulo reorderings, see *nval* and *ab*), or unless *N*(*w*) < *nval*(i) for all search points *w*, in which case *nab*(i,2) will not be modified.

info INTEGER.

0:	All intervals converged.
1--*mmax*:	The last *info* intervals did not converge.
mmax+1:	More than *mmax* intervals were generated.

Application Notes

This routine is intended to be called only by other LAPACK routines, thus the interface is less user-friendly. It is intended for two purposes:

(a) finding eigenvalues. In this case, ?laebz should have one or more initial intervals set up in *ab*, and ?laebz should be called with *ijob*=1. This sets up *nab*, and also counts the eigenvalues. Intervals with no eigenvalues would usually be thrown out at this point. Also, if not all the eigenvalues in an interval i are desired, *nab*(i,1) can be increased or *nab*(i,2) decreased. For example, set *nab*(i,1)=*nab*(i,2)-1 to get the largest eigenvalue. ?laebz is then called with *ijob*=2 and *mmax* no smaller than the value of *mout* returned by the call with *ijob*=1. After this (*ijob*=2) call, eigenvalues *nab*(i,1)+1 through *nab*(i,2) are approximately *ab*(i,1) (or *ab*(i,2)) to the tolerance specified by *abstol* and *reltol*.

(b) finding an interval (*a*',*b*'] containing eigenvalues *w*(f),...,*w*(l). In this case, start with a Gershgorin interval (*a*,*b*). Set up *ab* to contain 2 search intervals, both initially (*a*,*b*). One *nval* element should contain f-1 and the other should contain l, while *c* should contain *a* and *b*, respectively. *nab*(i,1) should be -1 and *nab*(i,2) should be *n*+1, to flag an error if the desired interval does not lie in (*a*,*b*). ?laebz is then called with *ijob*=3. On exit, if *w*(f-1) < *w*(f), then one of the intervals -- j -- will have *ab*(j,1)=*ab*(j,2) and *nab*(j,1)=*nab*(j,2)=f-1, while if, to the specified tolerance, *w*(f-k)=...=*w*(f+r), k > 0 and r ≥ 0, then the interval will have *n*(*ab*(j,1))=*nab*(j,1)=f-k and *n*(*ab*(j,2))=*nab*(j,2)=f+r. The cases *w*(l) < *w*(l+1) and *w*(l-r)=...=*w*(l+k) are handled similarly.

?laed0

Used by ?stedc. *Computes all eigenvalues and corresponding eigenvectors of an unreduced symmetric tridiagonal matrix using the divide and conquer method.*

Syntax

```
call slaed0( icompq, qsiz, n, d, e, q, ldq, qstore, ldqs, work, iwork, info )
call dlaed0( icompq, qsiz, n, d, e, q, ldq, qstore, ldqs, work, iwork, info )
call claed0( qsiz, n, d, e, q, ldq, qstore, ldqs, rwork, iwork, info )
call zlaed0( qsiz, n, d, e, q, ldq, qstore, ldqs, rwork, iwork, info )
```

Description

Real flavors of this routine compute all eigenvalues and (optionally) corresponding eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method.

Complex flavors claed0/zlaed0 compute all eigenvalues of a symmetric tridiagonal matrix which is one diagonal block of those from reducing a dense or band Hermitian matrix and corresponding eigenvectors of the dense or band matrix.

Input Parameters

icompq	INTEGER. Used with real flavors only. If *icompq* = 0, compute eigenvalues only.

If *icompq* = 1, compute eigenvectors of original dense symmetric matrix also.
On entry, the array *q* must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.
If *icompq* = 2, compute eigenvalues and eigenvectors of the tridiagonal matrix.

qsiz — INTEGER.
The dimension of the orthogonal/unitary matrix used to reduce the full matrix to tridiagonal form; *qsiz* ≥ *n* (for real flavors, *qsiz* ≥ *n* if *icompq* = 1).

n — INTEGER. The dimension of the symmetric tridiagonal matrix (*n* ≥ 0).

d, *e*, *rwork* — REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors. Arrays:
d(*) contains the main diagonal of the tridiagonal matrix. The dimension of *d* must be at least max(1, *n*).
e(*) contains the off-diagonal elements of the tridiagonal matrix. The dimension of *e* must be at least max(1, *n*–1).
rwork(*) is a workspace array used in complex flavors only. The dimension of *rwork* must be at least (1 +3*n*+2*n*lg(*n*)+3n^2), where lg(*n*) = smallest integer *k* such that $2^k \geq n$.

q, *qstore* — REAL for slaed0
DOUBLE PRECISION for dlaed0
COMPLEX for claed0
COMPLEX*16 for zlaed0.
Arrays: *q*(*ldq*, *), *qstore*(*ldqs*, *). The second dimension of these arrays must be at least max(1, *n*).
For real flavors:
If *icompq* = 0, array *q* is not referenced.
If *icompq* = 1, on entry, *q* is a subset of the columns of the orthogonal matrix used to reduce the full matrix to tridiagonal form corresponding to the subset of the full matrix which is being decomposed at this time.

If *icompq* = 2, on entry, *q* will be the identity matrix. The array *qstore* is a workspace array referenced only when *icompq* = 1. Used to store parts of the eigenvector matrix when the updating matrix multiplies take place.
For complex flavors:
On entry, *q* must contain an *qsiz*-by-*n* matrix whose columns are unitarily orthonormal. It is a part of the unitary matrix that reduces the full dense Hermitian matrix to a (reducible) symmetric tridiagonal matrix. The array *qstore* is a workspace array used to store parts of the eigenvector matrix when the updating matrix multiplies take place.

ldq — INTEGER. The first dimension of the array *q*; *ldq* ≥ max(1, *n*).

ldqs — INTEGER. The first dimension of the array *qstore*; *ldqs* ≥ max(1, *n*).

work — REAL for slaed0
DOUBLE PRECISION for dlaed0.
Workspace array, used in real flavors only.
If *icompq* = 0 or 1, the dimension of *work* must be at least $(1 + 3n + 2n\lg(n) + 2n^2)$, where lg(*n*) = smallest integer *k* such that $2^k \geq n$.
If *icompq* = 2, the dimension of *work* must be at least $(4n + n^2)$.

iwork — INTEGER.
Workspace array.
For real flavors, if *icompq* = 0 or 1, and for complex flavors, the dimension of *iwork* must be at least $(6 + 6n + 5n\lg(n))$.
For real flavors, if *icompq* = 2, the dimension of *iwork* must be at least $(3 + 5n)$.

Output Parameters

d — On exit, contains eigenvalues in ascending order.

e — On exit, the array has been destroyed.

q — If *icompq* = 2, on exit, *q* contains the eigenvectors of the tridiagonal matrix.

info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i* >0, the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns *i*/(*n*+1) through mod(*i*, *n*+1).

?laed1

Used by sstedc/dstedc*. Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is tridiagonal.*

Syntax

```
call slaed1( n, d, q, ldq, indxq, rho, cutpnt, work, iwork, info )
call dlaed1( n, d, q, ldq, indxq, rho, cutpnt, work, iwork, info )
```

Description

The routine ?laed1 computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. This routine is used only for the eigenproblem which requires all eigenvalues and eigenvectors of a tridiagonal matrix. ?laed7 handles the case in which eigenvalues only or eigenvalues and eigenvectors of a full symmetric matrix (which was reduced to tridiagonal form) are desired.

T = *Q*(in)(*D*(in)+ *rho***Z***Z*')**Q*'(in) = *Q*(out)**D*(out)**Q*'(out)

where *Z* = *Q*'*u*, *u* is a vector of length *n* with ones in the *cutpnt* and (*cutpnt*+1) -th elements and zeros elsewhere. The eigenvectors of the original matrix are stored in *Q*, and the eigenvalues are in *D*. The algorithm consists of three stages:

The first stage consists of deflating the size of the problem when there are multiple eigenvalues or if there is a zero in the *z* vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine ?laed2.

The second stage consists of calculating the updated eigenvalues. This is done by finding the roots of the secular equation via the routine ?laed4 (as called by ?laed3). This routine also calculates the eigenvectors of the current problem.

The final stage consists of computing the updated eigenvectors directly using the updated eigenvalues. The eigenvectors for the current problem are multiplied with the eigenvectors from the overall problem.

Input Parameters

n	INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).
d, *q*, *work*	REAL for slaed1 DOUBLE PRECISION for dlaed1. Arrays: *d*(*) contains the eigenvalues of the rank-1-perturbed matrix. The dimension of *d* must be at least max(1, *n*). *q*(*ldq*, *) contains the eigenvectors of the rank-1-perturbed matrix. The second dimension of *q* must be at least max(1, *n*). *work*(*) is a workspace array, dimension at least ($4n+n^2$).
ldq	INTEGER. The first dimension of the array *q*; *ldq* ≥ max(1, *n*).
indxq	INTEGER. Array, dimension (*n*). On entry, the permutation which separately sorts the two subproblems in *d* into ascending order.
rho	REAL for slaed1 DOUBLE PRECISION for dlaed1. The subdiagonal entry used to create the rank-1 modification.
cutpnt	INTEGER. The location of the last eigenvalue in the leading sub-matrix. min(1,*n*) ≤ *cutpnt* ≤ *n*/2.
iwork	INTEGER. Workspace array, dimension (4*n*).

Output Parameters

d	On exit, contains the eigenvalues of the repaired matrix.
q	On exit, *q* contains the eigenvectors of the repaired tridiagonal matrix.

indxq	On exit, contains the permutation which will reintegrate the subproblems back into sorted order, that is, *d(indxq*(i = 1, *n*)) will be in ascending order.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = 1, an eigenvalue did not converge.

?laed2

Used by sstedc/dstedc*. Merges eigenvalues and deflates secular equation. Used when the original matrix is tridiagonal.*

Syntax

```
call slaed2( k, n, n1, d, q, ldq, indxq, rho, z, dlamda, w, q2, indx, indxc,
indxp, coltyp, info )
```

```
call dlaed2( k, n, n1, d, q, ldq, indxq, rho, z, dlamda, w, q2, indx, indxc,
indxp, coltyp, info )
```

Description

The routine ?laed2 merges the two sets of eigenvalues together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more eigenvalues are close together or if there is a tiny entry in the *z* vector. For each such occurrence the order of the related secular equation problem is reduced by one.

Input Parameters

k	INTEGER. The number of non-deflated eigenvalues, and the order of the related secular equation ($0 \le k \le n$).
n	INTEGER. The dimension of the symmetric tridiagonal matrix ($n \ge 0$).
n1	INTEGER. The location of the last eigenvalue in the leading sub-matrix; $\min(1,n) \le n1 \le n/2$.
d, *q*, *z*	REAL for slaed2 DOUBLE PRECISION for dlaed2.

Arrays:
d(*) contains the eigenvalues of the two submatrices to be combined. The dimension of *d* must be at least max(1, *n*).
q(*ldq*, *) contains the eigenvectors of the two submatrices in the two square blocks with corners at (1,1), (*n1*,*n1*) and (*n1*+1,*n1*+1), (*n*,*n*). The second dimension of *q* must be at least max(1, *n*).
z(*) contains the updating vector (the last row of the first sub-eigenvector matrix and the first row of the second sub-eigenvector matrix).

ldq — INTEGER. The first dimension of the array *q*; *ldq* ≥ max(1, *n*).

indxq — INTEGER. Array, dimension (*n*).
On entry, the permutation which separately sorts the two subproblems in *d* into ascending order. Note that elements in the second half of this permutation must first have *n1* added to their values.

rho — REAL for slaed2
DOUBLE PRECISION for dlaed2.
On entry, the off-diagonal element associated with the rank-1 cut which originally split the two submatrices which are now being recombined.

indx, *indxp* — INTEGER.
Workspace arrays, dimension (*n*) each. Array *indx* contains the permutation used to sort the contents of *dlamda* into ascending order.
Array *indxp* contains the permutation used to place deflated values of *d* at the end of the array.
indxp(1:*k*) points to the nondeflated *d*-values and *indxp*(*k*+1:*n*) points to the deflated eigenvalues.

coltyp — INTEGER.
Workspace array, dimension (*n*).
During execution, a label which will indicate which of the following types a column in the *q2* matrix is:
1 : non-zero in the upper half only;
2 : dense;
3 : non-zero in the lower half only;

4 : deflated.

Output Parameters

d — On exit, *d* contains the trailing (*n*-*k*) updated eigenvalues (those which were deflated) sorted into increasing order.

q — On exit, *q* contains the trailing (*n*-*k*) updated eigenvectors (those which were deflated) in its last *n*-*k* columns.

indxq — Destroyed on exit.

rho — On exit, *rho* has been modified to the value required by ?laed3.

dlamda, *w*, *q2* — REAL for slaed2
DOUBLE PRECISION for dlaed2.
Arrays: *dlamda*(*n*), *w*(*n*), *q2*($n1^2$+(*n*-*n1*)^2).
The array *dlamda* contains a copy of the first *k* eigenvalues which is used by ?laed3 to form the secular equation.
The array *w* contains the first *k* values of the final deflation-altered *z*-vector which is passed to ?laed3.
The array *q2* contains a copy of the first *k* eigenvectors which is used by ?laed3 in a matrix multiply (sgemm/dgemm) to solve for the new eigenvectors.

indxc — INTEGER. Array, dimension (*n*).
The permutation used to arrange the columns of the deflated *q* matrix into three groups: the first group contains non-zero elements only at and above *n1*, the second contains non-zero elements only below *n1*, and the third is dense.

coltyp — On exit, *coltyp*(i) is the number of columns of type i, for i=1 to 4 only (see the definition of types in the description of *coltyp* in Input Parameters).

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

?laed3

Used by sstedc/dstedc*. Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is tridiagonal.*

Syntax

```
call slaed3( k, n, n1, d, q, ldq, rho, dlamda, q2, indx, ctot, w, s, info )
call dlaed3( k, n, n1, d, q, ldq, rho, dlamda, q2, indx, ctot, w, s, info )
```

Description

The routine ?laed3 finds the roots of the secular equation, as defined by the values in *d*, *w*, and *rho*, between 1 and *k*.

It makes the appropriate calls to ?laed4 and then updates the eigenvectors by multiplying the matrix of eigenvectors of the pair of eigensystems being combined by the matrix of eigenvectors of the *k*-by-*k* system which is solved here.

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but none are known.

Input Parameters

k	INTEGER. The number of terms in the rational function to be solved by ?laed4 ($k \geq 0$).
n	INTEGER. The number of rows and columns in the *q* matrix. $n \geq k$ (deflation may result in $n > k$).
n1	INTEGER. The location of the last eigenvalue in the leading sub-matrix; $\min(1,n) \leq n1 \leq n/2$.
q	REAL for slaed3 DOUBLE PRECISION for dlaed3. Array *q*(*ldq*, *). The second dimension of *q* must be at least max(1, *n*). Initially, the first *k* columns of this array are used as workspace.

ldq — INTEGER. The first dimension of the array *q*; *ldq* ≥ max(1, *n*).

rho — REAL for slaed3
DOUBLE PRECISION for dlaed3.
The value of the parameter in the rank one update equation.
rho ≥ 0 required.

dlamda, *q2*, *w* — REAL for slaed3
DOUBLE PRECISION for dlaed3.
Arrays: *dlamda*(*k*), *q2*(*ldq2*, *), *w*(*k*).
The first *k* elements of the array *dlamda* contain the old roots of the deflated updating problem. These are the poles of the secular equation.
The first *k* columns of the array *q2* contain the non-deflated eigenvectors for the split problem. The second dimension of *q2* must be at least max(1, *n*).
The first *k* elements of the array *w* contain the components of the deflation-adjusted updating vector.

indx — INTEGER. Array, dimension (*n*).
The permutation used to arrange the columns of the deflated *q* matrix into three groups (see ?laed2).
The rows of the eigenvectors found by ?laed4 must be likewise permuted before the matrix multiply can take place.

ctot — INTEGER. Array, dimension (4).
A count of the total number of the various types of columns in *q*, as described in *indx*. The fourth column type is any column which has been deflated.

s — REAL for slaed3
DOUBLE PRECISION for dlaed3.
Workspace array, dimension (*n1*+1)**k* .
Will contain the eigenvectors of the repaired matrix which will be multiplied by the previously accumulated eigenvectors to update the system.

Output Parameters

d — REAL for slaed3
DOUBLE PRECISION for dlaed3.

	Array, dimension at least max(1, *n*). *d*(i) contains the updated eigenvalues for 1 ≤ i ≤ *k*.
q	On exit, the columns 1 to *k* of *q* contain the updated eigenvectors.
dlamda	May be changed on output by having lowest order bit set to zero on Cray X-MP, Cray Y-MP, Cray-2, or Cray C-90, as described above.
w	Destroyed on exit.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = 1, an eigenvalue did not converge.

?laed4

Used by sstedc/dstedc. *Finds a single root of the secular equation.*

Syntax

```
call slaed4( n, i, d, z, delta, rho, dlam, info )
call dlaed4( n, i, d, z, delta, rho, dlam, info )
```

Description

This subroutine computes the *i*-th updated eigenvalue of a symmetric rank-one modification to a diagonal matrix whose elements are given in the array *d*, and that

D(i) < *D*(*j*) for *i* < *j*

and that *rho* > 0. This is arranged by the calling routine, and is no loss in generality. The rank-one modified system is thus

diag(*D*) + *rho***Z* * transpose(*Z*).

where we assume the Euclidean norm of *Z* is 1.

The method consists of approximating the rational functions in the secular equation by simpler interpolating rational functions.

Input Parameters

n	INTEGER. The length of all arrays.
i	INTEGER. The index of the eigenvalue to be computed; 1 ≤ i ≤ *n*.
d, *z*	REAL for slaed4 DOUBLE PRECISION for dlaed4 Arrays, dimension (*n*) each. The array *d* contains the original eigenvalues. It is assumed that they are in order, *d*(i) < *d*(*j*) for *i* < *j*. The array *z* contains the components of the updating vector *Z*.
rho	REAL for slaed4 DOUBLE PRECISION for dlaed4 The scalar in the symmetric updating formula.

Output Parameters

delta	REAL for slaed4 DOUBLE PRECISION for dlaed4 Array, dimension (*n*). If *n* ≠ 1, *delta* contains (*d*(*j*) – *lambda_i*) in its *j*-th component. If *n* = 1, then *delta*(1) = 1. The vector *delta* contains the information necessary to construct the eigenvectors.
dlam	REAL for slaed4 DOUBLE PRECISION for dlaed4 The computed *lambda_i*, the *i*-th updated eigenvalue.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = 1, the updating process failed.

?laed5

Used by sstedc/dstedc. *Solves the 2-by-2 secular equation.*

Syntax

```
call slaed5( i, d, z, delta, rho, dlam )
call dlaed5( i, d, z, delta, rho, dlam )
```

Description

This subroutine computes the *i*-th eigenvalue of a symmetric rank-one modification of a 2-by-2 diagonal matrix

```
diag(D) + rho*Z * transpose(Z).
```

The diagonal elements in the array *D* are assumed to satisfy

```
D(i) < D(j) for i < j .
```

We also assume *rho* > 0 and that the Euclidean norm of the vector *Z* is one.

Input Parameters

i	INTEGER. The index of the eigenvalue to be computed; $1 \le i \le 2$.
d, *z*	REAL for slaed5 DOUBLE PRECISION for dlaed5 Arrays, dimension (2) each. The array *d* contains the original eigenvalues. It is assumed that *d*(1) < *d*(2). The array *z* contains the components of the updating vector.
rho	REAL for slaed5 DOUBLE PRECISION for dlaed5 The scalar in the symmetric updating formula.

Output Parameters

delta	REAL for slaed5 DOUBLE PRECISION for dlaed5 Array, dimension (2).

The vector *delta* contains the information necessary to construct the eigenvectors.

dlam — REAL for slaed5
DOUBLE PRECISION for dlaed5
The computed *lambda_i*, the *i*-th updated eigenvalue.

?laed6

Used by sstedc/dstedc*. Computes one Newton step in solution of the secular equation.*

Syntax

```
call slaed6( kniter, orgati, rho, d, z, finit, tau, info )
call dlaed6( kniter, orgati, rho, d, z, finit, tau, info )
```

Description

This routine computes the positive or negative root (closest to the origin) of

$$f(x) = rho + \frac{z(1)}{d(1) - x} + \frac{z(2)}{d(2) - x} + \frac{z(3)}{d(3) - x}$$

It is assumed that if *orgati* = .TRUE. the root is between *d*(2) and *d*(3);otherwise it is between *d*(1) and *d*(2) This routine is called by ?laed4 when necessary. In most cases, the root sought is the smallest in magnitude, though it might not be in some extremely rare situations.

Input Parameters

kniter — INTEGER.
Refer to ?laed4 for its significance.

orgati — LOGICAL.
If *orgati* = .TRUE., the needed root is between *d*(2) and *d*(3); otherwise it is between *d*(1) and *d*(2). See ?laed4 for further details.

rho — REAL for slaed6
DOUBLE PRECISION for dlaed6

	Refer to the equation for $f(x)$ above.
d, *z*	REAL for slaed6 DOUBLE PRECISION for dlaed6 Arrays, dimension (3) each. The array *d* satisfies *d*(1) < *d*(2) < *d*(3). Each of the elements in the array *z* must be positive.
finit	REAL for slaed6 DOUBLE PRECISION for dlaed6 The value of $f(x)$ at 0. It is more accurate than the one evaluated inside this routine (if someone wants to do so).

Output Parameters

tau	REAL for slaed6 DOUBLE PRECISION for dlaed6 The root of the equation for f(x).
info	INTEGER. If *info* = 0, the execution is successful. If *info* = 1, failure to converge.

?laed7

Used by ?stedc*. Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is dense.*

Syntax

```
call slaed7( icompq, n, qsiz, tlvls, curlvl, curpbm, d, q, ldq, indxq, rho,
cutpnt, qstore, qptr, prmptr, perm, givptr, givcol, givnum, work, iwork, info
)
```

```
call dlaed7( icompq, n, qsiz, tlvls, curlvl, curpbm, d, q, ldq, indxq, rho,
cutpnt, qstore, qptr, prmptr, perm, givptr, givcol, givnum, work, iwork, info
)
```

```
call claed7( n, cutpnt, qsiz, tlvls, curlvl, curpbm, d, q, ldq, rho, indxq,
qstore, qptr, prmptr, perm, givptr, givcol, givnum, work, rwork, iwork, info
)
```

```
call zlaed7( n, cutpnt, qsiz, tlvls, curlvl, curpbm, d, q, ldq, rho, indxq,
qstore, qptr, prmptr, perm, givptr, givcol, givnum, work, rwork, iwork, info
)
```

Description

The routine ?laed7 computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. This routine is used only for the eigenproblem which requires all eigenvalues and optionally eigenvectors of a dense symmetric/Hermitian matrix that has been reduced to tridiagonal form. For real flavors, slaed1/dlaed1 handles the case in which all eigenvalues and eigenvectors of a symmetric tridiagonal matrix are desired.

T = *Q*(in)*(*D*(in)+*rho***Z***Z*')**Q*'(in) = *Q*(out)**D*(out)**Q*'(out)

where *Z* = *Q*'**u*, *u* is a vector of length *n* with ones in the *cutpnt* and (*cutpnt* + 1) -th elements and zeros elsewhere. The eigenvectors of the original matrix are stored in *Q*, and the eigenvalues are in *D*. The algorithm consists of three stages:

The first stage consists of deflating the size of the problem when there are multiple eigenvalues or if there is a zero in the *z* vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine slaed8/dlaed8 (for real flavors) or by the routine slaed2/dlaed2 (for complex flavors).

The second stage consists of calculating the updated eigenvalues. This is done by finding the roots of the secular equation via the routine ?laed4 (as called by ?laed9 or ?laed3). This routine also calculates the eigenvectors of the current problem.

The final stage consists of computing the updated eigenvectors directly using the updated eigenvalues. The eigenvectors for the current problem are multiplied with the eigenvectors from the overall problem.

Input Parameters

icompq	INTEGER. Used with real flavors only. If *icompq* = 0, compute eigenvalues only. If *icompq* = 1, compute eigenvectors of original dense symmetric matrix also. On entry, the array *q* must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.
n	INTEGER. The dimension of the symmetric tridiagonal matrix (*n* ≥ 0).
cutpnt	INTEGER. The location of the last eigenvalue in the leading sub-matrix. min(1,*n*) ≤ *cutpnt* ≤ *n* .
qsiz	INTEGER. The dimension of the orthogonal/unitary matrix used to reduce the full matrix to tridiagonal form; *qsiz* ≥ *n* (for real flavors, *qsiz* ≥ *n* if *icompq* = 1).
tlvls	INTEGER. The total number of merging levels in the overall divide and conquer tree.
curlvl	INTEGER. The current level in the overall merge routine, 0 ≤ *curlvl* ≤ *tlvls* .
curpbm	INTEGER. The current problem in the current level in the overall merge routine (counting from upper left to lower right).
d	REAL for slaed7/claed7 DOUBLE PRECISION for dlaed7/zlaed7. Array, dimension at least max(1, *n*). Array *d*(*) contains the eigenvalues of the rank-1-perturbed matrix.

q, *work*	REAL for slaed7 DOUBLE PRECISION for dlaed7 COMPLEX for claed7 COMPLEX*16 for zlaed7. Arrays: *q*(*ldq*, *) contains the the eigenvectors of the rank-1-perturbed matrix. The second dimension of *q* must be at least max(1, *n*). *work*(*) is a workspace array, dimension at least (3*n*+*qsiz***n*) for real flavors and at least (*qsiz***n*) for complex flavors.
ldq	INTEGER. The first dimension of the array *q*; *ldq* ≥ max(1, *n*).
rho	REAL for slaed7 /claed7 DOUBLE PRECISION for dlaed7/zlaed7. The subdiagonal element used to create the rank-1 modification.
qstore	REAL for slaed7/claed7 DOUBLE PRECISION for dlaed7/zlaed7. Array, dimension (n^2+1). Serves also as output parameter. Stores eigenvectors of submatrices encountered during divide and conquer, packed together. *qptr* points to beginning of the submatrices.
qptr	INTEGER. Array, dimension (*n*+2). Serves also as output parameter. List of indices pointing to beginning of submatrices stored in *qstore*. The submatrices are numbered starting at the bottom left of the divide and conquer tree, from left to right and bottom to top.
prmptr, *perm*, *givptr*	INTEGER. Arrays, dimension (*n* lg*n*) each. The array *prmptr*(*) contains a list of pointers which indicate where in *perm* a level's permutation is stored. *prmptr*(i+1) – *prmptr*(i) indicates the size of the permutation and also the size of the full, non-deflated problem. The array *perm*(*) contains the permutations (from deflation and sorting) to be applied to each eigenblock.

	The array *givptr*(*) contains a list of pointers which indicate where in *givcol* a level's Givens rotations are stored. *givptr*(i+1) - *givptr*(i) indicates the number of Givens rotations.
givcol	INTEGER. Array, dimension (2, $n \lg n$). Each pair of numbers indicates a pair of columns to take place in a Givens rotation.
givnum	REAL for slaed7/claed7 DOUBLE PRECISION for dlaed7/zlaed7. Array, dimension (2, $n \lg n$). Each number indicates the S value to be used in the corresponding Givens rotation.
iwork	INTEGER. Workspace array, dimension ($4n$).
rwork	REAL for claed7 DOUBLE PRECISION for zlaed7. Workspace array, dimension ($3n+2qsiz*n$). Used in complex flavors only.

Output Parameters

d	On exit, contains the eigenvalues of the repaired matrix.
q	On exit, *q* contains the eigenvectors of the repaired tridiagonal matrix.
indxq	INTEGER. Array, dimension (n). Contains the permutation which will reintegrate the subproblems back into sorted order, that is, *d*(*indxq*(i = 1, *n*)) will be in ascending order.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = 1, an eigenvalue did not converge.

?laed8

Used by ?stedc*. Merges eigenvalues and deflates secular equation. Used when the original matrix is dense.*

Syntax

call slaed8(*icompq, k, n, qsiz, d, q, ldq, indxq, rho, cutpnt, z, dlamda, q2, ldq2, w, perm, givptr, givcol, givnum, indxp, indx, info*)

call dlaed8(*icompq, k, n, qsiz, d, q, ldq, indxq, rho, cutpnt, z, dlamda, q2, ldq2, w, perm, givptr, givcol, givnum, indxp, indx, info*)

call claed8(*k, n, qsiz, q, ldq, d, rho, cutpnt, z, dlamda, q2, ldq2, w, indxp, indx, indxq, perm, givptr, givcol, givnum, info*)

call zlaed8(*k, n, qsiz, q, ldq, d, rho, cutpnt, z, dlamda, q2, ldq2, w, indxp, indx, indxq, perm, givptr, givcol, givnum, info*)

Description

This routine merges the two sets of eigenvalues together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more eigenvalues are close together or if there is a tiny element in the *z* vector. For each such occurrence the order of the related secular equation problem is reduced by one.

Input Parameters

icompq — INTEGER. Used with real flavors only.
If *icompq* = 0, compute eigenvalues only.
If *icompq* = 1, compute eigenvectors of original dense symmetric matrix also.
On entry, the array *q* must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.

n — INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).

cutpnt — INTEGER. The location of the last eigenvalue in the leading sub-matrix. $\min(1,n) \leq cutpnt \leq n$.

qsiz — INTEGER.

	The dimension of the orthogonal/unitary matrix used to reduce the full matrix to tridiagonal form; *qsiz* ≥ *n* (for real flavors, *qsiz* ≥ *n* if *icompq* = 1).
d, *z*	REAL for slaed8/claed8 DOUBLE PRECISION for dlaed8/zlaed8. Arrays, dimension at least max(1, *n*) each. The array *d*(*) contains the eigenvalues of the two submatrices to be combined. On entry, *z*(*) contains the updating vector (the last row of the first sub-eigenvector matrix and the first row of the second sub-eigenvector matrix). The contents of *z* are destroyed by the updating process.
q	REAL for slaed8 DOUBLE PRECISION for dlaed8 COMPLEX for claed8 COMPLEX*16 for zlaed8. Array *q*(*ldq*, *). The second dimension of *q* must be at least max(1, *n*). On entry, *q* contains the eigenvectors of the partially solved system which has been previously updated in matrix multiplies with other partially solved eigensystems. For real flavors, If *icompq* = 0, *q* is not referenced.
ldq	INTEGER. The first dimension of the array *q*; *ldq* ≥ max(1, *n*).
ldq2	INTEGER. The first dimension of the output array *q2*; *ldq2* ≥ max(1, *n*).
indxq	INTEGER. Array, dimension (*n*). The permutation which separately sorts the two sub-problems in *d* into ascending order. Note that elements in the second half of this permutation must first have *cutpnt* added to their values in order to be accurate.
rho	REAL for slaed8/claed8 DOUBLE PRECISION for dlaed8/zlaed8. On entry, the off-diagonal element associated with the rank-1 cut which originally split the two submatrices which are now being recombined.

Output Parameters

k	INTEGER. The number of non-deflated eigenvalues, and the order of the related secular equation.
d	On exit, contains the trailing (*n*-*k*) updated eigenvalues (those which were deflated) sorted into increasing order.
q	On exit, *q* contains the trailing (*n*-*k*) updated eigenvectors (those which were deflated) in its last (*n*-*k*) columns.
rho	On exit, *rho* has been modified to the value required by ?laed3.
dlamda, *w*	REAL for slaed8/claed8 DOUBLE PRECISION for dlaed8/zlaed8. Arrays, dimension (*n*) each. The array *dlamda*(*) contains a copy of the first *k* eigenvalues which will be used by ?laed3 to form the secular equation. The array *w*(*) will hold the first *k* values of the final deflation-altered *z*-vector and will be passed to ?laed3.
q2	REAL for slaed8 DOUBLE PRECISION for dlaed8 COMPLEX for claed8 COMPLEX*16 for zlaed8. Array *q2*(*ldq2*, *). The second dimension of *q2* must be at least max(1, *n*). Contains a copy of the first *k* eigenvectors which will be used by slaed7/dlaed7 in a matrix multiply (sgemm/dgemm) to update the new eigenvectors. For real flavors, If *icompq* = 0, *q2* is not referenced.
indxp, *indx*	INTEGER. Workspace arrays, dimension (*n*) each. The array *indxp*(*) will contain the permutation used to place deflated values of *d* at the end of the array. On output, *indxp*(1:*k*) points to the nondeflated *d*-values and *indxp*(*k*+1:*n*) points to the deflated eigenvalues. The array *indx*(*) will contain the permutation used to sort the contents of *d* into ascending order.
perm	INTEGER. Array, dimension (*n*).

	Contains the permutations (from deflation and sorting) to be applied to each eigenblock.
givptr	INTEGER. Contains the number of Givens rotations which took place in this subproblem.
givcol	INTEGER. Array, dimension (2, *n*). Each pair of numbers indicates a pair of columns to take place in a Givens rotation.
givnum	REAL for slaed8/claed8 DOUBLE PRECISION for dlaed8/zlaed8. Array, dimension (2, *n*). Each number indicates the *S* value to be used in the corresponding Givens rotation.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

?laed9

Used by sstedc/dstedc*. Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is dense.*

Syntax

```
call slaed9( k, kstart, kstop, n, d, q, ldq, rho, dlamda, w, s, lds, info )
call dlaed9( k, kstart, kstop, n, d, q, ldq, rho, dlamda, w, s, lds, info )
```

Description

This routine finds the roots of the secular equation, as defined by the values in *d*, *z*, and *rho*, between *kstart* and *kstop*. It makes the appropriate calls to slaed4/dlaed4 and then stores the new matrix of eigenvectors for use in calculating the next level of *z* vectors.

Input Parameters

k	INTEGER. The number of terms in the rational function to be solved by slaed4/dlaed4 (*k* ≥ 0).
kstart, *kstop*	INTEGER. The updated eigenvalues *lambda*(i),

kstart ≤ i ≤ *kstop* are to be computed.

1 ≤ *kstart* ≤ *kstop* ≤ *k*.

n INTEGER. The number of rows and columns in the *Q* matrix. *n* ≥ *k* (deflation may result in *n* > *k*).

q REAL for slaed9
DOUBLE PRECISION for dlaed9.
Workspace array, dimension (*ldq*, *).
The second dimension of *q* must be at least max(1, *n*).

ldq INTEGER. The first dimension of the array *q*; *ldq* ≥ max(1, *n*).

rho REAL for slaed9
DOUBLE PRECISION for dlaed9
The value of the parameter in the rank one update equation. *rho* ≥ 0 required.

dlamda, *w* REAL for slaed9
DOUBLE PRECISION for dlaed9
Arrays, dimension (*k*) each. The first *k* elements of the array *dlamda*(*) contain the old roots of the deflated updating problem. These are the poles of the secular equation.
The first *k* elements of the array *w*(*) contain the components of the deflation-adjusted updating vector.

lds INTEGER. The first dimension of the output array *s*; *lds* ≥ max(1, *k*).

Output Parameters

d REAL for slaed9
DOUBLE PRECISION for dlaed9
Array, dimension (*n*). *d*(i) contains the updated eigenvalues for *kstart* ≤ i ≤ *kstop*.

s REAL for slaed9
DOUBLE PRECISION for dlaed9.
Array, dimension (*lds*, *) .

	The second dimension of *s* must be at least max(1, *k*). Will contain the eigenvectors of the repaired matrix which will be stored for subsequent *z* vector calculation and multiplied by the previously accumulated eigenvectors to update the system.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = 1, the eigenvalue did not converge.

?laeda

Used by ?stedc. *Computes the Z vector determining the rank-one modification of the diagonal matrix. Used when the original matrix is dense.*

Syntax

call slaeda(*n*, *tlvls*, *curlvl*, *curpbm*, *prmptr*, *perm*, *givptr*, *givcol*, *givnum*, *q*, *qptr*, *z*, *ztemp*, *info*)

call dlaeda(*n*, *tlvls*, *curlvl*, *curpbm*, *prmptr*, *perm*, *givptr*, *givcol*, *givnum*, *q*, *qptr*, *z*, *ztemp*, *info*)

Description

The routine ?laeda computes the *Z* vector corresponding to the merge step in the *curlvl*-th step of the merge process with *tlvls* steps for the *curpbm*-th problem.

Input Parameters

n	INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).
tlvls	INTEGER. The total number of merging levels in the overall divide and conquer tree.
curlvl	INTEGER. The current level in the overall merge routine, $0 \leq curlvl \leq tlvls$.

curpbm — INTEGER. The current problem in the current level in the overall merge routine (counting from upper left to lower right).

prmptr, *perm*, *givptr* — INTEGER. Arrays, dimension ($n \lg n$) each.
The array *prmptr*(*) contains a list of pointers which indicate where in *perm* a level's permutation is stored. *prmptr*(i+1) – *prmptr*(i) indicates the size of the permutation and also the size of the full, non-deflated problem.
The array *perm*(*) contains the permutations (from deflation and sorting) to be applied to each eigenblock.
The array *givptr*(*) contains a list of pointers which indicate where in *givcol* a level's Givens rotations are stored. *givptr*(i+1) – *givptr*(i) indicates the number of Givens rotations.

givcol — INTEGER. Array, dimension (2, $n \lg n$).
Each pair of numbers indicates a pair of columns to take place in a Givens rotation.

givnum — REAL for slaeda
DOUBLE PRECISION for dlaeda.
Array, dimension (2, $n \lg n$).
Each number indicates the s value to be used in the corresponding Givens rotation.

q — REAL for slaeda
DOUBLE PRECISION for dlaeda.
Array, dimension (n^2).
Contains the square eigenblocks from previous levels, the starting positions for blocks are given by *qptr*.

qptr — INTEGER. Array, dimension (n+2). Contains a list of pointers which indicate where in *q* an eigenblock is stored. sqrt(*qptr*(i+1) – *qptr*(i)) indicates the size of the block.

ztemp — REAL for slaeda
DOUBLE PRECISION for dlaeda.
Workspace array, dimension (n).

Output Parameters

z — REAL for slaeda

	DOUBLE PRECISION for dlaeda. Array, dimension (*n*). Contains the updating vector (the last row of the first sub-eigenvector matrix and the first row of the second sub-eigenvector matrix).
info	INTEGER. If *info* = 0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

?laein

Computes a specified right or left eigenvector of an upper Hessenberg matrix by inverse iteration.

Syntax

call slaein(*rightv*, *noinit*, *n*, *h*, *ldh*, *wr*, *wi*, *vr*, *vi*, *b*, *ldb*, *work*, *eps3*, *smlnum*, *bignum*, *info*)

call dlaein(*rightv*, *noinit*, *n*, *h*, *ldh*, *wr*, *wi*, *vr*, *vi*, *b*, *ldb*, *work*, *eps3*, *smlnum*, *bignum*, *info*)

call claein(*rightv*, *noinit*, *n*, *h*, *ldh*, *w*, *v*, *b*, *ldb*, *rwork*, *eps3*, *smlnum*, *info*)

call zlaein(*rightv*, *noinit*, *n*, *h*, *ldh*, *w*, *v*, *b*, *ldb*, *rwork*, *eps3*, *smlnum*, *info*)

Description

The routine ?laein uses inverse iteration to find a right or left eigenvector corresponding to the eigenvalue (*wr*,*wi*) of a real upper Hessenberg matrix *H* (for real flavors slaein/dlaein) or to the eigenvalue *w* of a complex upper Hessenberg matrix *H* (for complex flavors claein/zlaein).

Input Parameters

rightv	LOGICAL. If *rightv* = .TRUE., compute right eigenvector; if *rightv* = .FALSE., compute left eigenvector.
noinit	LOGICAL.

If *noinit* = .TRUE., no initial vector is supplied in (*vr*,*vi*) or in *v* (for complex flavors);
if *noinit* = .FALSE., initial vector is supplied in (*vr*,*vi*) or in *v* (for complex flavors).

n — INTEGER. The order of the matrix *H* (*n* ≥ 0).

h — REAL for slaein
DOUBLE PRECISION for dlaein
COMPLEX for claein
COMPLEX*16 for zlaein.
Array *h*(*ldh*, *).
The second dimension of *h* must be at least max(1, *n*).
Contains the upper Hessenberg matrix *H*.

ldh — INTEGER. The first dimension of the array *h*; *ldh* ≥ max(1, *n*).

wr, *wi* — REAL for slaein
DOUBLE PRECISION for dlaein.
The real and imaginary parts of the eigenvalue of *H* whose corresponding right or left eigenvector is to be computed (for real flavors of the routine).

w — COMPLEX for claein
COMPLEX*16 for zlaein.
The eigenvalue of *H* whose corresponding right or left eigenvector is to be computed (for complex flavors of the routine).

vr, *vi* — REAL for slaein
DOUBLE PRECISION for dlaein.
Arrays, dimension (*n*) each. Used for real flavors only. On entry, if *noinit* = .FALSE. and *wi* = 0.0, *vr* must contain a real starting vector for inverse iteration using the real eigenvalue *wr*;

if *noinit* = .FALSE. and *wi* ≠ 0.0, *vr* and *vi* must contain the real and imaginary parts of a complex starting vector for inverse iteration using the complex eigenvalue (*wr*,*wi*);otherwise *vr* and *vi* need not be set.

v — COMPLEX for claein
COMPLEX*16 for zlaein.

	Array, dimension (*n*). Used for complex flavors only. On entry, if *noinit* = .FALSE., *v* must contain a starting vector for inverse iteration; otherwise *v* need not be set.
b	REAL for slaein DOUBLE PRECISION for dlaein COMPLEX for claein COMPLEX*16 for zlaein. Workspace array *b*(*ldb*, *). The second dimension of *b* must be at least max(1, *n*).
ldb	INTEGER. The first dimension of the array *b*; *ldb* ≥ *n*+1 for real flavors; *ldb* ≥ max(1, *n*) for complex flavors.
work	REAL for slaein DOUBLE PRECISION for dlaein. Workspace array, dimension (*n*). Used for real flavors only.
rwork	REAL for claein DOUBLE PRECISION for zlaein. Workspace array, dimension (*n*). Used for complex flavors only.
eps3, *smlnum*	REAL for slaein/claein DOUBLE PRECISION for dlaein/zlaein. *eps3* is a small machine-dependent value which is used to perturb close eigenvalues, and to replace zero pivots. *smlnum* is a machine-dependent value close to underflow threshold.
bignum	REAL for slaein DOUBLE PRECISION for dlaein. *bignum* is a machine-dependent value close to overflow threshold. Used for real flavors only.

Output Parameters

vr, *vi*	On exit, if *wi* = 0.0 (real eigenvalue), *vr* contains the computed real eigenvector; if *wi* ≠ 0.0 (complex eigenvalue), *vr* and *vi* contain the real and imaginary parts

of the computed complex eigenvector. The eigenvector is normalized so that the component of largest magnitude has magnitude 1; here the magnitude of a complex number (*x*,*y*) is taken to be |*x*| + |*y*|.
vi is not referenced if *wi* = 0.0.

v — On exit, *v* contains the computed eigenvector, normalized so that the component of largest magnitude has magnitude 1; here the magnitude of a complex number (*x*,*y*) is taken to be |*x*| + |*y*|.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = 1, inverse iteration did not converge. For real flavors, *vr* is set to the last iterate, and so is *vi*, if *wi* ≠ 0.0. For complex flavors, *v* is set to the last iterate.

?laev2

Computes the eigenvalues and eigenvectors of a 2-by-2 symmetric/Hermitian matrix.

Syntax

```
call slaev2( a, b, c, rt1, rt2, cs1, sn1 )
call dlaev2( a, b, c, rt1, rt2, cs1, sn1 )
call claev2( a, b, c, rt1, rt2, cs1, sn1 )
call zlaev2( a, b, c, rt1, rt2, cs1, sn1 )
```

Description

This routine performs the eigendecomposition of a 2-by-2 symmetric matrix

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix}$$ (for *slaev2/dlaev2*) or Hermitian matrix $$\begin{bmatrix} a & b \\ \text{conjg}(b) & c \end{bmatrix}$$

(for claev2/zlaev2).

On return, *rt1* is the eigenvalue of larger absolute value, *rt2* of smaller absolute value, and (*cs1*, *sn1*) is the unit right eigenvector for *rt1*, giving the decomposition

$$\begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} \cdot \begin{bmatrix} a & b \\ b & c \end{bmatrix} \cdot \begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} = \begin{bmatrix} rt1 & 0 \\ 0 & rt2 \end{bmatrix}$$

(for `slaev2`/`dlaev2`),

or

$$\begin{bmatrix} cs1 & \mathbf{conjg}(sn1) \\ -sn1 & cs1 \end{bmatrix} \cdot \begin{bmatrix} a & b \\ \mathbf{conjg}(b) & c \end{bmatrix} \cdot \begin{bmatrix} cs1 & -\mathbf{conjg}(sn1) \\ sn1 & cs1 \end{bmatrix} = \begin{bmatrix} rt1 & 0 \\ 0 & rt2 \end{bmatrix}$$

(for `claev2`/`zlaev2`).

Input Parameters

a, *b*, *c*	`REAL` for `slaev2`
	`DOUBLE PRECISION` for `dlaev2`
	`COMPLEX` for `claev2`
	`COMPLEX*16` for `zlaev2`.
	Elements of the input matrix.

Output Parameters

rt1, *rt2*	`REAL` for `slaev2`/`claev2` `DOUBLE PRECISION` for `dlaev2`/`zlaev2`. Eigenvalues of larger and smaller absolute value, respectively.
cs1	`REAL` for `slaev2`/`claev2` `DOUBLE PRECISION` for `dlaev2`/`zlaev2`.

sn1	`REAL` for `slaev2`
	`DOUBLE PRECISION` for `dlaev2`
	`COMPLEX` for `claev2`

COMPLEX*16 for zlaev2.
The vector (*cs1*, *sn1*) is the unit right eigenvector for rt1.

Application Notes

rt1 is accurate to a few ulps barring over/underflow. *rt2* may be inaccurate if there is massive cancellation in the determinant *a*c-b*b*; higher precision or correctly rounded or correctly truncated arithmetic would be needed to compute *rt2* accurately in all cases. *cs1* and *sn1* are accurate to a few ulps barring over/underflow. Overflow is possible only if *rt1* is within a factor of 5 of overflow. Underflow is harmless if the input data is 0 or exceeds underflow_threshold / macheps.

?laexc

Swaps adjacent diagonal blocks of a real upper quasi-triangular matrix in Schur canonical form, by an orthogonal similarity transformation.

Syntax

```
call slaexc( wantq, n, t, ldt, q, ldq, j1, n1, n2, work, info )
call dlaexc( wantq, n, t, ldt, q, ldq, j1, n1, n2, work, info )
```

Description

This routine swaps adjacent diagonal blocks T_{11} and T_{22} of order 1 or 2 in an upper quasi-triangular matrix T by an orthogonal similarity transformation.

T must be in Schur canonical form, that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elements equal and its off-diagonal elements of opposite sign.

Input Parameters

wantq	LOGICAL. If *wantq* = .TRUE., accumulate the transformation in the matrix Q; If *wantq* = .FALSE., do not accumulate the transformation.
n	INTEGER. The order of the matrix T ($n \geq 0$).
t, *q*	REAL for slaexc

	DOUBLE PRECISION for dlaexc Arrays: *t*(*ldt*,*) contains on entry the upper quasi-triangular matrix *T*, in Schur canonical form. The second dimension of *t* must be at least max(1, *n*). *q*(*ldq*,*) contains on entry, if *wantq* = .TRUE., the orthogonal matrix *Q*. If *wantq* = .FALSE., *q* is not referenced. The second dimension of *q* must be at least max(1, *n*).
ldt	INTEGER. The first dimension of *t*; at least max(1, *n*).
ldq	INTEGER. The first dimension of *q*; If *wantq* = .FALSE., then *ldq* ≥ 1. If *wantq* = .TRUE., then *ldq* ≥ max(1,*n*).
j1	INTEGER. The index of the first row of the first block T_{11}.
n1	INTEGER. The order of the first block T_{11} (*n1* = 0, 1, or 2).
n2	INTEGER. The order of the second block T_{22} (*n2* = 0, 1, or 2).
work	REAL for slaexc; DOUBLE PRECISION for dlaexc. Workspace array, DIMENSION (*n*).

Output Parameters

t	On exit, the updated matrix *T*, again in Schur canonical form.
q	On exit, if *wantq* = .TRUE., the updated matrix *Q*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = 1, the transformed matrix *T* would be too far from Schur form; the blocks are not swapped and *T* and *Q* are unchanged.

?lag2

Computes the eigenvalues of a 2-by-2 generalized eigenvalue problem, with scaling as necessary to avoid over-/underflow.

Syntax

```
call slag2( a, lda, b, ldb, safmin, scale1, scale2, wr1, wr2, wi )
call dlag2( a, lda, b, ldb, safmin, scale1, scale2, wr1, wr2, wi )
```

Description

This routine computes the eigenvalues of a 2 x 2 generalized eigenvalue problem *A* - *w* **B*, with scaling as necessary to avoid over-/underflow. The scaling factor, *s*, results in a modified eigenvalue equation

*s***A* - *w***B* ,

where *s* is a non-negative scaling factor chosen so that *w*, *w***B*, and *s***A* do not overflow and, if possible, do not underflow, either.

Input Parameters

a, *b*	REAL for slag2 DOUBLE PRECISION for dlag2 Arrays: *a*(*lda*,2) contains, on entry, the 2 x 2 matrix *A*. It is assumed that its 1-norm is less than 1/*safmin*. Entries less than sqrt(*safmin*)*norm(*A*) are subject to being treated as zero. *b*(*ldb*,2) contains, on entry, the 2 x 2 upper triangular matrix *B*. It is assumed that the one-norm of *B* is less than 1/*safmin*. The diagonals should be at least sqrt(*safmin*) times the largest element of *B* (in absolute value); if a diagonal is smaller than that, then +/- sqrt(*safmin*) will be used instead of that diagonal.
lda	INTEGER. The first dimension of *a*; *lda* ≥ 2.
ldb	INTEGER. The first dimension of *b*; *ldb* ≥ 2.
safmin	REAL for slag2;

DOUBLE PRECISION for dlag2.
The smallest positive number such that 1/*safmin* does not overflow. (This should always be ?lamch('S') - it is an argument in order to avoid having to call ?lamch frequently.)

Output Parameters

scale1
REAL for slag2;
DOUBLE PRECISION for dlag2.
A scaling factor used to avoid over-/underflow in the eigenvalue equation which defines the first eigenvalue. If the eigenvalues are complex, then the eigenvalues are (*wr1* +/- *wii*)/*scale1* (which may lie outside the exponent range of the machine), *scale1*=*scale2*, and *scale1* will always be positive.
If the eigenvalues are real, then the first (real) eigenvalue is *wr1*/*scale1*, but this may overflow or underflow, and in fact, *scale1* may be zero or less than the underflow threshhold if the exact eigenvalue is sufficiently large.

scale2
REAL for slag2;
DOUBLE PRECISION for dlag2.
A scaling factor used to avoid over-/underflow in the eigenvalue equation which defines the second eigenvalue. If the eigenvalues are complex, then *scale2*=*scale1*. If the eigenvalues are real, then the second (real) eigenvalue is *wr2*/*scale2*, but this may overflow or underflow, and in fact, *scale2* may be zero or less than the underflow threshold if the exact eigenvalue is sufficiently large.

wr1
REAL for slag2;
DOUBLE PRECISION for dlag2.
If the eigenvalue is real, then *wr1* is *scale1* times the eigenvalue closest to the (2,2) element of *A*inv(*B*).
If the eigenvalue is complex, then *wr1*=*wr2* is *scale1* times the real part of the eigenvalues.

wr2
REAL for slag2;
DOUBLE PRECISION for dlag2.

If the eigenvalue is real, then *wr2* is *scale2* times the other eigenvalue. If the eigenvalue is complex, then *wr1*=*wr2* is *scale1* times the real part of the eigenvalues.

wi — REAL for slag2; DOUBLE PRECISION for dlag2.
If the eigenvalue is real, then *wi* is zero. If the eigenvalue is complex, then *wi* is *scale1* times the imaginary part of the eigenvalues. *wi* will always be non-negative.

?lags2

Computes 2-by-2 orthogonal matrices U, V, and Q, and applies them to matrices A and B such that the rows of the transformed A and B are parallel.

Syntax

```
call slags2( upper, a1, a2, a3, b1, b2, b3, csu, snu, csv, snv, csq, snq )
call dlags2( upper, a1, a2, a3, b1, b2, b3, csu, snu, csv, snv, csq, snq )
```

Description

This routine computes 2-by-2 orthogonal matrices *U*, *V* and *Q*, such that if *upper* = .TRUE., then

$$U'*A*Q = U'*\begin{bmatrix} A_1 & A_2 \\ 0 & A_3 \end{bmatrix}*Q = \begin{bmatrix} x & 0 \\ x & x \end{bmatrix}$$

and

$$V'*B*Q = U'*\begin{bmatrix} B_1 & B_2 \\ 0 & B_3 \end{bmatrix}*Q = \begin{bmatrix} x & 0 \\ x & x \end{bmatrix}$$

or if *upper* = .FALSE., then

$$U'*A*Q = U'*\begin{bmatrix} A_1 & 0 \\ A_2 & A_3 \end{bmatrix}*Q = \begin{bmatrix} x & x \\ 0 & x \end{bmatrix}$$

and

$$V'*B*Q = U'*\begin{bmatrix} B_1 & 0 \\ B_2 & B_3 \end{bmatrix}*Q = \begin{bmatrix} x & x \\ 0 & x \end{bmatrix}$$

The rows of the transformed *A* and *B* are parallel, where

$$U=\begin{bmatrix} csu & snu \\ -snu & csu \end{bmatrix}, V = \begin{bmatrix} csv & snv \\ -snv & csv \end{bmatrix}, Q = \begin{bmatrix} csq & snq \\ -snq & csq \end{bmatrix}$$

Here Z' denotes the transpose of Z.

Input Parameters

upper	LOGICAL. If *upper* = .TRUE., the input matrices *A* and *B* are upper triangular; If *upper* = .FALSE., the input matrices *A* and *B* are lower triangular.
a1, *a2*, *a3*	REAL for slags2 DOUBLE PRECISION for dlags2 On entry, *a1*, *a2* and *a3* are elements of the input 2-by-2 upper (lower) triangular matrix *A*.
b1, *b2*, *b3*	REAL for slags2 DOUBLE PRECISION for dlags2 On entry, *b1*, *b2* and *b3* are elements of the input 2-by-2 upper (lower) triangular matrix *B*.

Output Parameters

csu, *snu*	REAL for slags2 DOUBLE PRECISION for dlags2 The desired orthogonal matrix *U*.
csv, *snv*	REAL for slags2 DOUBLE PRECISION for dlags2 The desired orthogonal matrix *V*.
csq, *snq*	REAL for slags2 DOUBLE PRECISION for dlags2 The desired orthogonal matrix *Q*.

?lagtf

Computes an LU factorization of a matrix T-λI, where T is a general tridiagonal matrix, and λ a scalar, using partial pivoting with row interchanges.

Syntax

```
call slagtf( n, a, lambda, b, c, tol, d, in, info )
call dlagtf( n, a, lambda, b, c, tol, d, in, info )
```

Description

This routine factorizes the matrix (*T - lambda*I*), where *T* is an *n*-by-*n* tridiagonal matrix and *lambda* is a scalar, as

*T - lambda*I = P*L*U*,

where *p* is a permutation matrix, *L* is a unit lower tridiagonal matrix with at most one non-zero sub-diagonal elements per column and *U* is an upper triangular matrix with at most two non-zero super-diagonal elements per column. The factorization is obtained by Gaussian elimination with partial pivoting and implicit row scaling. The parameter *lambda* is included in the routine so that ?lagtf may be used, in conjunction with ?lagts, to obtain eigenvectors of *T* by inverse iteration.

Input Parameters

n	INTEGER. The order of the matrix *T* (*n* ≥ 0).

a, *b*, *c*	REAL for slagtf DOUBLE PRECISION for dlagtf Arrays, dimension *a*(*n*), *b*(*n*-1), *c*(*n*-1): On entry, *a*(*) must contain the diagonal elements of the matrix *T*. On entry, *b*(*) must contain the (*n*-1) super-diagonal elements of *T*. On entry, *c*(*) must contain the (*n*-1) sub-diagonal elements of *T*.
tol	REAL for slagtf DOUBLE PRECISION for dlagtf On entry, a relative tolerance used to indicate whether or not the matrix (*T* - *lambda***I*) is nearly singular. *tol* should normally be chose as approximately the largest relative error in the elements of *T*. For example, if the elements of *T* are correct to about 4 significant figures, then *tol* should be set to about $5*10^{-4}$. If *tol* is supplied as less than eps, where eps is the relative machine precision, then the value eps is used in place of *tol*.

Output Parameters

a	On exit, *a* is overwritten by the *n* diagonal elements of the upper triangular matrix *U* of the factorization of *T*.
b	On exit, *b* is overwritten by the *n*-1 super-diagonal elements of the matrix *U* of the factorization of *T*.
c	On exit, *c* is overwritten by the *n*-1 sub-diagonal elements of the matrix *L* of the factorization of *T*.
d	REAL for slagtf DOUBLE PRECISION for dlagtf Array, dimension (*n*-2). On exit, *d* is overwritten by the *n*-2 second super-diagonal elements of the matrix *U* of the factorization of *T*.
in	INTEGER. Array, dimension (*n*).

On exit, *in* contains details of the permutation matrix *p*. If an interchange occurred at the k-th step of the elimination, then *in*(k) = 1, otherwise *in*(k) = 0. The element *in*(*n*) returns the smallest positive integer j such that

abs(*u*(j,j)) ≤ norm((*T* - *lambda***I*)(j))**tol*,

where norm(*A*(j)) denotes the sum of the absolute values of the j-th row of the matrix *A*.
If no such j exists then *in*(*n*) is returned as zero. If *in*(*n*) is returned as positive, then a diagonal element of *U* is small, indicating that (*T* - *lambda***I*) is singular or nearly singular.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*k*, the *k*-th parameter had an illegal value.

?lagtm

*Performs a matrix-matrix product of the form C = alpha*A*B+beta*C, where A is a tridiagonal matrix, B and C are rectangular matrices, and alpha and beta are scalars, which may be 0, 1, or -1.*

Syntax

```
call slagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb )
call dlagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb )
call clagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb )
call zlagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb )
```

Description

This routine performs a matrix-vector product of the form:

*B := alpha*A*X + beta*B*

where *A* is a tridiagonal matrix of order *n*, *B* and *X* are *n*-by-*nrhs* matrices, and *alpha* and *beta* are real scalars, each of which may be 0., 1., or -1.

Input Parameters

trans	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If *trans* = 'N', then *B* := *alpha***A***X* + *beta***B* (no transpose); If *trans* = 'T', then *B* := *alpha***A*^{T}**X* + *beta***B* (transpose); If *trans* = 'C', then *B* := *alpha***A*^{H}**X* + *beta***B* (conjugate transpose)
n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
nrhs	INTEGER. The number of right-hand sides, i.e., the number of columns in *X* and *B* (*nrhs* ≥ 0).
alpha, *beta*	REAL for slagtm/clagtm DOUBLE PRECISION for dlagtm/zlagtm The scalars α and β. *alpha* must be 0., 1., or -1.; otherwise, it is assumed to be 0. *beta* must be 0., 1., or -1.; otherwise, it is assumed to be 1.
dl, *d*, *du*	REAL for slagtm DOUBLE PRECISION for dlagtm COMPLEX for clagtm COMPLEX*16 for zlagtm. Arrays: *dl*(*n* - 1), *d*(*n*), *du*(*n* - 1). The array *dl* contains the (*n* - 1) sub-diagonal elements of *T*. The array *d* contains the *n* diagonal elements of *T*. The array *du* contains the (*n* - 1) super-diagonal elements of *T*.
x, *b*	REAL for slagtm DOUBLE PRECISION for dlagtm COMPLEX for clagtm COMPLEX*16 for zlagtm. Arrays: *x*(*ldx*,*) contains the *n*-by-*nrhs* matrix *X*. The second dimension of *x* must be at least max(1, *nrhs*).

	b(*ldb*,*) contains the *n*-by-*nrhs* matrix *B*. The second dimension of *b* must be at least max(1, *nrhs*).
ldx	INTEGER. The leading dimension of the array *x*; *ldx* ≥ max(1, *n*).
ldb	INTEGER. The leading dimension of the array *b*; *ldb* ≥ max(1, *n*).

Output Parameters

b	Overwritten by the matrix expression *B* := *alpha*A*X* + *beta*B*

?lagts

*Solves the system of equations (T - lambda*I)*x = y or (T - lambda*I)^T*x = y ,where T is a general tridiagonal matrix and lambda is a scalar, using the LU factorization computed by* ?lagtf*.*

Syntax

```
call slagts( job, n, a, b, c, d, in, y, tol, info )
call dlagts( job, n, a, b, c, d, in, y, tol, info )
```

Description

This routine may be used to solve for *x* one of the systems of equations:

(*T* – *lambda*I*)**x* = *y* or (*T* – *lambda*I*)'**x* = *y*,

where *T* is an *n*-by-*n* tridiagonal matrix, following the factorization of (*T* – *lambda*I*) as

T – *lambda*I* = *P*L*U*,

computed by the routine ?lagtf.

The choice of equation to be solved is controlled by the argument *job*, and in each case there is an option to perturb zero or very small diagonal elements of *U*, this option being intended for use in applications such as inverse iteration.

Input Parameters

job	INTEGER. Specifies the job to be performed by ?lagts as follows: = 1: The equations (*T* – *lambda***I*)*x* = *y* are to be solved, but diagonal elements of *U* are not to be perturbed. = -1: The equations (*T* – *lambda***I*)*x* = *y* are to be solved and, if overflow would otherwise occur, the diagonal elements of *U* are to be perturbed. See argument *tol* below. = 2: The equations (*T* – *lambda***I*)'*x* = *y* are to be solved, but diagonal elements of *U* are not to be perturbed. = -2: The equations (*T* – *lambda***I*)'*x* = *y* are to be solved and, if overflow would otherwise occur, the diagonal elements of *U* are to be perturbed. See argument *tol* below.
n	INTEGER. The order of the matrix *T* (*n* ≥ 0).
a, *b*, *c*, *d*	REAL for slagts DOUBLE PRECISION for dlagts Arrays, dimension *a*(*n*), *b*(*n*-1), *c*(*n*-1), *d*(*n*-2): On entry, *a*(*) must contain the diagonal elements of *U* as returned from ?lagtf. On entry, *b*(*) must contain the first super-diagonal elements of *U* as returned from ?lagtf. On entry, *c*(*) must contain the sub-diagonal elements of *L* as returned from ?lagtf. On entry, *d*(*) must contain the second super-diagonal elements of *U* as returned from ?lagtf.
in	INTEGER. Array, dimension (*n*). On entry, *in*(*) must contain details of the matrix *p* as returned from ?lagtf.
y	REAL for slagts DOUBLE PRECISION for dlagts Array, dimension (*n*). On entry, the right hand side vector *y*.
tol	REAL for slagtf DOUBLE PRECISION for dlagtf.

On entry, with *job* < 0, *tol* should be the minimum perturbation to be made to very small diagonal elements of *U*. *tol* should normally be chosen as about *eps**norm(*U*), where *eps* is the relative machine precision, but if *tol* is supplied as non-positive, then it is reset to *eps**max(abs(u(i,j))). If *job* > 0 then *tol* is not referenced.

Output Parameters

y — On exit, *y* is overwritten by the solution vector *x*.

tol — On exit, *tol* is changed as described in Input Parameters section above, only if *tol* is non-positive on entry. Otherwise *tol* is unchanged.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i* >0, overflow would occur when computing the *i*th element of the solution vector *x*. This can only occur when *job* is supplied as positive and either means that a diagonal element of *U* is very small, or that the elements of the right-hand side vector *y* are very large.

?lagv2

Computes the Generalized Schur factorization of a real 2-by-2 matrix pencil (A,B) where B is upper triangular.

Syntax

```
call slagv2( a, lda, b, ldb, alphar, alphai, beta, csl, snl, csr, snr )
call dlagv2( a, lda, b, ldb, alphar, alphai, beta, csl, snl, csr, snr )
```

Description

This routine computes the Generalized Schur factorization of a real 2-by-2 matrix pencil (*A*,*B*) where *B* is upper triangular. The routine computes orthogonal (rotation) matrices given by *csl*, *snl* and *csr*, *snr* such that:

1) if the pencil (*A*,*B*) has two real eigenvalues (include 0/0 or 1/0 types), then

$$\begin{bmatrix} a_{11} & a_{11} \\ 0 & a_{22} \end{bmatrix} = \begin{bmatrix} csl & snl \\ -snl & csl \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

$$\begin{bmatrix} b_{11} & b_{11} \\ 0 & b_{22} \end{bmatrix} = \begin{bmatrix} csl & snl \\ -snl & csl \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ 0 & b_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

2) if the pencil (*A*,*B*) has a pair of complex conjugate eigenvalues, then

$$\begin{bmatrix} a_{11} & a_{11} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} csl & snl \\ -snl & csl \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

$$\begin{bmatrix} b_{11} & 0 \\ 0 & b_{22} \end{bmatrix} = \begin{bmatrix} csl & snl \\ -snl & csl \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ 0 & b_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

where $b_{11} \geq b_{22} > 0$.

Input Parameters

a, *b*	`REAL` for `slagv2`
	`DOUBLE PRECISION` for `dlagv2`
	Arrays:
	a(*lda*,2) contains the 2-by-2 matrix *A*;
	b(*ldb*,2) contains the upper triangular 2-by-2 matrix *B*.

lda	`INTEGER`. The leading dimension of the array *a*; *lda* ≥ 2.

ldb	INTEGER. The leading dimension of the array *b*; *ldb* ≥ 2.

Output Parameters

a	On exit, *a* is overwritten by the "*A*-part" of the generalized Schur form.
b	On exit, *b* is overwritten by the "*B*-part" of the generalized Schur form.
alphar, *alphai*, *beta*	REAL for slagv2 DOUBLE PRECISION for dlagv2. Arrays, dimension (2) each. (*alphar*(k) + *i***alphai*(k))/*beta*(k) are the eigenvalues of the pencil (*A*,*B*), k=1,2 and *i* = sqrt(-1). Note that *beta*(k) may be zero.
csl, *snl*	REAL for slagv2 DOUBLE PRECISION for dlagv2 The cosine and sine of the left rotation matrix, respectively.
csr, *snr*	REAL for slagv2 DOUBLE PRECISION for dlagv2 The cosine and sine of the right rotation matrix, respectively.

?lahqr

Computes the eigenvalues and Schur factorization of an upper Hessenberg matrix, using the double-shift/single-shift QR algorithm.

Syntax

```
call slahqr( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz, info )
call dlahqr( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz, info )
call clahqr( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, info )
call zlahqr( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, info )
```

Description

This routine is an auxiliary routine called by ?hseqr to update the eigenvalues and Schur decomposition already computed by ?hseqr, by dealing with the Hessenberg submatrix in rows and columns *ilo* to *ihi*.

Input Parameters

wantt — LOGICAL.
If *wantt* = .TRUE., the full Schur form *T* is required;
If *wantt* = .FALSE., eigenvalues only are required.

wantz — LOGICAL.
If *wantz* = .TRUE., the matrix of Schur vectors *Z* is required;
If *wantz* = .FALSE., Schur vectors are not required.

n — INTEGER. The order of the matrix *H* (*n* ≥ 0).

ilo, *ihi* — INTEGER.

It is assumed that *h* is already upper quasi-triangular in rows and columns *ihi*+1:*n*, and that *h*(*ilo*,*ilo*-1) = 0 (unless *ilo* = 1). The routine ?lahqr works primarily with the Hessenberg submatrix in rows and columns *ilo* to *ihi*, but applies transformations to all of *h* if *wantt* = .TRUE.. Constraints:

1 ≤ *ilo* ≤ max(1,*ihi*); *ihi* ≤ *n*.

h, *z*

REAL for slahqr
DOUBLE PRECISION for dlahqr
COMPLEX for clahqr
COMPLEX*16 for zlahqr.
Arrays:
h(*ldh*,*) contains the upper Hessenberg matrix *h*.
The second dimension of *h* must be at least max(1, *n*).
z(*ldz*,*)
If *wantz* = .TRUE., then, on entry, *z* must contain the current matrix *z* of transformations accumulated by ?hseqr.
If *wantz* = .FALSE., then *z* is not referenced. The second dimension of *z* must be at least max(1, *n*).

ldh

INTEGER. The first dimension of *h*; at least max(1, *n*).

ldz

INTEGER. The first dimension of *z*; at least max(1, *n*).

iloz, *ihiz*

INTEGER. Specify the rows of *z* to which transformations must be applied if *wantz* = .TRUE..

1 ≤ *iloz* ≤ *ilo*; *ihi* ≤ *ihiz* ≤ *n*.

Output Parameters

h

On exit, if *info*= 0 and *wantt* = .TRUE., then,

- for slahqr/dlahqr, *h* is upper quasi-triangular in rows and columns *ilo*:*ihi* with any 2-by-2 diagonal blocks in standard form.
- for clahqr/zlahqr, *h* is upper triangular in rows and columns *ilo*:*ihi*.

If *info*= 0 and *wantt* = .FALSE., the contents of *h* are unspecified on exit. If *info* is positive, see description of *info* for the output state of *h*.

wr, *wi*

REAL for slahqr
DOUBLE PRECISION for dlahqr
Arrays, DIMENSION at least max(1, *n*) each. Used with real flavors only. The real and imaginary parts, respectively, of the computed eigenvalues *ilo* to *ihi* are stored in the corresponding elements of *wr* and *wi*. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of *wr* and *wi*, say the i-th and (i+1)-th, with *wi*(i)> 0 and *wi*(i+1) < 0.
If *wantt* = .TRUE., the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*, with *wr*(i) = *h*(i,i), and, if *h*(i:i+1, i:i+1) is a 2-by-2 diagonal block, *wi*(i) = sqrt(*h*(i+1,i)*h*(i,i+1)) and *wi*(i+1) = -*wi*(i).

w

COMPLEX for clahqr
COMPLEX*16 for zlahqr.
Array, DIMENSION at least max(1, *n*). Used with complex flavors only. The computed eigenvalues *ilo* to *ihi* are stored in the corresponding elements of *w*.
If *wantt* = .TRUE., the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*, with *w*(i) = *h*(i,i).

z

If *wantz* = .TRUE., then, on exit *z* has been updated; transformations are applied only to the submatrix *z*(*iloz*:*ihiz*, *ilo*:*ihi*).

info

INTEGER.
If *info* = 0, the execution is successful.
With *info* > 0,

- if *info* = *i*, ?lahqr failed to compute all the eigenvalues *ilo* to *ihi* in a total of 30 iterations per eigenvalue; elements i+1:*ihi* of *wr* and *wi* (for slahqr/dlahqr) or *w* (for clahqr/zlahqr) contain those eigenvalues which have been successfully computed.

- if *wantt* is .FALSE., then on exit the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns *ilo* thorugh *info* of the final output value of *h*.
- if *wantt* is .TRUE., then on exit

  (initial value of *h*)*$_{(*)}$ *u* = *u**(final value of *h*),

  where *u* is an orthognal matrix. The final value of *h* is upper Hessenberg and triangular in rows and columns *info*+1 through *ihi*.
- if *wantz* is .TRUE., then on exit

  (final value of *z*) = (initial value of *z*)* *u*,

  where *u* is an orthognal matrix in (*) regardless of the value of *wantt*.

?lahrd

Reduces the first nb columns of a general rectangular matrix A so that elements below the k-th subdiagonal are zero, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A.

Syntax

```
call slahrd( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call dlahrd( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call clahrd( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call zlahrd( n, k, nb, a, lda, tau, t, ldt, y, ldy )
```

Description

The routine reduces the first *nb* columns of a real/complex general *n*-by-(*n*-*k*+1) matrix *A* so that elements below the *k*-th subdiagonal are zero. The reduction is performed by an orthogonal/unitary similarity transformation *Q*'**A***Q*. The routine returns the matrices *V* and *T* which determine *Q* as a block reflector *I* - *V***T***V*', and also the matrix *Y* = *A***V***T*.

The matrix *Q* is represented as products of *nb* elementary reflectors:

Q = *H*(1) *H*(2)... *H*(*nb*)

Each *H*(i) has the form

H(i) = *I* - *tau***v***v*'

where *tau* is a real/complex scalar, and *v* is a real/complex vector.

This is an obsolete auxiliary routine. Please use the new routine ?lahr2 instead.

Input Parameters

n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
k	INTEGER. The offset for the reduction. Elements below the *k*-th subdiagonal in the first *nb* columns are reduced to zero.
nb	INTEGER. The number of columns to be reduced.
a	REAL for slahrd DOUBLE PRECISION for dlahrd COMPLEX for clahrd COMPLEX*16 for zlahrd. Array *a*(*lda*, *n*-*k*+1) contains the *n*-by-(*n*-*k*+1) general matrix *A* to be reduced.
lda	INTEGER. The first dimension of *a*; at least max(1, *n*).
ldt	INTEGER. The first dimension of the output array *t*; must be at least max(1, *nb*).
ldy	INTEGER. The first dimension of the output array *y*; must be at least max(1, *n*).

Output Parameters

a	On exit, the elements on and above the *k*-th subdiagonal in the first *nb* columns are overwritten with the corresponding elements of the reduced matrix; the elements below the *k*-th subdiagonal, with the array *tau*, represent the matrix *Q* as a product of elementary reflectors. The other columns of *a* are unchanged. See Application Notes below.
tau	REAL for slahrd DOUBLE PRECISION for dlahrd COMPLEX for clahrd COMPLEX*16 for zlahrd. Array, DIMENSION (*nb*). Contains scalar factors of the elementary reflectors.
t, *y*	REAL for slahrd DOUBLE PRECISION for dlahrd COMPLEX for clahrd COMPLEX*16 for zlahrd. Arrays, dimension *t*(*ldt*, *nb*), *y*(*ldy*, *nb*). The array *t* contains upper triangular matrix *T*. The array *y* contains the *n*-by-*nb* matrix *Y* .

Application Notes

For the elementary reflector *H*(i),

v(1:i+*k*–1) = 0, *v*(i+*k*) = 1; *v*(i+*k*+1:*n*) is stored on exit in *a*(i+*k*+1:*n*, i) and *tau* is stored in *tau*(i).

The elements of the vectors *v* together form the (*n*-*k*+1)-by-*nb* matrix *V* which is needed, with *T* and *Y*, to apply the transformation to the unreduced part of the matrix, using an update of the form:

A := (*I* – *V T V'*) * (*A* – *Y V'*).

The contents of *A* on exit are illustrated by the following example with *n* = 7, *k* = 3 and *nb* = 2:

$$\begin{bmatrix} a & h & a & a & a \\ a & h & a & a & a \\ a & h & a & a & a \\ h & h & a & a & a \\ v_1 & h & a & a & a \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \end{bmatrix}$$

where *a* denotes an element of the original matrix *A*, *h* denotes a modified element of the upper Hessenberg matrix *H*, and v_i denotes an element of the vector defining *H*(i).

?lahr2

Reduces the specified number of first columns of a general rectangular matrix A so that elements below the specified subdiagonal are zero, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A.

Syntax

```
call slahr2( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call dlahr2( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call clahr2( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call zlahr2( n, k, nb, a, lda, tau, t, ldt, y, ldy )
```

Description

The routine reduces the first *nb* columns of a real/complex general *n*-by-(*n*-*k*+1) matrix *A* so that elements below the *k*-th subdiagonal are zero. The reduction is performed by an orthogonal/unitary similarity transformation *Q*'**A***Q*. The routine returns the matrices *V* and *T* which determine *Q* as a block reflector *I* - *V***T***V*', and also the matrix *Y* = *A***V***T*.

The matrix *Q* is represented as products of *nb* elementary reflectors:

Q = *H*(1) *H*(2)... *H*(*nb*)

Each *H*(i) has the form

H(i) = *I* - *tau***v***v*'

where *tau* is a real/complex scalar, and *v* is a real/complex vector.

This is an auxiliary routine called by ?gehrd.

Input Parameters

n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
k	INTEGER. The offset for the reduction. Elements below the *k*-th subdiagonal in the first *nb* columns are reduced to zero (*k* < *n*).
nb	INTEGER. The number of columns to be reduced.
a	REAL for slahr2 DOUBLE PRECISION for dlahr2 COMPLEX for clahr2 COMPLEX*16 for zlahr2. Array, DIMENSION (*lda*, *n*-*k*+1) contains the *n*-by-(*n*-*k*+1) general matrix *A* to be reduced.
lda	INTEGER. The first dimension of the array *a*; *lda* ≥ max(1, *n*).
ldt	INTEGER. The first dimension of the output array *t*; *ldt* ≥ *nb*.
ldy	INTEGER. The first dimension of the output array *y*; *ldy* ≥ *n*.

Output Parameters

a	On exit, the elements on and above the *k*-th subdiagonal in the first *nb* columns are overwritten with the corresponding elements of the reduced matrix; the elements below the *k*-th subdiagonal, with the array *tau*, represent the matrix *Q* as a product of elementary reflectors. The other columns of *a* are unchanged. See Application Notes below.

tau	REAL for slahr2 DOUBLE PRECISION for dlahr2 COMPLEX for clahr2 COMPLEX*16 for zlahr2. Array, DIMENSION (*nb*). Contains scalar factors of the elementary reflectors.
t, *y*	REAL for slahr2 DOUBLE PRECISION for dlahr2 COMPLEX for clahr2 COMPLEX*16 for zlahr2. Arrays, dimension *t*(*ldt*, *nb*), *y*(*ldy*, *nb*). The array *t* contains upper triangular matrix *T*. The array *y* contains the *n*-by-*nb* matrix *Y* .

Application Notes

For the elementary reflector *H*(i),

v(1:i+*k*-1) = 0, *v*(i+*k*) = 1; *v*(i+*k*+1:*n*) is stored on exit in *a*(i+*k*+1:*n*, i) and *tau* is stored in *tau*(i).

The elements of the vectors *v* together form the (*n*-*k*+1)-by-*nb* matrix *V* which is needed, with *T* and *Y*, to apply the transformation to the unreduced part of the matrix, using an update of the form:

A := (*I* - *V***T***V*') * (*A* - *Y***V*').

The contents of *A* on exit are illustrated by the following example with *n* = 7, *k* = 3 and *nb* = 2:

$$\begin{bmatrix} a & a & a & a & a \\ a & a & a & a & a \\ a & a & a & a & a \\ h & h & a & a & a \\ v_1 & h & a & a & a \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \end{bmatrix}$$

where *a* denotes an element of the original matrix *A*, *h* denotes a modified element of the upper Hessenberg matrix *H*, and v_i denotes an element of the vector defining *H*(i).

?laic1

Applies one step of incremental condition estimation.

Syntax

```
call slaic1( job, j, x, sest, w, gamma, sestpr, s, c )
call dlaic1( job, j, x, sest, w, gamma, sestpr, s, c )
call claic1( job, j, x, sest, w, gamma, sestpr, s, c )
call zlaic1( job, j, x, sest, w, gamma, sestpr, s, c )
```

Description

The routine ?laic1 applies one step of incremental condition estimation in its simplest version.

Let *x*, $||x||_2 = 1$ (where $||a||_2$ denotes the 2-norm of *a*), be an approximate singular vector of an *j*-by-*j* lower triangular matrix *L*, such that

$||L*x||_2 = sest$

Then ?laic1 computes *sestpr*, *s*, *c* such that the vector

$$xhat = \begin{bmatrix} s*x \\ c \end{bmatrix}$$

is an approximate singular vector of

$$Lhat = \begin{bmatrix} L & 0 \\ w' & gamma \end{bmatrix}$$

in the sense that

$||Lhat*xhat||_2 = sestpr$.

Depending on *job*, an estimate for the largest or smallest singular value is computed.

Note that `[s c]'` and $sestpr^2$ is an eigenpair of the system (for `slaic1`/`claic`)

$$\mathrm{diag}(sest*sest, 0) + [alpha\ gamma] * \begin{bmatrix} alpha \\ gamma \end{bmatrix}$$

where $alpha = x'*w$;

or of the system (for `claic1`/`zlaic`)

$$\mathrm{diag}(sest*sest, 0) + [alpha\ gamma] * \begin{bmatrix} \mathrm{conjg}(alpha) \\ \mathrm{conjg}(gamma) \end{bmatrix}$$

where $alpha = \mathrm{conjg}(x)'*w$.

Input Parameters

job	`INTEGER`. If *job* =1, an estimate for the largest singular value is computed; If *job* =2, an estimate for the smallest singular value is computed;
j	`INTEGER`. Length of *x* and *w*.

x, *w*	`REAL` for `slaic1`
	`DOUBLE PRECISION` for `dlaic1`
	`COMPLEX` for `claic1`
	`COMPLEX*16` for `zlaic1`.
	Arrays, dimension (*j*) each. Contain vectors *x* and *w*, respectively.

sest	`REAL` for `slaic1`/`claic1`;

DOUBLE PRECISION for dlaic1/zlaic1.
Estimated singular value of *j*-by-*j* matrix *L*.

gamma — REAL for slaic1
DOUBLE PRECISION for dlaic1
COMPLEX for claic1
COMPLEX*16 for zlaic1.
The diagonal element *gamma*.

Output Parameters

sestpr — REAL for slaic1/claic1;
DOUBLE PRECISION for dlaic1/zlaic1.
Estimated singular value of (*j*+1)-by-(*j*+1) matrix *Lhat*.

s, *c* — REAL for slaic1
DOUBLE PRECISION for dlaic1
COMPLEX for claic1
COMPLEX*16 for zlaic1.
Sine and cosine needed in forming *xhat*.

?laln2

Solves a 1-by-1 or 2-by-2 linear system of equations of the specified form.

Syntax

```
call slaln2( ltrans, na, nw, smin, ca, a, lda, d1, d2, b, ldb, wr, wi, x, ldx, scale, xnorm, info )
```

```
call dlaln2( ltrans, na, nw, smin, ca, a, lda, d1, d2, b, ldb, wr, wi, x, ldx, scale, xnorm, info )
```

Description

The routine solves a system of the form

`(ca*A - w*D)*X = s*B`, or `(ca*A' - w*D)*X = s*B`

with possible scaling (*s*) and perturbation of *A* (*A'* means *A*-transpose.)

A is an *na*-by-*na* real matrix, *ca* is a real scalar, *D* is an *na*-by-*na* real diagonal matrix, *w* is a real or complex value, and *X* and *B* are *na*-by-1 matrices: real if *w* is real, complex if *w* is complex. The parameter *na* may be 1 or 2.

If *w* is complex, *X* and *B* are represented as *na*-by-2 matrices, the first column of each being the real part and the second being the imaginary part.

The routine computes the scaling factor *s* (≤ 1) so chosen that *X* can be computed without overflow. *X* is further scaled if necessary to assure that norm(*ca***A* - *w***D*)*norm(*X*) is less than overflow.

If both singular values of (*ca*&**A* - *w***D*) are less than *smin*, *smin***I* (where *I* stands for identity) will be used instead of (*ca***A* - *w***D*). If only one singular value is less than *smin*, one element of (*ca***A* - *w***D*) will be perturbed enough to make the smallest singular value roughly *smin*.

If both singular values are at least *smin*, (*ca***A* - *w***D*) will not be perturbed. In any case, the perturbation will be at most some small multiple of max(*smin*, *ulp**norm(*ca***A* - *w***D*)).

The singular values are computed by infinity-norm approximations, and thus will only be correct to a factor of 2 or so.

NOTE. All input quantities are assumed to be smaller than overflow by a reasonable factor (see *bignum*).

Input Parameters

trans	LOGICAL. If *trans* = .TRUE., *A*- transpose will be used. If *trans* = .FALSE., *A* will be used (not transposed.)
na	INTEGER. The size of the matrix *A*, possible values 1 or 2.
nw	INTEGER. This parameter must be 1 if *w* is real, and 2 if *w* is complex. Possible values 1 or 2.
smin	REAL for slaln2 DOUBLE PRECISION for dlaln2. The desired lower bound on the singular values of *A*.

This should be a safe distance away from underflow or overflow, for example, between (underflow/machine_precision) and (machine_precision * overflow). (See *bignum* and *ulp*).

ca — REAL for slaln2
DOUBLE PRECISION for dlaln2.
The coefficient by which *A* is multiplied.

a — REAL for slaln2
DOUBLE PRECISION for dlaln2.
Array, DIMENSION (*lda*,*na*).
The *na*-by-*na* matrix *A*.

lda — INTEGER. The leading dimension of *a*. Must be at least *na*.

d1, *d2* — REAL for slaln2
DOUBLE PRECISION for dlaln2.
The (1,1) and (2,2) elements in the diagonal matrix *D*, respectively. *d2* is not used if *nw* = 1.

b — REAL for slaln2
DOUBLE PRECISION for dlaln2.
Array, DIMENSION (*ldb*,*nw*). The *na*-by-*nw* matrix *B* (right-hand side). If *nw* =2 (*w* is complex), column 1 contains the real part of *B* and column 2 contains the imaginary part.

ldb — INTEGER. The leading dimension of *b*. Must be at least *na*.

wr, *wi* — REAL for slaln2
DOUBLE PRECISION for dlaln2.
The real and imaginary part of the scalar *w*, respectively. *wi* is not used if *nw* = 1.

ldx — INTEGER. The leading dimension of the output array *x*. Must be at least *na*.

Output Parameters

x — REAL for slaln2
DOUBLE PRECISION for dlaln2.
Array, DIMENSION (*ldx*,*nw*). The *na*-by-*nw* matrix *X* (unknowns), as computed by the routine. If *nw* = 2 (*w* is complex), on exit, column 1 will contain the real part of *X* and column 2 will contain the imaginary part.

scale	REAL for slaln2 DOUBLE PRECISION for dlaln2. The scale factor that *B* must be multiplied by to insure that overflow does not occur when computing *X*. Thus (*ca*A* – *w*D*) *X* will be *scale***B*, not *B* (ignoring perturbations of *A*.) It will be at most 1.
xnorm	REAL for slaln2 DOUBLE PRECISION for dlaln2. The infinity-norm of *X*, when *X* is regarded as an *na*-by-*nw* real matrix.
info	INTEGER. An error flag. It will be zero if no error occurs, a negative number if an argument is in error, or a positive number if (*ca*A* – *w*D*) had to be perturbed. The possible values are: If *info* = 0: no error occurred, and (*ca*A* – *w*D*) did not have to be perturbed. If *info* = 1: (*ca*A* – *w*D*) had to be perturbed to make its smallest (or only) singular value greater than *smin*.

NOTE. For higher speed, this routine does not check the inputs for errors.

?lals0

Applies back multiplying factors in solving the least squares problem using divide and conquer SVD approach. Used by ?gelsd.

Syntax

```
call slals0 ( icompq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm, givptr,
givcol, ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, work, info )

call dlals0 ( icompq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm, givptr,
givcol, ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, work, info )

call clals0( icompq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm, givptr,
givcol, ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, rwork, info )

call zlals0( icompq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm, givptr,
givcol, ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, rwork, info )
```

Description

The routine applies back the multiplying factors of either the left or right singular vector matrix of a diagonal matrix appended by a row to the right hand side matrix *B* in solving the least squares problem using the divide-and-conquer SVD approach.

For the left singular vector matrix, three types of orthogonal matrices are involved:

(1L) Givens rotations: the number of such rotations is *givptr*;the pairs of columns/rows they were applied to are stored in *givcol*;and the *c*- and *s*-values of these rotations are stored in *givnum*.

(2L) Permutation. The (*nl*+1)-st row of *B* is to be moved to the first row, and for j=2:*n*, *perm*(*j*)-th row of *B* is to be moved to the *j*-th row.

(3L) The left singular vector matrix of the remaining matrix.

For the right singular vector matrix, four types of orthogonal matrices are involved:

(1R) The right singular vector matrix of the remaining matrix.

(2R) If *sqre* = 1, one extra Givens rotation to generate the right null space.

(3R) The inverse transformation of (2L).

(4R) The inverse transformation of (1L).

Input Parameters

icompq	INTEGER. Specifies whether singular vectors are to be computed in factored form: If *icompq* = 0: Left singular vector matrix. If *icompq* = 1: Right singular vector matrix.
nl	INTEGER. The row dimension of the upper block. *nl* ≥ 1.
nr	INTEGER. The row dimension of the lower block. *nr* ≥ 1.
sqre	INTEGER. If *sqre* = 0: the lower block is an *nr*-by-*nr* square matrix. If *sqre* = 1: the lower block is an *nr*-by-(*nr*+1) rectangular matrix. The bidiagonal matrix has row dimension *n* = *nl* + *nr* + 1, and column dimension *m* = *n* + *sqre*.
nrhs	INTEGER. The number of columns of *B* and *bx*. Must be at least 1.
b	REAL for slals0 DOUBLE PRECISION for dlals0 COMPLEX for clals0 COMPLEX*16 for zlals0. Array, DIMENSION (*ldb*, *nrhs*). Contains the right hand sides of the least squares problem in rows 1 through *m*.
ldb	INTEGER. The leading dimension of *b*. Must be at least max(1,max(*m*, *n*)).
bx	REAL for slals0 DOUBLE PRECISION for dlals0 COMPLEX for clals0 COMPLEX*16 for zlals0. Workspace array, DIMENSION (*ldbx*, *nrhs*).
ldbx	INTEGER. The leading dimension of *bx*.
perm	INTEGER. Array, DIMENSION (*n*). The permutations (from deflation and sorting) applied to the two blocks.

givptr — INTEGER. The number of Givens rotations which took place in this subproblem.

givcol — INTEGER. Array, DIMENSION (*ldgcol*, 2). Each pair of numbers indicates a pair of rows/columns involved in a Givens rotation.

ldgcol — INTEGER. The leading dimension of *givcol*, must be at least *n*.

givnum — REAL for slals0/clals0
DOUBLE PRECISION for dlals0/zlals0
Array, DIMENSION (*ldgnum*, 2). Each number indicates the *c* or *s* value used in the corresponding Givens rotation.

ldgnum — INTEGER. The leading dimension of arrays *difr*, *poles* and *givnum*, must be at least *k*.

poles — REAL for slals0/clals0
DOUBLE PRECISION for dlals0/zlals0
Array, DIMENSION (*ldgnum*, 2). On entry, *poles*(1:*k*, 1) contains the new singular values obtained from solving the secular equation, and *poles*(1:*k*, 2) is an array containing the poles in the secular equation.

difl — REAL for slals0/clals0
DOUBLE PRECISION for dlals0/zlals0
Array, DIMENSION (*k*). On entry, *difl*(*i*) is the distance between *i*-th updated (undeflated) singular value and the *i*-th (undeflated) old singular value.

difr — REAL for slals0/clals0
DOUBLE PRECISION for dlals0/zlals0
Array, DIMENSION (*ldgnum*, 2). On entry, *difr*(*i*, 1) contains the distances between *i*-th updated (undeflated) singular value and the *i*+*1*-th (undeflated) old singular value. And *difr*(*i*, 2) is the normalizing factor for the *i*-th right singular vector.

z — REAL for slals0/clals0
DOUBLE PRECISION for dlals0/zlals0
Array, DIMENSION (*k*). Contains the components of the deflation-adjusted updating row vector.

K	INTEGER. Contains the dimension of the non-deflated matrix. This is the order of the related secular equation. $1 \leq k \leq n$.
c	REAL for slals0/clals0 DOUBLE PRECISION for dlals0/zlals0 Contains garbage if *sqre* =0 and the *c* value of a Givens rotation related to the right null space if *sqre* = 1.
s	REAL for slals0/clals0 DOUBLE PRECISION for dlals0/zlals0 Contains garbage if *sqre* =0 and the *s* value of a Givens rotation related to the right null space if *sqre* = 1.
work	REAL for slals0 DOUBLE PRECISION for dlals0 Workspace array, DIMENSION (*k*). Used with real flavors only.
rwork	REAL for clals0 DOUBLE PRECISION for zlals0 Workspace array, DIMENSION (*k**(1+*nrhs*) + 2**nrhs*). Used with complex flavors only.

Output Parameters

b	On exit, contains the solution *X* in rows 1 through *n*.
info	INTEGER. If *info* = 0: successful exit. If *info* = -*i* < 0, the *i*-th argument had an illegal value.

?lalsa

Computes the SVD of the coefficient matrix in compact form. Used by ?gelsd.

Syntax

call slalsa(*icompq, smlsiz, n, nrhs, b, ldb, bx, ldbx, u, ldu, vt, k, difl, difr, z, poles, givptr, givcol, ldgcol, perm, givnum, c, s, work, iwork, info*)

call dlalsa(*icompq, smlsiz, n, nrhs, b, ldb, bx, ldbx, u, ldu, vt, k, difl, difr, z, poles, givptr, givcol, ldgcol, perm, givnum, c, s, work, iwork, info*)

call clalsa(*icompq, smlsiz, n, nrhs, b, ldb, bx, ldbx, u, ldu, vt, k, difl, difr, z, poles, givptr, givcol, ldgcol, perm, givnum, c, s, rwork, iwork, info*)

call zlalsa(*icompq, smlsiz, n, nrhs, b, ldb, bx, ldbx, u, ldu, vt, k, difl, difr, z, poles, givptr, givcol, ldgcol, perm, givnum, c, s, rwork, iwork, info*)

Description

The routine is an itermediate step in solving the least squares problem by computing the SVD of the coefficient matrix in compact form. The singular vectors are computed as products of simple orthorgonal matrices.

If *icompq* = 0, ?lalsa applies the inverse of the left singular vector matrix of an upper bidiagonal matrix to the right hand side; and if *icompq* = 1, the routine applies the right singular vector matrix to the right hand side. The singular vector matrices were generated in the compact form by ?lalsa.

Input Parameters

icompq — INTEGER. Specifies whether the left or the right singular vector matrix is involved. If *icompq* = 0: left singular vector matrix is used
If *icompq* = 1: right singular vector matrix is used.

smlsiz — INTEGER. The maximum size of the subproblems at the bottom of the computation tree.

n	INTEGER. The row and column dimensions of the upper bidiagonal matrix.
nrhs	INTEGER. The number of columns of *b* and *bx*. Must be at least 1.
b	REAL for slalsa DOUBLE PRECISION for dlalsa COMPLEX for clalsa COMPLEX*16 for zlalsa Array, DIMENSION (*ldb*, *nrhs*). Contains the right hand sides of the least squares problem in rows 1 through *m*.
ldb	INTEGER. The leading dimension of *b* in the calling subprogram. Must be at least max(1,max(*m*, *n*)).
ldbx	INTEGER. The leading dimension of the output array *bx*.
u	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa Array, DIMENSION (*ldu*, *smlsiz*). On entry, *u* contains the left singular vector matrices of all subproblems at the bottom level.
ldu	INTEGER, *ldu* ≥ *n*. The leading dimension of arrays *u*, *vt*, *difl*, *difr*, *poles*, *givnum*, and *z*.
vt	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa Array, DIMENSION (*ldu*, *smlsiz* +1). On entry, contains the right singular vector matrices of all subproblems at the bottom level.
k	INTEGER array, DIMENSION (*n*).
difl	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa Array, DIMENSION (*ldu*, *nlvl*), where *nlvl* = int($\log_2$(*n* /(*smlsiz*+1))) + 1.
difr	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa

	Array, DIMENSION (*ldu*, 2**nlvl*). On entry, *difl*(*, *i*) and *difr*(*, 2*i* -1) record distances between singular values on the *i*-th level and singular values on the (*i* -1)-th level, and *difr*(*, 2*i*) record the normalizing factors of the right singular vectors matrices of subproblems on *i*-th level.
z	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa Array, DIMENSION (*ldu*, *nlvl* . On entry, *z*(1, *i*) contains the components of the deflation- adjusted updating the row vector for subproblems on the *i*-th level.
poles	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa Array, DIMENSION (*ldu*, 2**nlvl*). On entry, *poles*(*, 2*i*-1: 2*i*) contains the new and old singular values involved in the secular equations on the *i*-th level.
givptr	INTEGER. Array, DIMENSION (*n*). On entry, *givptr*(*i*) records the number of Givens rotations performed on the *i*-th problem on the computation tree.
givcol	INTEGER. Array, DIMENSION (*ldgcol*, 2**nlvl*). On entry, for each *i*, *givcol*(*, 2*i*-1: 2*i*) records the locations of Givens rotations performed on the *i*-th level on the computation tree.
ldgcol	INTEGER, *ldgcol* ≥ *n*. The leading dimension of arrays *givcol* and *perm*.
perm	INTEGER. Array, DIMENSION (*ldgcol*, *nlvl*). On entry, *perm*(*, *i*) records permutations done on the *i*-th level of the computation tree.
givnum	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa Array, DIMENSION (*ldu*, 2**nlvl*). On entry, *givnum*(*, 2*i*-1 : 2*i*) records the *c* and *s* values of Givens rotations performed on the *i*-th level on the computation tree.
c	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa

Array, DIMENSION (*n*). On entry, if the *i*-th subproblem is not square, *c*(*i*) contains the *c* value of a Givens rotation related to the right null space of the *i*-th subproblem.

s
REAL for slalsa/clalsa
DOUBLE PRECISION for dlalsa/zlalsa
Array, DIMENSION (*n*). On entry, if the *i*-th subproblem is not square, *s*(*i*) contains the *s*-value of a Givens rotation related to the right null space of the *i*-th subproblem.

work
REAL for slalsa
DOUBLE PRECISION for dlalsa
Workspace array, DIMENSION at least (*n*). Used with real flavors only.

rwork
REAL for clalsa
DOUBLE PRECISION for zlalsa
Workspace array, DIMENSION at least max(*n*, (*smlsz*+1)**nrhs**3). Used with complex flavors only.

iwork
INTEGER.
Workspace array, DIMENSION at least (3*n*).

Output Parameters

b
On exit, contains the solution *X* in rows 1 through *n*.

bx
REAL for slalsa
DOUBLE PRECISION for dlalsa
COMPLEX for clalsa
COMPLEX*16 for zlalsa
Array, DIMENSION (*ldbx*, *nrhs*). On exit, the result of applying the left or right singular vector matrix to *b*.

info
INTEGER. If *info* = 0: successful exit
If *info* = -*i* < 0, the *i*-th argument had an illegal value.

?lalsd

Uses the singular value decomposition of A to solve the least squares problem.

Syntax

```
call slalsd( uplo, smlsiz, n, nrhs, d, e, b, ldb, rcond, rank, work, iwork, info )
call dlalsd( uplo, smlsiz, n, nrhs, d, e, b, ldb, rcond, rank, work, iwork, info )
call clalsd( uplo, smlsiz, n, nrhs, d, e, b, ldb, rcond, rank, work, rwork, iwork, info )
call zlalsd( uplo, smlsiz, n, nrhs, d, e, b, ldb, rcond, rank, work, rwork, iwork, info )
```

Description

The routine uses the singular value decomposition of *A* to solve the least squares problem of finding *X* to minimize the Euclidean norm of each column of *A***X*-*B*, where *A* is *n*-by-*n* upper bidiagonal, and *X* and *B* are *n*-by-*nrhs*. The solution *X* overwrites *B*.

The singular values of *A* smaller than *rcond* times the largest singular value are treated as zero in solving the least squares problem; in this case a minimum norm solution is returned. The actual singular values are returned in *d* in ascending order.

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray XMP, Cray YMP, Cray C 90, or Cray 2.

It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

Input Parameters

uplo	CHARACTER*1. If *uplo* = 'U', *d* and *e* define an upper bidiagonal matrix. If *uplo* = 'L', *d* and *e* define a lower bidiagonal matrix.
smlsiz	INTEGER. The maximum size of the subproblems at the bottom of the computation tree.

n	INTEGER. The dimension of the bidiagonal matrix. $n \geq 0$.
nrhs	INTEGER. The number of columns of *B*. Must be at least 1.
d	REAL for slalsd/clalsd DOUBLE PRECISION for dlalsd/zlalsd Array, DIMENSION (*n*). On entry, *d* contains the main diagonal of the bidiagonal matrix.
e	REAL for slalsd/clalsd DOUBLE PRECISION for dlalsd/zlalsd Array, DIMENSION (*n*-1). Contains the super-diagonal entries of the bidiagonal matrix. On exit, *e* is destroyed.
b	REAL for slalsd DOUBLE PRECISION for dlalsd COMPLEX for clalsd COMPLEX*16 for zlalsd Array, DIMENSION (*ldb*,*nrhs*). On input, *b* contains the right hand sides of the least squares problem. On output, *b* contains the solution X.
ldb	INTEGER. The leading dimension of *b* in the calling subprogram. Must be at least max(1,*n*).
rcond	REAL for slalsd/clalsd DOUBLE PRECISION for dlalsd/zlalsd The singular values of *A* less than or equal to *rcond* times the largest singular value are treated as zero in solving the least squares problem. If *rcond* is negative, machine precision is used instead. For example, for the least squares problem diag(*S*)**X*=*B*, where diag(*S*) is a diagonal matrix of singular values, the solution is *X*(*i*)=*B*(*i*)/*S*(*i*) if *S*(*i*) is greater than *rcond* *max(*S*), and *X*(*i*)=0 if *S*(*i*) is less than or equal to *rcond* *max(*S*).
rank	INTEGER. The number of singular values of *A* greater than *rcond* times the largest singular value.
work	REAL for slalsd DOUBLE PRECISION for dlalsd COMPLEX for clalsd COMPLEX*16 for zlalsd

Workspace array.
DIMENSION for real flavors at least $(9n+2n*smlsiz+8n*nlvl+n*nrhs+(smlsiz+1)^2)$, where $nlvl = \max(0, \text{int}(\log_2(n/(smlsiz+1))) + 1)$.
DIMENSION for complex flavors is $(n*nrhs)$.

rwork — REAL for clalsd
DOUBLE PRECISION for zlalsd
Workspace array, used with complex flavors only.
DIMENSION at least $(9n + 2n*smlsiz + 8n*nlvl + 3*mlsiz*nrhs + (smlsiz+1)^2)$, where $nlvl = \max(0, \text{int}(\log_2(\min(m,n)/(smlsiz+1))) + 1)$.

iwork — INTEGER.
Workspace array of DIMENSION $(3n*nlvl + 11n)$.

Output Parameters

d — On exit, if $info = 0$, d contains singular values of the bidiagonal matrix.

e — On exit, destroyed.

b — On exit, b contains the solution X.

info — INTEGER.
If $info = 0$: successful exit.
If $info = -i < 0$, the i-th argument had an illegal value.
If $info > 0$: The algorithm failed to compute a singular value while working on the submatrix lying in rows and columns $info/(n+1)$ through $\text{mod}(info, n+1)$.

?lamrg

Creates a permutation list to merge the entries of two independently sorted sets into a single set sorted in acsending order.

Syntax

```
call slamrg( n1, n2, a, strd1, strd2, index )
call dlamrg( n1, n2, a, strd1, strd2, index )
```

Description

The routine creates a permutation list which will merge the elements of *a* (which is composed of two independently sorted sets) into a single set which is sorted in ascending order.

Input Parameters

n1, *n2*	INTEGER. These arguments contain the respective lengths of the two sorted lists to be merged.
a	REAL for slamrg DOUBLE PRECISION for dlamrg. Array, DIMENSION (*n1*+*n2*). The first *n1* elements of *a* contain a list of numbers which are sorted in either ascending or descending order. Likewise for the final *n2* elements.
strd1, *strd2*	INTEGER. These are the strides to be taken through the array *a*. Allowable strides are 1 and -1. They indicate whether a subset of *a* is sorted in ascending (*strdx* = 1) or descending (*strdx* = –1) order.

Output Parameters

index	INTEGER. Array, DIMENSION (*n1*+*n2*). On exit, this array will contain a permutation such that if *b*(*i*) = *a*(*index*(*i*)) for *i*=1, *n1*+*n2*, then *b* will be sorted in ascending order.

?laneg

Computes the Sturm count, the number of negative pivots encountered while factoring tridiagonal $T-sigma*I = L*D*L^T$.

Syntax

```
value = slaneg( n, d, lld, sigma, pivmin, r )
value = dlaneg( n, d, lld, sigma, pivmin, r )
```

Description

The routine computes the Sturm count, the number of negative pivots encountered while factoring tridiagonal $T-sigma*I = L*D*L^T$. This implementation works directly on the factors without forming the tridiagonal matrix T. The Sturm count is also the number of eigenvalues of T less than *sigma*. This routine is called from ?larb. The current routine does not use the *pivmin* parameter but rather requires IEEE-754 propagation of infinities and NaNs (NaN stands for 'Not A Number'). This routine also has no input range restrictions but does require default exception handling such that $x/0$ produces Inf when x is non-zero, and Inf/Inf produces NaN. (For more information see [Marques06]).

Input Parameters

n	INTEGER. The order of the matrix.
d	REAL for slaneg DOUBLE PRECISION for dlaneg Array, DIMENSION (*n*). Contains *n* diagonal elements of the matrix *D*.
lld	REAL for slaneg DOUBLE PRECISION for dlaneg Array, DIMENSION (*n*-1). Contains (*n*-1) elements *L*(i)**L*(i)**D*(i).
sigma	REAL for slaneg DOUBLE PRECISION for dlaneg Shift amount in *T*-*sigma***I* = *L***D***L****T*.
pivmin	REAL for slaneg DOUBLE PRECISION for dlaneg

	The minimum pivot in the Sturm sequence. May be used when zero pivots are encountered on non-IEEE-754 architectures.
r	INTEGER. The twist index for the twisted factorization that is used for the negcount.

Output Parameters

value	INTEGER. The number of negative pivots encountered while factoring.

?langb

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of general band matrix.

Syntax

val = slangb(*norm*, *n*, *kl*, *ku*, *ab*, *ldab*, *work*)

val = dlangb(*norm*, *n*, *kl*, *ku*, *ab*, *ldab*, *work*)

val = clangb(*norm*, *n*, *kl*, *ku*, *ab*, *ldab*, *work*)

val = zlangb(*norm*, *n*, *kl*, *ku*, *ab*, *ldab*, *work*)

Description

The function returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an *n*-by-*n* band matrix *A*, with *kl* sub-diagonals and *ku* super-diagonals.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(*A*), if *norm* = '1' or 'O' or 'o'

= normI(*A*), if *norm* = 'I' or 'i'

= normF(*A*), if *norm* = 'F', 'f', 'E' or 'e'

where norm1 denotes the 1-norm of a matrix (maximum column sum), normI denotes the infinity norm of a matrix (maximum row sum) and normF denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a consistent matrix norm.

Input Parameters

norm — CHARACTER*1. Specifies the vaule to be returned by the routine as described above.

n — INTEGER. The order of the matrix *A*. $n \geq 0$. When $n = 0$, ?langb is set to zero.

kl — INTEGER. The number of sub-diagonals of the matrix *A*. $kl \geq 0$.

ku — INTEGER. The number of super-diagonals of the matrix *A*. $ku \geq 0$.

ab — REAL for slangb
DOUBLE PRECISION for dlangb
COMPLEX for clangb
COMPLEX*16 for zlangb
Array, DIMENSION (*ldab*,*n*).
The band matrix *A*, stored in rows 1 to *kl*+*ku*+1. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:
$ab(ku+1+i-j,j) = a(i,j)$
for $\max(1,j-ku) \leq i \leq \min(n,j+kl)$.

ldab — INTEGER. The leading dimension of the array *ab*. $ldab \geq kl+ku+1$.

work — REAL for slangb/clangb
DOUBLE PRECISION for dlangb/zlangb
Workspace array, DIMENSION (max(1,*lwork*)), where $lwork \geq n$ when *norm* = 'I'; otherwise, *work* is not referenced.

Output Parameters

val — REAL for slangb/clangb

DOUBLE PRECISION for dlangb/zlangb
Value returned by the function.

?lange

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general rectangular matrix.

Syntax

```
val = slange( norm, m, n, a, lda, work )
val = dlange( norm, m, n, a, lda, work )
val = clange( norm, m, n, a, lda, work )
val = zlange( norm, m, n, a, lda, work )
```

Description

The function ?lange returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex matrix A.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(A), if *norm* = '1' or 'O' or 'o'

= normI(A), if *norm* = 'I' or 'i'

= normF(A), if *norm* = 'F', 'f', 'E' or 'e'

where norm1 denotes the 1-norm of a matrix (maximum column sum), normI denotes the infinity norm of a matrix (maximum row sum) and normF denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a consistent matrix norm.

Input Parameters

norm — CHARACTER*1. Specifies the vaule to be returned by the routine as described above.

m — INTEGER. The number of rows of the matrix A.

$m \geq 0$. When $m = 0$, ?lange is set to zero.

n	INTEGER. The number of columns of the matrix *A*. *n* ≥ 0. When *n* = 0, ?lange is set to zero.
a	REAL for slange DOUBLE PRECISION for dlange COMPLEX for clange COMPLEX*16 for zlange Array, DIMENSION (*lda*,*n*). The *m*-by-*n* matrix *A*.
lda	INTEGER. The leading dimension of the array *a*. *lda* ≥ max(*m*,1).
work	REAL for slange and clange. DOUBLE PRECISION for dlange and zlange. Workspace array, DIMENSION max(1,*lwork*), where *lwork* ≥ *m* when *norm* = 'I'; otherwise, *work* is not referenced.

Output Parameters

val	REAL for slange/clange DOUBLE PRECISION for dlange/zlange Value returned by the function.

?langt

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general tridiagonal matrix.

Syntax

```
val = slangt( norm, n, dl, d, du )
val = dlangt( norm, n, dl, d, du )
val = clangt( norm, n, dl, d, du )
val = zlangt( norm, n, dl, d, du )
```

Description

The routine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex tridiagonal matrix *A*.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(*A*), if *norm* = '1' or 'O'or 'o'

= normI(*A*), if *norm* = 'I' or 'i'

= normF(*A*), if *norm* = 'F', 'f', 'E' or 'e'

where norm1 denotes the 1-norm of a matrix (maximum column sum), normI denotes the infinity norm of a matrix (maximum row sum) and normF denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a consistent matrix norm.

Input Parameters

norm	CHARACTER*1. Specifies the vaule to be returned by the routine as described above.
n	INTEGER. The order of the matrix *A*. $n \geq 0$. When $n = 0$, ?langt is set to zero.
dl, *d*, *du*	REAL for slangt DOUBLE PRECISION for dlangt COMPLEX for clangt COMPLEX*16 for zlangt Arrays: *dl* (n-1), *d* (n), *du* (n-1). The array *dl* contains the (n-1) sub-diagonal elements of *A*. The array *d* contains the diagonal elements of *A*. The array *du* contains the (n-1) super-diagonal elements of *A*.

Output Parameters

val	REAL for slangt/clangt DOUBLE PRECISION for dlangt/zlangt Value returned by the function.

?lanhs

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of an upper Hessenberg matrix.

Syntax

```
val = slanhs( norm, n, a, lda, work )
val = dlanhs( norm, n, a, lda, work )
val = clanhs( norm, n, a, lda, work )
val = zlanhs( norm, n, a, lda, work )
```

Description

The function ?lanhs returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hessenberg matrix *A*.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(*A*), if *norm* = '1' or 'O' or 'o'

= normI(*A*), if *norm* = 'I' or 'i'

= normF(*A*), if *norm* = 'F', 'f', 'E' or 'e'

where norm1 denotes the 1-norm of a matrix (maximum column sum), normI denotes the infinity norm of a matrix (maximum row sum) and normF denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a consistent matrix norm.

Input Parameters

norm	CHARACTER*1. Specifies the vaule to be returned by the routine as described above.
n	INTEGER. The order of the matrix *A*. $n \geq 0$. When $n = 0$, ?lanhs is set to zero.
a	REAL for slanhs DOUBLE PRECISION for dlanhs COMPLEX for clanhs

	COMPLEX*16 for zlanhs Array, DIMENSION (*lda*,*n*). The *n*-by-*n* upper Hessenberg matrix *A*; the part of *A* below the first sub-diagonal is not referenced.
lda	INTEGER. The leading dimension of the array *a*. *lda* ≥ max(*n*,1).
work	REAL for slanhs and clanhs. DOUBLE PRECISION for dlange and zlange. Workspace array, DIMENSION (max(1,*lwork*)), where *lwork* ≥ *n* when *norm* = 'I'; otherwise, *work* is not referenced.

Output Parameters

val	REAL for slanhs/clanhs DOUBLE PRECISION for dlanhs/zlanhs Value returned by the function.

?lansb

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric band matrix.

Syntax

```
val = slansb( norm, uplo, n, k, ab, ldab, work )
val = dlansb( norm, uplo, n, k, ab, ldab, work )
val = clansb( norm, uplo, n, k, ab, ldab, work )
val = zlansb( norm, uplo, n, k, ab, ldab, work )
```

Description

The function ?lansb returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an *n*-by-*n* real/complex symmetric band matrix *A*, with *k* super-diagonals.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(*A*), if *norm* = '1' or 'O' or 'o'

= normI(*A*), if *norm* = 'I' or 'i'

= normF(*A*), if *norm* = 'F', 'f', 'E' or 'e'

where norm1 denotes the 1-norm of a matrix (maximum column sum), normI denotes the infinity norm of a matrix (maximum row sum) and normF denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a consistent matrix norm.

Input Parameters

norm — CHARACTER*1. Specifies the vaule to be returned by the routine as described above.

uplo — CHARACTER*1.
Specifies whether the upper or lower triangular part of the band matrix *A* is supplied. If *uplo* = 'U': upper triangular part is supplied; If *uplo* = 'L': lower triangular part is supplied.

n — INTEGER. The order of the matrix *A*. $n \geq 0$.
When *n* = 0, ?lansb is set to zero.

k — INTEGER. The number of super-diagonals or sub-diagonals of the band matrix *A*. $k \geq 0$.

ab — REAL for slansb
DOUBLE PRECISION for dlansb
COMPLEX for clansb
COMPLEX*16 for zlansb
Array, DIMENSION (*ldab*,*n*).
The upper or lower triangle of the symmetric band matrix *A*, stored in the first *k*+1 rows of *ab*. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:
if *uplo* = 'U', *ab*(*k*+1+*i*–*j*,*j*) = *a*(*i*,*j*)
for max(1,*j*–*k*) ≤ *i*≤ *j*;
if *uplo* = 'L', *ab*(1+*i*–*j*,*j*) = *a*(*i*,*j*) for
j≤*i*≤min(*n*,*j*+*k*).

ldab — INTEGER. The leading dimension of the array *ab*.

	ldab ≥ *k*+1.
work	REAL for slansb and clansb. DOUBLE PRECISION for dlansb and zlansb. Workspace array, DIMENSION (max(1, *lwork*)), where *lwork* ≥ *n* when *norm* = 'I' or '1' or 'O'; otherwise, *work* is not referenced.

Output Parameters

val	REAL for slansb/clansb DOUBLE PRECISION for dlansb/zlansb Value returned by the function.

?lanhb

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hermitian band matrix.

Syntax

val = clanhb(*norm*, *uplo*, *n*, *k*, *ab*, *ldab*, *work*)

val = zlanhb(*norm*, *uplo*, *n*, *k*, *ab*, *ldab*, *work*)

Description

The routine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an *n*-by-*n* Hermitian band matrix *A*, with *k* super-diagonals.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(*A*), if *norm* = '1' or 'O' or 'o'

= normI(*A*), if *norm* = 'I' or 'i'

= normF(*A*), if *norm* = 'F', 'f', 'E' or 'e'

where norm1 denotes the 1-norm of a matrix (maximum column sum), normI denotes the infinity norm of a matrix (maximum row sum) and normF denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a consistent matrix norm.

Input Parameters

norm — CHARACTER*1. Specifies the vaule to be returned by the routine as described above.

uplo — CHARACTER*1.
Specifies whether the upper or lower triangular part of the band matrix *A* is supplied.
If *uplo* = 'U': upper triangular part is supplied;
If *uplo* = 'L': lower triangular part is supplied.

n — INTEGER. The order of the matrix *A*. *n* ≥ 0. When *n* = 0, ?lanhb is set to zero.

k — INTEGER. The number of super-diagonals or sub-diagonals of the band matrix *A*.

k ≥ 0.

ab — COMPLEX for clanhb.
COMPLEX*16 for zlanhb.
Array, DIMENSION (*ldaB*,*n*). The upper or lower triangle of the Hermitian band matrix *A*, stored in the first *k*+1 rows of *ab*. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:
if *uplo* = 'U', *ab*(*k*+1+*i*–*j*,*j*) = *a*(*i*,*j*)

for max(1,*j*–*k*) ≤ *i* ≤ *j*;

if *uplo* = 'L', *ab*(1+*i*–*j*,*j*) = *a*(*i*,*j*) for *j* ≤ *i* ≤ min(*n*,*j*+*k*).
Note that the imaginary parts of the diagonal elements need not be set and are assumed to be zero.

ldab — INTEGER. The leading dimension of the array *ab*. *ldab* ≥ *k*+1.

work — REAL for clanhb.
DOUBLE PRECISION for zlanhb.
Workspace array, DIMENSION max(1, *lwork*), where

lwork ≥ *n* when *norm* = 'I' or '1' or 'O'; otherwise, *work* is not referenced.

Output Parameters

val — REAL for slanhb/clanhb
DOUBLE PRECISION for dlanhb/zlanhb
Value returned by the function.

?lansp

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric matrix supplied in packed form.

Syntax

```
val = slansp( norm, uplo, n, ap, work )
val = dlansp( norm, uplo, n, ap, work )
val = clansp( norm, uplo, n, ap, work )
val = zlansp( norm, uplo, n, ap, work )
```

Description

The function ?lansp returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix *A*, supplied in packed form.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(*A*), if *norm* = '1' or 'O' or 'o'

= normI(*A*), if *norm* = 'I' or 'i'

= normF(*A*), if *norm* = 'F', 'f', 'E' or 'e'

where norm1 denotes the 1-norm of a matrix (maximum column sum), normI denotes the infinity norm of a matrix (maximum row sum) and normF denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a consistent matrix norm.

Input Parameters

norm — CHARACTER*1. Specifies the vaule to be returned by the routine as described above.

uplo — CHARACTER*1.
Specifies whether the upper or lower triangular part of the symmetric matrix *A* is supplied.
If *uplo* = 'U': Upper triangular part of *A* is supplied
If *uplo* = 'L': Lower triangular part of *A* is supplied.

n — INTEGER. The order of the matrix *A*. $n \geq 0$. When $n = 0$, ?lansp is set to zero.

ap — REAL for slansp
DOUBLE PRECISION for dlansp
COMPLEX for clansp
COMPLEX*16 for zlansp
Array, DIMENSION ($n(n+1)/2$).
The upper or lower triangle of the symmetric matrix *A*, packed columnwise in a linear array. The j-th column of *A* is stored in the array *ap* as follows:
if *uplo* = 'U', $ap(i + (j-1)j/2) = A(i,j)$ for $1 \leq i \leq j$;
if *uplo* = 'L', $ap(i + (j-1)(2n-j)/2) = A(i,j)$ for $j \leq i \leq n$.

work — REAL for slansp and clansp.
DOUBLE PRECISION for dlansp and zlansp.
Workspace array, DIMENSION (max(1,*lwork*)), where $lwork \geq n$ when *norm* = 'I' or '1' or 'O'; otherwise, *work* is not referenced.

Output Parameters

val — REAL for slansp/clansp

DOUBLE PRECISION for dlansp/zlansp
Value returned by the function.

?lanhp

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix supplied in packed form.

Syntax

val = clanhp(*norm*, *uplo*, *n*, *ap*, *work*)

val = zlanhp(*norm*, *uplo*, *n*, *ap*, *work*)

Description

The function ?lanhp returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix *A*, supplied in packed form.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(*A*), if *norm* = '1' or 'O' or 'o'

= normI(*A*), if *norm* = 'I' or 'i'

= normF(*A*), if *norm* = 'F', 'f', 'E' or 'e'

where norm1 denotes the 1-norm of a matrix (maximum column sum), normI denotes the infinity norm of a matrix (maximum row sum) and normF denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a consistent matrix norm.

Input Parameters

norm — CHARACTER*1. Specifies the vaule to be returned by the routine as described above.

uplo — CHARACTER*1.
Specifies whether the upper or lower triangular part of the Hermitian matrix *A* is supplied.

If *uplo* = 'U': Upper triangular part of *A* is supplied
If *uplo* = 'L': Lower triangular part of *A* is supplied.

n — INTEGER. The order of the matrix *A*.

$n \geq 0$. When $n = 0$, ?lanhp is set to zero.

ap — COMPLEX for clanhp.
COMPLEX*16 for zlanhp.
Array, DIMENSION ($n(n+1)/2$). The upper or lower triangle of the Hermitian matrix *A*, packed columnwise in a linear array. The j-th column of *A* is stored in the array *ap* as follows:

if *uplo* = 'U', $ap(i + (j-1)j/2) = A(i,j)$ for $1 \leq i \leq j$;
if *uplo* = 'L', $ap(i + (j-1)(2n-j)/2) = A(i,j)$ for $j \leq i \leq n$.

work — REAL for clanhp.
DOUBLE PRECISION for zlanhp.
Workspace array, DIMENSION (max(1,*lwork*)), where *lwork* ≥ *n* when *norm* = 'I' or '1' or 'O'; otherwise, *work* is not referenced.

Output Parameters

val — REAL for clanhp.
DOUBLE PRECISION for zlanhp.
Value returned by the function.

?lanst/?lanht

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real symmetric or complex Hermitian tridiagonal matrix.

Syntax

```
val = slanst( norm, n, d, e )
val = dlanst( norm, n, d, e )
val = clanht( norm, n, d, e )
val = zlanht( norm, n, d, e )
```

Description

The functions ?lanst/?lanht return the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real symmetric or a complex Hermitian tridiagonal matrix A.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(A), if *norm* = '1' or 'O' or 'o'

= normI(A), if *norm* = 'I' or 'i'

= normF(A), if *norm* = 'F', 'f', 'E' or 'e'

where norm1 denotes the 1-norm of a matrix (maximum column sum), normI denotes the infinity norm of a matrix (maximum row sum) and normF denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a consistent matrix norm.

Input Parameters

norm	CHARACTER*1. Specifies the vaule to be returned by the routine as described above.
n	INTEGER. The order of the matrix A. $n \geq 0$. When $n = 0$, ?lanst/?lanht is set to zero.
d	REAL for slanst/clanht

DOUBLE PRECISION for dlanst/zlanht
Array, DIMENSION (*n*). The diagonal elements of *A*.

e — REAL for slanst
DOUBLE PRECISION for dlanst
COMPLEX for clanht
COMPLEX*16 for zlanht
Array, DIMENSION (*n*-1).
The (*n*-1) sub-diagonal or super-diagonal elements of *A*.

Output Parameters

val — REAL for slanst/clanht
DOUBLE PRECISION for dlanst/zlanht
Value returned by the function.

?lansy

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix.

Syntax

```
val = slansy( norm, uplo, n, a, lda, work )
val = dlansy( norm, uplo, n, a, lda, work )
val = clansy( norm, uplo, n, a, lda, work )
val = zlansy( norm, uplo, n, a, lda, work )
```

Description

The function ?lansy returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix *A*.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(*A*), if *norm* = '1' or 'O' or 'o'

= normI(*A*), if *norm* = 'I' or 'i'

= normF(*A*), if *norm* = 'F', 'f', 'E' or 'e'

where norm1 denotes the 1-norm of a matrix (maximum column sum), normI denotes the infinity norm of a matrix (maximum row sum) and normF denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a consistent matrix norm.

Input Parameters

norm	CHARACTER*1. Specifies the vaule to be returned by the routine as described above.
uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix *A* is to be referenced. = 'U': Upper triangular part of *A* is referenced. = 'L': Lower triangular part of *A* is referenced
n	INTEGER. The order of the matrix *A*. $n \geq 0$. When $n = 0$, ?lansy is set to zero.
a	REAL for slansy DOUBLE PRECISION for dlansy COMPLEX for clansy COMPLEX*16 for zlansy Array, DIMENSION (*lda*,*n*). The symmetric matrix *A*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.
lda	INTEGER. The leading dimension of the array *a*. *lda* ≥ max(*n*,1).
work	REAL for slansy and clansy. DOUBLE PRECISION for dlansy and zlansy. Workspace array, DIMENSION (max(1,*lwork*)), where *lwork* ≥ *n* when *norm* = 'I' or '1' or 'O'; otherwise, *work* is not referenced.

Output Parameters

val	REAL for slansy/clansy DOUBLE PRECISION for dlansy/zlansy Value returned by the function.

?lanhe

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix.

Syntax

val = clanhe(*norm*, *uplo*, *n*, *a*, *lda*, *work*)

val = zlanhe(*norm*, *uplo*, *n*, *a*, *lda*, *work*)

Description

The function ?lanhe returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix *A*.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(*A*), if *norm* = '1' or 'O'or 'o'

= normI(*A*), if *norm* = 'I' or 'i'

= normF(*A*), if *norm* = 'F', 'f', 'E' or 'e'

where norm1 denotes the 1-norm of a matrix (maximum column sum), normI denotes the infinity norm of a matrix (maximum row sum) and normF denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a consistent matrix norm.

Input Parameters

norm	CHARACTER*1. Specifies the vaule to be returned by the routine as described above.
uplo	CHARACTER*1.

	Specifies whether the upper or lower triangular part of the Hermitian matrix *A* is to be referenced. = 'U': Upper triangular part of *A* is referenced. = 'L': Lower triangular part of *A* is referenced
n	INTEGER. The order of the matrix *A*. *n* ≥ 0. When *n* = 0, ?lanhe is set to zero.
a	COMPLEX for clanhe. COMPLEX*16 for zlanhe. Array, DIMENSION (*lda*,*n*). The Hermitian matrix *A*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.
lda	INTEGER. The leading dimension of the array *a*. *lda* ≥ max(*n*,1).
work	REAL for clanhe. DOUBLE PRECISION for zlanhe. Workspace array, DIMENSION (max(1,*lwork*)) , where *lwork* ≥ *n* when *norm* = 'I' or '1' or 'O'; otherwise, *work* is not referenced.

Output Parameters

val	REAL for clanhe. DOUBLE PRECISION for zlanhe. Value returned by the function.

?lantb

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular band matrix.

Syntax

```
val = slantb( norm, uplo, diag, n, k, ab, ldab, work )
val = dlantb( norm, uplo, diag, n, k, ab, ldab, work )
val = clantb( norm, uplo, diag, n, k, ab, ldab, work )
val = zlantb( norm, uplo, diag, n, k, ab, ldab, work )
```

Description

The function ?lantb returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an *n*-by-*n* triangular band matrix *A*, with (*k* + 1) diagonals.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(*A*), if *norm* = '1' or 'O' or 'o'

= normI(*A*), if *norm* = 'I' or 'i'

= normF(*A*), if *norm* = 'F', 'f', 'E' or 'e'

where norm1 denotes the 1-norm of a matrix (maximum column sum), normI denotes the infinity norm of a matrix (maximum row sum) and normF denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a consistent matrix norm.

Input Parameters

norm	CHARACTER*1. Specifies the vaule to be returned by the routine as described above.
uplo	CHARACTER*1. Specifies whether the matrix *A* is upper or lower triangular. = 'U': Upper triangular = 'L': Lower triangular.

diag — CHARACTER*1.
Specifies whether or not the matrix *A* is unit triangular.
= 'N': Non-unit triangular
= 'U': Unit triangular.

n — INTEGER. The order of the matrix *A*. $n \geq 0$. When $n = 0$, ?lantb is set to zero.

k — INTEGER. The number of super-diagonals of the matrix *A* if *uplo* = 'U', or the number of sub-diagonals of the matrix *A* if *uplo* = 'L'. $k \geq 0$.

ab — REAL for slantb
DOUBLE PRECISION for dlantb
COMPLEX for clantb
COMPLEX*16 for zlantb
Array, DIMENSION (*ldab*,*n*). The upper or lower triangular band matrix *A*, stored in the first *k*+1 rows of *ab*.
The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:
if *uplo* = 'U', *ab*(*k*+1+*i*–*j*,*j*) = *a*(*i*,*j*) for max(1,*j*–*k*) ≤ i ≤ j;
if *uplo* = 'L', *ab*(1+*i*–*j*,*j*) = *a*(*i*,*j*) for *j*≤ *i*≤ min(*n*,*j*+*k*).
Note that when *diag* = 'U', the elements of the array *ab* corresponding to the diagonal elements of the matrix *A* are not referenced, but are assumed to be one.

ldab — INTEGER. The leading dimension of the array *ab*.
ldab ≥ *k*+1.

work — REAL for slantb and clantb.
DOUBLE PRECISION for dlantb and zlantb.
Workspace array, DIMENSION (max(1,*lwork*)), where *lwork* ≥ *n* when *norm* = 'I' ; otherwise, *work* is not referenced.

Output Parameters

val — REAL for slantb/clantb.
DOUBLE PRECISION for dlantb/zlantb.

Value returned by the function.

?lantp

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular matrix supplied in packed form.

Syntax

val = slantp(*norm*, *uplo*, *diag*, *n*, *ap*, *work*)

val = dlantp(*norm*, *uplo*, *diag*, *n*, *ap*, *work*)

val = clantp(*norm*, *uplo*, *diag*, *n*, *ap*, *work*)

val = zlantp(*norm*, *uplo*, *diag*, *n*, *ap*, *work*)

Description

The function ?lantp returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular matrix *A*, supplied in packed form.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(*A*), if *norm* = '1' or 'O' or 'o'

= normI(*A*), if *norm* = 'I' or 'i'

= normF(*A*), if *norm* = 'F', 'f', 'E' or 'e'

where norm1 denotes the 1-norm of a matrix (maximum column sum), normI denotes the infinity norm of a matrix (maximum row sum) and normF denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a consistent matrix norm.

Input Parameters

norm	CHARACTER*1. Specifies the vaule to be returned by the routine as described above.
uplo	CHARACTER*1.

Specifies whether the matrix *A* is upper or lower triangular.
= 'U': Upper triangular
= 'L': Lower triangular.

diag CHARACTER*1.
Specifies whether or not the matrix *A* is unit triangular.
= 'N': Non-unit triangular
= 'U': Unit triangular.

n INTEGER. The order of the matrix *A*.
$n \geq 0$. When $n = 0$, ?lantp is set to zero.

ap REAL for slantp
DOUBLE PRECISION for dlantp
COMPLEX for clantp
COMPLEX*16 for zlantp
Array, DIMENSION ($n(n+1)/2$).
The upper or lower triangular matrix *A*, packed columnwise in a linear array. The *j*-th column of *A* is stored in the array *ap* as follows:
if *uplo* = 'U', $AP(i + (j-1)j/2) = a(i,j)$ for $1 \leq i \leq j$;
if *uplo* = 'L', $ap(i + (j-1)(2n-j)/2) = a(i,j)$ for $j \leq i \leq n$.
Note that when *diag* = 'U', the elements of the array *ap* corresponding to the diagonal elements of the matrix *A* are not referenced, but are assumed to be one.

work REAL for slantp and clantp.
DOUBLE PRECISION for dlantp and zlantp.
Workspace array, DIMENSION (max(1, *lwork*)), where *lwork* ≥ *n* when *norm* = 'I' ; otherwise, *work* is not referenced.

Output Parameters

val REAL for slantp/clantp.
DOUBLE PRECISION for dlantp/zlantp.
Value returned by the function.

?lantr

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular matrix.

Syntax

```
val = slantr( norm, uplo, diag, m, n, a, lda, work )
val = dlantr( norm, uplo, diag, m, n, a, lda, work )
val = clantr( norm, uplo, diag, m, n, a, lda, work )
val = zlantr( norm, uplo, diag, m, n, a, lda, work )
```

Description

The function ?lantr returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular matrix *A*.

The value *val* returned by the function is:

val = max(abs(A_{ij})), if *norm* = 'M' or 'm'

= norm1(*A*), if *norm* = '1' or 'O' or 'o'

= normI(*A*), if *norm* = 'I' or 'i'

= normF(*A*), if *norm* = 'F', 'f', 'E' or 'e'

where norm1 denotes the 1-norm of a matrix (maximum column sum), normI denotes the infinity norm of a matrix (maximum row sum) and normF denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(A_{ij})) is not a consistent matrix norm.

Input Parameters

norm — CHARACTER*1. Specifies the vaule to be returned by the routine as described above.

uplo — CHARACTER*1.
Specifies whether the matrix *A* is upper or lower trapezoidal.
= 'U': Upper trapezoidal
= 'L': Lower trapezoidal.

Note that *A* is triangular instead of trapezoidal if *m* = *n*.

diag — CHARACTER*1.
Specifies whether or not the matrix *A* has unit diagonal.
= 'N': Non-unit diagonal
= 'U': Unit diagonal.

m — INTEGER. The number of rows of the matrix *A*. *m* ≥ 0, and if *uplo* = 'U', *m* ≤ *n*.
When *m* = 0, ?lantr is set to zero.

n — INTEGER. The number of columns of the matrix *A*. *n* ≥ 0, and if *uplo* = 'L', *n* ≤ *m*.
When *n* = 0, ?lantr is set to zero.

a — REAL for slantr
DOUBLE PRECISION for dlantr
COMPLEX for clantr
COMPLEX*16 for zlantr
Array, DIMENSION (*lda*,*n*).
The trapezoidal matrix *A* (*A* is triangular if *m* = *n*).
If *uplo* = 'U', the leading *m*-by-*n* upper trapezoidal part of the array *a* contains the upper trapezoidal matrix, and the strictly lower triangular part of *A* is not referenced.
If *uplo* = 'L', the leading *m*-by-*n* lower trapezoidal part of the array *a* contains the lower trapezoidal matrix, and the strictly upper triangular part of *A* is not referenced. Note that when *diag* = 'U', the diagonal elements of *A* are not referenced and are assumed to be one.

lda — INTEGER. The leading dimension of the array *a*.
lda ≥ max(*m*,1).

work — REAL for slantr/clantrp.
DOUBLE PRECISION for dlantr/zlantr.
Workspace array, DIMENSION (max(1,*lwork*)), where *lwork* ≥ *m* when *norm* = 'I' ; otherwise, *work* is not referenced.

Output Parameters

val	REAL for slantr/clantrp. DOUBLE PRECISION for dlantr/zlantr. Value returned by the function.

?lanv2

Computes the Schur factorization of a real 2-by-2 nonsymmetric matrix in standard form.

Syntax

```
call slanv2( a, b, c, d, rt1r, rt1i, rt2r, rt2i, cs, sn )
call dlanv2( a, b, c, d, rt1r, rt1i, rt2r, rt2i, cs, sn )
```

Description

The routine computes the Schur factorization of a real 2-by-2 nonsymmetric matrix in standard form:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} cs & -sn \\ sn & cs \end{bmatrix} \begin{bmatrix} aa & bb \\ cc & dd \end{bmatrix} \begin{bmatrix} cs & sn \\ -sn & cs \end{bmatrix}$$

where either

1. *cc* = 0 so that *aa* and *dd* are real eigenvalues of the matrix, or
2. *aa* = *dd* and *bb***cc* < 0, so that *aa* ± sqrt(*bb***cc*) are complex conjugate eigenvalues.

The routine was adjusted to reduce the risk of cancellation errors, when computing real eigenvalues, and to ensure, if possible, that abs(*rt1r*) ≥ abs(*rt2r*).

Input Parameters

a, *b*, *c*, *d*	REAL for slanv2 DOUBLE PRECISION for dlanv2. On entry, elements of the input matrix.

Output Parameters

a, *b*, *c*, *d*	On exit, overwritten by the elements of the standardized Schur form.
rt1r, *rt1i*, *rt2r*, *rt2i*	REAL for slanv2 DOUBLE PRECISION for dlanv2. The real and imaginary parts of the eigenvalues. If the eigenvalues are a complex conjugate pair, *rt1i* > 0.
cs, *sn*	REAL for slanv2 DOUBLE PRECISION for dlanv2. Parameters of the rotation matrix.

?lapll

Measures the linear dependence of two vectors.

Syntax

```
call slapll( n, x, incx, Y, incy, ssmin )
call dlapll( n, x, incx, Y, incy, ssmin )
call clapll( n, x, incx, Y, incy, ssmin )
call zlapll( n, x, incx, Y, incy, ssmin )
```

Description

Given two column vectors *x* and *y* of length *n*, let

A = (*x y*) be the *n*-by-2 matrix.

The routine ?lapll first computes the *QR* factorization of *A* as *A* = *Q***R* and then computes the SVD of the 2-by-2 upper triangular matrix *R*. The smaller singular value of *R* is returned in *ssmin*, which is used as the measurement of the linear dependency of the vectors *x* and *y*.

Input Parameters

n	INTEGER. The length of the vectors *x* and *y*.
x	REAL for slapll DOUBLE PRECISION for dlapll COMPLEX for clapll

COMPLEX*16 for zlapll
Array, DIMENSION (1+(*n*–1)*incx*).
On entry, *x* contains the *n*-vector x.

y — REAL for slapll
DOUBLE PRECISION for dlapll
COMPLEX for clapll
COMPLEX*16 for zlapll
Array, DIMENSION (1+(*n*-1)*incy*).
On entry, *y* contains the *n*-vector y.

incx — INTEGER. The increment between successive elements of x; *incx* > 0.

incy — INTEGER. The increment between successive elements of y; *incy* > 0.

Output Parameters

x — On exit, *x* is overwritten.

y — On exit, *y* is overwritten.

ssmin — REAL for slapll/clapll
DOUBLE PRECISION for dlapll/zlapll
The smallest singular value of the *n*-by-2 matrix *A* = (*x y*)
.

?lapmt

Performs a forward or backward permutation of the columns of a matrix.

Syntax

```
call slapmt( forwrd, m, n, x, ldx, k )
call dlapmt( forwrd, m, n, x, ldx, k )
call clapmt( forwrd, m, n, x, ldx, k )
call zlapmt( forwrd, m, n, x, ldx, k )
```

Description

The routine ?lapmt rearranges the columns of the *m*-by-*n* matrix *X* as specified by the permutation *k*(1),*k*(2),...,*k*(*n*) of the integers 1,...,*n*.

If *forwrd* = .TRUE., forward permutation:

X(*,*k*(*j*)) is moved to *X*(*,*j*) for *j*=1,2,...,*n*.

If *forwrd* = .FALSE., backward permutation:

X(*,*j*) is moved to *X*(*,*k*(*j*)) for *j* = 1,2,...,*n*.

Input Parameters

forwrd	LOGICAL. If *forwrd* = .TRUE., forward permutation If *forwrd* = .FALSE., backward permutation
m	INTEGER. The number of rows of the matrix *X*. *m* ≥ 0.
n	INTEGER. The number of columns of the matrix *X*. *n* ≥ 0.
x	REAL for slapmt DOUBLE PRECISION for dlapmt COMPLEX for clapmt COMPLEX*16 for zlapmt Array, DIMENSION (*ldx*,*n*). On entry, the *m*-by-*n* matrix *X*.
ldx	INTEGER. The leading dimension of the array *X*, *ldx* ≥ max(1,*m*).
k	INTEGER. Array, DIMENSION (*n*). On entry, *k* contains the permutation vector and is used as internal workspace.

Output Parameters

x	On exit, *x* contains the permuted matrix *X*.
k	On exit, *k* is reset to its original value.

?lapy2

Returns sqrt(x^2+y^2).

Syntax

val = slapy2(*x*, *y*)

val = dlapy2(*x*, *y*)

Description

The function ?lapy2 returns sqrt(x^2+y^2), avoiding unnecessary overflow or harmful underflow.

Input Parameters

x, *y*	REAL for slapy2 DOUBLE PRECISION for dlapy2 Specify the input values *x* and *y*.

Output Parameters

val	REAL for slapy2 DOUBLE PRECISION for dlapy2. Value returned by the function.

?lapy3

Returns sqrt($x^2+y^2+z^2$).

Syntax

val = slapy3(*x*, *y*, *z*)

val = dlapy3(*x*, *y*, *z*)

Description

The function ?lapy3 returns sqrt($x^2+y^2+z^2$), avoiding unnecessary overflow or harmful underflow.

Input Parameters

x, *y*, *z*	REAL for slapy3 DOUBLE PRECISION for dlapy3 Specify the input values *x*, *y* and *z*.

Output Parameters

val	REAL for slapy3 DOUBLE PRECISION for dlapy3. Value returned by the function.

?laqgb

Scales a general band matrix, using row and column scaling factors computed by ?gbequ.

Syntax

```
call slaqgb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, equed )
call dlaqgb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, equed )
call claqgb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, equed )
call zlaqgb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, equed )
```

Description

The routine equilibrates a general *m*-by-*n* band matrix *A* with *kl* subdiagonals and *ku* superdiagonals using the row and column scaling factors in the vectors *r* and *c*.

Input Parameters

m	INTEGER. The number of rows of the matrix *A*. *m* ≥ 0.
n	INTEGER. The number of columns of the matrix *A*. *n* ≥ 0.
kl	INTEGER. The number of subdiagonals within the band of *A*. *kl* ≥ 0.
ku	INTEGER. The number of superdiagonals within the band of *A*. *ku* ≥ 0.

ab
REAL for slaqgb
DOUBLE PRECISION for dlaqgb
COMPLEX for claqgb
COMPLEX*16 for zlaqgb
Array, DIMENSION (*ldab*,*n*). On entry, the matrix *A* in band storage, in rows 1 to *kl*+*ku*+1. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:
ab(*ku*+1+*i*-*j*,*j*) = *A*(*i*,*j*) for max(1,*j*-*ku*) ≤ i ≤ min(*m*,*j*+*kl*).

ldab
INTEGER. The leading dimension of the array *ab*.
lda ≥ *kl*+*ku*+1.

amax
REAL for slaqgb/claqgb
DOUBLE PRECISION for dlaqgb/zlaqgb
Absolute value of largest matrix entry.

r, *c*
REAL for slaqgb/claqgb
DOUBLE PRECISION for dlaqgb/zlaqgb
Arrays *r* (*m*), *c* (*n*). Contain the row and column scale factors for *A*, respectively.

rowcnd
REAL for slaqgb/claqgb
DOUBLE PRECISION for dlaqgb/zlaqgb
Ratio of the smallest *r*(*i*) to the largest *r*(*i*).

colcnd
REAL for slaqgb/claqgb
DOUBLE PRECISION for dlaqgb/zlaqgb
Ratio of the smallest *c*(*i*) to the largest *c*(*i*).

Output Parameters

ab
On exit, the equilibrated matrix, in the same storage format as *A*.
See *equed* for the form of the equilibrated matrix.

equed
CHARACTER*1.
Specifies the form of equilibration that was done.
If *equed* = 'N': No equilibration
If *equed* = 'R': Row equilibration, that is, *A* has been premultiplied by diag(*r*).
If *equed* = 'C': Column equilibration, that is, *A* has been postmultiplied by diag(*c*).

If *equed* = 'B': Both row and column equilibration, that is, *A* has been replaced by diag(*r*)*A*diag(*c*).

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if row or column scaling should be done based on the ratio of the row or column scaling factors. If *rowcnd* < *thresh*, row scaling is done, and if *colcnd* < *thresh*, column scaling is done. *large* and *small* are threshold values used to decide if row scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, row scaling is done.

?laqge

Scales a general rectangular matrix, using row and column scaling factors computed by ?geequ.

Syntax

```
call slaqge( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
call dlaqge( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
call claqge( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
call zlaqge( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
```

Description

The routine equilibrates a general *m*-by-*n* matrix *A* using the row and column scaling factors in the vectors *r* and *c*.

Input Parameters

m	INTEGER. The number of rows of the matrix *A*. *m* ≥ 0.
n	INTEGER. The number of columns of the matrix *A*. *n* ≥ 0.
a	REAL for slaqge DOUBLE PRECISION for dlaqge COMPLEX for claqge

COMPLEX*16 for zlaqge
Array, DIMENSION (*lda*,*n*). On entry, the *m*-by-*n* matrix *A*.

lda INTEGER. The leading dimension of the array *a*.
lda ≥ max(*m*,1).

r REAL for slanqge/claqge
DOUBLE PRECISION for dlaqge/zlaqge
Array, DIMENSION (*m*). The row scale factors for A.

c REAL for slanqge/claqge
DOUBLE PRECISION for dlaqge/zlaqge
Array, DIMENSION (*n*). The column scale factors for A.

rowcnd REAL for slanqge/claqge
DOUBLE PRECISION for dlaqge/zlaqge
Ratio of the smallest *r*(*i*) to the largest *r*(*i*).

colcnd REAL for slanqge/claqge
DOUBLE PRECISION for dlaqge/zlaqge
Ratio of the smallest *c*(*i*) to the largest *c*(*i*).

amax REAL for slanqge/claqge
DOUBLE PRECISION for dlaqge/zlaqge
Absolute value of largest matrix entry.

Output Parameters

a On exit, the equilibrated matrix.
See *equed* for the form of the equilibrated matrix.

equed CHARACTER*1.
Specifies the form of equilibration that was done.
If *equed* = 'N': No equilibration
If *equed* = 'R': Row equilibration, that is, *A* has been premultiplied by diag(*r*).
If *equed* = 'C': Column equilibration, that is, *A* has been postmultiplied by diag(*c*).
If *equed* = 'B': Both row and column equilibration, that is, *A* has been replaced by diag(*r*)*A*diag(*c*).

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if row or column scaling should be done based on the ratio of the row or column scaling factors. If *rowcnd* < *thresh*, row scaling is done, and if *colcnd* < *thresh*, column scaling is done. *large* and *small* are threshold values used to decide if row scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, row scaling is done.

?laqhb

Scales a Hermetian band matrix, using scaling factors computed by ?pbequ.

Syntax

call claqhb(*uplo*, *n*, *kd*, *ab*, *ldab*, *s*, *scond*, *amax*, *equed*)

call zlaqhb(*uplo*, *n*, *kd*, *ab*, *ldab*, *s*, *scond*, *amax*, *equed*)

Description

The routine equilibrates a Hermetian band matrix *A* using the scaling factors in the vector *s*.

Input Parameters

uplo — CHARACTER*1.
Specifies whether the upper or lower triangular part of the band matrix *A* is stored.
If *uplo* = 'U': upper triangular.
If *uplo* = 'L': lower triangular.

n — INTEGER. The order of the matrix *A*.
n ≥ 0.

kd — INTEGER. The number of super-diagonals of the matrix *A* if *uplo* = 'U', or the number of sub-diagonals if *uplo* = 'L'.
kd ≥ 0.

ab — COMPLEX for claqhb
COMPLEX*16 for zlaqhb

Array, DIMENSION (*ldab*,*n*). On entry, the upper or lower triangle of the band matrix *A*, stored in the first *kd*+1 rows of the array. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:
if *uplo* = 'U', *ab*(*kd*+1+*i*-*j*,*j*) = A(*i*,*j*) for max(1,*j*-*kd*) ≤ *i* ≤ *j*;
if *uplo* = 'L', *ab*(1+*i*-*j*,*j*) = A(*i*,*j*) for *j* ≤ *i* ≤ min(*n*,*j*+*kd*).

ldab — INTEGER. The leading dimension of the array *ab*.
ldab ≥ *kd*+1.

scond — REAL for claqsb
DOUBLE PRECISION for zlaqsb
Ratio of the smallest *s*(*i*) to the largest *s*(*i*).

amax — REAL for claqsb
DOUBLE PRECISION for zlaqsb
Absolute value of largest matrix entry.

Output Parameters

ab — On exit, if *info* = 0, the triangular factor *U* or *L* from the Cholesky factorization *A* = *U*'**U* or *A* = *L***L*' of the band matrix *A*, in the same storage format as *A*.

s — REAL for claqsb
DOUBLE PRECISION for zlaqsb
Array, DIMENSION (*n*). The scale factors for A.

equed — CHARACTER*1.
Specifies whether or not equilibration was done.
If *equed* = 'N': No equilibration.
If *equed* = 'Y': Equilibration was done, that is, *A* has been replaced by diag(*s*)**A***diag(*s*).

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if scaling should be based on the ratio of the scaling factors. If *scond* < *thresh*, scaling is done.

The values *large* and *small* are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, scaling is done.

?laqp2

Computes a QR factorization with column pivoting of the matrix block.

Syntax

```
call slaqp2( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
call dlaqp2( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
call claqp2( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
call zlaqp2( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
```

Description

The routine computes a *QR* factorization with column pivoting of the block A(*offset*+1:*m*,1:*n*). The block A(1:*offset*,1:*n*) is accordingly pivoted, but not factorized.

Input Parameters

m	INTEGER. The number of rows of the matrix *A*. *m* ≥ 0.
n	INTEGER. The number of columns of the matrix *A*. *n* ≥ 0.
offset	INTEGER. The number of rows of the matrix *A* that must be pivoted but no factorized. *offset* ≥ 0.
a	REAL for slaqp2 DOUBLE PRECISION for dlaqp2 COMPLEX for claqp2 COMPLEX*16 for zlaqp2 Array, DIMENSION (*lda*,*n*). On entry, the *m*-by-*n* matrix *A*.
lda	INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1,*m*).
jpvt	INTEGER. Array, DIMENSION (*n*).

On entry, if *jpvt*(*i*) ≠ 0, the *i*-th column of *A* is permuted to the front of *A***P* (a leading column); if *jpvt*(*i*) = 0, the *i*-th column of *A* is a free column.

vn1, *vn2*
REAL for slaqp2/claqp2
DOUBLE PRECISION for dlaqp2/zlaqp2
Arrays, DIMENSION (*n*) each. Contain the vectors with the partial and exact column norms, respectively.

work
REAL for slaqp2
DOUBLE PRECISION for dlaqp2
COMPLEX for claqp2
COMPLEX*16 for zlaqp2 Workspace array, DIMENSION (*n*).

Output Parameters

a
On exit, the upper triangle of block *A*(*offset*+1:*m*,1:*n*) is the triangular factor obtained; the elements in block *A*(*offset*+1:*m*,1:*n*) below the diagonal, together with the array *tau*, represent the orthogonal matrix *Q* as a product of elementary reflectors. Block *A*(1:*offset*,1:*n*) has been accordingly pivoted, but not factorized.

jpvt
On exit, if *jpvt*(*i*) = *k*, then the *i*-th column of *A***P* was the *k*-th column of *A*.

tau
REAL for slaqp2
DOUBLE PRECISION for dlaqp2
COMPLEX for claqp2
COMPLEX*16 for zlaqp2
Array, DIMENSION (min(*m*,*n*)).
The scalar factors of the elementary reflectors.

vn1, *vn2*
Contain the vectors with the partial and exact column norms, respectively.

?laqps

Computes a step of QR factorization with column pivoting of a real m-by-n matrix A by using BLAS level 3.

Syntax

```
call slaqps( m, n, offset, nb, kb, a, lda, jpvt, tau, vn1, vn2, auxv, f, ldf )
call dlaqps( m, n, offset, nb, kb, a, lda, jpvt, tau, vn1, vn2, auxv, f, ldf )
call claqps( m, n, offset, nb, kb, a, lda, jpvt, tau, vn1, vn2, auxv, f, ldf )
call zlaqps( m, n, offset, nb, kb, a, lda, jpvt, tau, vn1, vn2, auxv, f, ldf )
```

Description

This routine computes a step of *QR* factorization with column pivoting of a real *m*-by-*n* matrix *A* by using BLAS level 3. The routine tries to factorize *NB* columns from *A* starting from the row *offset*+1, and updates all of the matrix with BLAS level 3 routine ?gemm.

In some cases, due to catastrophic cancellations, ?laqps cannot factorize *NB* columns. Hence, the actual number of factorized columns is returned in *kb*.

Block *A*(1:*offset*,1:*n*) is accordingly pivoted, but not factorized.

Input Parameters

m	INTEGER. The number of rows of the matrix *A*. *m* ≥ 0.
n	INTEGER. The number of columns of the matrix *A*. *n* ≥ 0.
offset	INTEGER. The number of rows of *A* that have been factorized in previous steps.
nb	INTEGER. The number of columns to factorize.
a	REAL for slaqps DOUBLE PRECISION for dlaqps COMPLEX for claqps COMPLEX*16 for zlaqps

Array, DIMENSION (*lda*,*n*).
On entry, the *m*-by-*n* matrix *A*.

lda INTEGER. The leading dimension of the array *a*.

lda ≥ max(1,*m*).

jpvt INTEGER. Array, DIMENSION (*n*).
If *jpvt*(*I*) = *k* then column *k* of the full matrix *A* has been permuted into position *i* in AP.

vn1, *vn2* REAL for slaqps/claqps
DOUBLE PRECISION for dlaqps/zlaqps
Arrays, DIMENSION (*n*) each. Contain the vectors with the partial and exact column norms, respectively.

auxv REAL for slaqps
DOUBLE PRECISION for dlaqps
COMPLEX for claqps
COMPLEX*16 for zlaqps
Array, DIMENSION (*nb*). Auxiliary vector.

f REAL for slaqps
DOUBLE PRECISION for dlaqps
COMPLEX for claqps
COMPLEX*16 for zlaqps
Array, DIMENSION (*ldf*,*nb*). Matrix *F'* = *L***Y'***A*.

ldf INTEGER. The leading dimension of the array *f*.

ldf ≥ max(1,*n*).

Output Parameters

kb INTEGER. The number of columns actually factorized.

a On exit, block *A*(*offset*+1:*m*,1:*kb*) is the triangular factor obtained and block *A*(1:*offset*,1:*n*) has been accordingly pivoted, but no factorized. The rest of the matrix, block *A*(*offset*+1:*m*,*kb*+1:*n*) has been updated.

jpvt INTEGER array, DIMENSION (*n*). If *jpvt*(*I*) = *k* then column *k* of the full matrix *A* has been permuted into position *i* in AP.

tau REAL for slaqps
DOUBLE PRECISION for dlaqps

	COMPLEX for claqps COMPLEX*16 for zlaqps Array, DIMENSION (*kb*). The scalar factors of the elementary reflectors.
vn1, *vn2*	The vectors with the partial and exact column norms, respectively.
auxv	Auxiliary vector.
f	Matrix *F'* = *L***Y*'**A*.

?laqr0

Computes the eigenvalues of a Hessenberg matrix, and optionally the marixes from the Schur decomposition.

Syntax

```
call slaqr0( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz, work, lwork, info )
call dlaqr0( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz, work, lwork, info )
call claqr0( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, work, lwork, info )
call zlaqr0( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, work, lwork, info )
```

Description

This routine computes the eigenvalues of a Hessenberg matrix *H*, and, optionally, the matrices *T* and *Z* from the Schur decomposition $H=Z*T*Z^H$, where *T* is an upper quasi-triangular/triangular matrix (the Schur form), and *Z* is the orthogonal/unitary matrix of Schur vectors.

Optionally *Z* may be postmultiplied into an input orthogonal/unitary matrix *Q* so that this routine can give the Schur factorization of a matrix *A* which has been reduced to the Hessenberg form *H* by the orthogonal/unitary matrix *Q*: $A = Q*H*Q^H = (QZ)*H*(QZ)^H$.

Input Parameters

wantt	LOGICAL.

If *wantt* = .TRUE., the full Schur form T is required;
If *wantt* = .FALSE., only eigenvalues are required.

wantz — LOGICAL.
If *wantz* = .TRUE., the matrix of Schur vectors *Z* is required;
If *wantz* = .FALSE., Schur vectors are not required.

n — INTEGER. The order of the Hessenberg matrix *H*. (*n* ≥ 0).

ilo, *ihi* — INTEGER.
It is assumed that *H* is already upper triangular in rows and columns 1:*ilo*-1 and *ihi*+1:*n*, and if *ilo* > 1 then *H*(*ilo*, *ilo*-1) = 0.
ilo and *ihi* are normally set by a previous call to cgebal, and then passed to cgehrd when the matrix output by cgebal is reduced to Hessenberg form. Otherwise, *ilo* and *ihi* should be set to 1 and *n*, respectively.
If *n* > 0, then 1 ≤ *ilo* ≤ *ihi* ≤ *n*.
If *n*=0, then *ilo*=1 and *ihi*=0

h — REAL for slaqr0
DOUBLE PRECISION for dlaqr0
COMPLEX for claqr0
COMPLEX*16 for zlaqr0.
Array, DIMENSION (*ldh*, *n*), contains the upper Hessenberg matrix *H*.

ldh — INTEGER. The leading dimension of the array *h*. *ldh* ≥ max(1, *n*).

iloz, *ihiz* — INTEGER. Specify the rows of *Z* to which transformations must be applied if *wantz* is .TRUE., 1 ≤ *iloz* ≤ *ilo*; *ihi* ≤ *ihiz* ≤ *n*.

z — REAL for slaqr0
DOUBLE PRECISION for dlaqr0
COMPLEX for claqr0
COMPLEX*16 for zlaqr0.
Array, DIMENSION (*ldz*, *ihi*), contains the matrix *Z* if *wantz* is .TRUE.. If *wantz* is .FALSE., *z* is not referenced.

ldz	INTEGER. The leading dimension of the array *z*. If *wantz* is .TRUE., then *ldz* ≥ max(1, *ihiz*). Otherwise, *ldz* ≥ 1.
work	REAL for slaqr0 DOUBLE PRECISION for dlaqr0 COMPLEX for claqr0 COMPLEX*16 for zlaqr0. Workspace array with dimension *lwork*.
lwork	INTEGER. The dimension of the array *work*. *lwork* ≥ max(1,*n*) is sufficient, but for the optimal performance a greater workspace may be required, typically as large as 6**n*. It is recommended to use the worlspace query to determine the optimal workspace size. If *lwork*=–1, then the routine performs a workspace query: it estimates the optimal workspace size for the given values of the input parameters *n*, *ilo*, and *ihi*. The estimate is returned in *work*(1). No error messages related to the *lwork* is issued by xerbla. Neither *H* nor *Z* are accessed.

Output Parameters

h	If *info*=0 , and *wantt* is .TRUE., then *h* contains the upper quasi-triangular/triangular matrix *T* from the Schur decomposition (the Schur form). If *info*=0 , and *wantt* is .FALSE., then the contents of *h* are unspecified on exit. (The output values of *h* when *info* > 0 are given under the description of the *info* parameter below.) The routine may explicitly set *h*(i,j) for i>j and j=1,2,...*ilo*–1 or j=*ihi*+1, *ihi*+2,...*n*.
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
w	COMPLEX for claqr0 COMPLEX*16 for zlaqr0.

Arrays, DIMENSION(*n*). The computed eigenvalues of *h*(*ilo*:*ihi*, *ilo*:*ihi*) are stored in *w*(*ilo*:*ihi*). If *wantt* is .TRUE., then the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*, with *w*(i) = *h*(i,i).

wr, *wi*

REAL for slaqr0
DOUBLE PRECISION for dlaqr0
Arrays, DIMENSION(*ihi*) each. The real and imaginary parts, respectively, of the computed eigenvalues of *h*(*ilo*:*ihi*, *ilo*:*ihi*) are stored in the *wr*(*ilo*:*ihi*) and *wi*(*ilo*:*ihi*). If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of *wr* and *wi*, say the *i*-th and (*i*+1)-th, with *wi*(*i*)> 0 and *wi*(*i*+1) < 0. If *wantt* is .TRUE. , then the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*, with *wr*(i) = *h*(i,i), and if *h*(i:i+1,i:i+1)is a 2-by-2 diagonal block, then *wi*(i)=sqrt(-*h*(i+1,i)*h(i,i+1)).

z

If *wantz* is .TRUE., then *z(ilo:ihi, iloz:ihiz)* is replaced by *z(ilo:ihi, iloz:ihiz)*U*, where *U* is the orthogonal/unitary Schur factor of *h(ilo:ihi, ilo:ihi)*. If *wantz* is .FALSE., *z* is not referenced.
(The output values of *z* when *info* > 0 are given under the description of the *info* parameter below.)

info

INTEGER.
= 0: the execution is successful.
> 0: if *info* = *i*, then the routine failed to compute all the eigenvalues. Elements 1:*ilo*-1 and i+1:*n* of *wr* and *wi* contain those eigenvalues which have been successfully computed.
> 0: if *wantt* is .FALSE., then the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns *ilo* through *info* of the final output value of *h*.

> 0: if *wantt* is .TRUE., then (initial value of *h*)*U = U*(final value of *h*, where *U* is an orthogonal/unitary matrix. The final value of *h* is upper Hessenberg and quasi-triangular/triangular in rows and columns *info*+1 through *ihi*.
> 0: if *wantz* is .TRUE., then (final value of *z*(*ilo*:*ihi*, *iloz*:*ihiz*))=(initial value of *z*(*ilo*:*ihi*, *iloz*:*ihiz*)*U, where *U* is the orthogonal/unitary matrix in the previous expression (regardless of the value of *wantt*).
> 0: if *wantz* is .FALSE., then *z* is not accessed.

?laqr1

Sets a scalar multiple of the first column of the product of 2-by-2 or 3-by-3 matrix H and specified shifts.

Syntax

```
call slaqr1( n, h, ldh, sr1, si1, sr2, si2, v )
call dlaqr1( n, h, ldh, sr1, si1, sr2, si2, v )
call claqr1( n, h, ldh, s1, s2, v )
call zlaqr1( n, h, ldh, s1, s2, v )
```

Description

Given a 2-by-2 or 3-by-3 matrix *H*, this routine sets *v* to a scalar multiple of the first column of the product

K = (*H* - *s1*I*)*(*H* - *s2*I*), or *K* = (*H* - (*sr1* + i**si1*)*I*)*(*H* - (*sr2* + i**si2*)*I*)

scaling to avoid overflows and most underflows.

It is assumed that either 1) *sr1* = *sr2* and *si1* = -*si2*, or 2) *si1* = *si2* = 0.

This is useful for starting double implicit shift bulges in the QR algorithm.

Input Parameters

n INTEGER.
The order of the matrix *H*. *n* must be equal to 2 or 3.

sr1, *si2*, *sr2*, *si2*
REAL for slaqr1
DOUBLE PRECISION for dlaqr1
Shift values that define *K* in the formula above.

s1, *s2*
COMPLEX for claqr1
COMPLEX*16 for zlaqr1.
Shift values that define *K* in the formula above.

h
REAL for slaqr1
DOUBLE PRECISION for dlaqr1
COMPLEX for claqr1
COMPLEX*16 for zlaqr1.
Array, DIMENSION (*ldh*, *n*), contains 2-by-2 or 3-by-3 matrix *H* in the formula above.

ldh
INTEGER.
The leading dimension of the array *h* just as declared in the calling routine. *ldh* ≥ *n*.

Output Parameters

v
REAL for slaqr1
DOUBLE PRECISION for dlaqr1
COMPLEX for claqr1
COMPLEX*16 for zlaqr1.
Array with dimension (*n*).
A scalar multiple of the first column of the matrix *K* in the formula above.

?laqr2

Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggresive early deflation).

Syntax

```
call slaqr2( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns,
nd, sr, si, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
```

```
call dlaqr2( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns,
nd, sr, si, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
```

```
call claqr2( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns,
nd, sh, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
```

```
call zlaqr2( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns,
nd, sh, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
```

Description

This routine accepts as input an upper Hessenberg matrix *H* and performs an orthogonal/unitary similarity transformation designed to detect and deflate fully converged eigenvalues from a trailing principal submatrix. On output *H* has been overwritten by a new Hessenberg matrix that is a perturbation of an orthogonal/unitary similarity transformation of *H*. It is to be hoped that the final version of *H* has many zero subdiagonal entries.

This subroutine is identical to ?laqr3 except that it avoids recursion by calling ?lahqr instead of ?laqr4.

Input Parameters

wantt — LOGICAL.
If *wantt* = .TRUE., then the Hessenberg matrix *H* is fully updated so that the quasi-triangular/triangular Schur factor may be computed (in cooperation with the calling subroutine).
If *wantt* = .FALSE., then only enough of *H* is updated to preserve the eigenvalues.

wantz — LOGICAL.

If *wantz* = .TRUE., then the orthogonal/unitary matrix *Z* is updated so that the orthogonal/unitary Schur factor may be computed (in cooperation with the calling subroutine). If *wantz* = .FALSE., then *Z* is not referenced.

n — INTEGER. The order of the Hessenberg matrix *H* and (if *wantz* = .TRUE.) the order of the orthogonal/unitary matrix *Z*. .

ktop — INTEGER.
It is assumed that either *ktop*=1 or *h*(*ktop*,*ktop-1*)=0. *ktop* and *kbot* together determine an isolated block along the diagonal of the Hessenberg matrix.

kbot — INTEGER.
It is assumed without a check that either *kbot*=*n* or *h*(*kbot*+1,*kbot*)=0. *ktop* and *kbot* together determine an isolated block along the diagonal of the Hessenberg matrix.

nw — INTEGER.
Size of the deflation window. 1 ≤ *nw* ≤ (*kbot*-*ktop*+1).

h — REAL for slaqr2
DOUBLE PRECISION for dlaqr2
COMPLEX for claqr2
COMPLEX*16 for zlaqr2.
Array, DIMENSION (*ldh*, *n*), on input the initial *n*-by-*n* section of *h* stores the Hessenberg matrix *H* undergoing aggressive early deflation.

ldh — INTEGER. The leading dimension of the array *h* just as declared in the calling subroutine. *ldh*≥*n*.

iloz, *ihiz* — INTEGER. Specify the rows of *Z* to which transformations must be applied if *wantz* is .TRUE.. 1 ≤ *iloz* ≤ *ihiz* ≤ *n*.

z — REAL for slaqr2
DOUBLE PRECISION for dlaqr2
COMPLEX for claqr2
COMPLEX*16 for zlaqr2.
Array, DIMENSION (*ldz*, *ihi*), contains the matrix *Z* if *wantz* is .TRUE.. If *wantz* is .FALSE., then *z* is not referenced.

ldz	INTEGER. The leading dimension of the array *z* just as declared in the calling subroutine. *ldz* ≥ 1.
v	REAL for slaqr2 DOUBLE PRECISION for dlaqr2 COMPLEX for claqr2 COMPLEX*16 for zlaqr2. Workspace array with dimension (*ldv*, *nw*). An *nw*-by-*nw* work array.
ldv	INTEGER. The leading dimension of the array *v* just as declared in the calling subroutine. *ldv* ≥ *nw*.
nh	INTEGER. The number of column of *t*. *nh* ≥ *nw*.
t	REAL for slaqr2 DOUBLE PRECISION for dlaqr2 COMPLEX for claqr2 COMPLEX*16 for zlaqr2. Workspace array with dimension (*ldt*, *nw*).
ldt	INTEGER. The leading dimension of the array *t* just as declared in the calling subroutine. *ldt*≥*nw*.
nv	INTEGER. The number of rows of work array *wv* available for workspace. *nv*≥*nw*.
wv	REAL for slaqr2 DOUBLE PRECISION for dlaqr2 COMPLEX for claqr2 COMPLEX*16 for zlaqr2. Workspace array with dimension (*ldwv*, *nw*).
ldwv	INTEGER. The leading dimension of the array *wv* just as declared in the calling subroutine. *ldwv*≥*nw*.
work	REAL for slaqr2 DOUBLE PRECISION for dlaqr2 COMPLEX for claqr2 COMPLEX*16 for zlaqr2. Workspace array with dimension *lwork*.
lwork	INTEGER. The dimension of the array *work*.

lwork=2**nw*) is sufficient, but for the optimal performance a greater workspace may be required.
If *lwork*=-1, then the routine performs a workspace query: it estimates the optimal workspace size for the given values of the input parameters *n*, *nw*, *ktop*, and *kbot*. The estimate is returned in *work*(1). No error messages related to the *lwork* is issued by xerbla. Neither *H* nor *Z* are accessed.

Output Parameters

h
On output *h* has been transformed by an orthogonal/unitary similarity transformation, perturbed, and the returned to Hessenberg form that (it is to be hoped) has some zero subdiagonal entries.

work(1)
On exit *work*(1) is set to an estimate of the optimal value of *lwork* for the given values of the input parameters *n*, *nw*, *ktop*, and *kbot*.

z
If *wantz* is .TRUE., then the orthogonal/unitary similarity transformation is accumulated into *z(iloz:ihiz, ilo:ihi)* from the right.
If *wantz* is .FALSE., then *z* is unreferenced.

nd
INTEGER. The number of converged eigenvalues uncovered by the routine.

ns
INTEGER. The number of unconverged, that is approximate eigenvalues returned in *sr*, *si* or in *sh* that may be used as shifts by the calling subroutine.

sh
COMPLEX for claqr2
COMPLEX*16 for zlaqr2.
Arrays, DIMENSION (*kbot*).
The approximate eigenvalues that may be used for shifts are stored in the *sh*(*kbot*-*nd*-*ns*+1) through the *sh*(*kbot*-*nd*).
The converged eigenvalues are stored in the *sh*(*kbot*-*nd*+1) through the *sh*(*kbot*).

sr, *si*
REAL for slaqr2
DOUBLE PRECISION for dlaqr2
Arrays, DIMENSION (*kbot*) each.

The real and imaginary parts of the approximate eigenvalues that may be used for shifts are stored in the *sr*(*kbot*-*nd*-*ns*+1) through the *sr*(*kbot*-*nd*), and *si*(*kbot*-*nd*-*ns*+1) through the *si*(*kbot*-*nd*), respectively. The real and imaginary parts of converged eigenvalues are stored in the *sr*(*kbot*-*nd*+1) through the *sr*(*kbot*), and *si*(*kbot*-*nd*+1) through the *si*(*kbot*), respectively.

?laqr3

Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggresive early deflation).

Syntax

```
call slaqr3( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns,
nd, sr, si, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )

call dlaqr3( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns,
nd, sr, si, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )

call claqr3( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns,
nd, sh, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )

call zlaqr3( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns,
nd, sh, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
```

Description

This routine accepts as input an upper Hessenberg matrix *H* and performs an orthogonal/unitary similarity transformation designed to detect and deflate fully converged eigenvalues from a trailing principal submatrix. On output *H* has been overwritten by a new Hessenberg matrix that is a perturbation of an orthogonal/unitary similarity transformation of *H*. It is to be hoped that the final version of *H* has many zero subdiagonal entries.

Input Parameters

wantt LOGICAL.

If *wantt* = .TRUE., then the Hessenberg matrix *H* is fully updated so that the quasi-triangular/triangular Schur factor may be computed (in cooperation with the calling subroutine).
If *wantt* = .FALSE., then only enough of *H* is updated to preserve the eigenvalues.

wantz — LOGICAL.
If *wantz* = .TRUE., then the orthogonal/unitary matrix *Z* is updated so that the orthogonal/unitary Schur factor may be computed (in cooperation with the calling subroutine).
If *wantz* = .FALSE., then *Z* is not referenced.

n — INTEGER. The order of the Hessenberg matrix *H* and (if *wantz* = .TRUE.) the order of the orthogonal/unitary matrix *Z*.

ktop — INTEGER.
It is assumed that either *ktop*=1 or *h*(*ktop*,*ktop-1*)=0. *ktop* and *kbot* together determine an isolated block along the diagonal of the Hessenberg matrix.

kbot — INTEGER.
It is assumed without a check that either *kbot*=*n* or *h*(*kbot*+1,*kbot*)=0. *ktop* and *kbot* together determine an isolated block along the diagonal of the Hessenberg matrix.

nw — INTEGER.
Size of the deflation window. 1≤*nw*≤(*kbot*-*ktop*+1).

h — REAL for slaqr3
DOUBLE PRECISION for dlaqr3
COMPLEX for claqr3
COMPLEX*16 for zlaqr3.
Array, DIMENSION (*ldh*, *n*), on input the initial *n*-by-*n* section of *h* stores the Hessenberg matrix *H* undergoing aggressive early deflation.

ldh — INTEGER. The leading dimension of the array *h* just as declared in the calling subroutine. *ldh*≥*n*.

iloz, *ihiz* — INTEGER. Specify the rows of *Z* to which transformations must be applied if *wantz* is .TRUE.. 1≤*iloz*≤*ihiz*≤*n*.

z	REAL for slaqr3 DOUBLE PRECISION for dlaqr3 COMPLEX for claqr3 COMPLEX*16 for zlaqr3. Array, DIMENSION (*ldz*, *ihi*), contains the matrix *Z* if *wantz* is .TRUE.. If *wantz* is .FALSE., then *z* is not referenced.
ldz	INTEGER. The leading dimension of the array *z* just as declared in the calling subroutine. *ldz*≥1.
v	REAL for slaqr3 DOUBLE PRECISION for dlaqr3 COMPLEX for claqr3 COMPLEX*16 for zlaqr3. Workspace array with dimension (*ldv*, *nw*). An *nw*-by-*nw* work array.
ldv	INTEGER. The leading dimension of the array *v* just as declared in the calling subroutine. *ldv*≥*nw*.
nh	INTEGER. The number of column of *t*. *nh*≥*nw*.
t	REAL for slaqr3 DOUBLE PRECISION for dlaqr3 COMPLEX for claqr3 COMPLEX*16 for zlaqr3. Workspace array with dimension (*ldt*, *nw*).
ldt	INTEGER. The leading dimension of the array *t* just as declared in the calling subroutine. *ldt*≥*nw*.
nv	INTEGER. The number of rows of work array *wv* available for workspace. *nv*≥*nw*.
wv	REAL for slaqr3 DOUBLE PRECISION for dlaqr3 COMPLEX for claqr3 COMPLEX*16 for zlaqr3. Workspace array with dimension (*ldwv*, *nw*).
ldwv	INTEGER. The leading dimension of the array *wv* just as declared in the calling subroutine. *ldwv*≥*nw*.

work — REAL for slaqr3
DOUBLE PRECISION for dlaqr3
COMPLEX for claqr3
COMPLEX*16 for zlaqr3.
Workspace array with dimension *lwork*.

lwork — INTEGER. The dimension of the array *work*.
lwork=2*nw) is sufficient, but for the optimal performance a greater workspace may be required.
If *lwork*=-1, then the routine performs a workspace query: it estimates the optimal workspace size for the given values of the input parameters *n*, *nw*, *ktop*, and *kbot*. The estimate is returned in *work*(1). No error messages related to the *lwork* is issued by xerbla. Neither *H* nor *Z* are accessed.

Output Parameters

h — On output *h* has been transformed by an orthogonal/unitary similarity transformation, perturbed, and the returned to Hessenberg form that (it is to be hoped) has some zero subdiagonal entries.

work(1) — On exit *work*(1) is set to an estimate of the optimal value of *lwork* for the given values of the input parameters *n*, *nw*, *ktop*, and *kbot*.

z — If *wantz* is .TRUE., then the orthogonal/unitary similarity transformation is accumulated into *z(iloz:ihiz, ilo:ihi)* from the right.
If *wantz* is .FALSE., then *z* is unreferenced.

nd — INTEGER. The number of converged eigenvalues uncovered by the routine.

ns — INTEGER. The number of unconverged, that is approximate eigenvalues returned in *sr*, *si* or in *sh* that may be used as shifts by the calling subroutine.

sh — COMPLEX for claqr3
COMPLEX*16 for zlaqr3.
Arrays, DIMENSION (*kbot*).

	The approximate eigenvalues that may be used for shifts are stored in the *sh*(*kbot*-*nd*-*ns*+1) through the *sh*(*kbot*-*nd*). The converged eigenvalues are stored in the *sh*(*kbot*-*nd*+1) through the *sh*(*kbot*).
sr, *si*	REAL for slaqr3 DOUBLE PRECISION for dlaqr3 Arrays, DIMENSION (*kbot*) each. The real and imaginary parts of the approximate eigenvalues that may be used for shifts are stored in the *sr*(*kbot*-*nd*-*ns*+1) through the *sr*(*kbot*-*nd*), and *si*(*kbot*-*nd*-*ns*+1) through the *si*(*kbot*-*nd*), respectively. The real and imaginary parts of converged eigenvalues are stored in the *sr*(*kbot*-*nd*+1) through the *sr*(*kbot*), and *si*(*kbot*-*nd*+1) through the *si*(*kbot*), respectively.

?laqr4

Computes the eigenvalues of a Hessenberg matrix, and optionally the marices from the Schur decomposition.

Syntax

```
call slaqr4( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz, work, lwork, info )
call dlaqr4( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz, work, lwork, info )
call claqr4( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, work, lwork, info )
call zlaqr4( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, work, lwork, info )
```

Description

This routine computes the eigenvalues of a Hessenberg matrix H, and, optionally, the matrices T and Z from the Schur decomposition $H=Z*T*Z^H$, where T is an upper quasi-triangular/triangular matrix (the Schur form), and Z is the orthogonal/unitary matrix of Schur vectors.

Optionally *Z* may be postmultiplied into an input orthogonal/unitary matrix *Q* so that this routine can give the Schur factorization of a matrix *A* which has been reduced to the Hessenberg form *H* by the orthogonal/unitary matrix *Q*: *A* = *Q***H***Q*^{H} = (*QZ*)**H**(*QZ*)^H.

This routine implements one level of recursion for ?laqr0. It is a complete implementation of the small bulge multi-shift QR algorithm. It may be called by ?laqr0 and, for large enough deflation window size, it may be called by ?laqr3. This routine is identical to ?laqr0 except that it calls ?laqr2 instead of ?laqr3.

Input Parameters

wantt — LOGICAL.
If *wantt* = .TRUE., the full Schur form T is required;
If *wantt* = .FALSE., only eigenvalues are required.

wantz — LOGICAL.
If *wantz* = .TRUE., the matrix of Schur vectors *Z* is required;
If *wantz* = .FALSE., Schur vectors are not required.

n — INTEGER. The order of the Hessenberg matrix *H*. (*n* ≥ 0).

ilo, *ihi* — INTEGER.
It is assumed that *H* is already upper triangular in rows and columns 1:*ilo*-1 and *ihi*+1:*n*, and if *ilo* > 1 then *h*(*ilo*, *ilo*-1) = 0.
ilo and *ihi* are normally set by a previous call to cgebal, and then passed to cgehrd when the matrix output by cgebal is reduced to Hessenberg form. Otherwise, *ilo* and *ihi* should be set to 1 and *n*, respectively.
If *n* > 0, then 1 ≤ *ilo* ≤ *ihi* ≤ *n*.
If *n*=0, then *ilo*=1 and *ihi*=0

h — REAL for slaqr4
DOUBLE PRECISION for dlaqr4
COMPLEX for claqr4
COMPLEX*16 for zlaqr4.
Array, DIMENSION (*ldh*, *n*), contains the upper Hessenberg matrix *H*.

ldh — INTEGER. The leading dimension of the array *h*. *ldh* ≥ max(1, *n*).

iloz, *ihiz*	INTEGER. Specify the rows of *Z* to which transformations must be applied if *wantz* is .TRUE., $1 \le iloz \le ilo$; $ihi \le ihiz \le n$.
z	REAL for slaqr4 DOUBLE PRECISION for dlaqr4 COMPLEX for claqr4 COMPLEX*16 for zlaqr4. Array, DIMENSION (*ldz*, *ihi*), contains the matrix *Z* if *wantz* is .TRUE.. If *wantz* is .FALSE., *z* is not referenced.
ldz	INTEGER. The leading dimension of the array *z*. If *wantz* is .TRUE., then $ldz \ge \max(1, ihiz)$. Otherwise, $ldz \ge 1$.
work	REAL for slaqr4 DOUBLE PRECISION for dlaqr4 COMPLEX for claqr4 COMPLEX*16 for zlaqr4. Workspace array with dimension *lwork*.
lwork	INTEGER. The dimension of the array *work*. $lwork \ge \max(1,n)$ is sufficient, but for the optimal performance a greater workspace may be required, typically as large as $6*n$. It is recommended to use the worlspace query to determine the optimal workspace size. If $lwork=-1$, then the routine performs a workspace query: it estimates the optimal workspace size for the given values of the input parameters *n*, *ilo*, and *ihi*. The estimate is returned in *work*(1). No error messages related to the *lwork* is issued by xerbla. Neither *H* nor *Z* are accessed.

Output Parameters

h	If *info*=0 , and *wantt* is .TRUE., then *h* contains the upper quasi-triangular/triangular matrix *T* from the Schur decomposition (the Schur form). If *info*=0 , and *wantt* is .FALSE., then the contents of *h* are unspecified on exit.

(The output values of *h* when *info* > 0 are given under the description of the *info* parameter below.)
The routines may explicitly set *h*(i,j) for i>j and j=1,2,...*ilo*-1 or j=*ihi*+1, *ihi*+2,...*n*.

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

w — COMPLEX for claqr4
COMPLEX*16 for zlaqr4.
Arrays, DIMENSION(*n*). The computed eigenvalues of *h*(*ilo*:*ihi*, *ilo*:*ihi*) are stored in *w*(*ilo*:*ihi*). If *wantt* is .TRUE., then the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*, with *w*(i) = *h*(i,i).

wr, *wi* — REAL for slaqr4
DOUBLE PRECISION for dlaqr4
Arrays, DIMENSION(*ihi*) each. The real and imaginary parts, respectively, of the computed eigenvalues of *h*(*ilo*:*ihi*, *ilo*:*ihi*) are stored in the *wr*(*ilo*:*ihi*) and *wi*(*ilo*:*ihi*). If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of *wr* and *wi*, say the *i*-th and (*i*+1)-th, with *wi*(*i*)> 0 and *wi*(*i*+1) < 0. If *wantt* is .TRUE. , then the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*, with *wr*(i) = *h*(i,i), and if *h*(i:i+1,i:i+1)is a 2-by-2 diagonal block, then *wi*(i)=sqrt(-*h*(i+1,i)*h(i,i+1)).

z — If *wantz* is .TRUE., then *z(ilo:ihi, iloz:ihiz)* is replaced by *z(ilo:ihi, iloz:ihiz)*U*, where *U* is the orthogonal/unitary Schur factor of *h(ilo:ihi, ilo:ihi)*.
If *wantz* is .FALSE., *z* is not referenced.
(The output values of *z* when *info* > 0 are given under the description of the *info* parameter below.)

info — INTEGER.
= 0: the execution is successful.

> 0: if *info* = *i*, then the routine failed to compute all the eigenvalues. Elements 1:*ilo*–1 and i+1:*n* of *wr* and *wi* contain those eigenvalues which have been successfully computed.
> 0: if *wantt* is .FALSE., then the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns *ilo* through *info* of the final output value of *h*.
> 0: if *wantt* is .TRUE., then (initial value of *h*)***U* = *U**(final value of *h*, where *U* is an orthogonal/unitary matrix. The final value of *h* is upper Hessenberg and quasi-triangular/triangular in rows and columns *info*+1 through *ihi*.
> 0: if *wantz* is .TRUE., then (final value of *z*(*ilo*:*ihi*, *iloz*:*ihiz*))=(initial value of *z*(*ilo*:*ihi*, *iloz*:*ihiz*)***U*, where *U* is the orthogonal/unitary matrix in the previous expression (regardless of the value of *wantt*).
> 0: if *wantz* is .FALSE., then *z* is not accessed.

?laqr5

Performs a single small-bulge multi-shift QR sweep.

Syntax

```
call slaqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, sr, si, h, ldh,
iloz, ihiz, z, ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )

call dlaqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, sr, si, h, ldh,
iloz, ihiz, z, ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )

call claqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, s, h, ldh, iloz,
ihiz, z, ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )

call zlaqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, s, h, ldh, iloz,
ihiz, z, ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )
```

Description

This auxiliary routine called by ?laqr0 performs a single small-bulge multi-shift QR sweep.

Input Parameters

wantt — LOGICAL.
wantt = .TRUE. if the quasi-triangular/triangular Schur factor is computed.
wantt is set to .FALSE. otherwise.

wantz — LOGICAL.
wantz = .TRUE. if the orthogonal/unitary Schur factor is computed.
wantz is set to .FALSE. otherwise.

kacc22 — INTEGER. Possible values are 0, 1, or 2.
Specifies the computation mode of far-from-diagonal orthogonal updates.
= 0: the routine does not accumulate reflections and does not use matrix-matrix multiply to update far-from-diagonal matrix entries.
= 1: the routine accumulates reflections and uses matrix-matrix multiply to update the far-from-diagonal matrix entries.
= 2: the routine accumulates reflections, uses matrix-matrix multiply to update the far-from-diagonal matrix entries, and takes advantage of 2-by-2 block structure during matrix multiplies.

n — INTEGER. The order of the Hessenberg matrix *H* upon which the routine operates.

ktop, *kbot* — INTEGER.
It is assumed without a check that either *ktop*=1 or *h*(*ktop*,*ktop-1*)=0, and either *kbot*=*n* or *h*(*kbot*+1,*kbot*)=0.

nshfts — INTEGER.
Number of simultaneous shifts, must be positive and even.

sr, *si* — REAL for slaqr5
DOUBLE PRECISION for dlaqr5
Arrays, DIMENSION (*nshfts*) each.
sr contains the real parts and *si* contains the imaginary parts of the *nshfts* shifts of origin that define the multi-shift QR sweep.

s	COMPLEX for claqr5 COMPLEX*16 for zlaqr5. Arrays, DIMENSION (*nshfts*). *s* contains the shifts of origin that define the multi-shift QR sweep.
h	REAL for slaqr5 DOUBLE PRECISION for dlaqr5 COMPLEX for claqr5 COMPLEX*16 for zlaqr5. Array, DIMENSION (*ldh*, *n*), on input contains the Hessenberg matrix.
ldh	INTEGER. The leading dimension of the array *h* just as declared in the calling routine. *ldh* ≥ max(1, *n*).
iloz, *ihiz*	INTEGER. Specify the rows of *Z* to which transformations must be applied if *wantz* is .TRUE.. 1 ≤ *iloz* ≤ *ihiz* ≤ *n*.
z	REAL for slaqr5 DOUBLE PRECISION for dlaqr5 COMPLEX for claqr5 COMPLEX*16 for zlaqr5. Array, DIMENSION (*ldz*, *ihi*), contains the matrix *Z* if *wantz* is .TRUE.. If *wantz* is .FALSE., then *z* is not referenced.
ldz	INTEGER. The leading dimension of the array *z* just as declared in the calling routine. *ldz* ≥ *n*.
v	REAL for slaqr5 DOUBLE PRECISION for dlaqr5 COMPLEX for claqr5 COMPLEX*16 for zlaqr5. Workspace array with dimension (*ldv*, *nshfts*/2).
ldv	INTEGER. The leading dimension of the array *v* just as declared in the calling routine. *ldv* ≥ 3.
u	REAL for slaqr5 DOUBLE PRECISION for dlaqr5 COMPLEX for claqr5

COMPLEX*16 for zlaqr5.
Workspace array with dimension (*ldu*, 3**nshfts*-3).

ldu INTEGER. The leading dimension of the array *u* just as declared in the calling routine. *ldu* ≥ 3**nshfts*-3.

nh INTEGER. The number of column in the array *wh* available for workspace. *nh* ≥ 1.

wh REAL for slaqr5
DOUBLE PRECISION for dlaqr5
COMPLEX for claqr5
COMPLEX*16 for zlaqr5.
Workspace array with dimension (*ldwh*, *nh*)

ldwh INTEGER. The leading dimension of the array *wh* just as declared in the calling routine. *ldwh* ≥ 3**nshfts*-3

nv INTEGER. The number of rows of the array *wv* available for workspace. *nv* ≥ 1.

wv REAL for slaqr5
DOUBLE PRECISION for dlaqr5
COMPLEX for claqr5
COMPLEX*16 for zlaqr5.
Workspace array with dimension (*ldwv*, 3**nshfts*-3).

ldwv INTEGER. The leading dimension of the array *wv* just as declared in the calling routine. *ldwv* ≥ *nv*.

Output Parameters

h On output a multi-shift QR Sweep with shifts *sr*(j)+i**si*(j) or *s*(j) is applied to the isolated diagonal block in rows and columns *ktop* through *kbot* .

z If *wantz* is .TRUE., then the QR Sweep orthogonal/unitary similarity transformation is accumulated into *z(iloz:ihiz, ilo:ihi)* from the right.
If *wantz* is .FALSE., then *z* is unreferenced.

?laqsb

Scales a symmetric band matrix, using scaling factors computed by ?pbequ.

Syntax

```
call slaqsb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
call dlaqsb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
call claqsb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
call zlaqsb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
```

Description

The routine equilibrates a symmetric band matrix *A* using the scaling factors in the vector *s*.

Input Parameters

uplo
CHARACTER*1.
Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored.
If *uplo* = 'U': upper triangular.
If *uplo* = 'L': lower triangular.

n
INTEGER. The order of the matrix *A*.
n ≥ 0.

kd
INTEGER. The number of super-diagonals of the matrix *A* if *uplo* = 'U', or the number of sub-diagonals if *uplo* = 'L'.
kd ≥ 0.

ab
REAL for slaqsb
DOUBLE PRECISION for dlaqsb
COMPLEX for claqsb
COMPLEX*16 for zlaqsb
Array, DIMENSION (*ldab*,*n*). On entry, the upper or lower triangle of the symmetric band matrix *A*, stored in the first *kd*+1 rows of the array. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:

if *uplo* = 'U', *ab*(*kd*+1+*i*-*j*,*j*) = A(*i*,*j*) for max(1,*j*-*kd*) ≤ *i* ≤ *j*;

if *uplo* = 'L', *ab*(1+*i*-*j*,*j*) = A(*i*,*j*) for *j* ≤ *i* ≤ min(*n*,*j*+*kd*).

ldab — INTEGER. The leading dimension of the array *ab*.
ldab ≥ *kd*+1.

s — REAL for slaqsb/claqsb
DOUBLE PRECISION for dlaqsb/zlaqsb
Array, DIMENSION (*n*). The scale factors for A.

scond — REAL for slaqsb/claqsb
DOUBLE PRECISION for dlaqsb/zlaqsb
Ratio of the smallest *s*(*i*) to the largest *s*(*i*).

amax — REAL for slaqsb/claqsb
DOUBLE PRECISION for dlaqsb/zlaqsb
Absolute value of largest matrix entry.

Output Parameters

ab — On exit, if *info* = 0, the triangular factor *U* or *L* from the Cholesky factorization *A* = *U*'**U* or *A* = *L***L*' of the band matrix *A*, in the same storage format as *A*.

equed — CHARACTER*1.
Specifies whether or not equilibration was done.
If *equed* = 'N': No equilibration.
If *equed* = 'Y': Equilibration was done, that is, *A* has been replaced by diag(*s*)**A**diag(*s*).

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if scaling should be based on the ratio of the scaling factors. If *scond* < *thresh*, scaling is done. *large* and *small* are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, scaling is done.

?laqsp

Scales a symmetric/Hermitian matrix in packed storage, using scaling factors computed by ?ppequ.

Syntax

```
call slaqsp( uplo, n, ap, s, scond, amax, equed )
call dlaqsp( uplo, n, ap, s, scond, amax, equed )
call claqsp( uplo, n, ap, s, scond, amax, equed )
call zlaqsp( uplo, n, ap, s, scond, amax, equed )
```

Description

The routine ?laqsp equilibrates a symmetric matrix *A* using the scaling factors in the vector *s*.

Input Parameters

uplo — CHARACTER*1.
Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored.
If *uplo* = 'U': upper triangular.
If *uplo* = 'L': lower triangular.

n — INTEGER. The order of the matrix *A*. $n \geq 0$.

ap — REAL for slaqsp
DOUBLE PRECISION for dlaqsp
COMPLEX for claqsp
COMPLEX*16 for zlaqsp
Array, DIMENSION ($n(n+1)/2$).
On entry, the upper or lower triangle of the symmetric matrix *A*, packed columnwise in a linear array. The j-th column of *A* is stored in the array *ap* as follows:
if *uplo* = 'U', $ap(i + (j-1)j/2) = A(i,j)$ for $1 \leq i \leq j$;
if *uplo* = 'L', $ap(i + (j-1)(2n-j)/2) = A(i,j)$ for $j \leq i \leq n$.

s REAL for slaqsp/claqsp
DOUBLE PRECISION for dlaqsp/zlaqsp
Array, DIMENSION (*n*). The scale factors for A.

scond REAL for slaqsp/claqsp
DOUBLE PRECISION for dlaqsp/zlaqsp
Ratio of the smallest *s*(*i*) to the largest *s*(*i*).

amax REAL for slaqsp/claqsp
DOUBLE PRECISION for dlaqsp/zlaqsp
Absolute value of largest matrix entry.

Output Parameters

ap On exit, the equilibrated matrix: diag(*s*)*A*diag(*s*), in the same storage format as *A*.

equed CHARACTER*1.
Specifies whether or not equilibration was done.
If *equed* = 'N': No equilibration.
If *equed* = 'Y': Equilibration was done, that is, *A* has been replaced by diag(*s*)*A*diag(*s*).

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if scaling should be based on the ratio of the scaling factors. If *scond* < *thresh*, scaling is done. *large* and *small* are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, scaling is done.

?laqsy

Scales a symmetric/Hermitian matrix, using scaling factors computed by ?poequ.

Syntax

```
call slaqsy( uplo, n, a, lda, s, scond, amax, equed )
call dlaqsy( uplo, n, a, lda, s, scond, amax, equed )
call claqsy( uplo, n, a, lda, s, scond, amax, equed )
call zlaqsy( uplo, n, a, lda, s, scond, amax, equed )
```

Description

The routine equilibrates a symmetric matrix *A* using the scaling factors in the vector *s*.

Input Parameters

uplo — CHARACTER*1.
Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored.
If *uplo* = 'U': upper triangular.
If *uplo* = 'L': lower triangular.

n — INTEGER. The order of the matrix *A*.
n ≥ 0.

a — REAL for slaqsy
DOUBLE PRECISION for dlaqsy
COMPLEX for claqsy
COMPLEX*16 for zlaqsy
Array, DIMENSION (*lda*,*n*). On entry, the symmetric matrix *A*.
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda — INTEGER. The leading dimension of the array *a*.

	lda ≥ max(*n*,1).
s	REAL for slaqsy/claqsy DOUBLE PRECISION for dlaqsy/zlaqsy Array, DIMENSION (*n*). The scale factors for A.
scond	REAL for slaqsy/claqsy DOUBLE PRECISION for dlaqsy/zlaqsy Ratio of the smallest *s*(*i*) to the largest *s*(*i*).
amax	REAL for slaqsy/claqsy DOUBLE PRECISION for dlaqsy/zlaqsy Absolute value of largest matrix entry.

Output Parameters

a	On exit, if *equed* = 'Y', the equilibrated matrix: diag(*s*)*A*diag(*s*).
equed	CHARACTER*1. Specifies whether or not equilibration was done. If *equed* = 'N': No equilibration. If *equed* = 'Y': Equilibration was done, i.e., *A* has been replaced by diag(*s*)*A*diag(*s*).

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if scaling should be based on the ratio of the scaling factors. If *scond* < *thresh*, scaling is done. *large* and *small* are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, scaling is done.

?laqtr

Solves a real quasi-triangular system of equations, or a complex quasi-triangular system of special form, in real arithmetic.

Syntax

```
call slaqtr( ltran, lreal, n, t, ldt, b, w, scale, x, work, info )
call dlaqtr( ltran, lreal, n, t, ldt, b, w, scale, x, work, info )
```

Description

The routine ?laqtr solves the real quasi-triangular system

op(*T*) * *p* = *scale***c*, if *lreal* = .TRUE.

or the complex quasi-triangular systems

op(*T* + *iB*)*(*p*+*iq*) = *scale**(*c*+*id*), if *lreal* = .FALSE.

in real arithmetic, where *T* is upper quasi-triangular.

If *lreal* = .FALSE., then the first diagonal block of *T* must be 1-by-1, *B* is the specially structured matrix

$$B = \begin{bmatrix} b_1 & b_2 & \dots & \dots & b_n \\ & w & & & \\ & & w & & \\ & & & \dots & \\ & & & & w \end{bmatrix}$$

op*(A)* = *A* or *A*', *A*' denotes the conjugate transpose of matrix *A*.

On input,

$$x = \begin{bmatrix} c \\ d \end{bmatrix}, \text{ on output } x = \begin{bmatrix} p \\ q \end{bmatrix}$$

This routine is designed for the condition number estimation in routine ?trsna.

Input Parameters

ltran — LOGICAL.
On entry, *ltran* specifies the option of conjugate transpose:
= .FALSE., op(*T* + *iB*) = *T* + *iB*,

= .TRUE., op(*T* + *iB*) = (*T* + *iB*)'.

lreal — LOGICAL.
On entry, *lreal* specifies the input matrix structure:
= .FALSE., the input is complex
= .TRUE., the input is real.

n — INTEGER.
On entry, *n* specifies the order of *T* + *iB*. *n* ≥ 0.

t — REAL for slaqtr
DOUBLE PRECISION for dlaqtr
Array, dimension (*ldt*,*n*). On entry, *t* contains a matrix in Schur canonical form. If *lreal* = .FALSE., then the first diagonal block of *t* must be 1-by-1.

ldt — INTEGER. The leading dimension of the matrix *T*.
ldt ≥ max(1,*n*).

b — REAL for slaqtr
DOUBLE PRECISION for dlaqtr
Array, dimension (*n*). On entry, *b* contains the elements to form the matrix *B* as described above. If *lreal* = .TRUE., *b* is not referenced.

w — REAL for slaqtr
DOUBLE PRECISION for dlaqtr
On entry, *w* is the diagonal element of the matrix *B*.
If *lreal* = .TRUE., *w* is not referenced.

x — REAL for slaqtr
DOUBLE PRECISION for dlaqtr
Array, dimension (2*n*). On entry, *x* contains the right hand side of the system.

work — REAL for slaqtr
DOUBLE PRECISION for dlaqtr
Workspace array, dimension (*n*).

Output Parameters

scale — REAL for slaqtr
DOUBLE PRECISION for dlaqtr
On exit, *scale* is the scale factor.

x	On exit, *X* is overwritten by the solution.
info	INTEGER. If *info* = 0: successful exit. If *info* = 1: the some diagonal 1-by-1 block has been perturbed by a small number smin to keep nonsingularity. If *info* = 2: the some diagonal 2-by-2 block has been perturbed by a small number in ?laln2 to keep nonsingularity.

NOTE. For higher speed, this routine does not check the inputs for errors.

?lar1v

Computes the (scaled) r-th column of the inverse of the submatrix in rows b1 through bn of tridiagonal matrix.

Syntax

```
call slar1v( n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc,
negcnt, ztz, mingma, r, isuppz, nrminv, resid, rqcorr, work )

call dlar1v( n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc,
negcnt, ztz, mingma, r, isuppz, nrminv, resid, rqcorr, work )

call clar1v( n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc,
negcnt, ztz, mingma, r, isuppz, nrminv, resid, rqcorr, work )

call zlar1v( n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc,
negcnt, ztz, mingma, r, isuppz, nrminv, resid, rqcorr, work )
```

Description

The routine ?lar1v computes the (scaled) *r*-th column of the inverse of the submatrix in rows *b1* through *bn* of the tridiagonal matrix $L*D*L^T - \lambda*I$. When λ is close to an eigenvalue, the computed vector is an accurate eigenvector. Usually, *r* corresponds to the index where the eigenvector is largest in magnitude.

The following steps accomplish this computation :

- Stationary *qd* transform, $L*D*L^T - \lambda*I = L(+)*D(+)*L(+)^T$
- Progressive *qd* transform, $L*D*L^T - \lambda*I = U(-)*D(-)*U(-)^T$,
- Computation of the diagonal elements of the inverse of $L*D*L^T - \lambda*I$ by combining the above transforms, and choosing r as the index where the diagonal of the inverse is (one of the) largest in magnitude.
- Computation of the (scaled) r-th column of the inverse using the twisted factorization obtained by combining the top part of the stationary and the bottom part of the progressive transform.

Input Parameters

n	INTEGER. The order of the matrix $L*D*L^T$.
b1	INTEGER. First index of the submatrix of $L*D*L^T$.
bn	INTEGER. Last index of the submatrix of $L*D*L^T$.
lambda	REAL for slar1v/clar1v DOUBLE PRECISION for dlar1v/zlar1v The shift. To compute an accurate eigenvector, *lambda* should be a good approximation to an eigenvalue of $L*D*L^T$.
l	REAL for slar1v/clar1v DOUBLE PRECISION for dlar1v/zlar1v Array, DIMENSION (n-1). The (n-1) subdiagonal elements of the unit bidiagonal matrix L, in elements 1 to n-1.
d	REAL for slar1v/clar1v DOUBLE PRECISION for dlar1v/zlar1v Array, DIMENSION (n). The n diagonal elements of the diagonal matrix D.
ld	REAL for slar1v/clar1v DOUBLE PRECISION for dlar1v/zlar1v Array, DIMENSION (n-1). The n-1 elements L_i*D_i.
lld	REAL for slar1v/clar1v DOUBLE PRECISION for dlar1v/zlar1v Array, DIMENSION (n-1). The n-1 elements $L_i*L_i*D_i$.

pivmin
REAL for slar1v/clar1v
DOUBLE PRECISION for dlar1v/zlar1v
The minimum pivot in the Sturm sequence.

gaptol
REAL for slar1v/clar1v
DOUBLE PRECISION for dlar1v/zlar1v
Tolerance that indicates when eigenvector entries are negligible with respect to their contribution to the residual.

z
REAL for slar1v
DOUBLE PRECISION for dlar1v
COMPLEX for clar1v
COMPLEX*16 for zlar1v
Array, DIMENSION (*n*). All entries of *z* must be set to 0.

wantnc
LOGICAL.
Specifies whether *negcnt* has to be computed.

r
INTEGER.
The twist index for the twisted factorization used to compute *z*. On input, $0 \le r \le n$. If *r* is input as 0, *r* is set to the index where $(L*D*L^T - lambda*I)^{-1}$ is largest in magnitude. If $1 \le r \le n$, *r* is unchanged.

work
REAL for slar1v/clar1v
DOUBLE PRECISION for dlar1v/zlar1v
Workspace array, DIMENSION (4**n*).

Output Parameters

z
REAL for slar1v
DOUBLE PRECISION for dlar1v
COMPLEX for clar1v
COMPLEX*16 for zlar1v
Array, DIMENSION (*n*). The (scaled) *r*-th column of the inverse. *z*(*r*) is returned to be 1.

negcnt
INTEGER. If *wantnc* is .TRUE. then *negcnt* = the number of pivots < *pivmin* in the matrix factorization $L*D*L^T$, and *negcnt* = -1 otherwise.

ztz
REAL for slar1v/clar1v
DOUBLE PRECISION for dlar1v/zlar1v
The square of the 2-norm of *z*.

mingma — REAL for slar1v/clar1v
DOUBLE PRECISION for dlar1v/zlar1v
The reciprocal of the largest (in magnitude) diagonal element of the inverse of $L*D*L^T$ – *lambda*I*.

r — On output, *r* is the twist index used to compute *z*. Ideally, *r* designates the position of the maximum entry in the eigenvector.

isuppz — INTEGER. Array, DIMENSION (2). The support of the vector in *Z*, that is, the vector *z* is nonzero only in elements *isuppz*(1) through *isuppz*(2).

nrminv — REAL for slar1v/clar1v
DOUBLE PRECISION for dlar1v/zlar1v
Equals 1/sqrt(*ztz*).

resid — REAL for slar1v/clar1v
DOUBLE PRECISION for dlar1v/zlar1v
The residual of the FP vector.
resid = ABS(*mingma*)/sqrt(*ztz*).

rqcorr — REAL for slar1v/clar1v
DOUBLE PRECISION for dlar1v/zlar1v
The Rayleigh Quotient correction to *lambda*.
rqcorr = *mingma*/*ztz*.

?lar2v

Applies a vector of plane rotations with real cosines and real/complex sines from both sides to a sequence of 2-by-2 symmetric/Hermitian matrices.

Syntax

```
call slar2v( n, x, y, z, incx, c, s, incc )
call dlar2v( n, x, y, z, incx, c, s, incc )
call clar2v( n, x, y, z, incx, c, s, incc )
call zlar2v( n, x, y, z, incx, c, s, incc )
```

Description

The routine ?lar2v applies a vector of real/complex plane rotations with real cosines from both sides to a sequence of 2-by-2 real symmetric or complex Hermitian matrices, defined by the elements of the vectors x, y and z. For i = 1,2,...,*n*

$$\begin{bmatrix} x_i & z_i \\ \text{conjg}(z_i) & y_i \end{bmatrix} := \begin{bmatrix} c(i) & \text{conjg}(s(i)) \\ -s(i) & c(i) \end{bmatrix} \begin{bmatrix} x_i & z_i \\ \text{conjg}(z_i) & y_i \end{bmatrix} \begin{bmatrix} c(i) & -\text{conjg}(s(i)) \\ s(i) & c(i) \end{bmatrix}$$

Input Parameters

n	INTEGER. The number of plane rotations to be applied.
x, *y*, *z*	REAL for slar2v DOUBLE PRECISION for dlar2v COMPLEX for clar2v COMPLEX*16 for zlar2v Arrays, DIMENSION (1+(*n*-1)**incx*) each. Contain the vectors x, y and z, respectively. For all flavors of ?lar2v, elements of *x* and *y* are assumed to be real.
incx	INTEGER. The increment between elements of *x*, *y*, and *z*. *incx* > 0.
c	REAL for slar2v/clar2v DOUBLE PRECISION for dlar2v/zlar2v Array, DIMENSION (1+(*n*-1)**incc*). The cosines of the plane rotations.
s	REAL for slar2v DOUBLE PRECISION for dlar2v COMPLEX for clar2v COMPLEX*16 for zlar2v Array, DIMENSION (1+(*n*-1)**incc*). The sines of the plane rotations.
incc	INTEGER. The increment between elements of *c* and *s*. *incc* > 0.

Output Parameters

x, *y*, *z*	Vectors *x*, *y* and *z*, containing the results of transform.

?larf

Applies an elementary reflector to a general rectangular matrix.

Syntax

```
call slarf( side, m, n, v, incv, tau, c, ldc, work )
call dlarf( side, m, n, v, incv, tau, c, ldc, work )
call clarf( side, m, n, v, incv, tau, c, ldc, work )
call zlarf( side, m, n, v, incv, tau, c, ldc, work )
```

Description

The routine applies a real/complex elementary reflector *H* to a real/complex *m*-by-*n* matrix *C*, from either the left or the right. *H* is represented in the form

```
H = I - tau*v*v',
```

where *tau* is a real/complex scalar and v is a real/complex vector.

If *tau* = 0, then *H* is taken to be the unit matrix. For clarf/zlarf, to apply *H'* (the conjugate transpose of *H*), supply conjg(*tau*) instead of *tau*.

Input Parameters

side	CHARACTER*1. If *side* = 'L': form *H***C* If *side* = 'R': form *C***H*.
m	INTEGER. The number of rows of the matrix *C*.
n	INTEGER. The number of columns of the matrix *C*.
v	REAL for slarf DOUBLE PRECISION for dlarf COMPLEX for clarf COMPLEX*16 for zlarf Array, DIMENSION

(1 + (*m*-1)*abs(*incv*)) if *side* = 'L' or
(1 + (*n*-1)*abs(*incv*)) if *side* = 'R'. The vector *v* in the representation of *H*. *v* is not used if *tau* = 0.

incv — INTEGER. The increment between elements of *v*.

incv ≠ 0.

tau — REAL for slarf
DOUBLE PRECISION for dlarf
COMPLEX for clarf
COMPLEX*16 for zlarf
The value *tau* in the representation of *H*.

c — REAL for slarf
DOUBLE PRECISION for dlarf
COMPLEX for clarf
COMPLEX*16 for zlarf
Array, DIMENSION (*ldc*,*n*).
On entry, the *m*-by-*n* matrix *C*.

ldc — INTEGER. The leading dimension of the array *c*.

ldc ≥ max(1,*m*).

work — REAL for slarf
DOUBLE PRECISION for dlarf
COMPLEX for clarf
COMPLEX*16 for zlarf
Workspace array, DIMENSION
(*n*) if *side* = 'L' or
(*m*) if *side* = 'R'.

Output Parameters

c — On exit, *C* is overwritten by the matrix *H***C* if *side* = 'L', or *C***H* if *side* = 'R'.

?larfb

Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.

Syntax

```
call slarfb( side, trans, direct, storev, m, n, k, v, ldv, t, ldt, c, ldc,
work, ldwork )
call dlarfb( side, trans, direct, storev, m, n, k, v, ldv, t, ldt, c, ldc,
work, ldwork )
call clarfb( side, trans, direct, storev, m, n, k, v, ldv, t, ldt, c, ldc,
work, ldwork )
call zlarfb( side, trans, direct, storev, m, n, k, v, ldv, t, ldt, c, ldc,
work, ldwork )
```

Description

The routine ?larfb applies a complex block reflector *H* or its transpose *H'* to a complex *m*-by-*n* matrix *C* from either left or right.

Input Parameters

side — CHARACTER*1.
If *side* = 'L': apply *H* or *H'* from the left
If *side* = 'R': apply *H* or *H'* from the right

trans — CHARACTER*1.
If *trans* = 'N': apply *H* (No transpose)
If *trans* = 'C': apply *H'* (Conjugate transpose)

direct — CHARACTER*1.
Indicates how *H* is formed from a product of elementary reflectors
If *direct* = 'F': *H* = *H*(1) *H*(2) . . . *H*(*k*) (forward)
If *direct* = 'B': *H* = *H*(*k*) . . . *H*(2) *H*(1) (backward)

storev — CHARACTER*1.
Indicates how the vectors which define the elementary reflectors are stored:
If *storev* = 'C': Column-wise

	If *storev* = 'R': Row-wise
m	INTEGER. The number of rows of the matrix *C*.
n	INTEGER. The number of columns of the matrix *C*.
k	INTEGER. The order of the matrix *T* (equal to the number of elementary reflectors whose product defines the block reflector).
v	REAL for slarfb DOUBLE PRECISION for dlarfb COMPLEX for clarfb COMPLEX*16 for zlarfb Array, DIMENSION (*ldv*, *k*) if *storev* = 'C' (*ldv*, *m*) if *storev* = 'R' and *side* = 'L' (*ldv*, *n*) if *storev* = 'R' and *side* = 'R' The matrix *v*.
ldv	INTEGER. The leading dimension of the array v. If *storev* = 'C' and *side* = 'L', *ldv* ≥ max(1,*m*); if *storev* = 'C' and *side* = 'R', *ldv* ≥ max(1,*n*); if *storev* = 'R', *ldv* ≥ *k*.
t	REAL for slarfb DOUBLE PRECISION for dlarfb COMPLEX for clarfb COMPLEX*16 for zlarfb Array, DIMENSION (*ldt*,*k*). Contains the triangular *k*-by-*k* matrix *T* in the representation of the block reflector.
LDT	INTEGER. The leading dimension of the array *t*. *ldt* ≥ *k*.
c	REAL for slarfb DOUBLE PRECISION for dlarfb COMPLEX for clarfb COMPLEX*16 for zlarfb Array, DIMENSION (*ldc*,*n*). On entry, the *m*-by-*n* matrix *C*.
ldc	INTEGER. The leading dimension of the array *c*.

	ldc ≥ max(1,*m*).
work	REAL for slarfb DOUBLE PRECISION for dlarfb COMPLEX for clarfb COMPLEX*16 for zlarfb Workspace array, DIMENSION (*ldwork*, *k*).
ldwork	INTEGER. The leading dimension of the array *work*. If *side* = 'L', *ldwork* ≥ max(1, *n*); if *side* = 'R', *ldwork* ≥ max(1, *m*).

Output Parameters

c	On exit, *c* is overwritten by *H***C*, or *H*'**C*, or *C***H*, or *C***H*'.

?larfg

Generates an elementary reflector (Householder matrix).

Syntax

```
call slarfg( n, alpha, x, incx, tau )
call dlarfg( n, alpha, x, incx, tau )
call clarfg( n, alpha, x, incx, tau )
call zlarfg( n, alpha, x, incx, tau )
```

Description

The routine ?larfg generates a real/complex elementary reflector *H* of order *n*, such that

$$H^{*}\begin{bmatrix} alpha \\ x \end{bmatrix}\begin{bmatrix} beta \\ 0 \end{bmatrix}, H^{*}H = I,$$

where *alpha* and *beta* are scalars (with *beta* real for all flavors), and *x* is an (*n*-1)-element real/complex vector. *H* is represented in the form

$$H = I - tau * \begin{bmatrix} 1 \\ v \end{bmatrix} * \begin{bmatrix} 1 & v' \end{bmatrix}$$

where *tau* is a real/complex scalar and v is a real/complex (*n*-1)-element vector. Note that for clarfg/zlarfg, *H* is not Hermitian.

If the elements of *x* are all zero (and, for complex flavors, *alpha* is real), then *tau* = 0 and *H* is taken to be the unit matrix.

Otherwise, 1 ≤ *tau* ≤ 2 (for real flavors), or

1 ≤ Re(*tau*) ≤ 2 and abs(*tau*-1) ≤ 1 (for complex flavors).

Input Parameters

n	INTEGER. The order of the elementary reflector.
alpha	REAL for slarfg DOUBLE PRECISION for dlarfg COMPLEX for clarfg COMPLEX*16 for zlarfg On entry, the value alpha.
x	REAL for slarfg DOUBLE PRECISION for dlarfg COMPLEX for clarfg COMPLEX*16 for zlarfg Array, DIMENSION (1+(*n*-2)*abs(*incx*)). On entry, the vector *x*.
incx	INTEGER. The increment between elements of *x*. *incx* > 0.

Output Parameters

alpha	On exit, it is overwritten with the value *beta*.
x	On exit, it is overwritten with the vector *v*.
tau	REAL for slarfg DOUBLE PRECISION for dlarfg COMPLEX for clarfg

COMPLEX*16 for zlarfg The value *tau*.

?larft

*Forms the triangular factor T of a block reflector H = I - V*T*V^H.*

Syntax

```
call slarft( direct, storev, n, k, v, ldv, tau, t, ldt )
call dlarft( direct, storev, n, k, v, ldv, tau, t, ldt )
call clarft( direct, storev, n, k, v, ldv, tau, t, ldt )
call zlarft( direct, storev, n, k, v, ldv, tau, t, ldt )
```

Description

The routine ?larft forms the triangular factor *T* of a real/complex block reflector *H* of order *n*, which is defined as a product of *k* elementary reflectors.

If *direct* = 'F', *H* = *H*(1) *H*(2) . . . *H*(*k*) and *T* is upper triangular;

If *direct* = 'B', *H* = *H*(*k*) . . . *H*(2) *H*(1) and *T* is lower triangular.

If *storev* = 'C', the vector which defines the elementary reflector H(*i*) is stored in the *i*-th column of the array *v*, and *H* = *I* - *V***T***V*' .

If *storev* = 'R', the vector which defines the elementary reflector H(*i*) is stored in the *i*-th row of the array *v*, and *H* = *I* - *V*'**T***V*.

Input Parameters

direct — CHARACTER*1.
Specifies the order in which the elementary reflectors are multiplied to form the block reflector:
= 'F': *H* = *H*(1) *H*(2) . . . *H*(*k*) (forward)
= 'B': *H* = *H*(*k*) . . . *H*(2) *H*(1) (backward)

storev — CHARACTER*1.
Specifies how the vectors which define the elementary reflectors are stored (see also Application Notes below):
= 'C': column-wise
= 'R': row-wise.

n	INTEGER. The order of the block reflector *H*. *n* ≥ 0.
k	INTEGER. The order of the triangular factor *T* (equal to the number of elementary reflectors). *k* ≥ 1.
v	REAL for slarft DOUBLE PRECISION for dlarft COMPLEX for clarft COMPLEX*16 for zlarft Array, DIMENSION (*ldv*, *k*) if *storev* = 'C' or (*ldv*, *n*) if *storev* = 'R'. The matrix *V*.
ldv	INTEGER. The leading dimension of the array *v*. If *storev* = 'C', *ldv* ≥ max(1,*n*); if *storev* = 'R', *ldv* ≥ *k*.
tau	REAL for slarft DOUBLE PRECISION for dlarft COMPLEX for clarft COMPLEX*16 for zlarft Array, DIMENSION (*k*). *tau*(*i*) must contain the scalar factor of the elementary reflector *H*(*i*).
ldt	INTEGER. The leading dimension of the output array *t*. *ldt* ≥ *k*.

Output Parameters

t	REAL for slarft DOUBLE PRECISION for dlarft COMPLEX for clarft COMPLEX*16 for zlarft Array, DIMENSION (*ldt*,*k*). The *k*-by-*k* triangular factor *T* of the block reflector. If *direct* = 'F', *T* is upper triangular; if *direct* = 'B', *T* is lower triangular. The rest of the array is not used.
v	The matrix *V*.

Application Notes

The shape of the matrix V and the storage of the vectors which define the $H(i)$ is best illustrated by the following example with $n = 5$ and $k = 3$. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

$direct = \text{'F'}$ and $storev = \text{'C'}$:

$$\begin{bmatrix} 1 & & \\ v_1 & 1 & \\ v_1 & v_2 & 1 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \end{bmatrix}$$

$direct = \text{'F'}$ and $storev = \text{'R'}$:

$$\begin{bmatrix} 1 & v_1 & v_1 & v_1 & v_1 \\ & 1 & v_2 & v_2 & v_2 \\ & & 1 & v_3 & v_3 \end{bmatrix}$$

$direct = \text{'B'}$ and $storev = \text{'C'}$:

$$\begin{bmatrix} v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ 1 & v_2 & v_3 \\ & 1 & v_3 \\ & & 1 \end{bmatrix}$$

$direct = \text{'B'}$ and $storev = \text{'R'}$:

$$\begin{bmatrix} v_1 & v_1 & 1 & & \\ v_2 & v_2 & v_2 & 1 & \\ v_3 & v_3 & v_3 & v_3 & 1 \end{bmatrix}$$

?larfx

Applies an elementary reflector to a general rectangular matrix, with loop unrolling when the reflector has order ≥ 10.

Syntax

```
call slarfx( side, m, n, v, tau, c, ldc, work )
call dlarfx( side, m, n, v, tau, c, ldc, work )
call clarfx( side, m, n, v, tau, c, ldc, work )
call zlarfx( side, m, n, v, tau, c, ldc, work )
```

Description

The routine ?larfx applies a real/complex elementary reflector *H* to a real/complex *m*-by-*n* matrix *C*, from either the left or the right.

H is represented in the form

*H = I - tau*v*v'*, where *tau* is a real/complex scalar and v is a real/complex vector.

If *tau = 0*, then *H* is taken to be the unit matrix

Input Parameters

side	CHARACTER*1. If *side = 'L'*: form *H*C* If *side = 'R'*: form *C*H*.
m	INTEGER. The number of rows of the matrix *C*.
n	INTEGER. The number of columns of the matrix *C*.
v	REAL for slarfx DOUBLE PRECISION for dlarfx COMPLEX for clarfx COMPLEX*16 for zlarfx Array, DIMENSION (*m*) if *side = 'L'* or (*n*) if *side = 'R'*. The vector *v* in the representation of *H*.
tau	REAL for slarfx

DOUBLE PRECISION for dlarfx
COMPLEX for clarfx
COMPLEX*16 for zlarfx
The value *tau* in the representation of *H*.

c — REAL for slarfx
DOUBLE PRECISION for dlarfx
COMPLEX for clarfx
COMPLEX*16 for zlarfx
Array, DIMENSION (*ldc*,*n*). On entry, the *m*-by-*n* matrix C.

ldc — INTEGER. The leading dimension of the array *c*. *lda* ≥ (1,*m*).

work — REAL for slarfx
DOUBLE PRECISION for dlarfx
COMPLEX for clarfx
COMPLEX*16 for zlarfx
Workspace array, DIMENSION
(*n*) if *side* = 'L' or
(*m*) if *side* = 'R'.
work is not referenced if *H* has order < 11.

Output Parameters

c — On exit, C is overwritten by the matrix *H***C* if *side* = 'L', or *C***H* if *side* = 'R'.

?largv

Generates a vector of plane rotations with real cosines and real/complex sines.

Syntax

```
call slargv( n, x, incx, y, incy, c, incc )
call dlargv( n, x, incx, y, incy, c, incc )
call clargv( n, x, incx, y, incy, c, incc )
call zlargv( n, x, incx, y, incy, c, incc )
```

Description

The routine generates a vector of real/complex plane rotations with real cosines, determined by elements of the real/complex vectors *x* and *y*.

For slargv/dlargv:

$$\begin{bmatrix} c(i) & s(i) \\ -s(i) & c(i) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} a_i \\ 0 \end{bmatrix}, \text{for } i = 1,2,\ldots,n$$

For clargv/zlargv:

$$\begin{bmatrix} c(i) & s(i) \\ -\text{conjg}(s(i)) & c(i) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} r_i \\ 0 \end{bmatrix}, \text{for } i = 1,2,\ldots,n$$

where $c(i)^2 + abs(s(i))^2 = 1$ and the following conventions are used (these are the same as in clartg/zlartg but differ from the BLAS Level 1 routine crotg/zrotg):

If $y_i = 0$, then $c(i) = 1$ and $s(i) = 0$;

If $x_i = 0$, then $c(i) = 0$ and $s(i)$ is chosen so that r_i is real.

Input Parameters

n	INTEGER. The number of plane rotations to be generated.
x, *y*	REAL for slargv DOUBLE PRECISION for dlargv COMPLEX for clargv COMPLEX*16 for zlargv Arrays, DIMENSION (1+(*n*-1)**incx*) and (1+(*n*-1)**incy*), respectively. On entry, the vectors *x* and *y*.
incx	INTEGER. The increment between elements of *x*. *incx* > 0.
incy	INTEGER. The increment between elements of *y*. *incy* > 0.

incc INTEGER. The increment between elements of the output array *c*. *incc* > 0.

Output Parameters

x On exit, *x*(*i*) is overwritten by a_i (for real flavors), or by r_i (for complex flavors), for *i* = 1,...,*n*.

y On exit, the sines *s*(*i*) of the plane rotations.

c REAL for slargv/clargv
DOUBLE PRECISION for dlargv/zlargv
Array, DIMENSION (1+(*n*-1)**incc*). The cosines of the plane rotations.

?larnv

Returns a vector of random numbers from a uniform or normal distribution.

Syntax

```
call slarnv( idist, iseed, n, x )
call dlarnv( idist, iseed, n, x )
call clarnv( idist, iseed, n, x )
call zlarnv( idist, iseed, n, x )
```

Description

The routine ?larnv returns a vector of *n* random real/complex numbers from a uniform or normal distribution.

This routine calls the auxiliary routine ?laruv to generate random real numbers from a uniform (0,1) distribution, in batches of up to 128 using vectorisable code. The Box-Muller method is used to transform numbers from a uniform to a normal distribution.

Input Parameters

idist INTEGER. Specifies the distribution of the random numbers:
for slarnv and dlanrv:
= 1: uniform (0,1)
= 2: uniform (-1,1)

= 3: normal (0,1).
for `clarnv` and `zlanrv`:
= 1: real and imaginary parts each uniform (0,1)
= 2: real and imaginary parts each uniform (-1,1)
= 3: real and imaginary parts each normal (0,1)
= 4: uniformly distributed on the disc abs(z) < 1
= 5: uniformly distributed on the circle abs(z) = 1

iseed — `INTEGER`. Array, `DIMENSION` (4).
On entry, the seed of the random number generator; the array elements must be between 0 and 4095, and *iseed*(4) must be odd.

n — `INTEGER`. The number of random numbers to be generated.

Output Parameters

x — `REAL` for `slarnv`
`DOUBLE PRECISION` for `dlarnv`
`COMPLEX` for `clarnv`
`COMPLEX*16` for `zlarnv`
Array, `DIMENSION` (*n*). The generated random numbers.

iseed — On exit, the seed is updated.

?larra

Computes the splitting points with the specified threshold.

Syntax

```
call slarra( n, d, e, e2, spltol, tnrm, nsplit, isplit, info )
call dlarra( n, d, e, e2, spltol, tnrm, nsplit, isplit, info )
```

Description

This routine computes the splitting points with the specified threshold and sets any "small" off-diagonal elements to zero.

Input Parameters

n — `INTEGER`. The order of the matrix (*n* > 1).

d
REAL for slarra
DOUBLE PRECISION for dlarra
Array, DIMENSION (*n*).
Contains *n* diagonal elements of the tridiagonal matrix *T*.

e
REAL for slarra
DOUBLE PRECISION for dlarra
Array, DIMENSION (*n*).
First (*n*–1) entries contain the subdiagonal elements of the tridiagonal matrix *T*; *e*(*n*) need not be set.

e2
REAL for slarra
DOUBLE PRECISION for dlarra
Array, DIMENSION (*n*).
First (*n*–1) entries contain the squares of the subdiagonal elements of the tridiagonal matrix *T*; *e2*(*n*) need not be set.

spltol
REAL for slarra
DOUBLE PRECISION for dlarra
The threshold for splitting. Two criteria can be used:
spltol<0 : criterion based on absolute off-diagonal value;
spltol>0 : criterion that preserves relative accuracy.

tnrm
REAL for slarra
DOUBLE PRECISION for dlarra
The norm of the matrix.

Output Parameters

e
On exit, the entries *e*(*isplit*(i)), 1 ≤ i ≤ *nsplit*, are set to zero, the other entries of *e* are untouched.

e2
On exit, the entries *e2*(*isplit*(i)), 1 ≤ i ≤ *nsplit*, are set to zero.

nsplit
INTEGER.
The number of blocks the matrix *T* splits into. 1 ≤ *nsplit* ≤ *n*

isplit
INTEGER.
Array, DIMENSION (*n*).

	The splitting points, at which *T* breaks up into blocks. The first block consists of rows/columns 1 to *isplit*(1), the second of rows/columns *isplit*(1)+1 through *isplit*(2), and so on, and the *nsplit*-th consists of rows/columns *isplit*(*nsplit*-1)+1 through *isplit*(*nsplit*)=*n*.
info	INTEGER. = 0: successful exit.

?larrb

Provides limited bisection to locate eigenvalues for more accuracy.

Syntax

call slarrb(*n*, *d*, *lld*, *ifirst*, *ilast*, *rtol1*, *rtol2*, *offset*, *w*, *wgap*, *werr*, *work*, *iwork*, *pivmin*, *spdiam*, *twist*, *info*)

call dlarrb(*n*, *d*, *lld*, *ifirst*, *ilast*, *rtol1*, *rtol2*, *offset*, *w*, *wgap*, *werr*, *work*, *iwork*, *pivmin*, *spdiam*, *twist*, *info*)

Description

Given the relatively robust representation (RRR) $L*D*L^T$, the routine does "limited" bisection to refine the eigenvalues of $L*D*L^T$, *w*(*ifirst*-*offset*) through *w*(*ilast*-*offset*), to more accuracy. Initial guesses for these eigenvalues are input in *w*. The corresponding estimate of the error in these guesses and their gaps are input in *werr* and *wgap*, respectively. During bisection, intervals [*left*, *right*] are maintained by storing their mid-points and semi-widths in the arrays *w* and *werr* respectively.

Input Parameters

n	INTEGER. The order of the matrix.
d	REAL for slarrb DOUBLE PRECISION for dlarrb Array, DIMENSION (*n*). The *n* diagonal elements of the diagonal matrix *D*.
lld	REAL for slarrb DOUBLE PRECISION for dlarrb Array, DIMENSION (*n*-1).

	The n-1 elements $L_i*L_i*D_i$.
ifirst	INTEGER. The index of the first eigenvalue to be computed.
ilast	INTEGER. The index of the last eigenvalue to be computed.
rtol1, *rtol2*	REAL for slarrb DOUBLE PRECISION for dlarrb Tolerance for the convergence of the bisection intervals. An interval [*left*, *right*] has converged if RIGHT-LEFT.LT.MAX(*rtol1**gap, *rtol2**max(|left|,|right|)), where gap is the (estimated) distance to the nearest eigenvalue.
offset	INTEGER. Offset for the arrays *w*, *wgap* and *werr*, that is, the *ifirst*-*offset* through *ilast*-*offset* elements of these arrays are to be used.
w	REAL for slarrb DOUBLE PRECISION for dlarrb Array, DIMENSION (n). On input, *w*(*ifirst*-*offset*) through *w*(*ilast*-*offset*) are estimates of the eigenvalues of $L*D*L^T$ indexed *ifirst* through *ilast*.
wgap	REAL for slarrb DOUBLE PRECISION for dlarrb Array, DIMENSION (n-1). The estimated gaps between consecutive eigenvalues of $L*D*L^T$, that is, *wgap*(*i*-*offset*) is the gap between eigenvalues i and i+1. Note that if IFIRST.EQ.ILAST then *wgap*(*ifirst*-*offset*) must be set to 0.
werr	REAL for slarrb DOUBLE PRECISION for dlarrb Array, DIMENSION (n). On input, *werr*(*ifirst*-*offset*) through *werr*(*ilast*-*offset*) are the errors in the estimates of the corresponding elements in *w*.
work	REAL for slarrb DOUBLE PRECISION for dlarrb Workspace array, DIMENSION (2*n).
pivmin	REAL for slarrb DOUBLE PRECISION for dlarrb The minimum pivot in the Sturm sequence.

spdiam	REAL for slarrb DOUBLE PRECISION for dlarrb The spectral diameter of the matrix.
twist	INTEGER. The twist index for the twisted factorization that is used for the negcount. *twist* = *n*: Compute negcount from $L*D*L^T - lambda*i = L+ * D+ * L+^T$ *twist* = *n*: Compute negcount from $L*D*L^T - lambda*i = U- * D- * U-^T$ *twist* = *n*: Compute negcount from $L*D*L^T - lambda*i = N_r*D_r*N_r$
iwork	INTEGER. Workspace array, DIMENSION (2**n*).

Output Parameters

w	On output, the estimates of the eigenvalues are"refined".
wgap	On output, the gaps are refined.
werr	On output, "refined" errors in the estimates of *w*.
info	INTEGER. Error flag.

?larrc

Computes the number of eigenvalues of the symmetric tridiagonal matrix.

Syntax

```
call slarrc( jobt, n, vl, vu, d, e, pivmin, eigcnt, lcnt, rcnt, info )
call dlarrc( jobt, n, vl, vu, d, e, pivmin, eigcnt, lcnt, rcnt, info )
```

Description

This routine finds the number of eigenvalues of the symmetric tridiagonal matrix *T* or of its factorization *L*D*L**T* in the specified interval.

Input Parameters

jobt — CHARACTER*1.
= 'T': computes Sturm count for matrix *T*.
= 'L': computes Sturm count for matrix *L*D*L**T*.

n — INTEGER.
The order of the matrix. (*n* > 1).

vl,*vu* — REAL for slarrc
DOUBLE PRECISION for dlarrc
The lower and upper bounds for the eigenvalues.

d — REAL for slarrc
DOUBLE PRECISION for dlarrc
Array, DIMENSION (*n*).
If *jobt*= 'T': contains the *n* diagonal elements of the tridiagonal matrix *T*.
If *jobt*= 'L': contains the *n* diagonal elements of the diagonal matrix *D*.

e — REAL for slarrc
DOUBLE PRECISION for dlarrc
Array, DIMENSION (*n*).
If *jobt*= 'T': contains the (*n*-1) offdiagonal elements of the matrix *T*.
If *jobt*= 'L': contains the (*n*-1) offdiagonal elements of the matrix *L*.

pivmin — REAL for slarrc
DOUBLE PRECISION for dlarrc
The minimum pivot in the Sturm sequence for the matrix *T*.

Output Parameters

eigcnt — INTEGER.
The number of eigenvalues of the symmetric tridiagonal matrix *T* that are in the half-open interval (*vl*,*vu*].

lcnt,*rcnt* — INTEGER.
The left and right negcounts of the interval.

info — INTEGER.

Now it is not used and always is set to 0.

?larrd

Computes the eigenvalues of a symmetric tridiagonal matrix to suitable accuracy.

Syntax

```
call slarrd( range, order, n, vl, vu, il, iu, gers, reltol, d, e, e2, pivmin,
nsplit, isplit, m, w, werr, wl, wu, iblock, indexw, work, iwork, info )
```

```
call dlarrd( range, order, n, vl, vu, il, iu, gers, reltol, d, e, e2, pivmin,
nsplit, isplit, m, w, werr, wl, wu, iblock, indexw, work, iwork, info )
```

Description

The routine computes the eigenvalues of a symmetric tridiagonal matrix T to suitable accuracy. This is an auxiliary code to be called from ?stemr. The user may ask for all eigenvalues, all eigenvalues in the half-open interval $(vl, vu]$, or the il-th through iu-th eigenvalues.

To avoid overflow, the matrix must be scaled so that its largest element is no greater than $(overflow^{1/2}*underflow^{1/4})$ in absolute value, and for greatest accuracy, it should not be much smaller than that. (For more details see [Kahan66].

Input Parameters

range — CHARACTER.
= 'A': ("All") all eigenvalues will be found.
= 'V': ("Value") all eigenvalues in the half-open interval $(vl, vu]$ will be found.
= 'I': ("Index") the il-th through iu-th eigenvalues will be found.

order — CHARACTER.
= 'B': ("By block") the eigenvalues will be grouped by split-off block (see *iblock*, *isplit* below) and ordered from smallest to largest within the block.
= 'E': ("Entire matrix") the eigenvalues for the entire matrix will be ordered from smallest to largest.

n — INTEGER. The order of the tridiagonal matrix T ($n \geq 1$).

vl,*vu*
REAL for slarrd
DOUBLE PRECISION for dlarrd
If *range* = 'V': the lower and upper bounds of the interval to be searched for eigenvalues. Eigenvalues less than or equal to *vl*, or greater than *vu*, will not be returned. *vl* < *vu*.
If *range* = 'A' or 'I': not referenced.

il,*iu*
INTEGER.
If *range* = 'I': the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 ≤ *il* ≤ *iu* ≤ *n*, if *n* > 0; *il*=1 and *iu*=0 if *n*=0.
If *range* = 'A' or 'V': not referenced.

gers
REAL for slarrd
DOUBLE PRECISION for dlarrd
Array, DIMENSION (2*n).
The *n* Gerschgorin intervals (the i-th Gerschgorin interval is (*gers*(2*i-1), *gers*(2*i)).

reltol
REAL for slarrd
DOUBLE PRECISION for dlarrd
The minimum relative width of an interval. When an interval is narrower than *reltol* times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, that is converged. Note: this should always be at least *radix*machine epsilon*.

d
REAL for slarrd
DOUBLE PRECISION for dlarrd
Array, DIMENSION (*n*).
Contains *n* diagonal elements of the tridiagonal matrix *T*.

e
REAL for slarrd
DOUBLE PRECISION for dlarrd
Array, DIMENSION (*n*-1).
Contains (*n*-1) off-diagonal elements of the tridiagonal matrix *T*.

e2
REAL for slarrd
DOUBLE PRECISION for dlarrd
Array, DIMENSION (*n*-1).

Contains (*n*-1) squared off-diagonal elements of the tridiagonal matrix *T*.

pivmin — REAL for slarrd
DOUBLE PRECISION for dlarrd
The minimum pivot in the Sturm sequence for the matrix *T*.

nsplit — INTEGER.
The number of diagonal blocks the matrix *T*. 1 ≤ *nsplit* ≤ *n*

isplit — INTEGER.
Arrays, DIMENSION (*n*).
The splitting points, at which *T* breaks up into submatrices. The first submatrix consists of rows/columns 1 to *isplit*(1), the second of rows/columns *isplit*(1)+1 through *isplit*(2), and so on, and the *nsplit*-th consists of rows/columns *isplit*(*nsplit*-1)+1 through *isplit*(*nsplit*)=*n*.
(Only the first *nsplit* elements actually is used, but since the user cannot know a priori value of *nsplit*, *n* words must be reserved for *isplit*.)

work — REAL for slarrd
DOUBLE PRECISION for dlarrd
Workspace array, DIMENSION (4**n*).

iwork — INTEGER.
Workspace array, DIMENSION (4**n*).

Output Parameters

m — INTEGER.
The actual number of eigenvalues found. 0 ≤ *m* ≤ *n*. (See also the description of *info*=2, 3.)

w — REAL for slarrd
DOUBLE PRECISION for dlarrd
Array, DIMENSION (*n*).

The first *m* elements of *w* contain the eigenvalue approximations. ?laprd computes an interval $I_j = (a_j, b_j]$ that includes eigenvalue j. The eigenvalue approximation is given as the interval midpoint *w*(j)= $(a_j+b_j)/2$. The corresponding error is bounded by *werr*(j) = abs$(a_j-b_j)/2$.

werr

REAL for slarrd
DOUBLE PRECISION for dlarrd
Array, DIMENSION (*n*).
The error bound on the corresponding eigenvalue approximation in *w*.

wl, *wu*

REAL for slarrd
DOUBLE PRECISION for dlarrd
The interval (*wl*, *wu*] contains all the wanted eigenvalues.
If *range* = 'V': then *wl*=*vl* and *wu*=*vu*.
If *range* = 'A': then *wl* and *wu* are the global Gerschgorin bounds on the spectrum.
If *range* = 'I': then *wl* and *wu* are computed by ?laebz from the index range specified.

iblock

INTEGER.
Array, DIMENSION (*n*).
At each row/column j where *e*(j) is zero or small, the matrix *T* is considered to split into a block diagonal matrix. If *info* = 0, then *iblock*(i) specifies to which block (from 1 to the number of blocks) the eigenvalue *w*(i) belongs. (The routine may use the remaining *n*–*m* elements as workspace.)

indexw

INTEGER.
Array, DIMENSION (*n*).
The indices of the eigenvalues within each block (submatrix); for example, *indexw*(i)= j and *iblock*(i)=k imply that the i-th eigenvalue *w*(i) is the j-th eigenvalue in block k.

info

INTEGER.
= 0: successful exit.
< 0: if *info* = -i, the i-th argument has an illegal value
> 0: some or all of the eigenvalues fail to converge or are not computed:

=1 or 3: bisection fail to converge for some eigenvalues; these eigenvalues are flagged by a negative block number. The effect is that the eigenvalues may not be as accurate as the absolute and relative tolerances.
=2 or 3: *range*='I' only: not all of the eigenvalues *il*:*iu* are found.
=4: *range*='I', and the Gershgorin interval initially used is too small. No eigenvalues are computed.

?larre

Given the tridiagonal matrix T, sets small off-diagonal elements to zero and for each unreduced block T_i, finds base representations and eigenvalues.

Syntax

```
call slarre( range, n, vl, vu, il, iu, d, e, e2, rtol1, rtol2, spltol, nsplit,
isplit, m, w, werr, wgap, iblock, indexw, gers, pivmin, work, iwork, info )
```

```
call dlarre( range, n, vl, vu, il, iu, d, e, e2, rtol1, rtol2, spltol, nsplit,
isplit, m, w, werr, wgap, iblock, indexw, gers, pivmin, work, iwork, info )
```

Description

To find the desired eigenvalues of a given real symmetric tridiagonal matrix T, the routine sets any "small" off-diagonal elements to zero, and for each unreduced block T_i, it finds

- a suitable shift at one end of the block spectrum
- the base representation, $T_i - \sigma_i * I = L_i * D_i * L_i^T$, and
- eigenvalues of each $L_i * D_i * L_i^T$.

The representations and eigenvalues found are then used by ?stemr to compute the eigenvectors of a symmetric tridiagonal matrix. The accuracy varies depending on whether bisection is used to find a few eigenvalues or the dqds algorithm (subroutine ?lasq2) to compute all and discard any unwanted one. As an added benefit, ?larre also outputs the n Gerschgorin intervals for the matrices $L_i * D_i * L_i^T$.

Input Parameters

range — CHARACTER.
= 'A': ("All") all eigenvalues will be found.
= 'V': ("Value") all eigenvalues in the half-open interval (*vl*, *vu*] will be found.
= 'I': ("Index") the *il*-th through *iu*-th eigenvalues of the entire matrix will be found.

n — INTEGER. The order of the matrix. $n > 0$.

vl, *vu* — REAL for slarre
DOUBLE PRECISION for dlarre
If *range*='V', the lower and upper bounds for the eigenvalues. Eigenvalues less than or equal to *vl*, or greater than *vu*, are not returned. $vl < vu$.

il, *iu* — INTEGER.
If *range*='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. $1 \le il \le iu \le n$.

d — REAL for slarre
DOUBLE PRECISION for dlarre
Array, DIMENSION (*n*).
The *n* diagonal elements of the diagonal matrices *T*.

e — REAL for slarre
DOUBLE PRECISION for dlarre
Array, DIMENSION (*n*). The first (*n*-1) entries contain the subdiagonal elements of the tridiagonal matrix *T*; *e*(*n*) need not be set.

e2 — REAL for slarre
DOUBLE PRECISION for dlarre
Array, DIMENSION (*n*). The first (*n*-1) entries contain the squares of the subdiagonal elements of the tridiagonal matrix *T*; *e2*(*n*) need not be set.

rtol1, *rtol2* — REAL for slarre
DOUBLE PRECISION for dlarre

	Parameters for bisection. An interval `[LEFT,RIGHT]` has converged if `RIGHT-LEFT.LT.MAX( rtol1*gap, rtol2*max(|LEFT|,|RIGHT|) )`.

spltol	`REAL` for `slarre`
	`DOUBLE PRECISION` for `dlarre`
	The threshold for splitting.

work	`REAL` for `slarre`
	`DOUBLE PRECISION` for `dlarre`
	Workspace array, `DIMENSION` (6*n).

iwork	`INTEGER`. Workspace array, `DIMENSION` (5*n).

Output Parameters

vl, *vu*	On exit, if *range*='I' or ='A', contain the bounds on the desired part of the spectrum.
d	On exit, the n diagonal elements of the diagonal matrices D_i.
e	On exit, the subdiagonal elements of the unit bidiagonal matrices L_i. The entries *e*(*isplit*(i)), $1 \le i \le nsplit$, contain the base points $sigma_i$ on output.
e2	On exit, the entries *e2*(*isplit*(i)), $1 \le i \le nsplit$, have been set to zero.
nsplit	`INTEGER`. The number of blocks T splits into. $1 \le nsplit \le n$.
isplit	`INTEGER`. Array, `DIMENSION` (n). The splitting points, at which T breaks up into blocks. The first block consists of rows/columns 1 to *isplit*(1), the second of rows/columns *isplit*(1)+1 through *isplit*(2), etc., and the *nsplit*-th consists of rows/columns *isplit*(*nsplit*-1)+1 through *isplit*(*nsplit*)=*n*.
m	`INTEGER`. The total number of eigenvalues (of all the $L_i*D_i*L_i^T$) found.
w	`REAL` for `slarre` `DOUBLE PRECISION` for `dlarre`

Array, DIMENSION (n). The first m elements contain the eigenvalues. The eigenvalues of each of the blocks, $L_i*D_i*L_i^T$, are sorted in ascending order. The routine may use the remaining n-m elements as workspace.

werr
REAL for slarre
DOUBLE PRECISION for dlarre
Array, DIMENSION (n). The error bound on the corresponding eigenvalue in w.

wgap
REAL for slarre
DOUBLE PRECISION for dlarre
Array, DIMENSION (n). The separation from the right neighbor eigenvalue in w. The gap is only with respect to the eigenvalues of the same block as each block has its own representation tree. Exception: at the right end of a block the left gap is stored.

iblock
INTEGER. Array, DIMENSION (n).
The indices of the blocks (submatrices) associated with the corresponding eigenvalues in w; *iblock*(*i*)=1 if eigenvalue $w(i)$ belongs to the first block from the top, =2 if $w(i)$ belongs to the second block, etc.

indexw
INTEGER. Array, DIMENSION (n).
The indices of the eigenvalues within each block (submatrix); for example, *indexw*(*i*)= 10 and *iblock*(*i*)=2 imply that the i-th eigenvalue $w(i)$ is the 10-th eigenvalue in the second block.

gers
REAL for slarre
DOUBLE PRECISION for dlarre
Array, DIMENSION (2*n). The n Gerschgorin intervals (the i-th Gerschgorin interval is (*gers*(2*i-1), *gers*(2*i)).

pivmin
REAL for slarre
DOUBLE PRECISION for dlarre
The minimum pivot in the Sturm sequence for T.

info
INTEGER.
If *info* = 0: successful exit
If *info* > 0: A problem occured in ?larre. If *info* = 5, the Rayleigh Quotient Iteration failed to converge to full accuracy.

If *info* < 0: One of the called subroutines signaled an internal problem. Inspection of the corresponding parameter *info* for further information is required.

- If *info* = -1, there is a problem in ?larrd
- If *info* = -2, no base representation could be found in maxtry iterations. Increasing maxtry and recompilation might be a remedy.
- If *info* = -3, there is a problem in ?larrb when computing the refined root representation for ?lasq2.
- If *info* = -4, there is a problem in ?larrb when preforming bisection on the desired part of the spectrum.
- If *info* = -5, there is a problem in ?lasq2.
- If *info* = -6, there is a problem in ?lasq2.

?larrf

Finds a new relatively robust representation such that at least one of the eigenvalues is relatively isolated.

Syntax

call slarrf(*n*, *d*, *l*, *ld*, *clstrt*, *clend*, *w*, *wgap*, *werr*, *spdiam*, *clgapl*, *clgapr*, *pivmin*, *sigma*, *dplus*, *lplus*, *work*, *info*)

call dlarrf(*n*, *d*, *l*, *ld*, *clstrt*, *clend*, *w*, *wgap*, *werr*, *spdiam*, *clgapl*, *clgapr*, *pivmin*, *sigma*, *dplus*, *lplus*, *work*, *info*)

Description

Given the initial representation $L*D*L^T$ and its cluster of close eigenvalues (in a relative measure), $w(clstrt)$, $w(clstrt+1)$, ... $w(clend)$, the routine ?larrf finds a new relatively robust representation

$$L*D*L^T - \sigma_i*I = L(+)*D(+)*L(+)^T$$

such that at least one of the eigenvalues of $L(+)*D*(+)*L(+)^T$ is relatively isolated.

Input Parameters

n	INTEGER. The order of the matrix (subblock, if the matrix is splitted).
d	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION (n). The n diagonal elements of the diagonal matrix D.
l	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION (n-1). The (n-1) subdiagonal elements of the unit bidiagonal matrix L.
ld	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION (n-1). The n-1 elements L_i*D_i.
clstrt	INTEGER. The index of the first eigenvalue in the cluster.
clend	INTEGER. The index of the last eigenvalue in the cluster.
w	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION ≥ (*clend* -*clstrt*+1). The eigenvalue approximations of $L*D*L^T$ in ascending order. w(*clstrt*) through w(*clend*) form the cluster of relatively close eigenvalues.
wgap	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION ≥ (*clend* -*clstrt*+1). The separation from the right neighbor eigenvalue in w.
werr	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION ≥ (*clend* -*clstrt*+1). On input, *werr* contains the semiwidth of the uncertainty interval of the corresponding eigenvalue approximation in w.
spdiam	REAL for slarrf DOUBLE PRECISION for dlarrf

Estimate of the spectral diameter obtained from the Gerschgorin intervals.

clgapl, *clgapr*
REAL for slarrf
DOUBLE PRECISION for dlarrf
Absolute gap on each end of the cluster. Set by the calling routine to protect against shifts too close to eigenvalues outside the cluster.

pivmin
REAL for slarrf
DOUBLE PRECISION for dlarrf
The minimum pivot allowed in the Sturm sequence.

work
REAL for slarrf
DOUBLE PRECISION for dlarrf
Workspace array, DIMENSION (2*n).

Output Parameters

wgap
On output, the gaps are refined.

sigma
REAL for slarrf
DOUBLE PRECISION for dlarrf
The shift used to form $L(+)*D*(+)*L(+)^T$.

dplus
REAL for slarrf
DOUBLE PRECISION for dlarrf
Array, DIMENSION (n). The n diagonal elements of the diagonal matrix $D(+)$.

lplus
REAL for slarrf
DOUBLE PRECISION for dlarrf
Array, DIMENSION (n). The first (n-1) elements of *lplus* contain the subdiagonal elements of the unit bidiagonal matrix $L(+)$.

?larrj

Performs refinement of the initial estimates of the eigenvalues of the matrix T.

Syntax

```
call slarrj( n, d, e2, ifirst, ilast, rtol, offset, w, werr, work, iwork,
pivmin, spdiam, info )
```

```
call dlarrj( n, d, e2, ifirst, ilast, rtol, offset, w, werr, work, iwork,
pivmin, spdiam, info )
```

Description

Given the initial eigenvalue approximations of *T*, this routine does bisection to refine the eigenvalues of *T*, *w*(*ifirst-offset*) through *w*(*ilast-offset*) , to more accuracy. Initial guesses for these eigenvalues are input in *w*, the corresponding estimate of the error in these guesses in *werr*. During bisection, intervals [a,b] are maintained by storing their mid-points and semi-widths in the arrays *w* and *werr* respectively.

Input Parameters

n — INTEGER. The order of the matrix *T*.

d — REAL for slarrj
DOUBLE PRECISION for dlarrj
Array, DIMENSION (*n*).
Contains *n* diagonal elements of the matrix *T*.

e2 — REAL for slarrj
DOUBLE PRECISION for dlarrj
Array, DIMENSION (*n*–1).
Contains (*n*–1) squared sub-diagonal elements of the *T*.

ifirst — INTEGER.
The index of the first eigenvalue to be computed.

ilast — INTEGER.
The index of the last eigenvalue to be computed.

rtol — REAL for slarrj
DOUBLE PRECISION for dlarrj

	Tolerance for the convergence of the bisection intervals. An interval `[a,b]` is considered to be converged if `(b-a) ≤ rtol*max(|a|,|b|)`.
offset	`INTEGER`. Offset for the arrays *w* and *werr*, that is the *ifirst-offset* through *ilast-offset* elements of these arrays are to be used.

w	`REAL` for `slarrj`
	`DOUBLE PRECISION` for `dlarrj`
	Array, `DIMENSION` (*n*).
	On input, *w*(*ifirst-offset*) through *w*(*ilast-offset*) are estimates of the eigenvalues of *L*D*L**T* indexed *ifirst* through *ilast*.

werr	`REAL` for `slarrj`
	`DOUBLE PRECISION` for `dlarrj`
	Array, `DIMENSION` (*n*).
	On input, *werr*(*ifirst-offset*) through *werr*(*ilast-offset*) are the errors in the estimates of the corresponding elements in *w*.

work	`REAL` for `slarrj`
	`DOUBLE PRECISION` for `dlarrj`
	Workspace array, `DIMENSION` (2**n*).

iwork	`INTEGER`. Workspace array, `DIMENSION` (2**n*).

pivmin	`REAL` for `slarrj`
	`DOUBLE PRECISION` for `dlarrj`
	The minimum pivot in the Sturm sequence for the matrix *T*.

spdiam	`REAL` for `slarrj`
	`DOUBLE PRECISION` for `dlarrj`
	The spectral diameter of the matrix *T*.

Output Parameters

w	On exit, contains the refined estimates of the eigenvalues.

werr	On exit, contains the refined errors in the estimates of the corresponding elements in *w*.
info	INTEGER. Now it is not used and always is set to 0.

?larrk

Computes one eigenvalue of a symmetric tridiagonal matrix T to suitable accuracy.

Syntax

```
call slarrk( n, iw, gl, gu, d, e2, pivmin, reltol, w, werr, info )
call dlarrk( n, iw, gl, gu, d, e2, pivmin, reltol, w, werr, info )
```

Description

The routine computes one eigenvalue of a symmetric tridiagonal matrix *T* to suitable accuracy. This is an auxiliary code to be called from ?stemr.

To avoid overflow, the matrix must be scaled so that its largest element is no greater than $(overflow^{1/2}*underflow^{1/4})$ in absolute value, and for greatest accuracy, it should not be much smaller than that. (For more details see [[Kahan66]].

Input Parameters

n	INTEGER. The order of the matrix *T*. ($n \geq 1$).
iw	INTEGER. The index of the eigenvalue to be returned.
gl, *gu*	REAL for slarrk DOUBLE PRECISION for dlarrk An upper and a lower bound on the eigenvalue.
d	REAL for slarrk DOUBLE PRECISION for dlarrk Array, DIMENSION (n). Contains n diagonal elements of the matrix *T*.
e2	REAL for slarrk DOUBLE PRECISION for dlarrk Array, DIMENSION ($n-1$).

Contains (*n*-1) squared off-diagonal elements of the *T*.

pivmin — REAL for slarrk
DOUBLE PRECISION for dlarrk
The minimum pivot in the Sturm sequence for the matrix *T*.

reltol — REAL for slarrk
DOUBLE PRECISION for dlarrk
The minimum relative width of an interval. When an interval is narrower than *reltol* times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, that is converged. Note: this should always be at least *radix*machine epsilon*.

Output Parameters

w — REAL for slarrk
DOUBLE PRECISION for dlarrk
Contains the eigenvalue approximation.

werr — REAL for slarrk
DOUBLE PRECISION for dlarrk
Contains the error bound on the corresponding eigenvalue approximation in *w*.

info — INTEGER.
= 0: Eigenvalue converges
= -1: Eigenvalue does not converge

?larrr

Performs tests to decide whether the symmetric tridiagonal matrix T warrants expensive computations which guarantee high relative accuracy in the eigenvalues.

Syntax

```
call slarrr( n, d, e, info )
call dlarrr( n, d, e, info )
```

Description

The routine performs tests to decide whether the symmetric tridiagonal matrix T warrants expensive computations which guarantee high relative accuracy in the eigenvalues.

Input Parameters

n	INTEGER. The order of the matrix T. ($n > 0$).
d	REAL for slarrr DOUBLE PRECISION for dlarrr Array, DIMENSION (n). Contains n diagonal elements of the matrix T.
e	REAL for slarrr DOUBLE PRECISION for dlarrr Array, DIMENSION (n). The first $(n-1)$ entries contain sub-diagonal elements of the tridiagonal matrix T; $e(n)$ is set to 0.

Output Parameters

info	INTEGER. = 0: the matrix warrants computations preserving relative accuracy (default value). = -1: the matrix warrants computations guaranteeing only absolute accuracy.

?larrv

Computes the eigenvectors of the tridiagonal matrix $T = L*D*L^T$ given L, D and the eigenvalues of $L*D*L^T$.

Syntax

```
call slarrv( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, minrgp, rtol1,
rtol2, w, werr, wgap, iblock, indexw, gers, z, ldz, isuppz, work, iwork, info
)
```

```
call dlarrv( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, minrgp, rtol1,
rtol2, w, werr, wgap, iblock, indexw, gers, z, ldz, isuppz, work, iwork, info
)
```

```
call clarrv( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, minrgp, rtol1,
rtol2, w, werr, wgap, iblock, indexw, gers, z, ldz, isuppz, work, iwork, info
)
```

```
call zlarrv( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, minrgp, rtol1,
rtol2, w, werr, wgap, iblock, indexw, gers, z, ldz, isuppz, work, iwork, info
)
```

Description

The routine ?larrv computes the eigenvectors of the tridiagonal matrix $T = L*D*L^T$ given L, D and approximations to the eigenvalues of $L*D*L^T$.

The input eigenvalues should have been computed by slarre for real flavors (slarrv/clarrv) and by dlarre for double precision flavors (dlarre/zlarre).

Input Parameters

n	INTEGER. The order of the matrix. $n \geq 0$.
vl, *vu*	REAL for slarrv/clarrv DOUBLE PRECISION for dlarrv/zlarrv Lower and upper bounds respectively of the interval that contains the desired eigenvalues. $vl < vu$. Needed to compute gaps on the left or right end of the extremal eigenvalues in the desired range.
d	REAL for slarrv/clarrv

DOUBLE PRECISION for dlarrv/zlarrv
Array, DIMENSION (*n*). On entry, the *n* diagonal elements of the diagonal matrix *D*.

l — REAL for slarrv/clarrv
DOUBLE PRECISION for dlarrv/zlarrv
Array, DIMENSION (*n*).
On entry, the (*n*-1) subdiagonal elements of the unit bidiagonal matrix *L* are contained in elements 1 to *n*-1 of *L* if the matrix is not splitted. At the end of each block the corresponding shift is stored as given by slarre for real flavors and by dlarre for double precision flavors.

pivmin — REAL for slarrv/clarrv
DOUBLE PRECISION for dlarrv/zlarrv
The minimum pivot allowed in the Sturm sequence.

isplit — INTEGER. Array, DIMENSION (*n*).
The splitting points, at which *T* breaks up into blocks. The first block consists of rows/columns 1 to *isplit*(1), the second of rows/columns *isplit*(1)+1 through *isplit*(2), etc.

m — INTEGER. The total number of eigenvalues found.
$0 \le m \le n$. If *range* = 'A', $m = n$, and if *range* = 'I', $m = iu - il + 1$.

dol, *dou* — INTEGER.
If you want to compute only selected eigenvectors from all the eigenvalues supplied, specify an index range *dol*:*dou*. Or else apply the setting *dol*=1, *dou*=*m*. Note that *dol* and *dou* refer to the order in which the eigenvalues are stored in *w*.
If you want to compute only selected eigenpairs, then the columns *dol*-1 to *dou*+1 of the eigenvector space *Z* contain the computed eigenvectors. All other columns of *Z* are set to zero.

minrgp, *rtol1*, *rtol2* — REAL for slarrv/clarrv
DOUBLE PRECISION for dlarrv/zlarrv
Parameters for bisection. An interval [LEFT,RIGHT] has converged if RIGHT-LEFT.LT.MAX(*rtol1**gap, *rtol2**max(|LEFT|,|RIGHT|)).

w
REAL for slarrv/clarrv
DOUBLE PRECISION for dlarrv/zlarrv
Array, DIMENSION (*n*). The first *m* elements of *w* contain the approximate eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block (the output array *w* from ?larre is expected here). These eigenvalues are set with respect to the shift of the corresponding root representation for their block.

werr
REAL for slarrv/clarrv
DOUBLE PRECISION for dlarrv/zlarrv
Array, DIMENSION (*n*). The first *m* elements contain the semiwidth of the uncertainty interval of the corresponding eigenvalue in *w*.

wgap
REAL for slarrv/clarrv
DOUBLE PRECISION for dlarrv/zlarrv
Array, DIMENSION (*n*). The separation from the right neighbor eigenvalue in *w*.

iblock
INTEGER. Array, DIMENSION (*n*).
The indices of the blocks (submatrices) associated with the corresponding eigenvalues in *w*; *iblock*(*i*)=1 if eigenvalue *w*(*i*) belongs to the first block from the top, =2 if *w*(*i*) belongs to the second block, etc.

indexw
INTEGER. Array, DIMENSION (*n*).
The indices of the eigenvalues within each block (submatrix); for example, *indexw*(*i*)= 10 and *iblock*(*i*)=2 imply that the *i*-th eigenvalue *w*(*i*) is the 10-th eigenvalue in the second block.

gers
REAL for slarrv/clarrv
DOUBLE PRECISION for dlarrv/zlarrv
Array, DIMENSION (2**n*). The *n* Gerschgorin intervals (the *i*-th Gerschgorin interval is (*gers*(2**i*-1), *gers*(2**i*)). The Gerschgorin intervals should be computed from the original unshifted matrix.

ldz
INTEGER. The leading dimension of the output array *Z*. *ldz* ≥ 1, and if *jobz* = 'V', *ldz* ≥ max(1,*n*).

work REAL for slarrv/clarrv
DOUBLE PRECISION for dlarrv/zlarrv
Workspace array, DIMENSION (12*n).

iwork INTEGER.
Workspace array, DIMENSION (7*n).

Output Parameters

d On exit, d may be overwritten.

l On exit, l is overwritten.

w On exit, w holds the eigenvalues of the unshifted matrix.

werr On exit, *werr* contains refined values of its input approximations.

wgap On exit, *wgap* contains refined values of its input approximations. Very small gaps are changed.

z REAL for slarrv
DOUBLE PRECISION for dlarrv
COMPLEX for clarrv
COMPLEX*16 for zlarrv
Array, DIMENSION (ldz, max(1,m)).
If *info* = 0, the first m columns of z contain the orthonormal eigenvectors of the matrix T corresponding to the input eigenvalues, with the i-th column of z holding the eigenvector associated with $w(i)$.

NOTE. The user must ensure that at least max(1,m) columns are supplied in the array z.

isuppz INTEGER .
Array, DIMENSION (2*max(1,m)). The support of the eigenvectors in z, that is, the indices indicating the nonzero elements in z. The i-th eigenvector is nonzero only in elements *isuppz*(2i-1) through *isuppz*(2i).

info INTEGER.
If *info* = 0: successful exit

If *info* > 0: A problem occured in ?larrv. If *info* = 5, the Rayleigh Quotient Iteration failed to converge to full accuracy.
If *info* < 0: One of the called subroutines signaled an internal problem. Inspection of the corresponding parameter *info* for further information is required.

- If *info* = -1, there is a problem in ?larrb when refining a child eigenvalue;
- If *info* = -2, there is a problem in ?larrf when computing the relatively robust representation (RRR) of a child. When a child is inside a tight cluster, it can be difficult to find an RRR. A partial remedy from the user's point of view is to make the parameter *minrgp* smaller and recompile. However, as the orthogonality of the computed vectors is proportional to 1/*minrgp*, you should be aware that you might be trading in precision when you decrease *minrgp*.
- If *info* = -3, there is a problem in ?larrb when refining a single eigenvalue after the Rayleigh correction was rejected.

?lartg

Generates a plane rotation with real cosine and real/complex sine.

Syntax

```
call slartg( f, g, cs, sn, r )
call dlartg( f, g, cs, sn, r )
call clartg( f, g, cs, sn, r )
call zlartg( f, g, cs, sn, r )
```

Description

The routine generates a plane rotation so that

$$\begin{bmatrix} cs & sn \\ -\text{conjg}(sn) & cs \end{bmatrix} \cdot \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

where $cs^2 + |sn|^2 = 1$

This is a slower, more accurate version of the BLAS Level 1 routine ?rotg, except for the following differences.

For slartg/dlartg:

f and g are unchanged on return;

If $g=0$, then $cs=1$ and $sn=0$;

If $f=0$ and $g \neq 0$, then $cs=0$ and $sn=1$ without doing any floating point operations (saves work in ?bdsqr when there are zeros on the diagonal);

If f exceeds g in magnitude, cs will be positive.

For clartg/zlartg:

f and g are unchanged on return;

If $g=0$, then $cs=1$ and $sn=0$;

If $f=0$, then $cs=0$ and sn is chosen so that r is real.

Input Parameters

f, *g*	REAL for slartg DOUBLE PRECISION for dlartg COMPLEX for clartg COMPLEX*16 for zlartg The first and second component of vector to be rotated.

Output Parameters

cs	REAL for slartg/clartg DOUBLE PRECISION for dlartg/zlartg The cosine of the rotation.
sn	REAL for slartg

DOUBLE PRECISION for dlartg
COMPLEX for clartg
COMPLEX*16 for zlartg
The sine of the rotation.

r REAL for slartg
DOUBLE PRECISION for dlartg
COMPLEX for clartg
COMPLEX*16 for zlartg
The nonzero component of the rotated vector.

?lartv

Applies a vector of plane rotations with real cosines and real/complex sines to the elements of a pair of vectors.

Syntax

```
call slartv( n, x, incx, y, incy, c, s, incc )
call dlartv( n, x, incx, y, incy, c, s, incc )
call clartv( n, x, incx, y, incy, c, s, incc )
call zlartv( n, x, incx, y, incy, c, s, incc )
```

Description

The routine applies a vector of real/complex plane rotations with real cosines to elements of the real/complex vectors *x* and *y*. For i = 1,2,...,*n*

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} := \begin{bmatrix} c(i) & s(i) \\ -\text{conjg}(s(i)) & c(i) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

Input Parameters

n INTEGER. The number of plane rotations to be applied.

x, *y* REAL for slartv

	DOUBLE PRECISION for dlartv COMPLEX for clartv COMPLEX*16 for zlartv Arrays, DIMENSION (1+(*n*-1)**incx*) and (1+(*n*-1)**incy*), respectively. The input vectors *x* and *y*.
incx	INTEGER. The increment between elements of *x*. *incx* > 0.
incy	INTEGER. The increment between elements of *y*. *incy* > 0.
c	REAL for slartv/clartv DOUBLE PRECISION for dlartv/zlartv Array, DIMENSION (1+(*n*-1)**incc*). The cosines of the plane rotations.
s	REAL for slartv DOUBLE PRECISION for dlartv COMPLEX for clartv COMPLEX*16 for zlartv Array, DIMENSION (1+(*n*-1)**incc*). The sines of the plane rotations.
incc	INTEGER. The increment between elements of *c* and *s*. *incc* > 0.

Output Parameters

x, *y*	The rotated vectors *x* and *y*.

?laruv

Returns a vector of n random real numbers from a uniform distribution.

Syntax

```
call slaruv( iseed, n, x )
call dlaruv( iseed, n, x )
```

Description

The routine ?laruv returns a vector of *n* random real numbers from a uniform (0,1) distribution (n ≤ 128).

This is an auxiliary routine called by ?larnv.

Input Parameters

iseed	INTEGER. Array, DIMENSION (4). On entry, the seed of the random number generator; the array elements must be between 0 and 4095, and *iseed*(4) must be odd.
n	INTEGER. The number of random numbers to be generated. *n* ≤ 128.

Output Parameters

x	REAL for slaruv DOUBLE PRECISION for dlaruv Array, DIMENSION (*n*). The generated random numbers.
seed	On exit, the seed is updated.

?larz

Applies an elementary reflector (as returned by ?tzrzf) to a general matrix.

Syntax

```
call slarz( side, m, n, l, v, incv, tau, c, ldc, work )
call dlarz( side, m, n, l, v, incv, tau, c, ldc, work )
call clarz( side, m, n, l, v, incv, tau, c, ldc, work )
call zlarz( side, m, n, l, v, incv, tau, c, ldc, work )
```

Description

The routine ?larz applies a real/complex elementary reflector *H* to a real/complex *m*-by-*n* matrix *C*, from either the left or the right. *H* is represented in the form

*H = I - tau*v*v'*,

where *tau* is a real/complex scalar and v is a real/complex vector.

If *tau = 0*, then *H* is taken to be the unit matrix.

For complex flavors, to apply H '(the conjugate transpose of H), supply conjg(tau) instead of tau.

H is a product of k elementary reflectors as returned by ?tzrzf.

Input Parameters

side	CHARACTER*1. If *side* = 'L': form H*C If *side* = 'R': form C*H
m	INTEGER. The number of rows of the matrix C.
n	INTEGER. The number of columns of the matrix C.
l	INTEGER. The number of entries of the vector v containing the meaningful part of the Householder vectors. If *side* = 'L', $m \geq L \geq 0$, if *side* = 'R', $n \geq L \geq 0$.
v	REAL for slarz DOUBLE PRECISION for dlarz COMPLEX for clarz COMPLEX*16 for zlarz Array, DIMENSION (1+(*l*-1)*abs(*incv*)). The vector v in the representation of H as returned by ?tzrzf. v is not used if *tau* = 0.
incv	INTEGER. The increment between elements of v. *incv* $\neq$ 0.
tau	REAL for slarz DOUBLE PRECISION for dlarz COMPLEX for clarz COMPLEX*16 for zlarz The value *tau* in the representation of H.
c	REAL for slarz DOUBLE PRECISION for dlarz COMPLEX for clarz COMPLEX*16 for zlarz Array, DIMENSION (*ldc*,*n*). On entry, the m-by-n matrix C.

ldc	INTEGER. The leading dimension of the array *c*. *ldc* ≥ max(1,*m*).
work	REAL for slarz DOUBLE PRECISION for dlarz COMPLEX for clarz COMPLEX*16 for zlarz Workspace array, DIMENSION (*n*) if *side* = 'L' or (*m*) if *side* = 'R'.

Output Parameters

c	On exit, *C* is overwritten by the matrix *H***C* if *side* = 'L', or *C***H* if *side* = 'R'.

?larzb

Applies a block reflector or its transpose/conjugate-transpose to a general matrix.

Syntax

```
call slarzb( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc,
work, ldwork )
call dlarzb( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc,
work, ldwork )
call clarzb( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc,
work, ldwork )
call zlarzb( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc,
work, ldwork )
```

Description

The routine applies a real/complex block reflector H or its transpose H^T (or H for complex flavors) to a real/complex distributed m-by-n matrix C from the left or the right. Currently, only *storev* = 'R' and *direct* = 'B' are supported.

Input Parameters

side CHARACTER*1.
If *side* = 'L': apply *H* or *H'* from the left
If *side* = 'R': apply *H* or *H'* from the right

trans CHARACTER*1.
If *trans* = 'N': apply *H* (No transpose)
If *trans*='C': apply *H'* (Transpose/conjugate transpose)

direct CHARACTER*1.
Indicates how *H* is formed from a product of elementary reflectors
= 'F': *H* = *H*(1) *H*(2)... *H*(*k*) (forward, not supported)
= 'B': *H* = *H*(*k*)... *H*(2) *H*(1) (backward)

storev CHARACTER*1.
Indicates how the vectors which define the elementary reflectors are stored:
= 'C': Column-wise (not supported)
= 'R': Row-wise.

m INTEGER. The number of rows of the matrix *C*.

n INTEGER. The number of columns of the matrix *C*.

k INTEGER. The order of the matrix *T* (equal to the number of elementary reflectors whose product defines the block reflector).

l INTEGER. The number of columns of the matrix *V* containing the meaningful part of the Householder reflectors.
If *side* = 'L', *m* ≥ l ≥ 0, if *side* = 'R', *n* ≥ l ≥ 0.

v REAL for slarzb
DOUBLE PRECISION for dlarzb
COMPLEX for clarzb
COMPLEX*16 for zlarzb
Array, DIMENSION (*ldv*, nv).
If *storev* = 'C', nv = *k*;
if *storev* = 'R', nv = *l*.

ldv INTEGER. The leading dimension of the array *v*.
If *storev* = 'C', *ldv* ≥ *l*; if *storev* = 'R', *ldv* ≥ *k*.

t	REAL for slarzb DOUBLE PRECISION for dlarzb COMPLEX for clarzb COMPLEX*16 for zlarzb Array, DIMENSION (*ldt*,*k*). The triangular *k*-by-*k* matrix *T* in the representation of the block reflector.
ldt	INTEGER. The leading dimension of the array *t*. *ldt* ≥ *k*.
c	REAL for slarzb DOUBLE PRECISION for dlarzb COMPLEX for clarzb COMPLEX*16 for zlarzb Array, DIMENSION (*ldc*,*n*). On entry, the *m*-by-*n* matrix *C*.
ldc	INTEGER. The leading dimension of the array *c*. *ldc* ≥ max(1,*m*).
work	REAL for slarzb DOUBLE PRECISION for dlarzb COMPLEX for clarzb COMPLEX*16 for zlarzb Workspace array, DIMENSION (*ldwork*, *k*).
ldwork	INTEGER. The leading dimension of the array *work*. If *side* = 'L', *ldwork* ≥ max(1, *n*); if *side* = 'R', *ldwork* ≥ max(1, *m*).

Output Parameters

c	On exit, *C* is overwritten by *H***C*, or *H*'**C*, or *C***H*, or *C***H*'.

?larzt

*Forms the triangular factor T of a block reflector H = I - V*T*V^H.*

Syntax

```
call slarzt( direct, storev, n, k, v, ldv, tau, t, ldt )
call dlarzt( direct, storev, n, k, v, ldv, tau, t, ldt )
call clarzt( direct, storev, n, k, v, ldv, tau, t, ldt )
call zlarzt( direct, storev, n, k, v, ldv, tau, t, ldt )
```

Description

The routine forms the triangular factor *T* of a real/complex block reflector *H* of order > *n*, which is defined as a product of *k* elementary reflectors.

If *direct* = 'F', *H* = *H*(1) *H*(2) . . . *H*(*k*) and *T* is upper triangular.

If *direct* = 'B', *H* = *H*(*k*) . . . *H*(2) *H*(1) and *T* is lower triangular.

If *storev* = 'C', the vector which defines the elementary reflector *H*(*i*) is stored in the *i*-th column of the array *v*, and *H* = *I* - *V***T***V*'

If *storev* = 'R', the vector which defines the elementary reflector *H*(*i*) is stored in the *i*-th row of the array *v*, and *H* = *I* - *V*'**T***V*

Currently, only *storev* = 'R' and *direct* = 'B' are supported.

Input Parameters

direct — CHARACTER*1.
Specifies the order in which the elementary reflectors are multiplied to form the block reflector:
If *direct* = 'F': *H* = *H*(1) *H*(2) . . . *H*(*k*) (forward, not supported)
If *direct* = 'B': *H* = *H*(*k*) . . . *H*(2) *H*(1) (backward)

storev — CHARACTER*1.
Specifies how the vectors which define the elementary reflectors are stored (see also Application Notes below):
If *storev* = 'C': column-wise (not supported)
If *storev* = 'R': row-wise

n	INTEGER. The order of the block reflector *H*. *n* ≥ 0.
k	INTEGER. The order of the triangular factor *T* (equal to the number of elementary reflectors). *k* ≥ 1.
v	REAL for slarzt DOUBLE PRECISION for dlarzt COMPLEX for clarzt COMPLEX*16 for zlarzt Array, DIMENSION (*ldv*, *k*) if *storev* = 'C' (*ldv*, *n*) if *storev* = 'R' The matrix *V*.
ldv	INTEGER. The leading dimension of the array *v*. If *storev* = 'C', *ldv* ≥ max(1,*n*); if *storev* = 'R', *ldv* ≥ *k*.
tau	REAL for slarzt DOUBLE PRECISION for dlarzt COMPLEX for clarzt COMPLEX*16 for zlarzt Array, DIMENSION (*k*). *tau*(*i*) must contain the scalar factor of the elementary reflector *H*(*i*).
ldt	INTEGER. The leading dimension of the output array *t*. *ldt* ≥ *k*.

Output Parameters

t	REAL for slarzt DOUBLE PRECISION for dlarzt COMPLEX for clarzt COMPLEX*16 for zlarzt Array, DIMENSION (*ldt*,*k*). The *k*-by-*k* triangular factor *T* of the block reflector. If *direct* = 'F', *T* is upper triangular; if *direct* = 'B', *T* is lower triangular. The rest of the array is not used.
v	The matrix *V*. See Application Notes below.

Application Notes

The shape of the matrix V and the storage of the vectors which define the $H(i)$ is best illustrated by the following example with $n = 5$ and $k = 3$. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

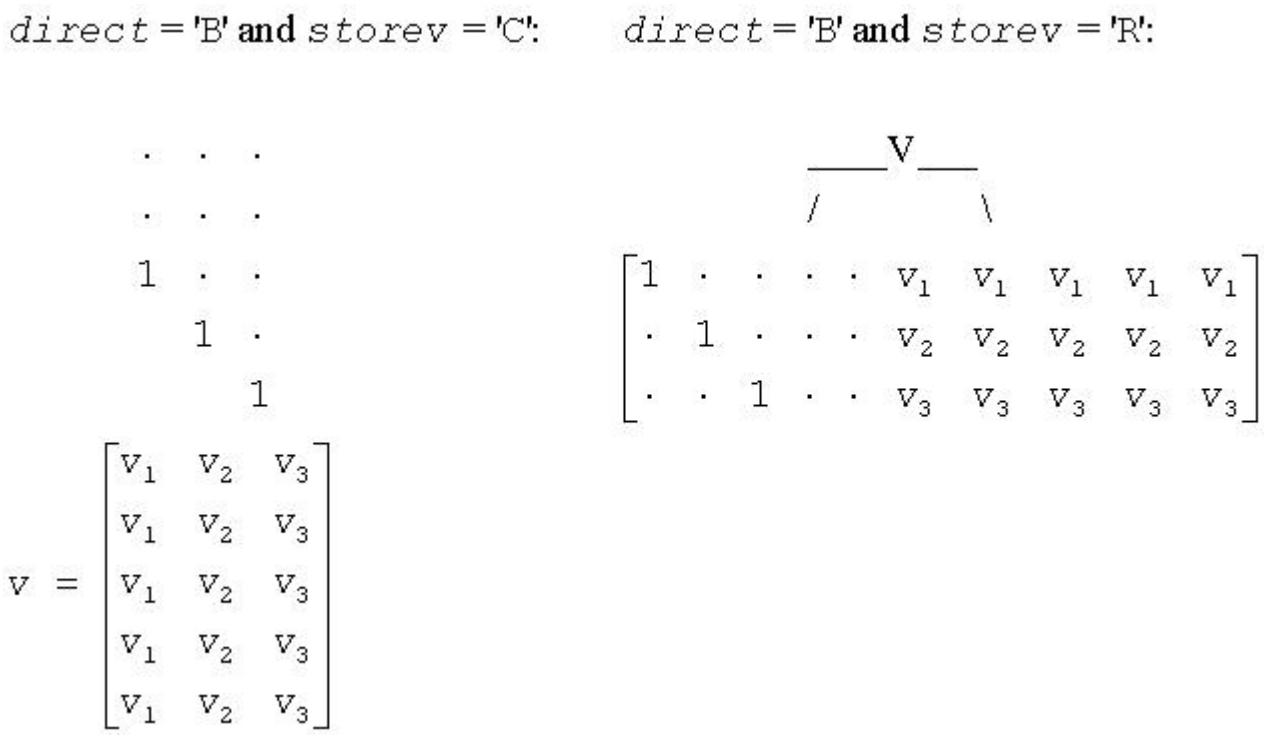
direct = 'F' and *storev* = 'C':

$$V = \begin{bmatrix} v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \end{bmatrix}$$

$$\begin{matrix} . & . & . \\ . & . & . \\ 1 & . & . \\ & 1 & . \\ & & 1 \end{matrix}$$

direct = 'F' and *storev* = 'R':

$$\overbrace{\begin{bmatrix} v_1 & v_1 & v_1 & v_1 & v_1 & \cdot & \cdot & \cdot & \cdot & 1 \\ v_2 & v_2 & v_2 & v_2 & v_2 & \cdot & \cdot & \cdot & 1 & \\ v_3 & v_3 & v_3 & v_3 & v_3 & \cdot & \cdot & 1 & & \end{bmatrix}}^{V}$$



?las2

Computes singular values of a 2-by-2 triangular matrix.

Syntax

```
call slas2( f, g, h, ssmin, ssmax )
call dlas2( f, g, h, ssmin, ssmax )
```

Description

The routine `?las2` computes the singular values of the 2-by-2 matrix

$$\begin{bmatrix} f & g \\ 0 & h \end{bmatrix}$$

On return, *ssmin* is the smaller singular value and *SSMAX* is the larger singular value.

Input Parameters

f, *g*, *h*	REAL for slas2 DOUBLE PRECISION for dlas2 The (1,1), (1,2) and (2,2) elements of the 2-by-2 matrix, respectively.

Output Parameters

ssmin, *ssmax*	REAL for slas2 DOUBLE PRECISION for dlas2 The smaller and the larger singular values, respectively.

Application Notes

Barring over/underflow, all output quantities are correct to within a few units in the last place (*ulps*), even in the absence of a guard digit in addition/subtraction. In ieee arithmetic, the code works correctly if one matrix element is infinite. Overflow will not occur unless the largest singular value itself overflows, or is within a few ulps of overflow. (On machines with partial overflow, like the Cray, overflow may occur if the largest singular value is within a factor of 2 of overflow.) Underflow is harmless if underflow is gradual. Otherwise, results may correspond to a matrix modified by perturbations of size near the underflow threshold.

?lascl

Multiplies a general rectangular matrix by a real scalar defined as cto/cfrom.

Syntax

```
call slascl( type, kl, ku, cfrom, cto, m, n, a, lda, info )
call dlascl( type, kl, ku, cfrom, cto, m, n, a, lda, info )
call clascl( type, kl, ku, cfrom, cto, m, n, a, lda, info )
call zlascl( type, kl, ku, cfrom, cto, m, n, a, lda, info )
```

Description

The routine ?lascl multiplies the *m*-by-*n* real/complex matrix *A* by the real scalar *cto*/*cfrom*. The operation is performed without over/underflow as long as the final result *cto***A*(*i*,*j*)/*cfrom* does not over/underflow.

type specifies that *A* may be full, upper triangular, lower triangular, upper Hessenberg, or banded.

Input Parameters

type — CHARACTER*1. *type* indices the storage *type* of the input matrix.
= 'G': *A* is a full matrix.
= 'L': *A* is a lower triangular matrix.
= 'U': *A* is an upper triangular matrix.
= 'H': *A* is an upper Hessenberg matrix.
= 'B': *A* is a symmetric band matrix with lower bandwidth *kl* and upper bandwidth *ku* and with the only the lower half stored
= 'Q': *A* is a symmetric band matrix with lower bandwidth *kl* and upper bandwidth *ku* and with the only the upper half stored.
= 'Z': *A* is a band matrix with lower bandwidth *kl* and upper bandwidth *ku*.

kl — INTEGER. The lower bandwidth of *A*. Referenced only if *type* = 'B', 'Q' or 'Z'.

ku	INTEGER. The upper bandwidth of *A*. Referenced only if *type* = 'B', 'Q' or 'Z'.
cfrom, *cto*	REAL for slascl/clascl DOUBLE PRECISION for dlascl/zlascl The matrix *A* is multiplied by *cto*/*cfrom*. *A*(*i*,*j*) is computed without over/underflow if the final result *cto***A*(*i*,*j*)/*cfrom* can be represented without over/underflow. *cfrom* must be nonzero.
m	INTEGER. The number of rows of the matrix *A*. *m* ≥ 0.
n	INTEGER. The number of columns of the matrix *A*. *n* ≥ 0.
a	REAL for slascl DOUBLE PRECISION for dlascl COMPLEX for clascl COMPLEX*16 for zlascl Array, DIMENSION (*lda*, *n*). The matrix to be multiplied by *cto*/*cfrom*. See *type* for the storage type.
lda	INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1,*m*).

Output Parameters

a	The multiplied matrix *A*.
info	INTEGER. If *info* = 0 - successful exit If *info* = -*i* < 0, the *i*-th argument had an illegal value.

?lasd0

Computes the singular values of a real upper bidiagonal n-by-m matrix B with diagonal d and off-diagonal e. Used by ?bdsdc.

Syntax

```
call slasd0( n, sqre, d, e, u, ldu, vt, ldvt, smlsiz, iwork, work, info )
call dlasd0( n, sqre, d, e, u, ldu, vt, ldvt, smlsiz, iwork, work, info )
```

Description

Using a divide and conquer approach, the routine ?lasd0 computes the singular value decomposition (SVD) of a real upper bidiagonal *n*-by-*m* matrix *B* with diagonal *d* and offdiagonal *e*, where *m* = *n* + *sqre*.

The algorithm computes orthogonal matrices *U* and *VT* such that *B* = *U*S*VT*. The singular values *S* are overwritten on *d*.

The related subroutine ?lasda computes only the singular values, and optionally, the singular vectors in compact form.

Input Parameters

n	INTEGER. On entry, the row dimension of the upper bidiagonal matrix. This is also the dimension of the main diagonal array *d*.
sqre	INTEGER. Specifies the column dimension of the bidiagonal matrix. If *sqre* = 0: the bidiagonal matrix has column dimension *m* = *n*. If *sqre* = 1: the bidiagonal matrix has column dimension *m* = *n*+1.
d	REAL for slasd0 DOUBLE PRECISION for dlasd0 Array, DIMENSION (*n*). On entry, *d* contains the main diagonal of the bidiagonal matrix.
e	REAL for slasd0 DOUBLE PRECISION for dlasd0 Array, DIMENSION (*m*-1). Contains the subdiagonal entries of the bidiagonal matrix. On exit, *e* is destroyed.
ldu	INTEGER. On entry, leading dimension of the output array *u*.
ldvt	INTEGER. On entry, leading dimension of the output array *vt*.
smlsiz	INTEGER. On entry, maximum size of the subproblems at the bottom of the computation tree.
iwork	INTEGER. Workspace array, dimension must be at least (8 * *n*).

work REAL for slasd0
DOUBLE PRECISION for dlasd0
Workspace array, dimension must be at least (3 * m^2 + 2 *m).

Output Parameters

d On exit *d*, If *info* = 0, contains singular values of the bidiagonal matrix.

u REAL for slasd0
DOUBLE PRECISION for dlasd0
Array, DIMENSION at least (*ldq*, *n*). On exit, *u* contains the left singular vectors.

vt REAL for slasd0
DOUBLE PRECISION for dlasd0
Array, DIMENSION at least (*ldvt*, *m*). On exit, *vt*' contains the right singular vectors.

info INTEGER.
If *info* = 0: successful exit.
If *info* = -*i* < 0, the *i*-th argument had an illegal value.
If *info* = 1, an singular value did not converge.

?lasd1

Computes the SVD of an upper bidiagonal matrix B of the specified size. Used by ?bdsdc.

Syntax

```
call slasd1( nl, nr, sqre, d, alpha, beta, u, ldu, vt, ldvt, idxq, iwork, work, info )
call dlasd1( nl, nr, sqre, d, alpha, beta, u, ldu, vt, ldvt, idxq, iwork, work, info )
```

Description

This routine computes the SVD of an upper bidiagonal *n*-by-*m* matrix *B*, where *n* = *nl* + *nr* + 1 and *m* = *n* + *sqre*.

The routine ?lasd1 is called from ?lasd0.

A related subroutine ?lasd7 handles the case in which the singular values (and the singular vectors in factored form) are desired.

?lasd1 computes the SVD as follows:

$$VT = U(in)^* \begin{bmatrix} D1(in) & 0 & 0 & 0 \\ Z1' & a & Z2' & b \\ 0 & 0 & D2(in) & 0 \end{bmatrix} ^*\mathrm{VT}(in)$$

= U(out)*(D(out) 0)*VT(out)

where Z' = (Z1' a Z2' b) = u' VT ', and u is a vector of dimension m with alpha and beta in the nl+1 and nl+2-th entries and zeros elsewhere; and the entry b is empty if sqre = 0.

The left singular vectors of the original matrix are stored in u, and the transpose of the right singular vectors are stored in vt, and the singular values are in d. The algorithm consists of three stages:

1. The first stage consists of deflating the size of the problem when there are multiple singular values or when there are zeros in the z vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine ?lasd2.
2. The second stage consists of calculating the updated singular values. This is done by finding the square roots of the roots of the secular equation via the routine ?lasd4 (as called by ?lasd3). This routine also calculates the singular vectors of the current problem.
3. The final stage consists of computing the updated singular vectors directly using the updated singular values. The singular vectors for the current problem are multiplied with the singular vectors from the overall problem.

Input Parameters

nl	INTEGER. The row dimension of the upper block. nl ≥ 1.
nr	INTEGER. The row dimension of the lower block. nr ≥ 1.

sqre INTEGER.
If *sqre* = 0: the lower block is an *nr*-by-*nr* square matrix.
If *sqre* = 1: the lower block is an *nr*-by-(*nr*+1) rectangular matrix. The bidiagonal matrix has row dimension *n* = *nl* + *nr* + 1, and column dimension *m* = *n* + *sqre*.

d REAL for slasd1
DOUBLE PRECISION for dlasd1
Array, DIMENSION (*nl*+*nr*+1). *n* = *nl*+*nr*+1. On entry *d*(1:*nl*,1:*nl*) contains the singular values of the upper block; and *d*(*nl*+2:*n*) contains the singular values of the lower block.

alpha REAL for slasd1
DOUBLE PRECISION for dlasd1
Contains the diagonal element associated with the added row.

beta REAL for slasd1
DOUBLE PRECISION for dlasd1
Contains the off-diagonal element associated with the added row.

u REAL for slasd1
DOUBLE PRECISION for dlasd1
Array, DIMENSION (*ldu*, *n*). On entry *u*(1:*nl*, 1:*nl*) contains the left singular vectors of the upper block; *u*(*nl*+2:*n*, *nl*+2:*n*) contains the left singular vectors of the lower block.

ldu INTEGER. The leading dimension of the array *U*.
ldu ≥ max(1, *n*).

vt REAL for slasd1
DOUBLE PRECISION for dlasd1
Array, DIMENSION (*ldvt*, *m*), where *m* = *n* + *sqre*.
On entry *vt*(1:*nl*+1, 1:*nl*+1)' contains the right singular vectors of the upper block; *vt*(*nl*+2:*m*, *nl*+2:*m*)' contains the right singular vectors of the lower block.

ldvt INTEGER. The leading dimension of the array *vt*.
ldvt ≥ max(1, *M*).

iwork INTEGER.

Workspace array, DIMENSION ($4n$).

work
REAL for slasd1
DOUBLE PRECISION for dlasd1
Workspace array, DIMENSION ($3m_2 + 2m$).

Output Parameters

d
On exit $d(1:n)$ contains the singular values of the modified matrix.

alpha
On exit, the diagonal element associated with the added row deflated by max(abs(*alpha*), abs(*beta*), abs(D(I))), I = 1,n.

beta
On exit, the off-diagonal element associated with the added row deflated by max(abs(*alpha*), abs(*beta*), abs(D(I))), I = 1,n.

u
On exit u contains the left singular vectors of the bidiagonal matrix.

vt
On exit vt' contains the right singular vectors of the bidiagonal matrix.

idxq
INTEGER
Array, DIMENSION (n). Contains the permutation which will reintegrate the subproblem just solved back into sorted order, that is, $d(idxq(i = 1, n))$ will be in ascending order.

info
INTEGER.
If $info = 0$: successful exit.
If $info = -i < 0$, the i-th argument had an illegal value.
If $info = 1$, an singular value did not converge.

?lasd2

Merges the two sets of singular values together into a single sorted set. Used by ?bdsdc.

Syntax

call slasd2(*nl*, *nr*, *sqre*, *k*, *d*, *z*, *alpha*, *beta*, *u*, *ldu*, *vt*, *ldvt*, *dsigma*, *u2*, *ldu2*, *vt2*, *ldvt2*, *idxp*, *idx*, *idxp*, *idxq*, *coltyp*, *info*)

call dlasd2(*nl*, *nr*, *sqre*, *k*, *d*, *z*, *alpha*, *beta*, *u*, *ldu*, *vt*, *ldvt*, *dsigma*, *u2*, *ldu2*, *vt2*, *ldvt2*, *idxp*, *idx*, *idxp*, *idxq*, *coltyp*, *info*)

Description

The routine ?lasd2 merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more singular values are close together or if there is a tiny entry in the *z* vector. For each such occurrence the order of the related secular equation problem is reduced by one.

The routine ?lasd2 is called from ?lasd1.

Input Parameters

nl — INTEGER. The row dimension of the upper block.
nl ≥ 1.

nr — INTEGER. The row dimension of the lower block.
nr ≥ 1.

sqre — INTEGER.
If *sqre* = 0): the lower block is an *nr*-by-*nr* square matrix
If *sqre* = 1): the lower block is an *nr*-by-(*nr*+1) rectangular matrix. The bidiagonal matrix has *n* = *nl* + *nr* + 1 rows and *m* = *n* + *sqre* ≥ *n* columns.

d — REAL for slasd2
DOUBLE PRECISION for dlasd2
Array, DIMENSION (*n*). On entry *d* contains the singular values of the two submatrices to be combined.

alpha — REAL for slasd2
DOUBLE PRECISION for dlasd2

	Contains the diagonal element associated with the added row.
beta	REAL for slasd2 DOUBLE PRECISION for dlasd2 Contains the off-diagonal element associated with the added row.
u	REAL for slasd2 DOUBLE PRECISION for dlasd2 Array, DIMENSION (*ldu*, *n*). On entry *u* contains the left singular vectors of two submatrices in the two square blocks with corners at (1,1), (*nl*, *nl*), and (*nl*+2, *nl*+2), (*n*,*n*).
ldu	INTEGER. The leading dimension of the array *u*. *ldu* ≥ *n*.
ldu2	INTEGER. The leading dimension of the output array *u2*. *ldu2* ≥ *n*.
vt	REAL for slasd2 DOUBLE PRECISION for dlasd2 Array, DIMENSION (*ldvt*, *m*). On entry, *vt*' contains the right singular vectors of two submatrices in the two square blocks with corners at (1,1), (*nl*+1, *nl*+1), and (*nl*+2, *nl*+2), (*m*, *m*).
ldvt	INTEGER. The leading dimension of the array *vt*. *ldvt* ≥ *m*.
ldvt2	INTEGER. The leading dimension of the output array *vt2*. *ldvt2* ≥ *m*.
idxp	INTEGER. Workspace array, DIMENSION (*n*). This will contain the permutation used to place deflated values of *D* at the end of the array. On output *idxp*(2:*k*) points to the nondeflated *d*-values and *idxp*(*k*+1:*n*) points to the deflated singular values.
idx	INTEGER. Workspace array, DIMENSION (*n*). This will contain the permutation used to sort the contents of *d* into ascending order.

coltyp INTEGER.
Workspace array, DIMENSION (*n*). As workspace, this array contains a label that indicates which of the following types a column in the *u2* matrix or a row in the *vt2* matrix is:
1 : non-zero in the upper half only
2 : non-zero in the lower half only
3 : dense
4 : deflated.

idxq INTEGER. Array, DIMENSION (*n*). This parameter contains the permutation that separately sorts the two sub-problems in *D* into ascending order. Note that entries in the first half of this permutation must first be moved one position backward; and entries in the second half must first have *nl*+1 added to their values.

Output Parameters

k INTEGER. Contains the dimension of the non-deflated matrix,
This is the order of the related secular equation. $1 \leq k \leq n$.

d On exit *D* contains the trailing (*n*-*k*) updated singular values (those which were deflated) sorted into increasing order.

u On exit *u* contains the trailing (*n*-*k*) updated left singular vectors (those which were deflated) in its last *n*-*k* columns.

z REAL for slasd2
DOUBLE PRECISION for dlasd2
Array, DIMENSION (*n*). On exit, *z* contains the updating row vector in the secular equation.

dsigma REAL for slasd2
DOUBLE PRECISION for dlasd2
Array, DIMENSION (*n*). Contains a copy of the diagonal elements (*k*–1 singular values and one zero) in the secular equation.

u2 REAL for slasd2
DOUBLE PRECISION for dlasd2

	Array, DIMENSION (*ldu2*, *n*). Contains a copy of the first *k*–1 left singular vectors which will be used by ?lasd3 in a matrix multiply (?gemm) to solve for the new left singular vectors. *u2* is arranged into four blocks. The first block contains a column with 1 at *nl*+1 and zero everywhere else; the second block contains non-zero entries only at and above *nl*; the third contains non-zero entries only below *nl*+1; and the fourth is dense.
vt	On exit, *vt*' contains the trailing (*n*-*k*) updated right singular vectors (those which were deflated) in its last *n*-*k* columns. In case *sqre* =1, the last row of *vt* spans the right null space.
vt2	REAL for slasd2 DOUBLE PRECISION for dlasd2 Array, DIMENSION (*ldvt2*, *n*). *vt2*' contains a copy of the first *k* right singular vectors which will be used by ?lasd3 in a matrix multiply (?gemm) to solve for the new right singular vectors. *vt2* is arranged into three blocks. The first block contains a row that corresponds to the special 0 diagonal element in *sigma*; the second block contains non-zeros only at and before *nl* +1; the third block contains non-zeros only at and after *nl* +2.
idxc	INTEGER. Array, DIMENSION (*n*). This will contain the permutation used to arrange the columns of the deflated *u* matrix into three groups: the first group contains non-zero entries only at and above *nl*, the second contains non-zero entries only below *nl*+2, and the third is dense.
coltyp	On exit, it is an array of dimension 4, with *coltyp*(*i*) being the dimension of the *i*-th type columns.
info	INTEGER. If *info* = 0): successful exit If *info* = –*i* < 0, the *i*-th argument had an illegal value.

?lasd3

Finds all square roots of the roots of the secular equation, as defined by the values in D and Z, and then updates the singular vectors by matrix multiplication. Used by ?bdsdc.

Syntax

```
call slasd3( nl, nr, sqre, k, d, q, ldq, dsigma, u, ldu, u2, ldu2, vt, ldvt,
vt2, ldvt2, idxc, ctot, z, info )
```

```
call dlasd3( nl, nr, sqre, k, d, q, ldq, dsigma, u, ldu, u2, ldu2, vt, ldvt,
vt2, ldvt2, idxc, ctot, z, info )
```

Description

The routine ?lasd3 finds all the square roots of the roots of the secular equation, as defined by the values in *D* and *Z*.

It makes the appropriate calls to ?lasd4 and then updates the singular vectors by matrix multiplication.

The routine ?lasd3 is called from ?lasd1.

Input Parameters

nl — INTEGER. The row dimension of the upper block.
nl ≥ 1.

nr — INTEGER. The row dimension of the lower block.
nr ≥ 1.

sqre — INTEGER.
If *sqre* = 0): the lower block is an *nr*-by-*nr* square matrix.
If *sqre* = 1): the lower block is an *nr*-by-(*nr*+1) rectangular matrix. The bidiagonal matrix has *n* = *nl* + *nr* + 1 rows and *m* = *n* + *sqre* ≥ *n* columns.

k — INTEGER.The size of the secular equation, 1 ≤ *k* ≤ *n*.

q — REAL for slasd3
DOUBLE PRECISION for dlasd3
Workspace array, DIMENSION at least (*ldq*, *k*).

ldq	INTEGER. The leading dimension of the array *Q*. *ldq* ≥ *k*.
dsigma	REAL for slasd3 DOUBLE PRECISION for dlasd3 Array, DIMENSION (*k*). The first *k* elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.
ldu	INTEGER. The leading dimension of the array *u*. *ldu* ≥ *n*.
u2	REAL for slasd3 DOUBLE PRECISION for dlasd3 Array, DIMENSION (*ldu2*, *n*). The first *k* columns of this matrix contain the non-deflated left singular vectors for the split problem.
ldu2	INTEGER. The leading dimension of the array *u2*. *ldu2* ≥ *n*.
ldvt	INTEGER. The leading dimension of the array *vt*. *ldvt* ≥ *n*.
vt2	REAL for slasd3 DOUBLE PRECISION for dlasd3 Array, DIMENSION (*ldvt2*, *n*). The first *k* columns of *vt2*' contain the non-deflated right singular vectors for the split problem.
ldvt2	INTEGER. The leading dimension of the array *vt2*. *ldvt2* ≥ *n*.
idxc	INTEGER. Array, DIMENSION (*n*). The permutation used to arrange the columns of u (and rows of *vt*) into three groups: the first group contains non-zero entries only at and above (or before) *nl* +1; the second contains non-zero entries only at and below (or after) *nl*+2; and the third is dense. The first column of *u* and the row of *vt* are treated separately, however. The rows of the singular vectors found by ?lasd4 must be likewise permuted before the matrix multiplies can take place.

ctot INTEGER. Array, DIMENSION (4). A count of the total number of the various types of columns in *u* (or rows in *vt*), as described in *idxc*.
The fourth column type is any column which has been deflated.

z REAL for slasd3
DOUBLE PRECISION for dlasd3
Array, DIMENSION (*k*). The first *k* elements of this array contain the components of the deflation-adjusted updating row vector.

Output Parameters

d REAL for slasd3
DOUBLE PRECISION for dlasd3
Array, DIMENSION (*k*). On exit the square roots of the roots of the secular equation, in ascending order.

u REAL for slasd3
DOUBLE PRECISION for dlasd3
Array, DIMENSION (*ldu*, *n*).
The last *n* - *k* columns of this matrix contain the deflated left singular vectors.

vt REAL for slasd3
DOUBLE PRECISION for dlasd3
Array, DIMENSION (*ldvt*, *m*).
The last *m* – *k* columns of *vt*' contain the deflated right singular vectors.

vt2 Destroyed on exit.

z Destroyed on exit.

info INTEGER.
If *info* = 0): successful exit.
If *info* = –*i* < 0, the *i*-th argument had an illegal value.
If *info* = 1, an singular value did not converge.

Application Notes

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray XMP, Cray YMP, Cray C 90, or Cray 2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

?lasd4

Computes the square root of the `i`*-th updated eigenvalue of a positive symmetric rank-one modification to a positive diagonal matrix. Used by* `?bdsdc`.

Syntax

```
call slasd4( n, i, d, z, delta, rho, sigma, work, info )
call dlasd4( n, i, d, z, delta, rho, sigma, work, info )
```

Description

This routine computes the square root of the `i`-th updated eigenvalue of a positive symmetric rank-one modification to a positive diagonal matrix whose entries are given as the squares of the corresponding entries in the array `d`, and that `0 ≤ d(i) < d(j)` for `i < j` and that `rho > 0`. This is arranged by the calling routine, and is no loss in generality. The rank-one modified system is thus

`diag( d )* diag( d ) + rho*Z*Z_transpose,`

where we assume the Euclidean norm of `Z` is 1.The method consists of approximating the rational functions in the secular equation by simpler interpolating rational functions.

Input Parameters

`n`	`INTEGER`. The length of all arrays.
`i`	`INTEGER`. The index of the eigenvalue to be computed. `1 ≤ i ≤ n`.

`d`	`REAL` for `slasd4`
	`DOUBLE PRECISION` for `dlasd4`
	Array, `DIMENSION` (`n`).

The original eigenvalues. It is assumed that they are in order, $0 \le d(i) < d(j)$ for $i < j$.

z
REAL for slasd4
DOUBLE PRECISION for dlasd4
Array, DIMENSION (n).
The components of the updating vector.

rho
REAL for slasd4
DOUBLE PRECISION for dlasd4
The scalar in the symmetric updating formula.

work
REAL for slasd4
DOUBLE PRECISION for dlasd4
Workspace array, DIMENSION (n).
If $n \ne 1$, *work* contains ($d(j)$ + *sigma_i*) in its j-th component.
If $n = 1$, then *work*(1) = 1.

Output Parameters

delta
REAL for slasd4
DOUBLE PRECISION for dlasd4
Array, DIMENSION (n).
If $n \ne 1$, *delta* contains ($d(j)$ – *sigma_i*) in its j-th component.
If $n = 1$, then *delta* (1) = 1. The vector *delta* contains the information necessary to construct the (singular) eigenvectors.

sigma
REAL for slasd4
DOUBLE PRECISION for dlasd4
The computed *sigma_i*, the i-th updated eigenvalue.

info
INTEGER.
= 0: successful exit
> 0: If *info* = 1, the updating process failed.

?lasd5

Computes the square root of the i-th eigenvalue of a positive symmetric rank-one modification of a 2-by-2 diagonal matrix.Used by ?bdsdc.

Syntax

```
call slasd5( i, d, z, delta, rho, dsigma, work )
call dlasd5( i, d, z, delta, rho, dsigma, work )
```

Description

This routine computes the square root of the *i*-th eigenvalue of a positive symmetric rank-one modification of a 2-by-2 diagonal matrix `diag( d )* diag( d ) + rho*Z*Z_transpose`

The diagonal entries in the array *d* are assumed to satisfy `0 ≤ d(i) < d(j)` for `i < j` .We also assume `rho > 0` and that the Euclidean norm of the vector *Z* is one.

Input Parameters

i	*INTEGER*.The index of the eigenvalue to be computed. `i = 1` or `i = 2`.

d	`REAL` for `slasd5`
	`DOUBLE PRECISION` for `dlasd5`
	Array, `dimension` (2).
	The original eigenvalues. We assume `0 ≤ d(1) < d(2)`.

z	`REAL` for `slasd5`
	`DOUBLE PRECISION` for `dlasd5`
	Array, `dimension` (2).
	The components of the updating vector.

rho	`REAL` for `slasd5`
	`DOUBLE PRECISION` for `dlasd5`
	The scalar in the symmetric updating formula.

work	`REAL` for `slasd5` `DOUBLE PRECISION` for `dlasd5`. Workspace array, `dimension` (2). Contains (`d(j) + sigma_i`) in its *j*-th component.

Output Parameters

delta	REAL for slasd5 DOUBLE PRECISION for dlasd5. Array, dimension (2). Contains (*d(j) - sigma_i*) in its *j*-th component. The vector *delta* contains the information necessary to construct the eigenvectors.
dsigma	REAL for slasd5 DOUBLE PRECISION for dlasd5. The computed *sigma_i* , the *i*-th updated eigenvalue.

?lasd6

Computes the SVD of an updated upper bidiagonal matrix obtained by merging two smaller ones by appending a row. Used by ?bdsdc.

Syntax

```
call slasd6( icompq, nl, nr, sqre, d, vf, vl, alpha, beta, idxq, perm, givptr,
givcol, ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, work, iwork,
info )
```

```
call dlasd6( icompq, nl, nr, sqre, d, vf, vl, alpha, beta, idxq, perm, givptr,
givcol, ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, work, iwork,
info )
```

Description

The routine ?lasd6 computes the *SVD* of an updated upper bidiagonal matrix *B* obtained by merging two smaller ones by appending a row. This routine is used only for the problem which requires all singular values and optionally singular vector matrices in factored form. *B* is an *n*-by-*m* matrix with *n = nl + nr + 1* and *m = n + sqre*. A related subroutine, ?lasd1, handles the case in which all singular values and singular vectors of the bidiagonal matrix are desired. ?lasd6 computes the *SVD* as follows:

$$B = U(in)* \begin{bmatrix} D1(in) & 0 & 0 & 0 \\ Z1' & a & Z2' & b \\ 0 & 0 & D2(in) & 0 \end{bmatrix} *\mathrm{VT}(in)$$

= *U(out*(D(out)*VT(out)*

where *Z*' = (*Z1' a Z2' b*) = *u' VT'*, and *u* is a vector of dimension *m* with *alpha* and *beta* in the *nl*+1 and *nl*+2-th entries and zeros elsewhere; and the entry b is empty if *sqre* = 0.

The singular values of *B* can be computed using *D1*, *D2*, the first components of all the right singular vectors of the lower block, and the last components of all the right singular vectors of the upper block. These components are stored and updated in *vf* and *vl*, respectively, in ?lasd6. Hence *U* and *VT* are not explicitly referenced.

The singular values are stored in *D*. The algorithm consists of two stages:

1. The first stage consists of deflating the size of the problem when there are multiple singular values or if there is a zero in the *Z* vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine ?lasd7.
2. The second stage consists of calculating the updated singular values. This is done by finding the roots of the secular equation via the routine ?lasd4 (as called by ?lasd8). This routine also updates *vf* and *vl* and computes the distances between the updated singular values and the old singular values. ?lasd6 is called from ?lasda.

Input Parameters

icompq	INTEGER. Specifies whether singular vectors are to be computed in factored form: = 0: Compute singular values only = 1: Compute singular vectors in factored form as well.
nl	INTEGER. The row dimension of the upper block. *nl* ≥ 1.
nr	INTEGER. The row dimension of the lower block. *nr* ≥ 1.
sqre	INTEGER . = 0: the lower block is an *nr*-by-*nr* square matrix.

= 1: the lower block is an *nr*-by-(*nr*+1) rectangular matrix. The bidiagonal matrix has row dimension *n*=*nl*+*nr*+1, and column dimension *m* = *n* + *sqre*.

d
REAL for slasd6
DOUBLE PRECISION for dlasd6
Array, dimension (*nl*+*nr*+1). On entry *d*(1:*nl*,1:*nl*) contains the singular values of the upper block, and *d*(*nl*+2:*n*) contains the singular values of the lower block.

vf
REAL for slasd6
DOUBLE PRECISION for dlasd6
Array, dimension (*m*).
On entry, *vf*(1:*nl*+1) contains the first components of all right singular vectors of the upper block; and *vf*(*nl*+2:*m*) contains the first components of all right singular vectors of the lower block.

vl
REAL for slasd6
DOUBLE PRECISION for dlasd6
Array, dimension (*m*).
On entry, *vl*(1:*nl*+1) contains the last components of all right singular vectors of the upper block; and *vl*(*nl*+2:*m*) contains the last components of all right singular vectors of the lower block.

alpha
REAL for slasd6
DOUBLE PRECISION for dlasd6
Contains the diagonal element associated with the added row.

beta
REAL for slasd6
DOUBLE PRECISION for dlasd6
Contains the off-diagonal element associated with the added row.

ldgcol
INTEGER.The leading dimension *of* the output array *givcol*, must be at least *n*.

ldgnum
INTEGER.
The leading dimension of the output arrays *givnum* and *poles*, must be at least *n*.

work
REAL for slasd6
DOUBLE PRECISION for dlasd6

Workspace array, dimension (4*m*).

iwork

INTEGER.

Workspace array, dimension (3*n*).

Output Parameters

d

On exit *d*(1:*n*) contains the singular values of the modified matrix.

vf

On exit, *vf* contains the first components of all right singular vectors of the bidiagonal matrix.

vl

On exit, *vl* contains the last components of all right singular vectors of the bidiagonal matrix.

alpha

On exit, the diagonal element associated with the added row deflated by max(abs(*alpha*), abs(*beta*), abs(D(I))), I = 1,n.

beta

On exit, the off-diagonal element associated with the added row deflated by max(abs(*alpha*), abs(*beta*), abs(D(I))), I = 1,n.

idxq

INTEGER.
Array, dimension (*n*). This contains the permutation which will reintegrate the subproblem just solved back into sorted order, that is, *d*(*idxq*(*i* = 1, *n*)) will be in ascending order.

perm

INTEGER.

Array, dimension (*n*). The permutations (from deflation and sorting) to be applied to each block. Not referenced if *icompq* = 0.

givptr

INTEGER. The number of Givens rotations which took place in this subproblem. Not referenced if *icompq* = 0.

givcol

INTEGER.

Array, dimension (*ldgcol*, 2). Each pair of numbers indicates a pair of columns to take place in a Givens rotation. Not referenced if *icompq* = 0.

givnum

REAL for slasd6
DOUBLE PRECISION for dlasd6

Array, dimension (*ldgnum*, 2). Each number indicates the *C* or *S* value to be used in the corresponding Givens rotation. Not referenced if *icompq* = 0.

poles
REAL for slasd6
DOUBLE PRECISION for dlasd6
Array, dimension (*ldgnum*, 2). On exit, *poles*(1,*) is an array containing the new singular values obtained from solving the secular equation, and *poles*(2,*) is an array containing the poles in the secular equation. Not referenced if *icompq* = 0.

difl
REAL for slasd6
DOUBLE PRECISION for dlasd6
Array, dimension (*n*). On exit, *difl*(*i*) is the distance between *i*-th updated (undeflated) singular value and the *i*-th (undeflated) old singular value.

difr
REAL for slasd6
DOUBLE PRECISION for dlasd6
Array, dimension (*ldgnum*, 2) if *icompq* = 1 and dimension (*n*) if *icompq* = 0.
On exit, *difr*(*i*, 1) is the distance between *i*-th updated (undeflated) singular value and the *i*+1-th (undeflated) old singular value. If *icompq* = 1, *difr*(1: *k*, 2) is an array containing the normalizing factors for the right singular vector matrix.
See ?lasd8 for details on *difl* and *difr*.

z
REAL for slasd6
DOUBLE PRECISION for dlasd6
Array, dimension (*m*).
The first elements of this array contain the components of the deflation-adjusted updating row vector.

k
INTEGER. Contains the dimension of the non-deflated matrix.
This is the order of the related secular equation. 1 ≤ *k* ≤ *n*.

c
REAL for slasd6
DOUBLE PRECISION for dlasd6
c contains garbage if *sqre* =0 and the *C*-value of a Givens rotation related to the right null space if

	sqre = 1.
s	REAL for slasd6 DOUBLE PRECISION for dlasd6 *s* contains garbage if *sqre* =0 and the *S*-value of a Givens rotation related to the right null space if *sqre* = 1.
info	INTEGER. = 0: successful exit. < 0: if *info* = -*i*, the *i*-th argument had an illegal value. > 0: if *info* = 1, an singular value did not converge

?lasd7

Merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. Used by ?bdsdc.

Syntax

call slasd7(*icompq*, *nl*, *nr*, *sqre*, *k*, *d*, *z*, *zw*, *vf*, *vfw*, *vl*, *vlw*, *alpha*, *beta*, *dsigma*, *idx*, *idxp*, *idxq*, *perm*, *givptr*, *givcol*, *ldgcol*, *givnum*, *ldgnum*, *c*, *s*, *info*)

call dlasd7(*icompq*, *nl*, *nr*, *sqre*, *k*, *d*, *z*, *zw*, *vf*, *vfw*, *vl*, *vlw*, *alpha*, *beta*, *dsigma*, *idx*, *idxp*, *idxq*, *perm*, *givptr*, *givcol*, *ldgcol*, *givnum*, *ldgnum*, *c*, *s*, *info*)

Description

The routine ?lasd7 merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more singular values are close together or if there is a tiny entry in the *z* vector. For each such occurrence the order of the related secular equation problem is reduced by one. ?lasd7 is called from ?lasd6.

Input Parameters

icompq	INTEGER. Specifies whether singular vectors are to be computed in compact form, as follows: = 0: Compute singular values only.

= 1: Compute singular vectors of upper bidiagonal matrix in compact form.

nl INTEGER. The row dimension of the upper block.

nl ≥ 1.

nr INTEGER. The row dimension of the lower block.

nr ≥ 1.

sqre INTEGER.
= 0: the lower block is an *nr*-by-*nr* square matrix.
= 1: the lower block is an *nr*-by-(*nr*+1) rectangular matrix.
The bidiagonal matrix has *n* = *nl* + *nr* + 1 rows and *m* = *n* + *sqre* ≥ *n* columns.

d REAL for slasd7
DOUBLE PRECISION for dlasd7
Array, DIMENSION (*n*). On entry *d* contains the singular values of the two submatrices to be combined.

zw REAL for slasd7
DOUBLE PRECISION for dlasd7
Array, DIMENSION (*m*).
Workspace for z.

vf REAL for slasd7
DOUBLE PRECISION for dlasd7
Array, DIMENSION (*m*). On entry, *vf*(1:*nl*+1) contains the first components of all right singular vectors of the upper block; and *vf*(*nl*+2:*m*) contains the first components of all right singular vectors of the lower block.

vfw REAL for slasd7
DOUBLE PRECISION for dlasd7
Array, DIMENSION (*m*).
Workspace for vf.

vl REAL for slasd7
DOUBLE PRECISION for dlasd7
Array, DIMENSION (*m*).

	On entry, *vl*(1:*nl*+1) contains the last components of all right singular vectors of the upper block; and *vl*(*nl*+2:*m*) contains the last components of all right singular vectors of the lower block.
VLW	REAL for slasd7 DOUBLE PRECISION for dlasd7 Array, DIMENSION (*m*). Workspace for VL.
alpha	REAL for slasd7 DOUBLE PRECISION for dlasd7. Contains the diagonal element associated with the added row.
beta	REAL for slasd7 DOUBLE PRECISION for dlasd7 Contains the off-diagonal element associated with the added row.
idx	INTEGER. Workspace array, DIMENSION (*n*). This will contain the permutation used to sort the contents of *d* into ascending order.
idxp	INTEGER. Workspace array, DIMENSION (*n*). This will contain the permutation used to place deflated values of *d* at the end of the array.
idxq	INTEGER. Array, DIMENSION (*n*). This contains the permutation which separately sorts the two sub-problems in *d* into ascending order. Note that entries in the first half of this permutation must first be moved one position backward; and entries in the second half must first have *nl*+1 added to their values.
ldgcol	INTEGER.The leading dimension of the output array *givcol*, must be at least *n*.
ldgnum	INTEGER. The leading dimension of the output array *givnum*, must be at least *n*.

Output Parameters

k — INTEGER. Contains the dimension of the non-deflated matrix, this is the order of the related secular equation.

$1 \leq k \leq n$.

d — On exit, *d* contains the trailing (*n*-*k*) updated singular values (those which were deflated) sorted into increasing order.

z — REAL for slasd7
DOUBLE PRECISION for dlasd7.
Array, DIMENSION (*m*).
On exit, *z* contains the updating row vector in the secular equation.

vf — On exit, *vf* contains the first components of all right singular vectors of the bidiagonal matrix.

vl — On exit, *vl* contains the last components of all right singular vectors of the bidiagonal matrix.

dsigma — REAL for slasd7
DOUBLE PRECISION for dlasd7.
Array, DIMENSION (*n*). Contains a copy of the diagonal elements (*k*-1 singular values and one zero) in the secular equation.

idxp — On output, *idxp*(2: *k*) points to the nondeflated *d*-values and *idxp*(*k*+1:*n*) points to the deflated singular values.

perm — INTEGER.
Array, DIMENSION (*n*).
The permutations (from deflation and sorting) to be applied to each singular block. Not referenced if *icompq* = 0.

givptr — INTEGER.
The number of Givens rotations which took place in this subproblem. Not referenced if *icompq* = 0.

givcol — INTEGER.
Array, DIMENSION (*ldgcol*, 2). Each pair of numbers indicates a pair of columns to take place in a Givens rotation. Not referenced if *icompq* = 0.

givnum — REAL for slasd7
DOUBLE PRECISION for dlasd7.

	Array, DIMENSION (*ldgnum*, 2). Each number indicates the *C* or *S* value to be used in the corresponding Givens rotation. Not referenced if *icompq* = 0.
c	REAL for slasd7. DOUBLE PRECISION for dlasd7. *c* contains garbage if *sqre* =0 and the *C*-value of a Givens rotation related to the right null space if *sqre* = 1.
S	REAL for slasd7. DOUBLE PRECISION for dlasd7. *s* contains garbage if *sqre* =0 and the *S*-value of a Givens rotation related to the right null space if *sqre* = 1.
info	INTEGER. = 0: successful exit. < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?lasd8

Finds the square roots of the roots of the secular equation, and stores, for each element in D, the distance to its two nearest poles. Used by ?bdsdc.

Syntax

```
call slasd8( icompq, k, d, z, vf, vl, difl, difr, lddifr, dsigma, work, info )
call dlasd8( icompq, k, d, z, vf, vl, difl, difr, lddifr, dsigma, work, info )
```

Description

The routine ?lasd8 finds the square roots of the roots of the secular equation, as defined by the values in *dsigma* and *z*. It makes the appropriate calls to ?lasd4, and stores, for each element in *d*, the distance to its two nearest poles (elements in *dsigma*). It also updates the arrays *vf* and *vl*, the first and last components of all the right singular vectors of the original bidiagonal matrix. ?lasd8 is called from ?lasd6.

Input Parameters

icompq — INTEGER. Specifies whether singular vectors are to be computed in factored form in the calling routine:
= 0: Compute singular values only.
= 1: Compute singular vectors in factored form as well.

k — INTEGER. The number of terms in the rational function to be solved by ?lasd4. $k \geq 1$.

z — REAL for slasd8
DOUBLE PRECISION for dlasd8.
Array, DIMENSION (*k*).
The first *k* elements of this array contain the components of the deflation-adjusted updating row vector.

vf — REAL for slasd8
DOUBLE PRECISION for dlasd8.
Array, DIMENSION (*k*).
On entry, *vf* contains information passed through dbede8.

vl — REAL for slasd8
DOUBLE PRECISION for dlasd8.
Array, DIMENSION (*k*). On entry, *vl* contains information passed through dbede8.

lddifr — INTEGER. The leading dimension of the output array *difr*, must be at least *k*.

dsigma — REAL for slasd8
DOUBLE PRECISION for dlasd8.
Array, DIMENSION (*k*).
The first *k* elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.

work — REAL for slasd8
DOUBLE PRECISION for dlasd8.
Workspace array, DIMENSION at least (3*k*).

Output Parameters

d — REAL for slasd8
DOUBLE PRECISION for dlasd8.

	Array, DIMENSION (*k*). On output, *D* contains the updated singular values.
vf	On exit, *vf* contains the first *k* components of the first components of all right singular vectors of the bidiagonal matrix.
vl	On exit, *vl* contains the first *k* components of the last components of all right singular vectors of the bidiagonal matrix.
difl	REAL for slasd8 DOUBLE PRECISION for dlasd8. Array, DIMENSION (*k*). On exit, *difl*(*i*) = *d*(*i*) - *dsigma*(*i*).
difr	REAL for slasd8 DOUBLE PRECISION for dlasd8. Array, DIMENSION (*lddifr*, 2) if *icompq* = 1 and DIMENSION (*k*) if *icompq* = 0. On exit, *difr*(i,1) = *d*(*i*) - *dsigma*(i+1), *difr*(*k*,1) is not defined and will not be referenced. If *icompq* = 1, *difr*(1:*k*,2) is an array containing the normalizing factors for the right singular vector matrix.
info	INTEGER. = 0: successful exit. < 0: if *info* = -*i*, the *i*-th argument had an illegal value. > 0: If *info* = 1, an singular value did not converge.

?lasd9

Finds the square roots of the roots of the secular equation, and stores, for each element in D, the distance to its two nearest poles. Used by ?bdsdc.

Syntax

```
call slasd9( icompq, ldu, k, d, z, vf, vl, difl, difr, dsigma, work, info )
call dlasd9( icompq, ldu, k, d, z, vf, vl, difl, difr, dsigma, work, info )
```

Description

The routine ?lasd9 finds the square roots of the roots of the secular equation, as defined by the values in *dsigma* and *z*. It makes the appropriate calls to ?lasd4, and stores, for each element in *d*, the distance to its two nearest poles (elements in *dsigma*). It also updates the arrays *vf* and *vl*, the first and last components of all the right singular vectors of the original bidiagonal matrix. ?lasd9 is called from ?lasd7.

Input Parameters

icompq — INTEGER. Specifies whether singular vectors are to be computed in factored form in the calling routine:
If *icompq* = 0, compute singular values only;
If *icompq* = 1, compute singular vector matrices in factored form also.

k — INTEGER. The number of terms in the rational function to be solved by slasd4. $k \geq 1$.

dsigma — REAL for slasd9
DOUBLE PRECISION for dlasd9.
Array, DIMENSION(*k*).
The first *k* elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.

z — REAL for slasd9
DOUBLE PRECISION for dlasd9.
Array, DIMENSION (*k*). The first *k* elements of this array contain the components of the deflation-adjusted updating row vector.

vf — REAL for slasd9
DOUBLE PRECISION for dlasd9.
Array, DIMENSION(*k*). On entry, *vf* contains information passed through sbede8.

vl — REAL for slasd9
DOUBLE PRECISION for dlasd9.
Array, DIMENSION(*k*). On entry, *vl* contains information passed through sbede8.

work — REAL for slasd9
DOUBLE PRECISION for dlasd9.

	Workspace array, DIMENSION at least (3*k*).

Output Parameters

d	REAL for slasd9 DOUBLE PRECISION for dlasd9. Array, DIMENSION(*k*). *d*(*i*) contains the updated singular values.
vf	On exit, *vf* contains the first *k* components of the first components of all right singular vectors of the bidiagonal matrix.
vl	On exit, *vl* contains the first *k* components of the last components of all right singular vectors of the bidiagonal matrix.
difl	REAL for slasd9 DOUBLE PRECISION for dlasd9. Array, DIMENSION (*k*). On exit, *difl*(*i*) = *d*(*i*) - *dsigma*(*i*).
difr	REAL for slasd9 DOUBLE PRECISION for dlasd9. Array, DIMENSION (*ldu*, 2) if *icompq* =1 and DIMENSION (*k*) if *icompq* = 0. On exit, *difr*(*i*, 1) = *d*(*i*) - *dsigma*(*i*+1), *difr*(*k*, 1) is not defined and will not be referenced. If *icompq* = 1, *difr*(1:*k*, 2) is an array containing the normalizing factors for the right singular vector matrix.
info	INTEGER. = 0: successful exit. < 0: if *info* = -*i*, the *i*-th argument had an illegal value. > 0: If *info* = 1, an singular value did not converge

?lasda

Computes the singular value decomposition (SVD) of a real upper bidiagonal matrix with diagonal `d` *and off-diagonal* `e`*. Used by* `?bdsdc`*.*

Syntax

```
call slasda( icompq, smlsiz, n, sqre, d, e, u, ldu, vt, k, difl, difr, z,
poles, givptr, givcol, ldgcol, perm, givnum, c, s, work, iwork, info )
```

```
call dlasda( icompq, smlsiz, n, sqre, d, e, u, ldu, vt, k, difl, difr, z,
poles, givptr, givcol, ldgcol, perm, givnum, c, s, work, iwork, info )
```

Description

Using a divide and conquer approach, `?lasda` computes the singular value decomposition (*SVD*) of a real upper bidiagonal *n*-by-*m* matrix *B* with diagonal *d* and off-diagonal *e*, where *m = n + sqre*.

The algorithm computes the singular values in the *SVD B = U*S*VT*. The orthogonal matrices *U* and *VT* are optionally computed in compact form. A related subroutine `?lasd0` computes the singular values and the singular vectors in explicit form.

Input Parameters

icompq — INTEGER.
Specifies whether singular vectors are to be computed in compact form, as follows:
= 0: Compute singular values only.
= 1: Compute singular vectors of upper bidiagonal matrix in compact form.

smlsiz — INTEGER.
The maximum size of the subproblems at the bottom of the computation tree.

n — INTEGER. The row dimension of the upper bidiagonal matrix. This is also the dimension of the main diagonal array *d*.

sqre — INTEGER. Specifies the column dimension of the bidiagonal matrix.
If *sqre = 0*: the bidiagonal matrix has column dimension *m = n*

	If *sqre* = 1: the bidiagonal matrix has column dimension *m* = *n* + 1.
d	REAL for slasda DOUBLE PRECISION for dlasda. Array, DIMENSION (*n*). On entry, *d* contains the main diagonal of the bidiagonal matrix.
e	REAL for slasda DOUBLE PRECISION for dlasda. Array, DIMENSION (*m* - 1). Contains the subdiagonal entries of the bidiagonal matrix. On exit, *e* is destroyed.
ldu	INTEGER. The leading dimension of arrays *u*, *vt*, *difl*, *difr*, *poles*, *givnum*, and *z*. *ldu* ≥ *n*.
ldgcol	INTEGER. The leading dimension of arrays *givcol* and *perm*. *ldgcol* ≥ *n*.
work	REAL for slasda DOUBLE PRECISION for dlasda. Workspace array, DIMENSION ($6 * n + (smlsiz + 1)^2$).
iwork	INTEGER. Workspace array, *Dimension* must be at least (7 * *n*).

Output Parameters

d	On exit *d*, if *info* = 0, contains the singular values of the bidiagonal matrix.
u	REAL for slasda DOUBLE PRECISION for dlasda. Array, DIMENSION (*ldu*, *smlsiz*) if *icompq* =1. Not referenced if *icompq* = 0. If *icompq* = 1, on exit, *u* contains the left singular vector matrices of all subproblems at the bottom level.
vt	REAL for slasda DOUBLE PRECISION for dlasda. Array, DIMENSION (*ldu*, *smlsiz*+1) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, *vt*' contains the right singular vector matrices of all subproblems at the bottom level.

k INTEGER.
Array, DIMENSION (*n*) if *icompq* = 1 and
DIMENSION (1) if *icompq* = 0.
If *icompq* = 1, on exit, *k*(*i*) is the dimension of the *i*-th secular equation on the computation tree.

difl REAL for slasda
DOUBLE PRECISION for dlasda.
Array, DIMENSION (*ldu*, *nlvl*),
where *nlvl* = floor (log*2* (*n*/*smlsiz*))).

difr REAL for slasda
DOUBLE PRECISION for dlasda.
Array,
DIMENSION (*ldu*, 2 *nlvl*) if *icompq* = 1 and
DIMENSION (*n*) if *icompq* = 0.
If *icompq* = 1, on exit, *difl*(1:*n*, *i*) and *difr*(1:*n*,2*i* -1) record distances between singular values on the *i*-th level and singular values on the (*i* -1)-th level, and *difr*(1:*n*, 2i) contains the normalizing factors for the right singular vector matrix. See ?lasd8 for details.

z REAL for slasda
DOUBLE PRECISION for dlasda.
Array,
DIMENSION (*ldu*, *nlvl*) if *icompq* = 1 and
DIMENSION (*n*) if *icompq* = 0. The first *k* elements of *z*(1, *i*) contain the components of the deflation-adjusted updating row vector for subproblems on the *i*-th level.

poles REAL for slasda
DOUBLE PRECISION for dlasda
Array, DIMENSION (*ldu*, 2**nlvl*)
if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, *poles*(1, 2*i* - 1) and *poles*(1, 2*i*) contain the new and old singular values involved in the secular equations on the *i*-th level.

givptr INTEGER. Array, DIMENSION (*n*) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, *givptr*(*i*) records the number of Givens rotations performed on the *i*-th problem on the computation tree.

givcol	INTEGER . Array, DIMENSION (*ldgcol*, 2**nlvl*) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, for each *i*, *givcol*(1, 2 *i* - 1) and *givcol*(1, 2 *i*) record the locations of Givens rotations performed on the *i*-th level on the computation tree.
perm	INTEGER . Array, DIMENSION (*ldgcol*, *nlvl*) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, *perm* (1, *i*) records permutations done on the *i*-th level of the computation tree.
givnum	REAL for slasda DOUBLE PRECISION for dlasda. Array DIMENSION (*ldu*, 2**nlvl*) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, for each *i*, *givnum*(1, 2 *i* - 1) and *givnum*(1, 2 *i*) record the *C*- and *S*-values of Givens rotations performed on the *i*-th level on the computation tree.
c	REAL for slasda DOUBLE PRECISION for dlasda. Array, DIMENSION (*n*) if *icompq* = 1, and DIMENSION (1) if *icompq* = 0. If *icompq* = 1 and the *i*-th subproblem is not square, on exit, *c*(*i*) contains the *C*-value of a Givens rotation related to the right null space of the *i*-th subproblem.
s	REAL for slasda DOUBLE PRECISION for dlasda. Array, DIMENSION (*n*) *icompq* = 1, and DIMENSION (1) if *icompq* = 0. If *icompq* = 1 and the *i*-th subproblem is not square, on exit, *s*(*i*) contains the *S*-value of a Givens rotation related to the right null space of the *i*-th subproblem.
info	INTEGER. = 0: successful exit. < 0: if *info* = -*i*, the *i*-th argument had an illegal value > 0: If *info* = 1, an singular value did not converge

?lasdq

Computes the SVD of a real bidiagonal matrix with diagonal d and off-diagonal e. Used by ?bdsdc.

Syntax

```
call slasdq( uplo, sqre, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, work, info )
```

```
call dlasdq( uplo, sqre, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, work, info )
```

Description

The routine ?lasdq computes the singular value decomposition (*SVD*) of a real (upper or lower) bidiagonal matrix with diagonal *d* and off-diagonal *e*, accumulating the transformations if desired. Letting *B* denote the input bidiagonal matrix, the algorithm computes orthogonal matrices *Q* and *P* such that *B* = *Q***S***P*' (*P*' denotes the transpose of *P*). The singular values *S* are overwritten on *d*.

The input matrix *U* is changed to *U***Q* if desired.

The input matrix *VT* is changed to *P*' **VT* if desired.

The input matrix *C* is changed to *Q*'**C* if desired.

Input Parameters

uplo — CHARACTER*1. On entry, *uplo* specifies whether the input bidiagonal matrix is upper or lower bidiagonal.
If *uplo* = 'U' or 'u' , *B* is upper bidiagonal;
If *uplo* = 'L' or '*l*' , *B* is lower bidiagonal.

sqre — INTEGER.
= 0: then the input matrix is *n*-by-*n*.
= 1: then the input matrix is *n*-by-(*n*+1) if *uplu* = 'U' and (*n*+1)-by-*n* if *uplu*
= 'L'. The bidiagonal matrix has *n* = *nl* + *nr* + 1 rows and *m* = *n* + *sqre* ≥ *n* columns.

n — INTEGER. On entry, *n* specifies the number of rows and columns in the matrix. *n* must be at least 0.

ncvt	INTEGER. On entry, *ncvt* specifies the number of columns of the matrix *VT*. *ncvt* must be at least 0.
nru	INTEGER. On entry, *nru* specifies the number of rows of the matrix *U*. *nru* must be at least 0.
ncc	INTEGER. On entry, *ncc* specifies the number of columns of the matrix *C*. *ncc* must be at least 0.
d	REAL for *slasdq* DOUBLE PRECISION for *dlasdq*. Array, DIMENSION (*n*). On entry, *d* contains the diagonal entries of the bidiagonal matrix whose *SVD* is desired.
e	REAL for *slasdq* DOUBLE PRECISION for *dlasdq*. Array, DIMENSION is (*n*-1) if *sqre* = 0 and *n* if *sqre* = 1. On entry, the entries of *e* contain the off-diagonal entries of the bidiagonal matrix whose *SVD* is desired.
vt	REAL for slasdq DOUBLE PRECISION for dlasdq. Array, DIMENSION (*ldvt*, *ncvt*). On entry, contains a matrix which on exit has been premultiplied by *P'*, dimension *n*-by-*ncvt* if *sqre* = 0 and (*n*+1)-by-*ncvt* if *sqre* = 1 (not referenced if *ncvt*=0).
ldvt	INTEGER. On entry, *ldvt* specifies the leading dimension of *vt* as declared in the calling (sub) program. *ldvt* must be at least 1. If *ncvt* is nonzero, *ldvt* must also be at least *n*.
u	REAL for *slasdq* DOUBLE PRECISION for *dlasdq*. Array, DIMENSION (*ldu*, *n*). On entry, contains a matrix which on exit has been postmultiplied by *Q*, dimension *nru*-by-*n* if *sqre* = 0 and *nru*-by-(*n*+1) if *sqre* = 1 (not referenced if *nru*=0).
ldu	INTEGER. On entry, *ldu* specifies the leading dimension of *u* as declared in the calling (sub) program. *ldu* must be at least max(1, *nru*) .
c	REAL for *slasdq* DOUBLE PRECISION for *dlasdq*.

	Array, DIMENSION (*ldc*, *ncc*). On entry, contains an *n*-by-*ncc* matrix which on exit has been premultiplied by *Q*', dimension *n*-by-*ncc* if *sqre* = 0 and (*n*+1)-by-*ncc* if *sqre* = 1 (not referenced if *ncc*=0).
ldc	INTEGER. On entry, *ldc* specifies the leading dimension of *C* as declared in the calling (sub) program. *ldc* must be at least 1. If *ncc* is non-zero, *ldc* must also be at least *n*.
work	REAL for slasdq DOUBLE PRECISION for dlasdq. Array, DIMENSION (4*n*). This is a workspace array. Only referenced if one of *ncvt*, *nru*, or *ncc* is nonzero, and if *n* is at least 2.

Output Parameters

d	On normal exit, *d* contains the singular values in ascending order.
e	On normal exit, *e* will contain 0. If the algorithm does not converge, *d* and *e* will contain the diagonal and superdiagonal entries of a bidiagonal matrix orthogonally equivalent to the one given as input.
vt	On exit, the matrix has been premultiplied by *P*'.
u	On exit, the matrix has been postmultiplied by *Q*.
c	On exit, the matrix has been premultiplied by *Q*'.
info	INTEGER. On exit, a value of 0 indicates a successful exit. If *info* < 0, argument number -*info* is illegal. If *info* > 0, the algorithm did not converge, and *info* specifies how many superdiagonals did not converge.

?lasdt

Creates a tree of subproblems for bidiagonal divide and conquer. Used by ?bdsdc.

Syntax

```
call slasdt( n, lvl, nd, inode, ndiml, ndimr, msub )
call dlasdt( n, lvl, nd, inode, ndiml, ndimr, msub )
```

Description

The routine creates a tree of subproblems for bidiagonal divide and conquer.

Input Parameters

n	INTEGER. On entry, the number of diagonal elements of the bidiagonal matrix.
msub	INTEGER. On entry, the maximum row dimension each subproblem at the bottom of the tree can be of.

Output Parameters

lvl	INTEGER. On exit, the number of levels on the computation tree.
nd	INTEGER. On exit, the number of nodes on the tree.
inode	INTEGER. Array, DIMENSION (*n*). On exit, centers of subproblems.
ndiml	INTEGER . Array, DIMENSION (*n*). On exit, row dimensions of left children.
ndimr	INTEGER . Array, DIMENSION (*n*). On exit, row dimensions of right children.

?laset

Initializes the off-diagonal elements and the diagonal elements of a matrix to given values.

Syntax

```
call slaset( uplo, m, n, alpha, beta, a, lda )
call dlaset( uplo, m, n, alpha, beta, a, lda )
call claset( uplo, m, n, alpha, beta, a, lda )
call zlaset( uplo, m, n, alpha, beta, a, lda )
```

Description

The routine initializes an *m*-by-*n* matrix *A* to *beta* on the diagonal and *alpha* on the off-diagonals.

Input Parameters

uplo
: CHARACTER*1. Specifies the part of the matrix *A* to be set.
If *uplo* = 'U', upper triangular part is set; the strictly lower triangular part of *A* is not changed.
If *uplo* = 'L': lower triangular part is set; the strictly upper triangular part of *A* is not changed.
Otherwise: All of the matrix *A* is set.

m
: INTEGER. The number of rows of the matrix *A*. $m \geq 0$.

n
: INTEGER. The number of columns of the matrix *A*.
$n \geq 0$.

alpha, *beta*
: REAL for slaset
DOUBLE PRECISION for dlaset
COMPLEX for claset
COMPLEX*16 for zlaset.
The constants to which the off-diagonal and diagonal elements are to be set, respectively.

a
: REAL for slaset
DOUBLE PRECISION for dlaset
COMPLEX for claset
COMPLEX*16 for zlaset.

	Array, DIMENSION (*lda*, *n*). On entry, the *m*-by-*n* matrix *A*.
lda	INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1,*m*).

Output Parameters

a	On exit, the leading *m*-by-*n* submatrix of *A* is set as follows: if *uplo* = 'U', *A*(*i*,*j*) = *alpha*, 1≤ *i* ≤ *j*-1, 1≤ *j* ≤ *n*, if *uplo* = 'L', *A*(*i*,*j*) = *alpha*, *j*+1≤ *i* ≤ *m*, 1≤ *j* ≤ *n*, otherwise, *A*(*i*,*j*) = *alpha*, 1≤ *i* ≤ *m*, 1≤ *j* ≤ *n*, *i* ≠ *j*, and, for all *uplo*, *A*(*i*,*i*) = *beta*, 1≤ *i* ≤ min(*m*, *n*).

?lasq1

Computes the singular values of a real square bidiagonal matrix. Used by ?bdsqr.

Syntax

```
call slasq1( n, d, e, work, info )
call dlasq1( n, d, e, work, info )
```

Description

The routine ?lasq1 computes the singular values of a real *n*-by-*n* bidiagonal matrix with diagonal *d* and off-diagonal *e*. The singular values are computed to high relative accuracy, in the absence of denormalization, underflow and overflow.

Input Parameters

n	INTEGER.The number of rows and columns in the matrix. *n* ≥ 0.
d	REAL for slasq1 DOUBLE PRECISION for dlasq1.

Array, DIMENSION (n).
On entry, d contains the diagonal elements of the bidiagonal matrix whose SVD is desired.

e — REAL for slasq1
DOUBLE PRECISION for dlasq1.
Array, DIMENSION (n).
On entry, elements $e(1:n-1)$ contain the off-diagonal elements of the bidiagonal matrix whose SVD is desired.

work — REAL for slasq1
DOUBLE PRECISION for dlasq1.
Workspace array, DIMENSION ($4n$).

Output Parameters

d — On normal exit, d contains the singular values in decreasing order.

e — On exit, e is overwritten.

info — INTEGER.
= 0: successful exit;
< 0: if $info = -i$, the i-th argument had an illegal value;
> 0: the algorithm failed:
= 1, a split was marked by a positive value in e;
= 2, current block of z not diagonalized after $30*n$ iterations (in inner while loop);
= 3, termination criterion of outer while loop not met (program created more than n unreduced blocks.

?lasq2

Computes all the eigenvalues of the symmetric positive definite tridiagonal matrix associated with the qd array z to high relative accuracy. Used by ?bdsqr *and* ?stegr.

Syntax

```
call slasq2( n, z, info )
call dlasq2( n, z, info )
```

Description

The routine ?lasq2 computes all the eigenvalues of the symmetric positive definite tridiagonal matrix associated with the *qd* array *z* to high relative accuracy, in the absence of denormalization, underflow and overflow.

To see the relation of *z* to the tridiagonal matrix, let *L* be a unit lower bidiagonal matrix with subdiagonals *z*(2,4,6,,..) and let *U* be an upper bidiagonal matrix with 1's above and diagonal *z*(1,3,5,,..). The tridiagonal is *LU* or, if you prefer, the symmetric tridiagonal to which it is similar.

Input Parameters

n — INTEGER. The number of rows and columns in the matrix.
n ≥ 0.

z — REAL for slasq2
DOUBLE PRECISION for dlasq2.
Array, DIMENSION (4 * *n*).
On entry, *z* holds the *qd* array.

Output Parameters

z — On exit, entries 1 to *n* hold the eigenvalues in decreasing order, *z*(2**n*+1) holds the trace, and *z*(2**n*+2) holds the sum of the eigenvalues. If *n* > 2, then *z*(2**n*+3) holds the iteration count, *z*(2**n*+4) holds ndivs/nin^2, and *z*(2**n*+5) holds the percentage of shifts that failed.

info — INTEGER.
= 0: successful exit;
< 0: if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*, if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+ *j*);
> 0: the algorithm failed:
= 1, a split was marked by a positive value in *e*;
= 2, current block of *z* not diagonalized after 30**n* iterations (in inner while loop);
= 3, termination criterion of outer while loop not met (program created more than *n* unreduced blocks).

Application Notes

The routine ?lasq2 defines a logical variable, *ieee*, which is .TRUE. on machines which follow ieee-754 floating-point standard in their handling of infinities and NaNs, and .FALSE. otherwise. This variable is passed to ?lazq3.

?lasq3

Checks for deflation, computes a shift and calls dqds. Used by ?bdsqr.

Syntax

```
call slasq3( i0, n0, z, pp, dmin, sigma, desig, qmax, nfail, iter, ndiv, ieee
)
```

```
call dlasq3( i0, n0, z, pp, dmin, sigma, desig, qmax, nfail, iter, ndiv, ieee
)
```

Description

The routine ?lasq3 checks for deflation, computes a shift, and calls *dqds*. In case of failure, it changes shifts, and tries again until output is positive.

Input Parameters

i0	INTEGER. First index.
n0	INTEGER. Last index.
z	REAL for slasq3 DOUBLE PRECISION for dlasq3. Array, DIMENSION (4*n*). *z* holds the *qd* array.
pp	INTEGER. *pp*=0 for ping, *pp*=1 for pong.
desig	REAL for slasq3 DOUBLE PRECISION for dlasq3. Lower order part of *sigma*.
qmax	REAL for slasq3 DOUBLE PRECISION for dlasq3. Maximum value of *q*.
ieee	LOGICAL. Flag for ieee or non-ieee arithmetic (passed to ?lasq5).

Output Parameters

dmin	REAL for slasq3 DOUBLE PRECISION for dlasq3. Minimum value of *d*.
sigma	REAL for slasq3 DOUBLE PRECISION for dlasq3. Sum of shifts used in current segment.
desig	Lower order part of *sigma*.
nfail	INTEGER. Number of times shift was too big.
iter	INTEGER. Number of iterations.
ndiv	INTEGER. Number of divisions.

?lasq4

Computes an approximation to the smallest eigenvalue using values of d from the previous transform. Used by ?bdsqr.

Syntax

call slasq4(*i0*, *n0*, *z*, *pp*, *n0in*, *dmin*, *dmin1*, *dmin2*, *dn*, *dn1*, *dn2*, *tau*, *ttype*)

call dlasq4(*i0*, *n0*, *z*, *pp*, *n0in*, *dmin*, *dmin1*, *dmin2*, *dn*, *dn1*, *dn2*, *tau*, *ttype*)

Description

The routine computes an approximation *tau* to the smallest eigenvalue using values of d from the previous transform.

Input Parameters

i0	INTEGER. First index.
n0	INTEGER. Last index.
z	REAL for slasq4 DOUBLE PRECISION for dlasq4. Array, DIMENSION (4*n*).

	z holds the *qd* array.
pp	INTEGER. *pp*=0 for ping, *pp*=1 for pong.
n0in	INTEGER. The value of *n0* at start of eigtest.
dmin	REAL for slasq4 DOUBLE PRECISION for dlasq4. Minimum value of *d*.
dmin1	REAL for slasq4 DOUBLE PRECISION for dlasq4. Minimum value of *d*, excluding *d*(*n0*).
dmin2	REAL for slasq4 DOUBLE PRECISION for dlasq4. Minimum value of *d*, excluding *d*(*n0*) and *d*(*n0*-1).
dn	REAL for slasq4 DOUBLE PRECISION for dlasq4. Contains *d*(*n*).
dn1	REAL for slasq4 DOUBLE PRECISION for dlasq4. Contains *d*(*n*-1).
dn2	REAL for slasq4 DOUBLE PRECISION for dlasq4. Contains *d*(*n*-2).

Output Parameters

tau	REAL for slasq4 DOUBLE PRECISION for dlasq4. Shift.
ttype	INTEGER. Shift type.

?lasq5

Computes one dqds transform in ping-pong form.
Used by ?bdsqr and ?stegr.

Syntax

```
call slasq5( i0, n0, z, pp, tau, dmin, dmin1, dmin2, dn, dnm1, dnm2, ieee )
call dlasq5( i0, n0, z, pp, tau, dmin, dmin1, dmin2, dn, dnm1, dnm2, ieee )
```

Description

The routine computes one dqds transform in ping-pong form: one version for ieee machines, another for non-ieee machines.

Input Parameters

i0	INTEGER. First index.
n0	INTEGER. Last index.
z	REAL for slasq5 DOUBLE PRECISION for dlasq5. Array, DIMENSION (4*n*). *z* holds the qd array. *emin* is stored in *z*(4**n0*) to avoid an extra argument.
pp	INTEGER. *pp*=0 for ping, *pp*=1 for pong.
tau	REAL for slasq5 DOUBLE PRECISION for dlasq5. This is the shift.
ieee	LOGICAL. Flag for IEEE or non-IEEE arithmetic.

Output Parameters

dmin	REAL for slasq5 DOUBLE PRECISION for dlasq5. Minimum value of *d*.
dmin1	REAL for slasq5 DOUBLE PRECISION for dlasq5. Minimum value of *d*, excluding *d*(*n0*).
dmin2	REAL for slasq5 DOUBLE PRECISION for dlasq5. Minimum value of *d*, excluding *d*(*n0*) and *d*(*n0*-1).
dn	REAL for slasq5 DOUBLE PRECISION for dlasq5. Contains *d*(*n0*), the last value of *d*.
dnm1	REAL for slasq5 DOUBLE PRECISION for dlasq5. Contains *d*(*n0*-1).
dnm2	REAL for slasq5

DOUBLE PRECISION for dlasq5. Contains d(n0-2).

?lasq6

Computes one dqd transform in ping-pong form.
Used by ?bdsqr *and* ?stegr.

Syntax

```
call slasq6( i0, n0, z, pp, dmin, dmin1, dmin2, dn, dnm1, dnm2 )
call dlasq6( i0, n0, z, pp, dmin, dmin1, dmin2, dn, dnm1, dnm2 )
```

Description

The routine ?lasq6 computes one *dqd* (shift equal to zero) transform in ping-pong form, with protection against underflow and overflow.

Input Parameters

i0	INTEGER. First index.
n0	INTEGER. Last index.
z	REAL for slasq6 DOUBLE PRECISION for dlasq6. Array, DIMENSION (4*n*). *Z* holds the qd array. *emin* is stored in *z*(4**n0*) to avoid an extra argument.
pp	INTEGER. *pp*=0 for ping, *pp*=1 for pong.

Output Parameters

dmin	REAL for slasq6 DOUBLE PRECISION for dlasq6. Minimum value of *d*.
dmin1	REAL for slasq6 DOUBLE PRECISION for dlasq6. Minimum value of *d*, excluding *d*(*n0*).
dmin2	REAL for slasq6 DOUBLE PRECISION for dlasq6. Minimum value of *d*, excluding *d*(*n0*) and *d*(*n0*-1).
dn	REAL for slasq6

	DOUBLE PRECISION for dlasq6. Contains *d*(*n0*), the last value of *d*.
dnm1	REAL for slasq6 DOUBLE PRECISION for dlasq6. Contains *d*(*n0*-1).
dnm2	REAL for slasq6 DOUBLE PRECISION for dlasq6. Contains *d*(*n0*-2).

?lasr

Applies a sequence of plane rotations to a general rectangular matrix.

Syntax

```
call slasr( side, pivot, direct, m, n, c, s, a, lda )
call dlasr( side, pivot, direct, m, n, c, s, a, lda )
call clasr( side, pivot, direct, m, n, c, s, a, lda )
call zlasr( side, pivot, direct, m, n, c, s, a, lda )
```

Description

The routine applies a sequence of plane rotations to a real/complex matrix *A*, from the left or the right.

A := *P***A*, when *side* = 'L' (Left-hand side)

A := *A***P*', when *side* = 'R' (Right-hand side)

where *P* is an orthogonal matrix consisting of a sequence of plane rotations with *z* = *m* when *side* = 'L' and *z* = *n* when *side* = 'R'.

When *direct* = 'F' (Forward sequence), then

P = *P*(*z* - 1) ... *P*(2) *P*(1),

and when *direct* = 'B' (Backward sequence), then

P = *P*(1) *P*(2) ... *P*(*z* - 1),

where *P*(*k*) is a plane rotation matrix defined by the 2-by-2 plane rotation:

$$R(k) = \begin{bmatrix} c(k) & s(k) \\ -s(k) & c(k) \end{bmatrix}$$

When *pivot* = 'V' (Variable pivot), the rotation is performed for the plane (k, k + 1), that is, $P(k)$ has the form

$$P(k) = \begin{bmatrix} 1 & & & & & & \\ & \cdots & & & & & \\ & & 1 & & & & \\ & & & c(k) & s(k) & & \\ & & & -s(k) & c(k) & & \\ & & & & & 1 & \\ & & & & & & \cdots \\ & & & & & & & 1 \end{bmatrix}$$

where $R(k)$ appears as a rank-2 modification to the identity matrix in rows and columns k and k+1.

When *pivot* = 'T' (Top pivot), the rotation is performed for the plane (1, k+1), so $P(k)$ has the form

$$P(k)=\begin{bmatrix} c(k) & & & & s(k) & & \\ & 1 & & & & & \\ & & \cdots & & & & \\ & & & 1 & & & \\ -s(k) & & & & c(k) & & \\ & & & & & 1 & \\ & & & & & & \cdots \\ & & & & & & & 1 \end{bmatrix}$$

where $R(k)$ appears in rows and columns k and k+1.

Similarly, when *pivot* = 'B' (Bottom pivot), the rotation is performed for the plane (k, z), giving $P(k)$ the form

$$P(k)=\begin{bmatrix} 1 & & & & & & \\ & \cdots & & & & & \\ & & 1 & & & & \\ & & & c(k) & & & s(k) \\ & & & & 1 & & \\ & & & & & \cdots & \\ & & & & & & 1 \\ & & & -s(k) & & & c(k) \end{bmatrix}$$

where $R(k)$ appears in rows and columns k and z. The rotations are performed without ever forming $P(k)$ explicitly.

Input Parameters

side — CHARACTER*1. Specifies whether the plane rotation matrix P is applied to A on the left or the right.

= 'L': left, compute $A := P*A$
= 'R': right, compute $A := A*P'$

direct CHARACTER*1. Specifies whether P is a forward or backward sequence of plane rotations.
= 'F': forward, $P = P(z - 1)* ... * P(2) * P(1)$
= 'B': backward, $P = P(1)* P(2)* ... * P(z - 1)$

pivot CHARACTER*1. Specifies the plane for which $P(k)$ is a plane rotation matrix.
= 'V': Variable pivot, the plane (k, k+1)
= 'T': Top pivot, the plane (1, k+1)
= 'B': Bottom pivot, the plane (k, z)

m INTEGER. The number of rows of the matrix A.

If $m \leq 1$, an immediate return is effected.

n INTEGER. The number of columns of the matrix A.

If $n \leq 1$, an immediate return is effected.

c, *s* REAL for slasr/clasr
DOUBLE PRECISION for dlasr/zlasr.
Arrays, DIMENSION
(m-1) if *side* = 'L',
(n-1) if *side* = 'R' .
$c(k)$ and $s(k)$ contain the cosine and sine of the plane rotations respectively that define the 2-by-2 plane rotation part ($R(k)$) of the $P(k)$ matrix as described above in Description.

a REAL for slasr
DOUBLE PRECISION for dlasr
COMPLEX for clasr
COMPLEX*16 for zlasr.
Array, DIMENSION (*lda*, n).
The m-by-n matrix A.

lda INTEGER. The leading dimension of the array a.

$lda \geq \max(1,m)$.

Output Parameters

a	On exit, *A* is overwritten by *P*A* if *side* = 'R', or by *A*P'* if *side* = 'L'.

?lasrt

Sorts numbers in increasing or decreasing order.

Syntax

```
call slasrt( id, n, d, info )
call dlasrt( id, n, d, info )
```

Description

The routine ?lasrt sorts the numbers in *d* in increasing order (if *id* = 'I') or in decreasing order (if *id* = 'D'). It uses Quick Sort, reverting to Insertion Sort on arrays of size ≤ 20. Dimension of stack limits *n* to about 2^{32}.

Input Parameters

id	CHARACTER*1. = 'I': sort *d* in increasing order; = 'D': sort *d* in decreasing order.
n	INTEGER. The length of the array *d*.
d	REAL for slasrt DOUBLE PRECISION for dlasrt. On entry, the array to be sorted.

Output Parameters

d	On exit, *d* has been sorted into increasing order ($d(1) \le \ldots \le d(n)$) or into decreasing order ($d(1) \ge \ldots \ge d(n)$), depending on *id*.
info	INTEGER. = 0: successful exit < 0: if *info* = -i, the *i*-th argument had an illegal value.

?lassq

Updates a sum of squares represented in scaled form.

Syntax

```
call slassq( n, x, incx, scale, sumsq )
call dlassq( n, x, incx, scale, sumsq )
call classq( n, x, incx, scale, sumsq )
call zlassq( n, x, incx, scale, sumsq )
```

Description

The real routines slassq/dlassq return the values *scl* and *smsq* such that

$scl^2 * smsq = x(1)^2 + ... + x(n)^2 + scale^2 * sumsq$,

where $x(i) = x(1 + (i - 1) incx)$.

The value of *sumsq* is assumed to be non-negative and *scl* returns the value

$scl = \max(scale, abs(x(i)))$.

Values *scale* and *sumsq* must be supplied in *scale* and *sumsq*, and *scl* and *smsq* are overwritten on *scale* and *sumsq*, respectively.

The complex routines classq/zlassq return the values *scl* and *ssq* such that

$scl^2 * ssq = x(1)^2 + ... + x(n)^2 + scale^2 * sumsq$,

where $x(i) = abs (x(1 + (i - 1) incx))$.

The value of *sumsq* is assumed to be at least unity and the value of *ssq* will then satisfy $1.0 \le ssq \le sumsq + 2n$

scale is assumed to be non-negative and *scl* returns the value

$scl = \max(scale, abs(real(x(i))), abs(aimag(x(i))))$.

Values *scale* and *sumsq* must be supplied in *scale* and *sumsq*, and *scl* and *ssq* are overwritten on *scale* and *sumsq*, respectively.

All routines ?lassq make only one pass through the vector *x*.

Input Parameters

n	INTEGER. The number of elements to be used from the vector *x*.
x	REAL for slassq DOUBLE PRECISION for dlassq COMPLEX for classq COMPLEX*16 for zlassq. The vector for which a scaled sum of squares is computed: $x(i) = x(1 + (i - 1)\ incx), 1 \le i \le n$.
incx	INTEGER. The increment between successive values of the vector *x*. *incx* > 0.
scale	REAL for slassq/classq DOUBLE PRECISION for dlassq/zlassq. On entry, the value *scale* in the equation above.
sumsq	REAL for slassq/classq DOUBLE PRECISION for dlassq/zlassq. On entry, the value *sumsq* in the equation above.

Output Parameters

scale	On exit, *scale* is overwritten with *scl*, the scaling factor for the sum of squares.
sumsq	For real flavors: On exit, *sumsq* is overwritten with the value *smsq* in the equation above. For complex flavors: On exit, *sumsq* is overwritten with the value *ssq* in the equation above.

?lasv2

Computes the singular value decomposition of a 2-by-2 triangular matrix.

Syntax

```
call slasv2( f, g, h, ssmin, ssmax, snr, csr, snl, csl )
call dlasv2( f, g, h, ssmin, ssmax, snr, csr, snl, csl )
```

Description

The routine ?lasv2 computes the singular value decomposition of a 2-by-2 triangular matrix

$$\begin{bmatrix} f & g \\ 0 & h \end{bmatrix}$$

On return, abs(*ssmax*) is the larger singular value, abs(*ssmin*) is the smaller singular value, and (*csl*,*snl*) and (*csr*,*snr*) are the left and right singular vectors for abs(*ssmax*), giving the decomposition

$$\begin{bmatrix} csl & snl \\ -snl & csl \end{bmatrix} \begin{bmatrix} f & g \\ 0 & h \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix} = \begin{bmatrix} ssmax & 0 \\ 0 & ssmin \end{bmatrix}$$

Input Parameters

f, *g*, *h*	REAL for slasv2 DOUBLE PRECISION for dlasv2. The (1,1), (1,2) and (2,2) elements of the 2-by-2 matrix, respectively.

Output Parameters

ssmin, *ssmax*	REAL for slasv2

	DOUBLE PRECISION for dlasv2. abs(*ssmin*) and abs(*ssmax*) is the smaller and the larger singular value, respectively.
snl, *csl*	REAL for slasv2 DOUBLE PRECISION for dlasv2. The vector (*csl*, *snl*) is a unit left singular vector for the singular value abs(*ssmax*).
snr, *csr*	REAL for slasv2 DOUBLE PRECISION for dlasv2. The vector (*csr*, *snr*) is a unit right singular vector for the singular value abs(*ssmax*).

Application Notes

Any input parameter may be aliased with any output parameter.

Barring over/underflow and assuming a guard digit in subtraction, all output quantities are correct to within a few units in the last place (ulps).

In ieee arithmetic, the code works correctly if one matrix element is infinite. Overflow will not occur unless the largest singular value itself overflows or is within a few ulps of overflow. (On machines with partial overflow, like the Cray, overflow may occur if the largest singular value is within a factor of 2 of overflow.) Underflow is harmless if underflow is gradual. Otherwise, results may correspond to a matrix modified by perturbations of size near the underflow threshold.

?laswp

Performs a series of row interchanges on a general rectangular matrix.

Syntax

```
call slaswp( n, a, lda, k1, k2, ipiv, incx )
call dlaswp( n, a, lda, k1, k2, ipiv, incx )
call claswp( n, a, lda, k1, k2, ipiv, incx )
call zlaswp( n, a, lda, k1, k2, ipiv, incx )
```

Description

The routine performs a series of row interchanges on the matrix *A*. One row interchange is initiated for each of rows *k1* through *k2* of *A*.

Input Parameters

n	INTEGER. The number of columns of the matrix *A*.
a	REAL for slaswp DOUBLE PRECISION for dlaswp COMPLEX for claswp COMPLEX*16 for zlaswp. Array, DIMENSION (*lda*, *n*). On entry, the matrix of column dimension *n* to which the row interchanges will be applied.
lda	INTEGER. The leading dimension of the array *a*.
k1	INTEGER. The first element of *ipiv* for which a row interchange will be done.
k2	INTEGER. The last element of *ipiv* for which a row interchange will be done.
ipiv	INTEGER. Array, DIMENSION (*k2* * abs(*incx*)). The vector of pivot indices. Only the elements in positions *k1* through *k2* of *ipiv* are accessed. *ipiv*(*k*) = *l* implies rows *k* and *l* are to be interchanged.
incx	INTEGER. The increment between successive values of *ipiv*. If *ipiv* is negative, the pivots are applied in reverse order.

Output Parameters

a	On exit, the permuted matrix.

?lasy2

Solves the Sylvester matrix equation where the matrices are of order 1 or 2.

Syntax

```
call slasy2( ltranl, ltranr, isgn, n1, n2, tl, ldtl, tr, ldtr, b, ldb, scale, x, ldx, xnorm, info )
```

```
call dlasy2( ltranl, ltranr, isgn, n1, n2, tl, ldtl, tr, ldtr, b, ldb, scale, x, ldx, xnorm, info )
```

Description

The routine solves for the *n1*-by-*n2* matrix *X*, 1 ≤ *n1*, *n2* ≤ 2, in

op(*TL*) * *X* + *isgn* * *X* *op(*TR*) = *scale*B*,

where

TL is *n1*-by-*n1*,

TR is *n2*-by-*n2*,

B is *n1*-by-*n2*,

and *isgn* = 1 or -1. op(*T*) = *T* or *T*', where *T*' denotes the transpose of *T*.

Input Parameters

ltranl	LOGICAL. On entry, *ltranl* specifies the op(*TL*): = .FALSE., op(*TL*) = *TL*, = .TRUE., op(*TL*) = *TL*'.
ltranr	LOGICAL. On entry, *ltranr* specifies the op(*TR*): = .FALSE., op(*TR*) = *TR*, = .TRUE., op(*TR*) = *TR*'.
isgn	INTEGER. On entry, *isgn* specifies the sign of the equation as described before. *isgn* may only be 1 or -1.
n1	INTEGER. On entry, *n1* specifies the order of matrix *TL*. *n1* may only be 0, 1 or 2.

n2 INTEGER. On entry, *n2* specifies the order of matrix *TR*. *n2* may only be 0, 1 or 2.

tl REAL for slasy2
DOUBLE PRECISION for dlasy2.
Array, DIMENSION (*ldtl*,2).
On entry, *tl* contains an *n1*-by-*n1* matrix *TL*.

ldtl INTEGER.The leading dimension of the matrix *TL*.
ldtl ≥ max(1,*n1*).

tr REAL for slasy2
DOUBLE PRECISION for dlasy2.
Array, DIMENSION (*ldtr*,2). On entry, *tr* contains an *n2*-by-*n2* matrix *TR*.

ldtr INTEGER. The leading dimension of the matrix *tr*.
ldtr ≥ max(1,*n2*).

b REAL for slasy2
DOUBLE PRECISION for dlasy2.
Array, DIMENSION (*ldb*,2). On entry, the *n1*-by-*n2* matrix *b* contains the right-hand side of the equation.

ldb INTEGER. The leading dimension of the matrix *b*.
ldb ≥ max(1,*n1*).

ldx INTEGER. The leading dimension of the output matrix *x*.
ldx ≥ max(1,*n1*).

Output Parameters

scale REAL for slasy2
DOUBLE PRECISION for dlasy2.
On exit, *scale* contains the scale factor.
scale is chosen less than or equal to 1 to prevent the solution overflowing.

x REAL for slasy2
DOUBLE PRECISION for dlasy2.
Array, DIMENSION (*ldx*,2). On exit, *x* contains the *n1*-by-*n2* solution.

xnorm REAL for slasy2

	DOUBLE PRECISION for dlasy2. On exit, *xnorm* is the infinity-norm of the solution.
info	INTEGER. On exit, *info* is set to 0: successful exit. 1: *TL* and *TR* have too close eigenvalues, so *TL* or *TR* is perturbed to get a nonsingular equation.

NOTE. For higher speed, this routine does not check the inputs for errors.

?lasyf

Computes a partial factorization of a real/complex symmetric matrix, using the diagonal pivoting method.

Syntax

```
call slasyf( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call dlasyf( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call clasyf( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call zlasyf( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
```

Description

The routine *?lasyf* computes a partial factorization of a real/complex symmetric matrix *A* using the Bunch-Kaufman diagonal pivoting method. The partial factorization has the form:

$$A = \begin{bmatrix} I & U_{12} \\ 0 & U_{12} \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & D \end{bmatrix} \begin{bmatrix} I & 0 \\ U_{12}' & U_{12}' \end{bmatrix} \text{ if } uplo = \text{'U', or}$$

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} L_{12}' & L_{21}' \\ 0 & I \end{bmatrix} \text{ if } uplo = \text{'L'}$$

where the order of D is at most nb.

The actual order is returned in the argument kb, and is either nb or nb-1, or n if $n \leq nb$.

This is an auxiliary routine called by ?sytrf. It uses blocked code (calling Level 3 BLAS) to update the submatrix A_{11} (if *uplo* = 'U') or A_{22} (if *uplo* = 'L').

Input Parameters

uplo
CHARACTER*1.
Specifies whether the upper or lower triangular part of the symmetric matrix A is stored:
= 'U': Upper triangular
= 'L': Lower triangular

n
INTEGER. The order of the matrix A. $n \geq 0$.

nb
INTEGER. The maximum number of columns of the matrix A that should be factored. nb should be at least 2 to allow for 2-by-2 pivot blocks.

a
REAL for slasyf
DOUBLE PRECISION for dlasyf
COMPLEX for clasyf
COMPLEX*16 for zlasyf.
Array, DIMENSION (*lda*, *n*). On entry, the symmetric matrix A. If *uplo* = 'U', the leading n-by-n upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If *uplo* = 'L', the leading n-by-n lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

lda
INTEGER. The leading dimension of the array a. *lda* ≥ max(1,*n*).

w
REAL for slasyf
DOUBLE PRECISION for dlasyf
COMPLEX for clasyf
COMPLEX*16 for zlasyf.
Workspace array, DIMENSION (*ldw*, *nb*).

ldw	INTEGER. The leading dimension of the array *w*. *ldw* ≥ max(1,*n*).

Output Parameters

kb	INTEGER. The number of columns of *A* that were actually factored *kb* is either *nb*-1 or *nb*, or *n* if *n* ≤ *nb*.
a	On exit, *A* contains details of the partial factorization.
ipiv	INTEGER. Array, DIMENSION (*n*). Details of the interchanges and the block structure of *D*. If *uplo* = 'U', only the last *kb* elements of *ipiv* are set; if *uplo* = 'L', only the first *kb* elements are set. If *ipiv*(*k*) > 0, then rows and columns *k* and *ipiv*(*k*) were interchanged and *D*(*k*, *k*) is a 1-by-1 diagonal block. If *uplo* = 'U' and *ipiv*(*k*) = *ipiv*(*k*–1) < 0, then rows and columns *k*-1 and -*ipiv*(*k*) were interchanged and *D*(*k*–1:*k*, *k*–1:*k*) is a 2-by-2 diagonal block. If *uplo* = 'L' and *ipiv*(*k*) = *ipiv*(*k*+1) < 0, then rows and columns *k*+1 and –*ipiv*(*k*) were interchanged and *D*(*k*:*k*+1, *k*:*k*+1) is a 2-by-2 diagonal block.
info	INTEGER. = 0: successful exit > 0: if *info* = *k*, *D*(*k*, *k*) is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular.

?lahef

Computes a partial factorization of a complex Hermitian indefinite matrix, using the diagonal pivoting method.

Syntax

```
call clahef( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call zlahef( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
```

Description

The routine ?lahef computes a partial factorization of a complex Hermitian matrix A, using the Bunch-Kaufman diagonal pivoting method. The partial factorization has the form:

$$A = \begin{bmatrix} I & U_{12} \\ 0 & U_{12} \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & D \end{bmatrix} \begin{bmatrix} I & 0 \\ U_{12}' & U_{12}' \end{bmatrix} \text{ if } uplo = \text{'U'}, \text{ or}$$

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} L_{12}' & L_{21}' \\ 0 & I \end{bmatrix} \text{ if } uplo = \text{'L'}$$

where the order of D is at most nb.

The actual order is returned in the argument kb, and is either nb or nb-1, or n if $n \leq nb$.

Note that U' denotes the conjugate transpose of U.

This is an auxiliary routine called by ?hetrf. It uses blocked code (calling Level 3 BLAS) to update the submatrix A_{11} (if $uplo$ = 'U') or A_{22} (if $uplo$ = 'L').

Input Parameters

uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored: = 'U': upper triangular = 'L': lower triangular
n	INTEGER. The order of the matrix A. $n \geq 0$.
nb	INTEGER. The maximum number of columns of the matrix A that should be factored. nb should be at least 2 to allow for 2-by-2 pivot blocks.
a	COMPLEX for clahef COMPLEX*16 for zlahef.

Array, DIMENSION (*lda*, *n*).
On entry, the Hermitian matrix *A*.
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *A* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *A* is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *A* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *A* is not referenced.

lda — INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1,*n*).

w — COMPLEX for clahef
COMPLEX*16 for zlahef.
Workspace array, DIMENSION (*ldw*, *nb*).

ldw — INTEGER. The leading dimension of the array *w*. *ldw* ≥ max(1,*n*).

Output Parameters

kb — INTEGER. The number of columns of *A* that were actually factored *kb* is either *nb*-1 or *nb*, or *n* if *n* ≤ *nb*.

a — On exit, *A* contains details of the partial factorization.

ipiv — INTEGER.
Array, DIMENSION (*n*). Details of the interchanges and the block structure of *D*.
If *uplo* = 'U', only the last *kb* elements of *ipiv* are set; if *uplo* = 'L', only the first *kb* elements are set.
If *ipiv*(*k*) > 0, then rows and columns *k* and *ipiv*(*k*) are interchanged and *D*(*k*, *k*) is a 1-by-1 diagonal block.
If *uplo* = 'U' and *ipiv*(*k*) = *ipiv*(*k*–1) < 0, then rows and columns *k*-1 and -*ipiv*(*k*) are interchanged and *D*(*k*–1:*k*, *k*–1:*k*) is a 2-by-2 diagonal block.
If *uplo* = 'L' and *ipiv*(*k*) = *ipiv*(*k*+1) < 0, then rows and columns *k*+1 and –*ipiv*(*k*) are interchanged and *D*(*k*:*k*+1, *k*:*k*+1) is a 2-by-2 diagonal block.

info — INTEGER.
= 0: successful exit

> 0: if *info* = *k*, *D*(*k*, *k*) is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular.

?latbs

Solves a triangular banded system of equations.

Syntax

```
call slatbs( uplo, trans, diag, normin, n, kd, ab, ldab, x, scale, cnorm, info )
call dlatbs( uplo, trans, diag, normin, n, kd, ab, ldab, x, scale, cnorm, info )
call clatbs( uplo, trans, diag, normin, n, kd, ab, ldab, x, scale, cnorm, info )
call zlatbs( uplo, trans, diag, normin, n, kd, ab, ldab, x, scale, cnorm, info )
```

Description

The routine solves one of the triangular systems

*A*x = s*b*, or A^T**x = s*b*, or A^H**x = s*b* (for complex flavors)

with scaling to prevent overflow, where *A* is an upper or lower triangular band matrix. Here A^T denotes the transpose of *A*, A^H denotes the conjugate transpose of *A*, *x* and *b* are *n*-element vectors, and *s* is a scaling factor, usually less than or equal to 1, chosen so that the components of *x* will be less than the overflow threshold. If the unscaled problem will not cause overflow, the Level 2 BLAS routine ?tbsv is called. If the matrix *A* is singular (*A*(*j*, *j*) = 0 for some *j*), then *s* is set to 0 and a non-trivial solution to *A*x* = 0 is returned.

Input Parameters

uplo — CHARACTER*1.
Specifies whether the matrix *A* is upper or lower triangular.
= 'U': upper triangular
= 'L': lower triangular

trans — CHARACTER*1.
Specifies the operation applied to *A*.

= 'N': solve $A*x = s*b$ (no transpose)
= 'T': solve $A^T*x = s*b$ (transpose)
= 'C': solve $A^H*x = s*b$ (conjugate transpose)

diag — CHARACTER*1.
Specifies whether or not the matrix *A* is unit triangular
= 'N': non-unit triangular
= 'U': unit triangular

normin — CHARACTER*1.
Specifies whether *cnorm* has been set or not.
= 'Y': *cnorm* contains the column norms on entry;
= 'N': *cnorm* is not set on entry. On exit, the norms will be computed and stored in *cnorm*.

n — INTEGER. The order of the matrix *A*. $n \geq 0$.

kd — INTEGER. The number of subdiagonals or superdiagonals in the triangular matrix *A*. $kb \geq 0$.

ab — REAL for slatbs
DOUBLE PRECISION for dlatbs
COMPLEX for clatbs
COMPLEX*16 for zlatbs.
Array, DIMENSION (*ldab*, *n*).
The upper or lower triangular band matrix *A*, stored in the first *kb*+1 rows of the array. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:
if *uplo* = 'U', $ab(kd+1+i-j,j) = A(i,j)$ for $\max(1, j-kd) \leq i \leq j$;
if *uplo* = 'L', $ab(1+i-j,j) = A(i,j)$ for $j \leq i \leq \min(n, j+kd)$.

ldab — INTEGER. The leading dimension of the array *ab*. $ldab \geq kb+1$.

x — REAL for slatbs
DOUBLE PRECISION for dlatbs
COMPLEX for clatbs
COMPLEX*16 for zlatbs.
Array, DIMENSION (*n*).
On entry, the right hand side *b* of the triangular system.

cnorm	REAL for slatbs/clatbs DOUBLE PRECISION for dlatbs/zlatbs. Array, DIMENSION (*n*). If *NORMIN* = 'Y', *cnorm* is an input argument and *cnorm*(*j*) contains the norm of the off-diagonal part of the *j*-th column of *A*. If *trans* = 'N', *cnorm*(*j*) must be greater than or equal to the infinity-norm, and if *trans* = 'T' or 'C', *cnorm*(*j*) must be greater than or equal to the 1-norm.

Output Parameters

scale	REAL for slatbs/clatbs DOUBLE PRECISION for dlatbs/zlatbs. The scaling factor s for the triangular system as described above. If *scale* = 0, the matrix *A* is singular or badly scaled, and the vector *x* is an exact or approximate solution to *Ax* = 0.
cnorm	If *normin* = 'N', *cnorm* is an output argument and *cnorm*(*j*) returns the 1-norm of the off-diagonal part of the *j*-th column of *A*.
info	INTEGER. = 0: successful exit < 0: if *info* = -*k*, the *k*-th argument had an illegal value

?latdf

Uses the LU factorization of the n-by-n matrix computed by ?getc2 and computes a contribution to the reciprocal Dif-estimate.

Syntax

```
call slatdf( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpiv )
call dlatdf( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpiv )
call clatdf( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpiv )
call zlatdf( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpiv )
```

Description

The routine ?latdf uses the *LU* factorization of the *n*-by-*n* matrix *Z* computed by ?getc2 and computes a contribution to the reciprocal Dif-estimate by solving *Z*x = b* for *x*, and choosing the right-hand side *b* such that the norm of *x* is as large as possible. On entry *rhs = b* holds the contribution from earlier solved sub-systems, and on return *rhs = x*.

The factorization of *Z* returned by ?getc2 has the form *Z = P*L*U*Q*, where *p* and *Q* are permutation matrices. *L* is lower triangular with unit diagonal elements and *U* is upper triangular.

Input Parameters

ijob	INTEGER. *ijob = 2*: First compute an approximative null-vector *e* of *Z* using ?gecon, *e* is normalized, and solve for *Zx = ±e -f* with the sign giving the greater value of 2-norm(*x*). This option is about 5 times as expensive as default. *ijob ≠ 2* (default): Local look ahead strategy where all entries of the right-hand side *b* is chosen as either +1 or -1 .
n	INTEGER. The number of columns of the matrix *Z*.
z	REAL for slatdf/clatdf DOUBLE PRECISION for dlatdf/zlatdf. Array, DIMENSION (*ldz*, *n*) On entry, the *LU* part of the factorization of the *n*-by-*n* matrix *Z* computed by ?getc2: *Z = P*L*U*Q*.
ldz	INTEGER. The leading dimension of the array *Z*. *lda* ≥ max(1, *n*).
rhs	REAL for slatdf/clatdf DOUBLE PRECISION for dlatdf/zlatdf. Array, DIMENSION (*n*). On entry, *rhs* contains contributions from other subsystems.
rdsum	REAL for slatdf/clatdf DOUBLE PRECISION for dlatdf/zlatdf.

On entry, the sum of squares of computed contributions to the *D*if-estimate under computation by ?tgsyL, where the scaling factor *rdscal* has been factored out. If *trans* = 'T', *rdsum* is not touched.
Note that *rdsum* only makes sense when ?tgsy2 is called by ?tgsyL.

rdscal
REAL for slatdf/clatdf
DOUBLE PRECISION for dlatdf/zlatdf.
On entry, scaling factor used to prevent overflow in *rdsum*.
If *trans* = *T*', *rdscal* is not touched.
Note that *rdscal* only makes sense when ?tgsy2 is called by ?tgsyL.

ipiv
INTEGER.
Array, DIMENSION (*n*).
The pivot indices; for 1 ≤ i ≤ *n*, row *i* of the matrix has been interchanged with row *ipiv*(*i*).

jpiv
INTEGER.
Array, DIMENSION (*n*).
The pivot indices; for 1 ≤ j ≤ *n*, column *j* of the matrix has been interchanged with column *jpiv*(*j*).

Output Parameters

rhs
On exit, *rhs* contains the solution of the subsystem with entries according to the value of *ijob*.

rdsum
On exit, the corresponding sum of squares updated with the contributions from the current sub-system.
If *trans* = 'T', *rdsum* is not touched.

rdscal
On exit, *rdscal* is updated with respect to the current contributions in *rdsum*.
If *trans* = 'T', *rdscal* is not touched.

?latps

Solves a triangular system of equations with the matrix held in packed storage.

Syntax

```
call slatps( uplo, trans, diag, normin, n, ap, x, scale, cnorm, info )
call dlatps( uplo, trans, diag, normin, n, ap, x, scale, cnorm, info )
call clatps( uplo, trans, diag, normin, n, ap, x, scale, cnorm, info )
call zlatps( uplo, trans, diag, normin, n, ap, x, scale, cnorm, info )
```

Description

The routine ?latps solves one of the triangular systems

$A*x = s*b$, or $A^T*x = s*b$, or $A^H*x = s*b$ (for complex flavors)

with scaling to prevent overflow, where A is an upper or lower triangular matrix stored in packed form. Here A^T denotes the transpose of A, A^H denotes the conjugate transpose of A, x and b are n-element vectors, and s is a scaling factor, usually less than or equal to 1, chosen so that the components of x will be less than the overflow threshold. If the unscaled problem does not cause overflow, the Level 2 BLAS routine ?tpsv is called. If the matrix A is singular ($A(j, j) = 0$ for some j), then s is set to 0 and a non-trivial solution to $Ax = 0$ is returned.

Input Parameters

uplo CHARACTER*1.
Specifies whether the matrix A is upper or lower triangular.
= 'U': upper triangular
= 'L': uower triangular

trans CHARACTER*1.
Specifies the operation applied to A.
= 'N': solve $A*x = s*b$ (no transpose)
= 'T': solve $A^T*x = s*b$ (transpose)
= 'C': solve $A^H*x = s*b$ (conjugate transpose)

diag CHARACTER*1.
Specifies whether or not the matrix A is unit triangular.
= '*N*': non-unit triangular

= 'U': unit triangular

normin CHARACTER*1.
Specifies whether *cnorm* has been set or not.
= 'Y': *cnorm* contains the column norms on entry;
= 'N': *cnorm* is not set on entry. On exit, the norms will be computed and stored in cnorm.

n INTEGER. The order of the matrix *A*. $n \geq 0$.

ap REAL for slatps
DOUBLE PRECISION for dlatps
COMPLEX for clatps
COMPLEX*16 for zlatps.
Array, DIMENSION ($n(n+1)/2$).
The upper or lower triangular matrix *A*, packed columnwise in a linear array. The j-th column of *A* is stored in the array *ap* as follows:
if *uplo* = 'U', $ap(i + (j-1)j/2) = A(i,j)$ for $1 \leq i \leq j$;
if *uplo* = 'L', $ap(i + (j-1)(2n-j)/2) = A(i, j)$ for $j \leq i \leq n$.

x REAL for slatps DOUBLE PRECISION for dlatps
COMPLEX for clatps
COMPLEX*16 for zlatps.
Array, DIMENSION (n)
On entry, the right hand side b of the triangular system.

cnorm REAL for slatps/clatps
DOUBLE PRECISION for dlatps/zlatps.
Array, DIMENSION (n).
If *normin* = 'Y', *cnorm* is an input argument and *cnorm*(j) contains the norm of the off-diagonal part of the j-th column of *A*.
If *trans* = 'N', *cnorm*(j) must be greater than or equal to the infinity-norm, and if *trans* = 'T' or 'C' , *cnorm*(j) must be greater than or equal to the 1-norm.

Output Parameters

x	On exit, *x* is overwritten by the solution vector *x*.
scale	REAL for slatps/clatps DOUBLE PRECISION for dlatps/zlatps. The scaling factor *s* for the triangular system as described above. If *scale* = 0, the matrix *A* is singular or badly scaled, and the vector *x* is an exact or approximate solution to *A*x* = 0.
cnorm	If *normin* = 'N', *cnorm* is an output argument and *cnorm*(*j*) returns the 1-norm of the off-diagonal part of the *j*-th column of *A*.
info	INTEGER. = 0: successful exit < 0: if *info* = -*k*, the *k*-th argument had an illegal value

?latrd

Reduces the first nb rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.

Syntax

```
call slatrd( uplo, n, nb, a, lda, e, tau, w, ldw )
call dlatrd( uplo, n, nb, a, lda, e, tau, w, ldw )
call clatrd( uplo, n, nb, a, lda, e, tau, w, ldw )
call zlatrd( uplo, n, nb, a, lda, e, tau, w, ldw )
```

Description

The routine ?latrd reduces *nb* rows and columns of a real symmetric or complex Hermitian matrix *A* to symmetric/Hermitian tridiagonal form by an orthogonal/unitary similarity transformation *Q' A Q*, and returns the matrices *V* and *W* which are needed to apply the transformation to the unreduced part of *A*.

If *uplo* = 'U', ?latrd reduces the last *nb* rows and columns of a matrix, of which the upper triangle is supplied;

if *uplo* = 'L', ?latrd reduces the first *nb* rows and columns of a matrix, of which the lower triangle is supplied.

This is an auxiliary routine called by ?sytrd/?hetrd.

Input Parameters

uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix *A* is stored: = 'U': upper triangular = 'L': lower triangular
n	INTEGER. The order of the matrix *A*.
nb	INTEGER. The number of rows and columns to be reduced.
a	REAL for slatrd DOUBLE PRECISION for dlatrd COMPLEX for clatrd COMPLEX*16 for zlatrd. Array, DIMENSION (*lda*, *n*). On entry, the symmetric/Hermitian matrix *A* If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *A* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *A* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *A* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *A* is not referenced.
lda	INTEGER. The leading dimension of the array *a*. *lda* ≥ (1,*n*).
LDW	INTEGER. The leading dimension of the output array *w*. *ldw* ≥ max(1,*n*).

Output Parameters

a	On exit, if *uplo* = 'U', the last *nb* columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of *A*; the elements above the diagonal with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors; if *uplo* = 'L', the first *nb* columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of *a*; the elements below the diagonal with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.
e	REAL for slatrd/clatrd DOUBLE PRECISION for dlatrd/zlatrd. If *uplo* = 'U', *e*(*n*-*nb*:*n*-1) contains the superdiagonal elements of the last *nb* columns of the reduced matrix; if *uplo* = 'L', *e*(1:*nb*) contains the subdiagonal elements of the first *nb* columns of the reduced matrix.
tau	REAL for slatrd DOUBLE PRECISION for dlatrd COMPLEX for clatrd COMPLEX*16 for zlatrd. Array, DIMENSION (*lda*, *n*). The scalar factors of the elementary reflectors, stored in *tau*(*n*-*nb*:*n*-1) if *uplo* = 'U', and in *tau*(1:*nb*) if *uplo* = 'L'.
w	REAL for slatrd DOUBLE PRECISION for dlatrd COMPLEX for clatrd COMPLEX*16 for zlatrd. Array, DIMENSION (*lda*, *n*). The *n*-by-*nb* matrix *W* required to update the unreduced part of *A*.

Application Notes

If *uplo* = 'U', the matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*n*) *H*(*n*-1) . . . *H*(*n*-*nb*+1)

Each *H*(*i*) has the form

H(*i*) = *I* - *tau***v***v*'

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(*i*:*n*) = 0 and *v*(*i*-1) = 1; *v*(1: *i*-1) is stored on exit in *a*(1: *i*-1, *i*), and *tau* in *tau*(*i*-1).

If *uplo* = 'L', the matrix *Q* is represented as a product of elementary reflectors

Q = *H*(1) *H*(2) . . . *H*(*nb*)

Each *H*(*i*) has the form *H*(*i*) = *I* - *tau***v***v*'

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1: *i*) = 0 and *v*(*i*+1) = 1; *v*(*i*+1:n) is stored on exit in *a*(*i*+1:*n*, *i*), and *tau* in *tau*(*i*).

The elements of the vectors *v* together form the *n*-by-*nb* matrix *V* which is needed, with *W*, to apply the transformation to the unreduced part of the matrix, using a symmetric/Hermitian rank-2k update of the form:

A := *A* - *VW*' - *WV*'.

The contents of *a* on exit are illustrated by the following examples with *n* = 5 and *nb* = 2:

if *uplo* = 'U':

$$\begin{bmatrix} a & a & a & v_4 & v_4 \\ & a & a & v_4 & v_5 \\ & & a & 1 & v_5 \\ & & & d & 1 \\ & & & & d \end{bmatrix}$$

if *uplo* = 'L'

$$\begin{bmatrix} d & & & & \\ 1 & d & & & \\ v_1 & 1 & a & & \\ v_1 & v_2 & a & a & \\ v_1 & v_2 & a & a & a \end{bmatrix}$$

where *d* denotes a diagonal element of the reduced matrix, *a* denotes an element of the original matrix that is unchanged, and v_i denotes an element of the vector defining *H*(*i*).

?latrs

Solves a triangular system of equations with the scale factor set to prevent overflow.

Syntax

```
call slatrs( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
call dlatrs( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
call clatrs( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
call zlatrs( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
```

Description

The routine solves one of the triangular systems

$A*x = s*b$, or $A^T*x = s*b$, or $A^H*x = s*b$ (for complex flavors)

with scaling to prevent overflow. Here A is an upper or lower triangular matrix, A^T denotes the transpose of A, A^H denotes the conjugate transpose of A, x and b are n-element vectors, and s is a scaling factor, usually less than or equal to 1, chosen so that the components of x will be less than the overflow threshold. If the unscaled problem will not cause overflow, the Level 2 BLAS routine ?trsv is called. If the matrix A is singular ($A(j,j) = 0$ for some j), then s is set to 0 and a non-trivial solution to $Ax = 0$ is returned.

Input Parameters

uplo — CHARACTER*1.
Specifies whether the matrix A is upper or lower triangular.
= 'U': Upper triangular
= 'L': Lower triangular

trans — CHARACTER*1.
Specifies the operation applied to A.
= 'N': solve $A*x = s*b$ (no transpose)
= 'T': solve $A^T*x = s*b$ (transpose)
= 'C': solve $A^H*x = s*b$ (conjugate transpose)

diag — CHARACTER*1.
Specifies whether or not the matrix A is unit triangular.
= 'N': non-unit triangular
= 'N': non-unit triangular

normin CHARACTER*1.
Specifies whether *cnorm* has been set or not.
= 'Y': *cnorm* contains the column norms on entry;
= 'N': *cnorm* is not set on entry. O
n exit, the norms will be computed and stored in *cnorm*.

n INTEGER. The order of the matrix *A*. *n* ≥ 0

a REAL for slatrs
DOUBLE PRECISION for dlatrs
COMPLEX for clatrs
COMPLEX*16 for zlatrs.
Array, DIMENSION (*lda*, *n*). Contains the triangular matrix *A*.
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of the array *a* contains the upper triangular matrix, and the strictly lower triangular part of *A* is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of the array *a* contains the lower triangular matrix, and the strictly upper triangular part of *A* is not referenced.
If *diag* = 'U', the diagonal elements of *A* are also not referenced and are assumed to be 1.

lda INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1, *n*).

x REAL for slatrs
DOUBLE PRECISION for dlatrs
COMPLEX for clatrs
COMPLEX*16 for zlatrs.
Array, DIMENSION (*n*).
On entry, the right hand side *b* of the triangular system.

cnorm REAL for slatrs/clatrs
DOUBLE PRECISION for dlatrs/zlatrs.
Array, DIMENSION (*n*).
If *normin* = 'Y', *cnorm* is an input argument and *cnorm* (*j*) contains the norm of the off-diagonal part of the *j*-th column of *A*.

If *trans* = 'N', *cnorm* (*j*) must be greater than or equal to the infinity-norm, and if *trans* = 'T' or 'C', *cnorm*(*j*) must be greater than or equal to the 1-norm.

Output Parameters

x	On exit, *x* is overwritten by the solution vector *x*.
scale	REAL for slatrs/clatrs DOUBLE PRECISION for dlatrs/zlatrs. Array, DIMENSION (*lda*, *n*). The scaling factor *s* for the triangular system as described above. If *scale* = 0, the matrix *A* is singular or badly scaled, and the vector *x* is an exact or approximate solution to *Ax* = 0.
cnorm	If *normin* = 'N', *cnorm* is an output argument and *cnorm*(*j*) returns the 1-norm of the off-diagonal part of the *j*-th column of *A*.
info	INTEGER. = 0: successful exit < 0: if *info* = -*k*, the *k*-th argument had an illegal value

Application Notes

A rough bound on *x* is computed; if that is less than overflow, ?trsv is called, otherwise, specific code is used which checks for possible overflow or divide-by-zero at every operation.

A columnwise scheme is used for solving *Ax* = *b*. The basic algorithm if *A* is lower triangular is

```
x[1:n] := b[1:n]
for j = 1, ..., n
x(j) := x(j) / A(j,j)
x[j+1:n] := x[j+1:n] - x(j)*a[j+1:n,j]
end
```

Define bounds on the components of *x* after *j* iterations of the loop:

M(*j*) = bound on *x*[1:*j*]

G(*j*) = bound on *x*[*j*+1:*n*]

Initially, let $M(0) = 0$ and $G(0) = \max\{x(i),\ i=1,...,n\}$.

Then for iteration j+1 we have

$M(j+1) \le G(j) / |a(j+1,j+1)|$

$G(j+1) \le G(j) + M(j+1)*|a[j+2:n,j+1]|$

$\le G(j)(1 + cnorm(j+1)/|a(j+1,j+1)|,$

where $cnorm(j+1)$ is greater than or equal to the infinity-norm of column j+1 of a, not counting the diagonal. Hence

$$G(j) \le G(0) \prod_{1 \le i \le j} (1 + cnorm(i)/|A(i,i)|)$$

and

$$|x(j)| \le (G(0)/|A(j,j)|) \prod_{1 \le i \le j} (1 + cnorm(i)/|A(i,i)|)$$

Since $|x(j)| \le M(j)$, we use the Level 2 BLAS routine ?trsv if the reciprocal of the largest $M(j)$, $j=1,..,n$, is larger than $\max(underflow,\ 1/overflow)$.

The bound on $x(j)$ is also used to determine when a step in the columnwise method can be performed without fear of overflow. If the computed bound is greater than a large constant, x is scaled to prevent overflow, but if the bound overflows, x is set to 0, $x(j)$ to 1, and scale to 0, and a non-trivial solution to $Ax = 0$ is found.

Similarly, a row-wise scheme is used to solve $A^Tx = b$ or $A^Hx = b$. The basic algorithm for A upper triangular is

```
for j = 1, ..., n
x(j) := ( b(j) - A[1:j-1,j]' x[1:j-1]) / A(j,j)
end
```

We simultaneously compute two bounds

$G(j) = \text{bound on } (b(i) - A[1:i-1,i]'*x[1:i-1]),\ 1 \le i \le j$

$M(j) = \text{bound on } x(i),\ 1 \le i \le j$

The initial values are $G(0) = 0, M(0) = \max\{ b(i),\ i=1,..,n\}$, and we add the constraint $G(j) \ge G(j-1)$ and $M(j) \ge M(j-1)$ for $j \ge 1$.

Then the bound on $x(j)$ is

$M(j) \le M(j-1) *(1 + cnorm(j)) / | A(j,j)|$

$$\le M(0) \prod_{1 \le i \le j} (1 + cnorm(i)/|A(i,i)|)$$

and we can safely call ?trsv if $1/M(n)$ and $1/G(n)$ are both greater than $\max(underflow, 1/overflow)$.

?latrz

Factors an upper trapezoidal matrix by means of orthogonal/unitary transformations.

Syntax

```
call slatrz( m, n, l, a, lda, tau, work )
call dlatrz( m, n, l, a, lda, tau, work )
call clatrz( m, n, l, a, lda, tau, work )
call zlatrz( m, n, l, a, lda, tau, work )
```

Description

The routine ?latrz factors the m-by-(m+l) real/complex upper trapezoidal matrix

$[A1\ A2] = [A(1:m,1:m)\ A(1:\ m,\ n-l+1:n)]$

as (R 0)* Z, by means of orthogonal/unitary transformations. Z is an (m+l)-by-(m+l) orthogonal/unitary matrix and R and $A1$ are m-by -m upper triangular matrices.

Input Parameters

m — INTEGER. The number of rows of the matrix *A*. *m* ≥ 0.

n — INTEGER. The number of columns of the matrix *A*. *n* ≥ 0.

l — INTEGER. The number of columns of the matrix *A* containing the meaningful part of the Householder vectors.
n-*m* ≥ *l* ≥ 0.

a — REAL for slatrz
DOUBLE PRECISION for dlatrz
COMPLEX for clatrz
COMPLEX*16 for zlatrz.
Array, DIMENSION (*lda*, *n*).
On entry, the leading *m*-by-*n* upper trapezoidal part of the array *a* must contain the matrix to be factorized.

lda — INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1,*m*).

work — REAL for slatrz
DOUBLE PRECISION for dlatrz
COMPLEX for clatrz
COMPLEX*16 for zlatrz.
Workspace array, DIMENSION (*m*).

Output Parameters

a — On exit, the leading *m*-by-*m* upper triangular part of *a* contains the upper triangular matrix *R*, and elements *n*-*l*+1 to *n* of the first *m* rows of *a*, with the array *tau*, represent the orthogonal/unitary matrix *Z* as a product of *m* elementary reflectors.

tau — REAL for slatrz
DOUBLE PRECISION for dlatrz
COMPLEX for clatrz
COMPLEX*16 for zlatrz.
Array, DIMENSION (*m*).
The scalar factors of the elementary reflectors.

Application Notes

The factorization is obtained by Householder's method. The k-th transformation matrix, $z(k)$, which is used to introduce zeros into the (m - k + 1)-th row of A, is given in the form

$$Z(k) = \begin{bmatrix} I & 0 \\ 0 & T(k) \end{bmatrix}$$

where

$$T(k) = \mathrm{I} - tau^{*}\, u(k)^{*} T(k)', \qquad u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

tau is a scalar and $z(k)$ is an l-element vector. tau and $z(k)$ are chosen to annihilate the elements of the k-th row of $A2$.

The scalar tau is returned in the k-th element of tau and the vector $u(k)$ in the k-th row of $A2$, such that the elements of $z(k)$ are in `a( k, l + 1 ), ..., a( k, n )`.

The elements of r are returned in the upper triangular part of $A1$.

Z is given by

`Z = Z( 1 ) Z( 2 ) ... Z( m ).`

?lauu2

Computes the product $U*U^H$ or L^H*L, where U and L are upper or lower triangular matrices (unblocked algorithm).

Syntax

```
call slauu2( uplo, n, a, lda, info )
call dlauu2( uplo, n, a, lda, info )
call clauu2( uplo, n, a, lda, info )
call zlauu2( uplo, n, a, lda, info )
```

Description

The routine ?lauu2 computes the product $U*U'$ or $L'*L$, where the triangular factor U or L is stored in the upper or lower triangular part of the array a.

If $uplo$ = 'U' or 'u' , then the upper triangle of the result is stored, overwriting the factor U in A.

If $uplo$ = 'L' or 'l', then the lower triangle of the result is stored, overwriting the factor L in A.

This is the unblocked form of the algorithm, calling BLAS Level 2 Routines.

Input Parameters

uplo	CHARACTER*1. Specifies whether the triangular factor stored in the array a is upper or lower triangular: = 'U': Upper triangular = 'L': Lower triangular
n	INTEGER. The order of the triangular factor U or L. $n \geq 0$.
a	REAL for slauu2 DOUBLE PRECISION for dlauu2 COMPLEX for clauu2 COMPLEX*16 for zlauu2. Array, DIMENSION (lda, n). On entry, the triangular factor U or L.

lda	INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1,*n*).

Output Parameters

a	On exit, if *uplo* = 'U', the upper triangle of *A* is overwritten with the upper triangle of the product *U***U*'; if *uplo* = 'L', the lower triangle of *A* is overwritten with the lower triangle of the product *L*'**L*.
info	INTEGER. = 0: successful exit < 0: if *info* = -*k*, the *k*-th argument had an illegal value

?lauum

Computes the product $U*U^H$ or L^H*L, where U and L are upper or lower triangular matrices (blocked algorithm).

Syntax

```
call slauum( uplo, n, a, lda, info )
call dlauum( uplo, n, a, lda, info )
call clauum( uplo, n, a, lda, info )
call zlauum( uplo, n, a, lda, info )
```

Description

The routine ?lauum computes the product *U***U*' or *L*'**L*, where the triangular factor *U* or *L* is stored in the upper or lower triangular part of the array *a*.

If *uplo* = 'U' or 'u', then the upper triangle of the result is stored, overwriting the factor *U* in *A*.

If *uplo* = 'L' or 'l', then the lower triangle of the result is stored, overwriting the factor *L* in *A*.

This is the blocked form of the algorithm, calling BLAS Level 3 Routines.

Input Parameters

uplo	CHARACTER*1. Specifies whether the triangular factor stored in the array *a* is upper or lower triangular: = 'U': Upper triangular = 'L': Lower triangular
n	INTEGER. The order of the triangular factor *U* or *L*. *n* ≥ 0.
a	REAL for slauum DOUBLE PRECISION for dlauum COMPLEX for clauum COMPLEX*16 for zlauum . Array, DIMENSION (*lda*, *n*). On entry, the triangular factor *U* or *L*.
lda	INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1,*n*).

Output Parameters

a	On exit, if *uplo* = 'U', the upper triangle of *A* is overwritten with the upper triangle of the product *U***U*'; if *uplo* = 'L', the lower triangle of *A* is overwritten with the lower triangle of the product *L*'**L*.
info	INTEGER. = 0: successful exit < 0: if *info* = -*k*, the *k*-th argument had an illegal value

?lazq3

Checks for deflation, computes a shift and calls dqds.

Syntax

```
call slazq3( i0, n0, z, pp, dmin, sigma, desig, qmax, nfail, iter, ndiv, ieee,
ttype, dmin1, dmin2, dn, dn1, dn2, tau )
```

```
call dlazq3( i0, n0, z, pp, dmin, sigma, desig, qmax, nfail, iter, ndiv, ieee,
ttype, dmin1, dmin2, dn, dn1, dn2, tau )
```

Description

The routine ?lazq3 checks for deflation, computes a shift (*tau*) and calls *dqds*. In case of failure, it changes shifts, and tries again until output is positive.

This routine is a thread safe version of ?lasq3 routine, which passes *ttype*, *dmin1*, *dmin2*, *dn*, *dn1*, *dn2*, and *tau* through the argument list in place of declaring them in a SAVE statement.

Input Parameters

i0 — INTEGER.
First index.

n0 — INTEGER.
Last index.

z — REAL for slasq3
DOUBLE PRECISION for dlasq3.
Array, DIMENSION (4*n). *z* holds the *qd* array.

pp — INTEGER.
pp=0 for ping, *pp*=1 for pong.

desig — REAL for slazq3
DOUBLE PRECISION for dlazq3.
Lower order part of *sigma*.

qmax — REAL for slazq3
DOUBLE PRECISION for dlazq3.
Maximum value of *q*.

ieee — LOGICAL.
Flag for IEEE or non-IEEE arithmetic (passed to the routine).

ttype — INTEGER. Shift type.

dmin1 — REAL for slazq3
DOUBLE PRECISION for dlazq3.
Minimum value of *d*, excluding *d*(*n0*). Should be 0 on entry at the first iteration and should not be modified further.

dmin2 — REAL for slazq3
DOUBLE PRECISION for dlazq3.
Minimum value of *d*, excluding *d*(*n0*) and *d*(*n0*-1). Should be 0 on entry at the first iteration and should not be modified further.

dn — REAL for slazq3

DOUBLE PRECISION for dlazq3.
Contains d(n). Should be 0 on entry at the first iteration and should not be modified further.

dn1 REAL for slazq3
DOUBLE PRECISION for dlazq3.
Contains d(n-1). Should be 0 on entry at the first iteration and should not be modified further.

dn2 REAL for slazq3
DOUBLE PRECISION for dlazq3.
Contains d(n-2). Should be 0 on entry at the first iteration and should not be modified further.

tau REAL for slazq3
DOUBLE PRECISION for dlazq3.
Shift value

Output Parameters

dmin REAL for slazq3
DOUBLE PRECISION for dlazq3.
Minimum value of *d*.

sigma REAL for slazq3
DOUBLE PRECISION for dlazq3.
Sum of shifts used in current segment.

desig Lower order part of *sigma*.

nfail INTEGER.
Number of times shift was too big.

iter INTEGER.
Number of iterations.

ndiv INTEGER.
Number of divisions.

ttype Shift type.

dmin1 Minimum value of *d*, excluding *d(n0)*.

dmin2 Minimum value of *d*, excluding *d(n0)* and *d(n0-1)*.

dn *d(n)*.

dn1 *d(n-1)*.

dn2	*d*(*n*-2).
tau	Shift value.

?lazq4

Computes an approximation to the smallest eigenvalue using values of d from the previous transform.

Syntax

```
call slazq4( i0, n0, z, pp, n0in, dmin, dmin1, dmin2, dn, dn1, dn2, tau,
ttype, g )
```

```
call dlazq4( i0, n0, z, pp, n0in, dmin, dmin1, dmin2, dn, dn1, dn2, tau,
ttype, g )
```

Description

The routine computes an approximation *tau* to the smallest eigenvalue using values of *d* from the previous transform.

This routine is a thread safe version of ?lasq4 routine, which passes *g* through the argument list in place of declaring *g* in a SAVE statement.

Input Parameters

i0	INTEGER. First index.
n0	INTEGER. Last index.
z	REAL for slazq4 DOUBLE PRECISION for dlazq4. Array, DIMENSION (4*n). *z* holds the *qd* array.
pp	INTEGER. *pp*=0 for ping, *pp*=1 for pong.
n0in	INTEGER. The value of *n0* at start of eigtest.
dmin	REAL for slazq4 DOUBLE PRECISION for dlazq4.

Minimum value of *d*.

dmin1
REAL for slazq4
DOUBLE PRECISION for dlazq4.
Minimum value of *d*, excluding *d*(*n0*).

dmin2
REAL for slazq4
DOUBLE PRECISION for dlazq4.
Minimum value of *d*, excluding *d*(*n0*) and *d*(*n0*-1).

dn
REAL for slazq4
DOUBLE PRECISION for dlazq4.
Contains *d*(*n*).

dn1
REAL for slazq4
DOUBLE PRECISION for dlazq4.
Contains *d*(*n*-1).

dn2
REAL for slazq4
DOUBLE PRECISION for dlazq4.
Contains *d*(*n*-2).

g
REAL for slazq4
DOUBLE PRECISION for dlazq4.
Shift coefficient.

Output Parameters

tau
REAL for slazq4
DOUBLE PRECISION for dlazq4.
Shift value.

ttype
INTEGER.
Shift type.

g
Shift coefficient.

?org2l/?ung2l

Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by ?geqlf *(unblocked algorithm).*

Syntax

```
call sorg2l( m, n, k, a, lda, tau, work, info )
call dorg2l( m, n, k, a, lda, tau, work, info )
call cung2l( m, n, k, a, lda, tau, work, info )
call zung2l( m, n, k, a, lda, tau, work, info )
```

Description

The routine ?org2l/?ung2l generates an m-by-n real/complex matrix Q with orthonormal columns, which is defined as the last n columns of a product of k elementary reflectors of order m:

$Q = H(k) \ldots H(2) H(1)$ as returned by ?geqlf.

Input Parameters

m	INTEGER. The number of rows of the matrix Q. $m \geq 0$.
n	INTEGER. The number of columns of the matrix Q. $m \geq n \geq 0$.
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q. $n \geq k \geq 0$.
a	REAL for sorg2l DOUBLE PRECISION for dorg2l COMPLEX for cung2l COMPLEX*16 for zung2l. Array, DIMENSION (lda,n). On entry, the ($n-k+i$)-th column must contain the vector which defines the elementary reflector $H(i)$, for i = 1,2,..., k, as returned by ?geqlf in the last k columns of its array argument A.

lda	INTEGER. The first dimension of the array *a*. *lda* ≥ max(1,*m*).
tau	REAL for sorg2l DOUBLE PRECISION for dorg2l COMPLEX for cung2l COMPLEX*16 for zung2l. Array, DIMENSION (*k*). *tau*(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by ?geqlf.
work	REAL for sorg2l DOUBLE PRECISION for dorg2l COMPLEX for cung2l COMPLEX*16 for zung2l. Workspace array, DIMENSION (*n*).

Output Parameters

a	On exit, the *m*-by-*n* matrix *Q*.
info	INTEGER. = 0: successful exit < 0: if *info* = -*i*, the *i*-th argument has an illegal value

?org2r/?ung2r

Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by ?geqrf *(unblocked algorithm).*

Syntax

```
call sorg2r( m, n, k, a, lda, tau, work, info )
call dorg2r( m, n, k, a, lda, tau, work, info )
call cung2r( m, n, k, a, lda, tau, work, info )
call zung2r( m, n, k, a, lda, tau, work, info )
```

Description

The routine ?org2r/?ung2r generates an *m*-by-*n* real/complex matrix *Q* with orthonormal columns, which is defined as the first *n* columns of a product of *k* elementary reflectors of order *m*

Q = *H*(1) *H*(2) . . . *H*(*k*)

as returned by ?geqrf.

Input Parameters

m	INTEGER. The number of rows of the matrix *Q*. *m* ≥ 0.
n	INTEGER. The number of columns of the matrix *Q*. *m* ≥ *n* ≥ 0.
k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. *n* ≥ *k* ≥ 0.
a	REAL for sorg2r DOUBLE PRECISION for dorg2r COMPLEX for cung2r COMPLEX*16 for zung2r. Array, DIMENSION (*lda*, *n*). On entry, the *i*-th column must contain the vector which defines the elementary reflector *H*(*i*), for *i* = 1,2,..., *k*, as returned by ?geqrf in the first *k* columns of its array argument *a*.
lda	INTEGER. The first DIMENSION of the array *a*. *lda* ≥ max(1,*m*).
tau	REAL for sorg2r DOUBLE PRECISION for dorg2r COMPLEX for cung2r COMPLEX*16 for zung2r. Array, DIMENSION (*k*). *tau*(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by ?geqrf.
work	REAL for sorg2r DOUBLE PRECISION for dorg2r

COMPLEX for cung2r
COMPLEX*16 for zung2r.
Workspace array, DIMENSION (*n*).

Output Parameters

a	On exit, the *m*-by-*n* matrix *Q*.
info	INTEGER. = 0: successful exit < 0: if *info* = -*i*, the *i*-th argument has an illegal value

?orgl2/?ungl2

Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by ?gelqf *(unblocked algorithm).*

Syntax

```
call sorgl2( m, n, k, a, lda, tau, work, info )
call dorgl2( m, n, k, a, lda, tau, work, info )
call cungl2( m, n, k, a, lda, tau, work, info )
call zungl2( m, n, k, a, lda, tau, work, info )
```

Description

The routine ?orgl2/?ungl2 generates a *m*-by-*n* real/complex matrix *Q* with orthonormal rows, which is defined as the first *m* rows of a product of *k* elementary reflectors of order n

Q = *H*(*k*) . . . *H*(2) *H*(1) or *Q* = *H*(*k*) ' . . . *H*(2)' *H*(1)'

as returned by ?gelqf.

Input Parameters

m	INTEGER. The number of rows of the matrix *Q*. *m* ≥ 0.
n	INTEGER. The number of columns of the matrix *Q*. *n* ≥ *m*.
k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. *m* ≥ *k* ≥ 0.

a	REAL for sorgl2 DOUBLE PRECISION for dorgl2 COMPLEX for cungl2 COMPLEX*16 for zungl2. Array, DIMENSION (*lda*, *n*). On entry, the *i*-th row must contain the vector which defines the elementary reflector *H*(*i*), for *i* = 1,2,..., *k*, as returned by ?gelqf in the first *k* rows of its array argument *a*.
lda	INTEGER. The first dimension of the array *a*. *lda* ≥ max(1,*m*).
tau	REAL for sorgl2 DOUBLE PRECISION for dorgl2 COMPLEX for cungl2 COMPLEX*16 for zungl2. Array, DIMENSION (*k*). *tau*(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by ?gelqf.
work	REAL for sorgl2 DOUBLE PRECISION for dorgl2 COMPLEX for cungl2 COMPLEX*16 for zungl2. Workspace array, DIMENSION (*m*).

Output Parameters

a	On exit, the *m*-by-*n* matrix *Q*.
info	INTEGER. = 0: successful exit < 0: if *info* = -*i*, the *i*-th argument has an illegal value.

?orgr2/?ungr2

Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by `?gerqf` *(unblocked algorithm).*

Syntax

```
call sorgr2( m, n, k, a, lda, tau, work, info )
call dorgr2( m, n, k, a, lda, tau, work, info )
call cungr2( m, n, k, a, lda, tau, work, info )
call zungr2( m, n, k, a, lda, tau, work, info )
```

Description

The routine `?orgr2/?ungr2` generates an *m*-by-*n* real matrix *Q* with orthonormal rows, which is defined as the last *m* rows of a product of *k* elementary reflectors of order *n*

Q = *H*(1) *H*(2) . . . *H*(*k*) or *Q* = *H*(1)' *H*(2)' . . . *H*(*k*)'

as returned by ?gerqf.

Input Parameters

m — INTEGER. The number of rows of the matrix *Q*. *m* ≥ 0.

n — INTEGER. The number of columns of the matrix *Q*. *n* ≥ *m*

k — INTEGER.
The number of elementary reflectors whose product defines the matrix *Q*. *m* ≥ *k* ≥ 0.

a — REAL for sorgr2
DOUBLE PRECISION for dorgr2
COMPLEX for cungr2
COMPLEX*16 for zungr2.
Array, DIMENSION (*lda*, *n*).
On entry, the (*m*- *k*+i)-th row must contain the vector which defines the elementary reflector *H*(*i*), for *i* = 1,2,..., *k*, as returned by ?gerqf in the last *k* rows of its array argument *a*.

lda	INTEGER. The first dimension of the array *a*. *lda* ≥ max(1,*m*).
tau	REAL for sorgr2 DOUBLE PRECISION for dorgr2 COMPLEX for cungr2 COMPLEX*16 for zungr2. Array, DIMENSION (*k*). *tau*(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by ?gerqf.
work	REAL for sorgr2 DOUBLE PRECISION for dorgr2 COMPLEX for cungr2 COMPLEX*16 for zungr2. Workspace array, DIMENSION (*m*).

Output Parameters

a	On exit, the *m*-by-*n* matrix *Q*.
info	INTEGER. = 0: successful exit < 0: if *info* = -*i*, the *i*-th argument has an illegal value

?orm2l/?unm2l

Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by ?geqlf *(unblocked algorithm).*

Syntax

```
call sorm2l( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dorm2l( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunm2l( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call zunm2l( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Description

The routine ?orm2l/?unm2l overwrites the general real/complex *m*-by-*n* matrix *C* with

*Q***C* if *side* = 'L' and *trans* = 'N', or

Q'**C* if *side* = 'L' and *trans* = 'T' (for real flavors) or

trans = 'C' (for complex flavors), or

*C***Q* if *side* = '*R*' and *trans* = 'N', or

*C***Q*' if *side* = 'R' and *trans* = 'T' (for real flavors) or

trans = 'C' (for complex flavors)

where *Q* is a real orthogonal or complex unitary matrix defined as the product of *k* elementary reflectors

Q = *H*(*k*) . . . *H*(2) *H*(1)

as returned by ?geqlf. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side	CHARACTER*1. = 'L': apply *Q* or *Q*' from the left = 'R': apply *Q* or *Q*' from the right
trans	CHARACTER*1. = 'N': apply *Q* (no transpose) = 'T': apply *Q*' (transpose, for real flavors) = 'C': apply *Q*' (conjugate transpose, for complex flavors)
m	INTEGER. The number of rows of the matrix *C*. *m* ≥ 0.
n	INTEGER. The number of columns of the matrix *C*. *n* ≥ 0.
k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. If *side* = 'L', *m* ≥ *k* ≥ 0; if *side* = 'R', *n* ≥ *k* ≥ 0.
a	REAL for sorm2l DOUBLE PRECISION for dorm2l COMPLEX for cunm2l COMPLEX*16 for zunm2l. Array, DIMENSION (*lda*,*k*).

The *i*-th column must contain the vector which defines the elementary reflector *H*(*i*), for *i* = 1,2,..., *k*, as returned by ?geqlf in the last *k* columns of its array argument *a*. The array *a* is modified by the routine but restored on exit.

lda INTEGER. The leading dimension of the array *a*.

If *side* = 'L', *lda* ≥ max(1, *m*)

if *side* = 'R', *lda* ≥ max(1, *n*).

tau REAL for sorm2l
DOUBLE PRECISION for dorm2l
COMPLEX for cunm2l
COMPLEX*16 for zunm2l.
Array, DIMENSION (*k*). *tau*(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by ?geqlf.

c REAL for sorm2l
DOUBLE PRECISION for dorm2l
COMPLEX for cunm2l
COMPLEX*16 for zunm2l.
Array, DIMENSION (*LDc*, *n*).
On entry, the *m*-by-*n* matrix *C*.

ldc INTEGER. The leading dimension of the array *C*. *ldc* ≥ max(1,*m*).

work REAL for sorm2l
DOUBLE PRECISION for dorm2l
COMPLEX for cunm2l
COMPLEX*16 for zunm2l.
Workspace array, DIMENSION:
(*n*) if *side* = 'L',
(*m*) if *side* = 'R'.

Output Parameters

c On exit, *c* is overwritten by *QC*,or *Q'C*, or *CQ'*, or *CQ*.

info INTEGER.
= 0: successful exit
< 0: if *info* = -*i*, the *i*-th argument had an illegal value

?orm2r/?unm2r

Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by `?geqrf` *(unblocked algorithm).*

Syntax

```
call sorm2r( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dorm2r( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunm2r( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call zunm2r( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Description

The routine `?orm2r/?unm2r` overwrites the general real/complex *m*-by-*n* matrix *C* with

*Q***C* if *side* = 'L' and *trans* = 'N', or

Q'**C* if *side* = 'L' and *trans* = 'T' (for real flavors) or

trans = 'C' (for complex flavors), or

*C***Q* if *side* = 'R' and *trans* = 'N', or

*C***Q*' if *side* = 'R' and *trans* = 'T' (for real flavors) or

trans = 'C' (for complex flavors)

where *Q* is a real orthogonal or complex unitary matrix defined as the product of *k* elementary reflectors

Q = *H*(1) *H*(2) . . . *H*(*k*)

as returned by `?geqrf`. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side	CHARACTER*1. = 'L': apply *Q* or *Q*' from the left = 'R': apply *Q* or *Q*' from the right
trans	CHARACTER*1. = 'N': apply *Q* (no transpose) = 'T': apply *Q*' (transpose, for real flavors)

= 'C': apply Q' (conjugate transpose, for complex flavors)

m INTEGER. The number of rows of the matrix C. $m \geq 0$.

n INTEGER. The number of columns of the matrix C. $n \geq 0$.

k INTEGER. The number of elementary reflectors whose product defines the matrix Q.

If *side* = 'L', $m \geq k \geq 0$;

if *side* = 'R', $n \geq k \geq 0$.

a REAL for sorm2r
DOUBLE PRECISION for dorm2r
COMPLEX for cunm2r
COMPLEX*16 for zunm2r.
Array, DIMENSION (*lda*,*k*).
The i-th column must contain the vector which defines the elementary reflector $H(i)$, for $i = 1,2,\ldots, k$, as returned by ?geqrf in the first k columns of its array argument *a*.
The array *a* is modified by the routine but restored on exit.

lda INTEGER. The leading dimension of the array *a*.

If *side* = 'L', *lda* ≥ max(1, *m*);

if *side* = 'R', *lda* ≥ max(1, *n*).

tau REAL for sorm2r
DOUBLE PRECISION for dorm2r
COMPLEX for cunm2r
COMPLEX*16 for zunm2r.
Array, DIMENSION (*k*).
tau(i) must contain the scalar factor of the elementary reflector $H(i)$, as returned by ?geqrf.

c REAL for sorm2r
DOUBLE PRECISION for dorm2r
COMPLEX for cunm2r
COMPLEX*16 for zunm2r.
Array, DIMENSION (*LDc*, *n*).
On entry, the m-by-n matrix C.

ldc INTEGER. The leading dimension of the array *c*. *ldc* ≥ max(1,*m*).

work	REAL for sorm2r DOUBLE PRECISION for dorm2r COMPLEX for cunm2r COMPLEX*16 for zunm2r. Workspace array, DIMENSION (*n*) if *side* = 'L', (*m*) if *side* = 'R'.

Output Parameters

c	On exit, *c* is overwritten by *QC*, or *Q'C*, or *CQ'*, or *CQ*.
info	INTEGER. = 0: successful exit < 0: if *info* = -i, the *i*-th argument had an illegal value

?orml2/?unml2

Multiplies a general matrix by the orthogonal/unitary matrix from a LQ factorization determined by ?gelqf *(unblocked algorithm).*

Syntax

```
call sorml2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dorml2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunml2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call zunml2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Description

The routine ?orml2/?unml2 overwrites the general real/complex *m*-by-*n* matrix *C* with

*Q*C* if *side* = 'L' and *trans* = 'N', or

*Q'*C* if *side* = 'L' and *trans* = 'T' (for real flavors) or

trans = 'C' (for complex flavors), or

*C*Q* if *side* = 'R' and *trans* = 'N', or

*C*Q'* if *side* = 'R' and *trans* = 'T' (for real flavors) or

trans = 'C' (for complex flavors)

where *Q* is a real orthogonal or complex unitary matrix defined as the product of *k* elementary reflectors

Q = *H*(*k*) . . . *H*(2) *H*(1) or *Q* = *H*(*k*)' . . . *H*(2)' *H*(1)'

as returned by ?gelqf. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side	CHARACTER*1. = 'L': apply *Q* or *Q*' from the left = 'R': apply *Q* or *Q*' from the right
trans	CHARACTER*1. = 'N': apply *Q* (no transpose) = 'T': apply *Q*' (transpose, for real flavors) = 'C': apply *Q*' (conjugate transpose, for complex flavors)
m	INTEGER. The number of rows of the matrix *C*. *m* ≥ 0.
n	INTEGER. The number of columns of the matrix *C*. *n* ≥ 0.
k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. If *side* = 'L', *m* ≥ *k* ≥ 0; if *side* = 'r', *n* ≥ *k* ≥ 0.
a	REAL for sorml2 DOUBLE PRECISION for dorml2 COMPLEX for cunml2 COMPLEX*16 for zunml2. Array, DIMENSION (*lda*, *m*) if *side* = 'L', (*lda*, *n*) if *side* = 'R' The *i*-th row must contain the vector which defines the elementary reflector *H*(*i*), for *i* = 1,2,..., *k*, as returned by ?gelqf in the first *k* rows of its array argument *a*. The array *a* is modified by the routine but restored on exit.
lda	INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1,*k*).

tau REAL for sorml2
DOUBLE PRECISION for dorml2
COMPLEX for cunml2
COMPLEX*16 for zunml2.
Array, DIMENSION (*k*).
tau(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by ?gelqf.

c REAL for sorml2
DOUBLE PRECISION for dorml2
COMPLEX for cunml2
COMPLEX*16 for zunml2.
Array, DIMENSION (*ldc*, *n*) On entry, the *m*-by-*n* matrix *C*.

ldc INTEGER. The leading dimension of the array *c*. *ldc* ≥ max(1,*m*).

work REAL for sorml2
DOUBLE PRECISION for dorml2
COMPLEX for cunml2
COMPLEX*16 for zunml2.
Workspace array, DIMENSION
(*n*) if *side* = 'L',
(*m*) if *side* = 'R'

Output Parameters

c On exit, *c* is overwritten by *QC*, or *Q'C*, or *CQ*', or *CQ*.

info INTEGER.
= 0: successful exit
< 0: if *info* = -*i*, the *i*-th argument had an illegal value

?ormr2/?unmr2

Multiplies a general matrix by the orthogonal/unitary matrix from a RQ factorization determined by `?gerqf` *(unblocked algorithm).*

Syntax

```
call sormr2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dormr2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunmr2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call zunmr2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Description

The routine `?ormr2/?unmr2` overwrites the general real/complex *m*-by-*n* matrix *C* with

*Q*C* if *side* = 'L' and *trans* = 'N', or

*Q'*C* if *side* = 'L' and *trans* = 'T' (for real flavors) or

trans = 'C' (for complex flavors), or

*C*Q* if *side* = 'R' and *trans* = 'N', or

*C*Q'* if *side* = 'R' and *trans* = 'T' (for real flavors) or

trans = 'C' (for complex flavors)

where *Q* is a real orthogonal or complex unitary matrix defined as the product of *k* elementary reflectors

Q = *H*(1) *H*(2) . . . *H*(*k*) or *Q* = *H*(1)' *H*(2)' . . . *H*(*k*)'

as returned by `?gerqf`. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side — CHARACTER*1.
= 'L': apply *Q* or *Q'* from the left
= 'R': apply *Q* or *Q'* from the right

trans — CHARACTER*1.
= 'N': apply *Q* (no transpose)
= 'T': apply *Q'* (transpose, for real flavors)

= 'C': apply Q' (conjugate transpose, for complex flavors)

m INTEGER. The number of rows of the matrix C. $m \geq 0$.

n INTEGER. The number of columns of the matrix C. $n \geq 0$.

k INTEGER. The number of elementary reflectors whose product defines the matrix Q.

If $side$ = 'L', $m \geq k \geq 0$;

if $side$ = 'r', $n \geq k \geq 0$.

a REAL for sormr2
DOUBLE PRECISION for dormr2
COMPLEX for cunmr2
COMPLEX*16 for zunmr2.
Array, DIMENSION
(lda, m) if $side$ = 'L',
(lda, n) if $side$ = 'r'
The i-th row must contain the vector which defines the elementary reflector $H(i)$, for $i = 1,2,\ldots,k$, as returned by ?gerqf in the last k rows of its array argument a. The array a is modified by the routine but restored on exit.

lda INTEGER.

The leading dimension of the array a. $lda \geq \max(1,k)$.

tau REAL for sormr2
DOUBLE PRECISION for dormr2
COMPLEX for cunmr2
COMPLEX*16 for zunmr2.
Array, DIMENSION (k).
$tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by ?gerqf.

c REAL for sormr2
DOUBLE PRECISION for dormr2
COMPLEX for cunmr2
COMPLEX*16 for zunmr2.
Array, DIMENSION (ldc, n).
On entry, the m-by-n matrix C.

ldc	INTEGER. The leading dimension of the array *c*. *ldc* ≥ max(1,*m*).
work	REAL for sormr2 DOUBLE PRECISION for dormr2 COMPLEX for cunmr2 COMPLEX*16 for zunmr2. Workspace array, DIMENSION (*n*) if *side* = 'L', (*m*) if *side* = 'R'

Output Parameters

c	On exit, *c* is overwritten by *QC*, or *Q'C*, or *CQ*', or *CQ*.
info	INTEGER. = 0: successful exit < 0: if *info* = -*i*, the *i*-th argument had an illegal value

?ormr3/?unmr3

Multiplies a general matrix by the orthogonal/unitary matrix from a RZ factorization determined by ?tzrzf *(unblocked algorithm).*

Syntax

```
call sormr3( side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info )
call dormr3( side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info )
call cunmr3( side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info )
call zunmr3( side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info )
```

Description

The routine ?ormr3/?unmr3 overwrites the general real/complex *m*-by-*n* matrix *C* with

*Q*C* if *side* = 'L' and *trans* = 'N', or

Q'*C* if *side* = 'L' and *trans* = 'T' (for real flavors) or

trans = 'C' (for complex flavors), or

*C***Q* if *side* = 'R' and *trans* = 'N', or

*C***Q*' if *side* = 'R' and *trans* = 'T' (for real flavors) or

trans = 'C' (for complex flavors)

where *Q* is a real orthogonal or complex unitary matrix defined as the product of *k* elementary reflectors

Q = *H*(1) *H*(2) . . . *H*(*k*)

as returned by ?tzrzf. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side	CHARACTER*1. = 'L': apply *Q* or *Q*' from the left = 'R': apply *Q* or *Q*' from the right
trans	CHARACTER*1. = 'N': apply *Q* (no transpose) = 'T': apply *Q*' (transpose, for real flavors) = 'C': apply *Q*' (conjugate transpose, for complex flavors)
m	INTEGER. The number of rows of the matrix *C*. *m* ≥ 0.
n	INTEGER. The number of columns of the matrix *C*. *n* ≥ 0.
k	INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. If *side* = 'L', *m* ≥ *k* ≥ 0; if *side* = 'r' , *n* ≥ *k* ≥ 0.
l	INTEGER. The number of columns of the matrix *A* containing the meaningful part of the Householder reflectors. If *side* = 'L', *m* ≥ *l* ≥ 0, if *side* = 'R', *n* ≥ *l* ≥ 0.
a	REAL for sormr3 DOUBLE PRECISION for dormr3 COMPLEX for cunmr3 COMPLEX*16 for zunmr3. Array, DIMENSION (*lda*, *m*) if *side* = 'L',

(*lda*, *n*) if *side* = 'r'
The *i*-th row must contain the vector which defines the elementary reflector *H*(*i*), for *i* = 1,2,...,*k*, as returned by ?tzrzf in the last *k* rows of its array argument *a*. The array *a* is modified by the routine but restored on exit.

lda — INTEGER.
The leading dimension of the array *a*. *lda* ≥ max(1,*k*).

tau — REAL for sormr3
DOUBLE PRECISION for dormr3
COMPLEX for cunmr3
COMPLEX*16 for zunmr3.
Array, DIMENSION (*k*).
tau(*i*) must contain the scalar factor of the elementary reflector H(*i*), as returned by ?tzrzf.

c — REAL for sormr3
DOUBLE PRECISION for dormr3
COMPLEX for cunmr3
COMPLEX*16 for zunmr3.
Array, DIMENSION (*ldc*, *n*).
On entry, the *m*-by-*n* matrix *C*.

ldc — INTEGER. The leading dimension of the array *C*. *ldc* ≥ max(1,*m*).

work — REAL for sormr3
DOUBLE PRECISION for dormr3
COMPLEX for cunmr3
COMPLEX*16 for zunmr3.
Workspace array, DIMENSION
(*n*) if *side* = 'L',
(*m*) if *side* = 'R'.

Output Parameters

c — On exit, *c* is overwritten by *QC*, or *Q'C*, or *CQ*', or *CQ*.

info — INTEGER.
= 0: successful exit
< 0: if *info* = -*i*, the *i*-th argument had an illegal value

?pbtf2

Computes the Cholesky factorization of a symmetric/ Hermitian positive-definite band matrix (unblocked algorithm).

Syntax

```
call spbtf2( uplo, n, kd, ab, ldab, info )
call dpbtf2( uplo, n, kd, ab, ldab, info )
call cpbtf2( uplo, n, kd, ab, ldab, info )
call zpbtf2( uplo, n, kd, ab, ldab, info )
```

Description

The routine computes the Cholesky factorization of a real symmetric or complex Hermitian positive definite band matrix `A`.

The factorization has the form

`A = U'*U`, if `uplo = 'U'`, or

`A = L*L'`, if `uplo = 'L'`,

where `U` is an upper triangular matrix, `U'` is the transpose of `U`, and `L` is lower triangular. This is the unblocked version of the algorithm, calling BLAS Level 2 Routines.

Input Parameters

uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix `A` is stored: = 'U': upper triangular = 'L': lower triangular
n	INTEGER. The order of the matrix `A`. $n \geq 0$.
kd	INTEGER. The number of super-diagonals of the matrix `A` if `uplo = 'U'`, or the number of sub-diagonals if `uplo = 'L'`. $kd \geq 0$.
ab	REAL for spbtf2

DOUBLE PRECISION for dpbtf2
COMPLEX for cpbtf2
COMPLEX*16 for zpbtf2.
Array, DIMENSION (*ldab*, *n*).
On entry, the upper or lower triangle of the symmetric/Hermitian band matrix *A*, stored in the first *kd*+1 rows of the array. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:
if *uplo* = 'U', *ab*(*kd*+1+*i* –*j*,*j*) = *A*(*i*, *j* for max(1, *j*-*kd*) ≤ *i* ≤ *j*;
if *uplo* = 'L', *ab*(1+*i* –*j*,*j*) = *A*(*i*, *j* for *j* ≤ *i* ≤ min(*n*, *j*+*kd*).

ldab — INTEGER. The leading dimension of the array *ab*. *ldab* ≥ *kd*+1.

Output Parameters

ab — On exit, If *info* = 0, the triangular factor *U* or *L* from the Cholesky factorization *A* = *U*'**U* or *A* = *L***L*' of the band matrix *A*, in the same storage format as *A*.

info — INTEGER.
= 0: successful exit
< 0: if *info* = –*k*, the *k*-th argument had an illegal value
> 0: if *info* = *k*, the leading minor of order *k* is not positive definite, and the factorization could not be completed.

?potf2

Computes the Cholesky factorization of a symmetric/Hermitian positive-definite matrix (unblocked algorithm).

Syntax

```
call spotf2( uplo, n, a, lda, info )
call dpotf2( uplo, n, a, lda, info )
call cpotf2( uplo, n, a, lda, info )
call zpotf2( uplo, n, a, lda, info )
```

Description

The routine ?potf2 computes the Cholesky factorization of a real symmetric or complex Hermitian positive definite matrix *A*. The factorization has the form

A = *U*'**U*, if *uplo* = 'U', or

A = *L***L*', if *uplo* = 'L',

where *U* is an upper triangular matrix and *L* is lower triangular.

This is the unblocked version of the algorithm, calling BLAS Level 2 Routines

Input Parameters

uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix *A* is stored. = 'U': upper triangular = 'L': lower triangular
n	INTEGER. The order of the matrix *A*. *n* ≥ 0.
a	REAL for spotf2 DOUBLE PRECISION or dpotf2 COMPLEX for cpotf2 COMPLEX*16 for zpotf2. Array, DIMENSION (*lda*, *n*). On entry, the symmetric/Hermitian matrix *A*.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *A* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *A* is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *A* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *A* is not referenced.

lda — INTEGER. The leading dimension of the array *a*.
lda ≥ max(1,*n*).

Output Parameters

a — On exit, If *info* = 0, the factor *U* or *L* from the Cholesky factorization *A* = *U*'**U*, or *A* = *L***L*'.

info — INTEGER.
= 0: successful exit
< 0: if *info* = -*k*, the *k*-th argument had an illegal value
> 0: if *info* = *k*, the leading minor of order *k* is not positive definite, and the factorization could not be completed.

?ptts2

*Solves a tridiagonal system of the form A*X=B using the L*D*L^H factorization computed by* ?pttrf.

Syntax

```
call sptts2( n, nrhs, d, e, b, ldb )
call dptts2( n, nrhs, d, e, b, ldb )
call cptts2( iuplo, n, nrhs, d, e, b, ldb )
call zptts2( iuplo, n, nrhs, d, e, b, ldb )
```

Description

The routine ?ptts2 solves a tridiagonal system of the form

*A*X = B*

Real flavors sptts2/dptts2 use the *L D L'* factorization of *A* computed by spttrf/dpttrf, and complex flavors cptts2/zptts2 use the *U' D U* or *L D L'* factorization of *A* computed by cpttrf/zpttrf.

D is a diagonal matrix specified in the vector *d*, *U* (or *L*) is a unit bidiagonal matrix whose superdiagonal (subdiagonal) is specified in the vector *e*, and *X* and *B* are *n*-by-*nrhs* matrices.

Input Parameters

iuplo — INTEGER. Used with complex flavors only.
Specifies the form of the factorization and whether the vector *e* is the superdiagonal of the upper bidiagonal factor *U* or the subdiagonal of the lower bidiagonal factor *L*.
= 1: *A = U'*D*U*, *e* is the superdiagonal of *U*;
= 0: *A = L*D*L'*, *e* is the subdiagonal of *L*

n — INTEGER. The order of the tridiagonal matrix *A*. *n* ≥ 0.

nrhs — INTEGER. The number of right hand sides, that is, the number of columns of the matrix *B*. nrhs ≥ 0.

d — REAL for sptts2/cptts2
DOUBLE PRECISION for dptts2/zptts2.
Array, DIMENSION (*n*).
The *n* diagonal elements of the diagonal matrix *D* from the factorization of *A*.

e — REAL for sptts2
DOUBLE PRECISION for dptts2
COMPLEX for cptts2
COMPLEX*16 for zptts2.
Array, DIMENSION (*n*-1).
Contains the (*n*-1) subdiagonal elements of the unit bidiagonal factor *L* from the *LDL'* factorization of *A* (for real flavors, or for complex flavors when *iuplo = 0*).
For complex flavors when *iuplo = 1*, *e* contains the (*n*-1) superdiagonal elements of the unit bidiagonal factor *U* from the factorization *A = U'*D*U*.

B — REAL for sptts2/cptts2
DOUBLE PRECISION for dptts2/zptts2.
Array, DIMENSION (*ldb*, *nrhs*).

	On entry, the right hand side vectors *B* for the system of linear equations.
ldb	INTEGER. The leading dimension of the array *B*. *ldb* ≥ max(1,*n*).

Output Parameters

b	On exit, the solution vectors, *X*.

?rscl

Multiplies a vector by the reciprocal of a real scalar.

Syntax

```
call srscl( n, sa, sx, incx )
call drscl( n, sa, sx, incx )
call csrscl( n, sa, sx, incx )
call zdrscl( n, sa, sx, incx )
```

Description

The routine ?rscl multiplies an *n*-element real/complex vector *x* by the real scalar 1/*a*. This is done without overflow or underflow as long as the final result *x*/*a* does not overflow or underflow.

Input Parameters

n	INTEGER. The number of components of the vector *x*.
sa	REAL for srscl/csrscl DOUBLE PRECISION for drscl/zdrscl. The scalar a which is used to divide each component of the vector *x*. sa must be ≥ 0, or the subroutine will divide by zero.
sx	REAL for srscl DOUBLE PRECISION for drscl COMPLEX for csrscl COMPLEX*16 for zdrscl.

	Array, DIMENSION (1+(*n*-1)*abs(*incx*)). The *n*-element vector *x*.
incx	INTEGER. The increment between successive values of the vector *sx*. If *incx* > 0, *sx*(1) = *x*(*1*) and *sx*(1+(i-1)*incx) = *x*(*i*), 1< *i* ≤ *n*.

Output Parameters

sx	On exit, the result *x*/*a*.

?sygs2/?hegs2

Reduces a symmetric/Hermitian definite generalized eigenproblem to standard form, using the factorization results obtained from ?potrf *(unblocked algorithm).*

Syntax

```
call ssygs2( itype, uplo, n, a, lda, b, ldb, info )
call dsygs2( itype, uplo, n, a, lda, b, ldb, info )
call chegs2( itype, uplo, n, a, lda, b, ldb, info )
call zhegs2( itype, uplo, n, a, lda, b, ldb, info )
```

Description

The routine ?sygs2/?hegs2 reduces a real symmetric-definite or a complex Hermitian-definite generalized eigenproblem to standard form.

If *itype* = 1, the problem is

*A*x* = λ*B*x*,

and *A* is overwritten by inv(*U*')* *A*inv(*U*) or inv(*L*)*A*inv(*L*').

If *itype* = 2 or 3, the problem is *A*B*x* = λ*x*, or *B*A*x* = λ*x*,

and *A* is overwritten by *U*A*U*' or *L*'*A*L*. *B* must have been previously factorized as *U*'*U* or *L*L*' by ?potrf.

Input Parameters

itype	INTEGER. = 1: compute inv(*U*')***A**inv(*U*), or inv(*L*)***A**inv(*L*'); = 2 or 3: compute *U***A***U*', or *L*'**A***L*.
uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix *A* is stored, and how *B* has been factorized. = 'U': upper triangular = 'L': lower triangular
n	INTEGER. The order of the matrices *A* and *B*. *n* ≥ 0.
a	REAL for ssygs2 DOUBLE PRECISION for dsygs2 COMPLEX for chegs2 COMPLEX*16 for zhegs2. Array, DIMENSION (*lda*, *n*). On entry, the symmetric/Hermitian matrix *A*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *A* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *A* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *A* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *A* is not referenced.
lda	INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1,*n*).
b	REAL for ssygs2 DOUBLE PRECISION for dsygs2 COMPLEX for chegs2 COMPLEX*16 for zhegs2. Array, DIMENSION (*ldb*, *n*). The triangular factor from the Cholesky factorization of *B* as returned by ?potrf.
ldb	INTEGER. The leading dimension of the array *B*. *ldb* ≥ max(1,*n*).

Output Parameters

a	On exit, If *info* = 0, the transformed matrix, stored in the same format as *A*.
info	INTEGER. = 0: successful exit. < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?sytd2/?hetd2

Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation(unblocked algorithm).

Syntax

```
call ssytd2( uplo, n, a, lda, d, e, tau, info )
call dsytd2( uplo, n, a, lda, d, e, tau, info )
call chetd2( uplo, n, a, lda, d, e, tau, info )
call zhetd2( uplo, n, a, lda, d, e, tau, info )
```

Description

The routine ?sytd2/?hetd2 reduces a real symmetric/complex Hermitian matrix *A* to real symmetric tridiagonal form *T* by an orthogonal/unitary similarity transformation: *Q'*A*Q* = *T*.

Input Parameters

uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix *A* is stored: = 'U': upper triangular = 'L': lower triangular
n	INTEGER. The order of the matrix *A*. *n* ≥ 0.
a	REAL for ssytd2 DOUBLE PRECISION for dsytd2

COMPLEX for chetd2
COMPLEX*16 for zhetd2.
Array, DIMENSION (*lda*, *n*).
On entry, the symmetric/Hermitian matrix *A*.
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *A* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *A* is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *A* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *A* is not referenced.

lda

INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1,*n*).

Output Parameters

a

On exit, if *uplo* = 'U', the diagonal and first superdiagonal of *A* are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements above the first superdiagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors;
if *uplo* = 'L', the diagonal and first subdiagonal of *A* are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements below the first subdiagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.

d

REAL for ssytd2/chetd2
DOUBLE PRECISION for dsytd2/zhetd2.
Array, DIMENSION (*n*).
The diagonal elements of the tridiagonal matrix *T*:
d(*i*) = *a*(*i*,*i*).

e

REAL for ssytd2/chetd2
DOUBLE PRECISION for dsytd2/zhetd2.
Array, DIMENSION (*n*-1).
The off-diagonal elements of the tridiagonal matrix *T*:
e(*i*) = *a*(*i*,*i*+1) if *uplo* = 'U',
e(*i*) = *a*(*i*+1,*i*) if *uplo* = 'L'.

tau	REAL for ssytd2 DOUBLE PRECISION for dsytd2 COMPLEX for chetd2 COMPLEX*16 for zhetd2. Array, DIMENSION (*n*-1). The scalar factors of the elementary reflectors .
info	INTEGER. = 0: successful exit < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?sytf2

Computes the factorization of a real/complex symmetric indefinite matrix, using the diagonal pivoting method (unblocked algorithm).

Syntax

```
call ssytf2( uplo, n, a, lda, ipiv, info )
call dsytf2( uplo, n, a, lda, ipiv, info )
call csytf2( uplo, n, a, lda, ipiv, info )
call zsytf2( uplo, n, a, lda, ipiv, info )
```

Description

The routine ?sytf2 computes the factorization of a real/complex symmetric matrix *A* using the Bunch-Kaufman diagonal pivoting method:

A = *U***D***U*' or *A* = *L***D***L*'

where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, *U*' is the transpose of *U*, and *D* is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling BLAS Level 2 Routines.

Input Parameters

uplo	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored = 'U': upper triangular

= 'L': lower triangular

n INTEGER. The order of the matrix *A*. *n* ≥ 0.

a REAL for ssytf2
DOUBLE PRECISION for dsytf2
COMPLEX for csytf2
COMPLEX*16 for zsytf2.
Array, DIMENSION (*lda*, *n*).
On entry, the symmetric matrix *A*.
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *A* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *A* is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *A* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *A* is not referenced.

lda INTEGER.
The leading dimension of the array *a*. *lda* ≥ max(1,*n*).

Output Parameters

a On exit, the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L*.

ipiv INTEGER.
Array, DIMENSION (*n*).
Details of the interchanges and the block structure of *D*
If *ipiv*(*k*) > 0, then rows and columns *k* and *ipiv*(*k*) were interchanged and *D*(*k*,*k*) is a 1-by-1 diagonal block.
If *uplo* = 'U' and *ipiv*(*k*) = *ipiv*(*k*–1) < 0, then rows and columns *k*-1 and -*ipiv*(*k*) were interchanged and *D*(*k*,*k*) is a 2-by-2 diagonal block. If *uplo* = 'L' and *ipiv*(*k*) = *ipiv*(*k*+1)< 0, then rows and columns *k*+1 and -*ipiv*(*k*) were interchanged and *D*(*k*:*k*+1,*k*:*k*+1) is a 2-by-2 diagonal block.

info INTEGER.
= 0: successful exit
< 0: if *info* = -*k*, the *k*–th argument had an illegal value

> 0: if *info* = *k*, *D*(*k*,*k*) is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?hetf2

Computes the factorization of a complex Hermitian matrix, using the diagonal pivoting method (unblocked algorithm).

Syntax

```
call chetf2( uplo, n, a, lda, ipiv, info )
call zhetf2( uplo, n, a, lda, ipiv, info )
```

Description

The routine computes the factorization of a complex Hermitian matrix *A* using the Bunch-Kaufman diagonal pivoting method:

A = *U***D***U*' or *A* = *L***D***L*'

where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, *U*' is the conjugate transpose of *U*, and *D* is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling BLAS Level 2 Routines.

Input Parameters

uplo — CHARACTER*1.
Specifies whether the upper or lower triangular part of the Hermitian matrix *A* is stored:
= 'U': Upper triangular
= 'L': Lower triangular

n — INTEGER. The order of the matrix *A*. *n* ≥ 0.

A — COMPLEX for chetf2
COMPLEX*16 for zhetf2.
Array, DIMENSION (*lda*, *n*).
On entry, the Hermitian matrix *A*.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *A* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *A* is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *A* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *A* is not referenced.

lda — INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1,*n*).

Output Parameters

a — On exit, the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L*.

ipiv — INTEGER. Array, DIMENSION (*n*).
Details of the interchanges and the block structure of *D*
If *ipiv*(*k*) > 0, then rows and columns *k* and *ipiv*(*k*) were interchanged and *D*(*k*,*k*) is a 1-by-1 diagonal block.
If *uplo* = 'U' and *ipiv*(*k*) = *ipiv*(*k*–1) < 0, then rows and columns *k*-1 and -*ipiv*(*k*) were interchanged and *D*(*k*-1:*k*,*k*-1:*k*) is a 2-by-2 diagonal block.
If *uplo* = 'L' and *ipiv*(*k*) = *ipiv*(*k*+1) < 0, then rows and columns *k*+1 and -*ipiv*(*k*) were interchanged and *D*(*k*:*k*+1, *k*:*k*+1) is a 2-by-2 diagonal block.

info — INTEGER.
= 0: successful exit
< 0: if *info* = –*k*, the *k*-th argument had an illegal value
> 0: if *info* = *k*, *D*(*k*,*k*) is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?tgex2

Swaps adjacent diagonal blocks in an upper (quasi) triangular matrix pair by an orthogonal/unitary equivalence transformation.

Syntax

```
call stgex2( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, n1, n2,
work, lwork, info )
call dtgex2( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, n1, n2,
work, lwork, info )
call ctgex2( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, info )
call ztgex2( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, info )
```

Description

The real routines stgex2/dtgex2 swap adjacent diagonal blocks (A11, B11) and (A22, B22) of size 1-by-1 or 2-by-2 in an upper (quasi) triangular matrix pair (A, B) by an orthogonal equivalence transformation. (A, B) must be in generalized real Schur canonical form (as returned by sgges/dgges), that is, A is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks. B is upper triangular.

The complex routines ctgex2/ztgex2 swap adjacent diagonal 1-by-1 blocks (A11, B11) and (A22, B22) in an upper triangular matrix pair (A, B) by an unitary equivalence transformation.

(A, B) must be in generalized Schur canonical form, that is, A and B are both upper triangular.

All routines optionally update the matrices Q and Z of generalized Schur vectors:

```
Q(in) *A(in)* Z(in)' = Q(out)*A(out)* Z(out)'
Q(in)* B(in)* Z(in)' = Q(out)* B(out)* Z(out)'
```

Input Parameters

wantq	LOGICAL. If *wantq* = .TRUE. : update the left transformation matrix Q; If *wantq* = .FALSE. : do not update Q.
wantz	LOGICAL.

	If *wantz* = .TRUE. : update the right transformation matrix *Z*; If *wantz* = .FALSE.: do not update *Z*.
n	INTEGER. The order of the matrices *A* and *B*. *n* ≥ 0.
a, *b*	REAL for stgex2 DOUBLE PRECISION for dtgex2 COMPLEX for ctgex2 COMPLEX*16 for ztgex2. Arrays, DIMENSION (*lda*, *n*) and (*ldb*, *n*), respectively. On entry, the matrices *A* and *B* in the pair (*A*, *B*).
lda	INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1,*n*).
ldb	INTEGER. The leading dimension of the array *b*. *ldb* ≥ max(1,*n*).
Q, *z*	REAL for stgex2 DOUBLE PRECISION for dtgex2 COMPLEX for ctgex2 COMPLEX*16 for ztgex2. Arrays, DIMENSION (*ldq*, *n*) and (*ldz*, *n*), respectively. On entry, if *wantq* = .TRUE., *q* contains the orthogonal/unitary matrix *Q*, and if *wantz* = .TRUE., *z* contains the orthogonal/unitary matrix *Z*.
ldq	INTEGER. The leading dimension of the array *Q*. *ldq* ≥ 1. If *wantq* = .TRUE., *ldq* ≥ *n*.
ldz	INTEGER. The leading dimension of the array *z*. *ldz* ≥ 1. If *wantz* = .TRUE., *ldz* ≥ *n*.
j1	INTEGER. The index to the first block (*A*11, *B*11). 1 ≤ *j1* ≤ *n*.
n1	INTEGER. Used with real flavors only. The order of the first block (*A*11, *B*11). *n1* = 0, 1 or 2.
n2	INTEGER. Used with real flavors only. The order of the second block (*A*22, *B*22). *n2* = 0, 1 or 2.
work	REAL for stgex2 DOUBLE PRECISION for dtgex2.

Workspace array, DIMENSION (max(1,*lwork*)). Used with real flavors only.

lwork INTEGER. The dimension of the array *work*.

lwork ≥ max(*n**(*n2*+*n1*), 2*(*n2*+*n1*)^2)

Output Parameters

a On exit, the updated matrix *A*.

B On exit, the updated matrix *B*.

Q On exit, the updated matrix *Q*.
Not referenced if *wantq* = .FALSE..

z On exit, the updated matrix *Z*.
Not referenced if *wantz* = .FALSE..

info INTEGER.
=0: Successful exit For stgex2/dtgex2: If *info* = 1, the transformed matrix (*A*, *B*) would be too far from generalized Schur form; the blocks are not swapped and (*A*, *B*) and (*Q*, *Z*) are unchanged. The problem of swapping is too ill-conditioned. If *info* = -16: *lwork* is too small. Appropriate value for *lwork* is returned in *work*(1).
For ctgex2/ztgex2:
If *info* = 1, the transformed matrix pair (*A*, *B*) would be too far from generalized Schur form; the problem is ill-conditioned.

?tgsy2

Solves the generalized Sylvester equation (unblocked algorithm).

Syntax

```
call stgsy2( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f,
ldf, scale, rdsum, rdscal, iwork, pq, info )
```

```
call dtgsy2( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f,
ldf, scale, rdsum, rdscal, iwork, pq, info )
```

```
call ctgsy2( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f,
ldf, scale, rdsum, rdscal, iwork, pq, info )
```

```
call ztgsy2( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f,
ldf, scale, rdsum, rdscal, iwork, pq, info )
```

Description

The routine `?tgsy2` solves the generalized Sylvester equation:

$$AR - LB = \text{scale} * C \qquad (1)$$
$$DR - LE = \text{scale} * F,$$

using Level 1 and 2 BLAS, where R and L are unknown m-by-n matrices, (A, D), (B, E) and (C, F) are given matrix pairs of size m-by -m, n-by-n and m-by-n, respectively. For `stgsy2`/`dtgsy2`, pairs (A, D) and (B, E) must be in generalized Schur canonical form, that is, A, B are upper quasi triangular and D, E are upper triangular. For `ctgsy2`/`ztgsy2`, matrices A, B, D and E are upper triangular (that is, (A, D) and (B, E) in generalized Schur form).

The solution (r, L) overwrites (C, F). $0 \le scale \le 1$ is an output scaling factor chosen to avoid overflow.

In matrix notation, solving equation (1) corresponds to solve

$Zx = \text{scale}* b$,

where Z is defined as

$$Z = \begin{bmatrix} \text{kron}(I_n, A) & -\text{kron}(B', I_m) \\ \text{kron}(I_n, D) & -\text{kron}(E', I_m) \end{bmatrix} \quad (2)$$

Here I_k is the identity matrix of size k and X' is the transpose of X. kron(X, Y) denotes the Kronecker product between the matrices X and Y.

If *trans* = 'T' , solve the transposed (conjugate transposed) system

$Z'y$ = scale* b

for y, which is equivalent to solve for r and L in

$$\begin{aligned} A' R + D' L &= \text{scale} * C \\ R B' + L E' &= \text{scale} * (-F) \end{aligned} \quad (3)$$

This case is used to compute an estimate of Dif[(A, D), (B, E)] = *sigma_min*(Z) using reverse communication with ?lacon.

?tgsy2 also (for *ijob* ≥ 1) contributes to the computation in ?tgsyl of an upper bound on the separation between two matrix pairs. Then the input (A, D), (B, E) are sub-pencils of the matrix pair (two matrix pairs) in ?tgsyl. See ?tgsyl for details.

Input Parameters

trans	CHARACTER*1. If *trans* = 'N', solve the generalized Sylvester equation (1); If *trans* = 'T': solve the 'transposed' system (3).
ijob	INTEGER. Specifies what kind of functionality is to be performed. If *ijob* = 0: solve (1) only. If *ijob* = 1: a contribution from this subsystem to a Frobenius norm-based estimate of the separation between two matrix pairs is computed (look ahead strategy is used);

	If *ijob* = 2: a contribution from this subsystem to a Frobenius norm-based estimate of the separation between two matrix pairs is computed (?gecon on sub-systems is used). Not referenced if *trans* = 'T'.
m	INTEGER. On entry, *m* specifies the order of *A* and *D*, and the row dimension of *C*, *F*, *r* and *L*.
n	INTEGER. On entry, *n* specifies the order of *B* and *E*, and the column dimension of *C*, *F*, *r* and *L*.
a, *b*	REAL for stgsy2 DOUBLE PRECISION for dtgsy2 COMPLEX for ctgsy2 COMPLEX*16 for ztgsy2. Arrays, DIMENSION (*lda*, *m*) and (*ldb*, *n*), respectively. On entry, *a* contains an upper (quasi) triangular matrix *A* and *B* contains an upper (quasi) triangular matrix *b*.
lda	INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1, *m*).
ldb	INTEGER. The leading dimension of the array *b*. *ldb* ≥ max(1, *n*).
c, *f*	REAL for stgsy2 DOUBLE PRECISION for dtgsy2 COMPLEX for ctgsy2 COMPLEX*16 for ztgsy2. Arrays, DIMENSION (*ldc*, *n*) and (*ldf*, *n*), respectively. On entry, *c* contains the right-hand-side of the first matrix equation in (1) and *f* contains the right-hand-side of the second matrix equation in (1).
ldc	INTEGER. The leading dimension of the array *c*. *ldc* ≥ max(1, *m*).
d, *e*	REAL for stgsy2 DOUBLE PRECISION for dtgsy2 COMPLEX for ctgsy2 COMPLEX*16 for ztgsy2.

Arrays, DIMENSION (*ldd*, *m*) and (*lde*, *n*), respectively. On entry, *d* contains an upper triangular matrix *D* and *e* contains an upper triangular matrix *E*.

ldd — INTEGER. The leading dimension of the array *d*. *ldd* ≥ max(1, *m*).

lde — INTEGER. The leading dimension of the array *e*. *lde* ≥ max(1, *n*).

ldf — INTEGER. The leading dimension of the array *f*. *ldf* ≥ max(1, *m*).

rdsum — REAL for stgsy2/ctgsy2
DOUBLE PRECISION for dtgsy2/ztgsy2.
On entry, the sum of squares of computed contributions to the Dif-estimate under computation by ?tgsyL, where the scaling factor *rdscal* has been factored out.

rdscal — REAL for stgsy2/ctgsy2
DOUBLE PRECISION for dtgsy2/ztgsy2.
On entry, scaling factor used to prevent overflow in *rdsum*.

iwork — INTEGER. Used with real flavors only.
Workspace array, DIMENSION (*m*+*n*+2).

Output Parameters

c — On exit, if *ijob* = 0, *c* has been overwritten by the solution *R*.

f — On exit, if *ijob* = 0, *f* has been overwritten by the solution *L*.

scale — REAL for stgsy2/ctgsy2
DOUBLE PRECISION for dtgsy2/ztgsy2.
On exit, 0 ≤ *scale* ≤ 1. If 0 < *scale* < 1, the solutions *R* and *L* (*C* and *F* on entry) hold the solutions to a slightly perturbed system, but the input matrices *A*, *B*, *D* and *E* have not been changed. If *scale* = 0, *r* and *L* hold the solutions to the homogeneous system with *C* = *F* = 0. Normally *scale* = 1.

rdsum	On exit, the corresponding sum of squares updated with the contributions from the current sub-system. If *trans* = 'T', *rdsum* is not touched. Note that *rdsum* only makes sense when ?tgsy2 is called by ?tgsyl.
rdscal	On exit, *rdscal* is updated with respect to the current contributions in *rdsum*. If *trans* = 'T' , *rdscal* is not touched. Note that *rdscal* only makes sense when ?tgsy2 is called by ?tgsyl.
pq	INTEGER. Used with real flavors only. On exit, the number of subsystems (of size 2-by-2, 4-by-4 and 8-by-8) solved by the routine stgsy2/dtgsy2.
info	INTEGER. On exit, if *info* is set to = 0: Successful exit < 0: If *info* = -*i*, the *i*-th argument had an illegal value. > 0: The matrix pairs (*A*, *D*) and (*B*, *E*) have common or very close eigenvalues.

?trti2

Computes the inverse of a triangular matrix (unblocked algorithm).

Syntax

```
call strti2( uplo, diag, n, a, lda, info )
call dtrti2( uplo, diag, n, a, lda, info )
call ctrti2( uplo, diag, n, a, lda, info )
call ztrti2( uplo, diag, n, a, lda, info )
```

Description

The routine ?trti2 computes the inverse of a real/complex upper or lower triangular matrix.

This is the Level 2 BLAS version of the algorithm.

Input Parameters

uplo CHARACTER*1.
Specifies whether the matrix *A* is upper or lower triangular.
= 'U': upper triangular
= 'L': lower triangular

diag CHARACTER*1.
Specifies whether or not the matrix *A* is unit triangular.
= 'N': non-unit triangular
= 'N': non-unit triangular

n INTEGER. The order of the matrix *A*. $n \geq 0$.

a REAL for strti2
DOUBLE PRECISION for dtrti2
COMPLEX for ctrti2
COMPLEX*16 for ztrti2.
Array, DIMENSION (*lda*, *n*).
On entry, the triangular matrix *A*.
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of the array *a* contains the upper triangular matrix, and the strictly lower triangular part of *A* is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of the array *a* contains the lower triangular matrix, and the strictly upper triangular part of *A* is not referenced.
If *diag* = 'U', the diagonal elements of *A* are also not referenced and are assumed to be 1.

lda INTEGER. The leading dimension of the array *a*. *lda* ≥ max(1,*n*).

Output Parameters

a On exit, the (triangular) inverse of the original matrix, in the same storage format.

info INTEGER.
= 0: successful exit
< 0: if *info* = -*k*, the *k*-th argument had an illegal value

clag2z

Converts a complex single precision matrix to a complex double precision matrix.

Syntax

```
call clag2z( m, n, sa, ldsa, a, lda, info )
```

Description

The routine converts a complex single precision matrix *SA* to a complex double precision matrix *A*.

Note that while it is possible to overflow while converting from double to single, it is not possible to overflow when converting from single to double.

This is a helper routine so there is no argument checking.

Input Parameters

m	INTEGER. The number of lines of the matrix *A* (*m* ≥ 0).
n	INTEGER. The number of columns in the matrix *A* (*n* ≥ 0).
ldsa	INTEGER. The leading dimension of the array *sa*; *ldsa* ≥ max(1, *m*).
a	DOUBLE PRECISION array, DIMENSION (*lda*, *n*). On entry, contains the *m*-by-*n* coefficient matrix *A*.
lda	INTEGER. The leading dimension of the array *a*; *lda* ≥ max(1, *m*).

Output Parameters

sa	REAL array, DIMENSION (*ldsa*, *n*). On exit, contains the *m*-by-*n* coefficient matrix *SA*.
info	INTEGER. If *info* = 0, the execution is successful.

dlag2s

Converts a double precision matrix to a single precision matrix.

Syntax

```
call dlag2s( m, n, a, lda, sa, ldsa, info )
```

Description

The routine converts a double precision matrix *SA* to a single precision matrix *A*.

RMAX is the overflow for the single precision arithmetic. dlag2s checks that all the entries of *A* are between -RMAX and RMAX. If not, the convertion is aborted and a flag is raised.

This is a helper routine so there is no argument checking.

Input Parameters

m	INTEGER. The number of lines of the matrix *A* (*m* ≥ 0).
n	INTEGER. The number of columns in the matrix *A* (*n* ≥ 0).
a	DOUBLE PRECISION array, DIMENSION (*lda*, *n*). On entry, contains the *m*-by-*n* coefficient matrix *A*.
lda	INTEGER. The leading dimension of the array *a*; *lda* ≥ max(1, *m*).
ldsa	INTEGER. The leading dimension of the array *sa*; *ldsa* ≥ max(1, *m*).

Output Parameters

sa	REAL array, DIMENSION (*ldsa*, *n*). On exit, if *info* = 0, contains the *m*-by-*n* coefficient matrix *SA*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *k*, the (*i*, *j*) entry of the matrix *A* has overflowed when moving from double precision to single precision. *k* is given by *k* = (*i*-1)*lda+*j*.

slag2d

Converts a single precision matrix to a double precision matrix.

Syntax

```
call slag2d( m, n, sa, ldsa, a, lda, info )
```

Description

The routine converts a single precision matrix *SA* to a double precision matrix *A*.

Note that while it is possible to overflow while converting from double to single, it is not possible to overflow when converting from single to double.

This is a helper routine so there is no argument checking.

Input Parameters

m	INTEGER. The number of lines of the matrix *A* (*m* ≥ 0).
n	INTEGER. The number of columns in the matrix *A* (*n* ≥ 0).
ldsa	INTEGER. The leading dimension of the array *sa*; *ldsa* ≥ max(1, *m*).
a	DOUBLE PRECISION array, DIMENSION (*lda*, *n*). On entry, contains the *m*-by-*n* coefficient matrix *A*.
lda	INTEGER. The leading dimension of the array *a*; *lda* ≥ max(1, *m*).

Output Parameters

sa	REAL array, DIMENSION (*ldsa*, *n*). On exit, if *info* = 0, contains the *m*-by-*n* coefficient matrix *SA*.
info	INTEGER. If *info* = 0, the execution is successful.

zlag2c

Converts a complex double precision matrix to a complex single precision matrix.

Syntax

```
call zlag2c( m, n, a, lda, sa, ldsa, info )
```

Description

The routine converts a double precision complex matrix *SA* to a single precision complex matrix *A*.

RMAX is the overflow for the single precision arithmetic. zlag2c checks that all the entries of *A* are between -RMAX and RMAX. If not, the convertion is aborted and a flag is raised.

This is a helper routine so there is no argument checking.

Input Parameters

m	INTEGER. The number of lines of the matrix *A* (*m* ≥ 0).
n	INTEGER. The number of columns in the matrix *A* (*n* ≥ 0).
a	DOUBLE PRECISION array, DIMENSION (*lda*, *n*). On entry, contains the *m*-by-*n* coefficient matrix *A*.
lda	INTEGER. The leading dimension of the array *a*; *lda* ≥ max(1, *m*).
ldsa	INTEGER. The leading dimension of the array *sa*; *ldsa* ≥ max(1, *m*).

Output Parameters

sa	REAL array, DIMENSION (*ldsa*, *n*). On exit, if *info* = 0, contains the *m*-by-*n* coefficient matrix *SA*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *k*, the (*i*, *j*) entry of the matrix *A* has overflowed when moving from double precision to single precision. *k* is given by *k* = (*i*-1)*lda+*j*.

Utility Functions and Routines

This section describes LAPACK utility functions and routines. Summary information about these routines is given in the following table:

Table 5-2 LAPACK Utility Routines

Routine Name	Data Types	Description
ilaver		Returns the version of the Lapack library.
ilaenv		Environmental enquiry function which returns values for tuning algorithmic performance.
iparmq		Environmental enquiry function which returns values for tuning algorithmic performance.
ieeeck		Checks if the infinity and NaN arithmetic is safe. Called by ilaenv.
lsame		Tests two characters for equality regardless of case.
lsamen		Tests two character strings for equality regardless of case.
?labad	s, d	Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.
?lamch	s, d	Determines machine parameters for floating-point arithmetic.
?lamc1	s, d	Called from ?lamc2. Determines machine parameters given by *beta*, *t*, *rnd*, *ieee1*.
?lamc2	s, d	Used by ?lamch. Determines machine parameters specified in its arguments list.
?lamc3	s, d	Called from ?lamc1-?lamc5. Intended to force *a* and *b* to be stored prior to doing the addition of *a* and *b*.
?lamc4	s, d	This is a service routine for ?lamc2.
?lamc5	s, d	Called from ?lamc2. Attempts to compute the largest machine floating-point number, without overflow.

Routine Name	Data Types	Description
second/dsecnd		Return user time for a process.
xerbla		Error handling routine called by LAPACK routines.

ilaver

Returns the version of the Lapack library.

Syntax

```
call ilaver( vers_major, vers_minor, vers_patch )
```

Description

This routine returns the version of the Lapack library.

Output Parameters

vers_major INTEGER.
Returns the major version of the LAPACK library.

vers_minor INTEGER.
Returns the minor version from the major version of the LAPACK library.

vers_patch INTEGER.
Returns the patch version from the minor version of the LAPACK library.

ilaenv

Environmental enquiry function which returns values for tuning algorithmic performance.

Syntax

```
value = ilaenv( ispec, name, opts, n1, n2, n3, n4 )
```

Description

Enquiry function `ilaenv` is called from the LAPACK routines to choose problem-dependent parameters for the local environment. See *ispec* below for a description of the parameters.

This version provides a set of parameters which should give good, but not optimal, performance on many of the currently available computers. Users are encouraged to modify this subroutine to set the tuning parameters for their particular machine using the option and problem size information in the arguments.

This routine will not function correctly if it is converted to all lower case. Converting it to all upper case is allowed.

Input Parameters

ispec — INTEGER.
Specifies the parameter to be returned as the value of `ilaenv`:
= 1: the optimal blocksize; if this value is 1, an unblocked algorithm will give the best performance.
= 2: the minimum block size for which the block routine should be used; if the usable block size is less than this value, an unblocked routine should be used.
= 3: the crossover point (in a block routine, for *n* less than this value, an unblocked routine should be used)
= 4: the number of shifts, used in the nonsymmetric eigenvalue routines (deprecated)
= 5: the minimum column dimension for blocking to be used; rectangular blocks must have dimension at least *k*-by-*m*, where `k` is given by `ilaenv(2,...)` and `m` by `ilaenv(5,...)`
= 6: the crossover point for the SVD (when reducing an *m*-by-*n* matrix to bidiagonal form, if `max(m,n)/min(m,n)` exceeds this value, a *QR* factorization is used first to reduce the matrix to a triangular form.)
= 7: the number of processors
= 8: the crossover point for the multishift *QR* and *QZ* methods for nonsymmetric eigenvalue problems (deprecated).

	= 9: maximum size of the subproblems at the bottom of the computation tree in the divide-and-conquer algorithm (used by ?gelsd and ?gesdd) =10: ieee NaN arithmetic can be trusted not to trap =11: infinity arithmetic can be trusted not to trap 12 ≤ *ispec* ≤ 16: ?hseqr or one of its subroutines, see iparmq for detailed explanation.
name	CHARACTER*(*). The name of the calling subroutine, in either upper case or lower case.
opts	CHARACTER*(*). The character options to the subroutine *name*, concatenated into a single character string. For example, *uplo* = 'U', *trans* = 'T', and *diag* = 'N' for a triangular routine would be specified as *opts* = 'UTN'.
n1, *n2*, *n3*, *n4*	INTEGER. Problem dimensions for the subroutine *name*; these may not all be required.

Output Parameters

value	INTEGER. If *value* ≥ 0: the value of the parameter specified by *ispec*; If *value* = -k < 0: the k-th argument had an illegal value.

Application Notes

The following conventions have been used when calling ilaenv from the LAPACK routines:

1. *opts* is a concatenation of all of the character options to subroutine *name*, in the same order that they appear in the argument list for *name*, even if they are not used in determining the value of the parameter specified by *ispec*.
2. The problem dimensions *n1*, *n2*, *n3*, *n4* are specified in the order that they appear in the argument list for *name*. *n1* is used first, *n2* second, and so on, and unused problem dimensions are passed a value of -1.
3. The parameter value returned by ilaenv is checked for validity in the calling subroutine. For example, ilaenv is used to retrieve the optimal blocksize for strtri as follows:

```
nb = ilaenv( 1, 'strtri', uplo // diag, n, -1, -1, -1> )
if( nb.le.1 ) nb = max( 1, n )
```

Below is an example of ilaenv usage in C language:

```
#include <stdio.h>

#include "mkl.h"

int main(void)

{

   int size = 1000;

   int ispec = 1;

   int dummy = -1;

   int blockSize1 = ilaenv(&ispec, "dsytrd", "U", &size, &dummy, &dummy,
&dummy);

   int blockSize2 = ilaenv(&ispec, "dormtr", "LUN", &size, &size, &dummy,
&dummy);

   printf("DSYTRD blocksize = %d\n", blockSize1);

   printf("DORMTR blocksize = %d\n", blockSize2);

   return 0;

}
```

iparmq

Environmental enquiry function which returns values for tuning algorithmic performance.

Syntax

value = iparmq(*ispec*, *name*, *opts*, *n*, *ilo*, *ihi*, *lwork*)

Description

This function sets problem and machine dependent parameters useful for ?hseqr and its subroutines. It is called whenever ilaenv is called with 12≤*ispec*≤16.

Input Parameters

ispec INTEGER.
Specifies the parameter to be returned as the value of iparmq:

= 12: (*inmin*) Matrices of order *nmin* or less are sent directly to ?lahqr, the implicit double shift QR algorithm. *nmin* must be at least 11.
= 13: (*inwin*) Size of the deflation window. This is best set greater than or equal to the number of simultaneous shifts *ns*. Larger matrices benefit from larger deflation windows.
= 14: (*inibl*) Determines when to stop nibbling and invest in an (expensive) multi-shift QR sweep. If the aggressive early deflation subroutine finds *ld* converged eigenvalues from an order *nw* deflation window and *ld*>(*nw***nibble*)/100, then the next QR sweep is skipped and early deflation is applied immediately to the remaining active diagonal block. Setting iparmq(*ispec*=14)=0 causes TTQRE to skip a multi-shift QR sweep whenever early deflation finds a converged eigenvalue. Setting iparmq(*ispec*=14) greater than or equal to 100 prevents TTQRE from skipping a multi-shift QR sweep.
= 15: (*nshfts*) The number of simultaneous shifts in a multi-shift QR iteration.
= 16: (*iacc22*) iparmq is set to 0, 1 or 2 with the following meanings.
0: During the multi-shift QR sweep, ?laqr5 does not accumulate reflections and does not use matrix-matrix multiply to update the far-from-diagonal matrix entries.
1: During the multi-shift QR sweep, ?laqr5 and/or ?laqr3 accumulates reflections and uses matrix-matrix multiply to update the far-from-diagonal matrix entries.
2: During the multi-shift QR sweep, ?laqr5 accumulates reflections and takes advantage of 2-by-2 block structure during matrix-matrix multiplies.
(If ?trrm is slower than ?gemm, then iparmq(*ispec*=16)=1 may be more efficient than iparmq(*ispec*=16)=2 despite the greater level of arithmetic work implied by the latter choice.)

name CHARACTER*(*). The name of the calling subroutine.

opts CHARACTER*(*). This is a concatenation of the string arguments to TTQRE.

n INTEGER. *n* is the order of the Hessenberg matrix *H*.

ilo, *ihi*	INTEGER. It is assumed that *H* is already upper triangular in rows and columns 1:*ilo*-1 and *ihi*+1:*n*.
lwork	INTEGER. The amount of workspace available.

Output Parameters

value	INTEGER. If *value* ≥ 0: the value of the parameter specified by *iparmq*; If *value* = -k < 0: the k-th argument had an illegal value.

Application Notes

The following conventions have been used when calling ilaenv from the LAPACK routines:

1. *opts* is a concatenation of all of the character options to subroutine *name*, in the same order that they appear in the argument list for *name*, even if they are not used in determining the value of the parameter specified by *ispec*.
2. The problem dimensions *n1*, *n2*, *n3*, *n4* are specified in the order that they appear in the argument list for *name*. *n1* is used first, *n2* second, and so on, and unused problem dimensions are passed a value of -1.
3. The parameter value returned by ilaenv is checked for validity in the calling subroutine. For example, ilaenv is used to retrieve the optimal blocksize for strtri as follows:

```
nb = ilaenv( 1, 'strtri', uplo // diag, n, -1, -1, -1> )
if( nb.le.1 ) nb = max( 1, n )
```

ieeeck

Checks if the infinity and NaN arithmetic is safe.
Called by ilaenv.

Syntax

```
ival = ieeeck( ispec, zero, one )
```

Description

The function ieeeck is called from ilaenv to verify that infinity and possibly NaN arithmetic is safe, that is, will not trap.

Input Parameters

ispec	INTEGER. Specifies whether to test just for inifinity arithmetic or both for infinity and NaN arithmetic: If *ispec* = 0: Verify infinity arithmetic only. If *ispec* = 1: Verify infinity and NaN arithmetic.
zero	REAL. Must contain the value 0.0 This is passed to prevent the compiler from optimizing away this code.
one	REAL. Must contain the value 1.0 This is passed to prevent the compiler from optimizing away this code.

Output Parameters

ival	INTEGER. If *ival* = 0: Arithmetic failed to produce the correct answers. If *ival* = 1: Arithmetic produced the correct answers.

lsame

Tests two characters for equality regardless of case.

Syntax

```
val = lsame(ca, cb)
```

Description

This logical function returns .TRUE. if *ca* is the same letter as *cb* regardless of case.

Input Parameters

ca, *cb*	CHARACTER*1. Specify the single characters to be compared.

Output Parameters

val	LOGICAL. Result of the comparison.

lsamen

Tests two character strings for equality regardless of case.

Syntax

val = lsamen(*n*, *ca*, *cb*)

Description

This logical function tests if the first *n* letters of the string *ca* are the same as the first *n* letters of *cb*, regardless of case. The function lsamen returns .TRUE. if *ca* and *cb* are equivalent except for case and .FALSE. otherwise. lsamen also returns .FALSE. if len(*ca*) or len(*cb*) is less than *n*.

Input Parameters

n	INTEGER. The number of characters in *ca* and *cb* to be compared.
ca, *cb*	CHARACTER*(*). Specify two character strings of length at least *n* to be compared. Only the first *n* characters of each string will be accessed.

Output Parameters

val	LOGICAL. Result of the comparison.

?labad

Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.

Syntax

```
call slabad( small, large )
call dlabad( small, large )
```

Description

This routine takes as input the values computed by `slamch/dlamch` for underflow and overflow, and returns the square root of each of these values if the log of *large* is sufficiently large. This subroutine is intended to identify machines with a large exponent range, such as the Crays, and redefine the underflow and overflow limits to be the square roots of the values computed by `?lamch`. This subroutine is needed because `?lamch` does not compensate for poor arithmetic in the upper half of the exponent range, as is found on a Cray.

Input Parameters

small
`REAL` for `slabad`
`DOUBLE PRECISION` for `dlabad`.
The underflow threshold as computed by `?lamch`.

large
`REAL` for `slabad`
`DOUBLE PRECISION` for `dlabad`.
The overflow threshold as computed by `?lamch`.

Output Parameters

small
On exit, if `log10(`*large*`)` is sufficiently large, the square root of *small*, otherwise unchanged.

large
On exit, if `log10(`*large*`)` is sufficiently large, the square root of *large*, otherwise unchanged.

?lamch

Determines machine parameters for floating-point arithmetic.

Syntax

```
val = slamch( cmach )
val = dlamch( cmach )
```

Description

The function ?lamch determines single precision and double precision machine parameters.

Input Parameters

cmach — CHARACTER*1. Specifies the value to be returned by ?lamch:
= 'E' or 'e', *val* = *eps*
= 'S' or 's' , *val* = *sfmin*
= 'B' or 'b', *val* = *base*
= 'P' or 'p', *val* = *eps***base*
= 'N' or 'n', *val* = *t*
= 'R' or 'r', *val* = *rnd*
= 'M' or 'm', *val* = *emin*
= 'U' or 'u', *val* = *rmin*
= 'L' or 'l', *val* = *emax*
= 'O' or 'o', *val* = *rmax*
where
eps = relative machine precision;
sfmin = safe minimum, such that 1/*sfmin* does not overflow;
base = base of the machine;
prec = *eps***base*;
t = number of (base) digits in the mantissa;
rnd = 1.0 when rounding occurs in addition, 0.0 otherwise;
emin = minimum exponent before (gradual) underflow;
rmin = *underflow_threshold* - *base***(*emin*-1);
emax = largest exponent before overflow;
rmax = *overflow_threshold* - (*base****emax*)*(1-*eps*).

Output Parameters

val	REAL for slamch DOUBLE PRECISION for dlamch Value returned by the function.

?lamc1

Called from ?lamc2*. Determines machine parameters given by* *beta*, *t*, *rnd*, *ieee1*.

Syntax

```
call slamc1( beta, t, rnd, ieee1 )
call dlamc1( beta, t, rnd, ieee1 )
```

Description

The routine ?lamc1 determines machine parameters given by *beta*, *t*, *rnd*, *ieee1*.

Output Parameters

beta	INTEGER. The base of the machine.
t	INTEGER. The number of (*beta*) digits in the mantissa.
rnd	LOGICAL. Specifies whether proper rounding (*rnd* = .TRUE.) or chopping (*rnd* = .FALSE.) occurs in addition. This may not be a reliable guide to the way in which the machine performs its arithmetic.
ieee1	LOGICAL. Specifies whether rounding appears to be done in the *ieee* 'round to nearest' style.

?lamc2

Used by ?lamch*. Determines machine parameters specified in its arguments list.*

Syntax

```
call slamc2( beta, t, rnd, eps, emin, rmin, emax, rmax )
call dlamc2( beta, t, rnd, eps, emin, rmin, emax, rmax )
```

Description

The routine ?lamc2 determines machine parameters specified in its arguments list.

Output Parameters

beta	INTEGER. The base of the machine.
t	INTEGER. The number of (*beta*) digits in the mantissa.
rnd	LOGICAL. Specifies whether proper rounding (*rnd* = .TRUE.) or chopping (*rnd* = .FALSE.) occurs in addition. This may not be a reliable guide to the way in which the machine performs its arithmetic.
eps	REAL for slamc2 DOUBLE PRECISION for dlamc2 The smallest positive number such that *fl*(1.0 - *eps*) < 1.0, where *fl* denotes the computed value.
emin	INTEGER. The minimum exponent before (gradual) underflow occurs.
rmin	REAL for slamc2 DOUBLE PRECISION for dlamc2 The smallest normalized number for the machine, given by $base^{emin-1}$, where *base* is the floating point value of *beta*.
emax	INTEGER.The maximum exponent before overflow occurs.
rmax	REAL for slamc2 DOUBLE PRECISION for dlamc2

The largest positive number for the machine, given by $base^{emax(1 - eps)}$, where *base* is the floating point value of *beta*.

?lamc3

Called from ?lamc1-?lamc5*. Intended to force* *a* *and* *b* *to be stored prior to doing the addition of* *a* *and* *b*.

Syntax

```
val = slamc3( a, b )
val = dlamc3( a, b )
```

Description

The routine is intended to force *A* and *B* to be stored prior to doing the addition of *A* and *B*, for use in situations where optimizers might hold one of these in a register.

Input Parameters

a, *b*
REAL for slamc3
DOUBLE PRECISION for dlamc3
The values *a* and *b*.

Output Parameters

val
REAL for slamc3
DOUBLE PRECISION for dlamc3
The result of adding values *a* and *b*.

?lamc4

This is a service routine for ?lamc2*.*

Syntax

```
call slamc4( emin, start, base )
call dlamc4( emin, start, base )
```

Description

This is a service routine for ?lamc2.

Input Parameters

start	REAL for slamc4 DOUBLE PRECISION for dlamc4 The starting point for determining *emin*.
base	INTEGER. The base of the machine.

Output Parameters

emin	INTEGER. The minimum exponent before (gradual) underflow, computed by setting *a = start* and dividing by *base* until the previous *a* can not be recovered.

?lamc5

Called from ?lamc2. Attempts to compute the largest machine floating-point number, without overflow.

Syntax

```
call slamc5( beta, p, emin, ieee, emax, rmax)
call dlamc5( beta, p, emin, ieee, emax, rmax)
```

Description

The routine ?lamc5 attempts to compute *rmax*, the largest machine floating-point number, without overflow. It assumes that *emax* + abs(*emin*) sum approximately to a power of 2. It will fail on machines where this assumption does not hold, for example, the Cyber 205 (*emin* = -28625, *emax* = 28718). It will also fail if the value supplied for *emin* is too large (that is, too close to zero), probably with overflow.

Input Parameters

beta	INTEGER. The base of floating-point arithmetic.
p	INTEGER. The number of base *beta* digits in the mantissa of a floating-point value.

emin INTEGER. The minimum exponent before (gradual) underflow.

ieee LOGICAL. A logical flag specifying whether or not the arithmetic system is thought to comply with the IEEE standard.

Output Parameters

emax INTEGER. The largest exponent before overflow.

rmax REAL for slamc5
DOUBLE PRECISION for dlamc5
The largest machine floating-point number.

second/dsecnd

Return user time for a process.

Syntax

```
val = second()
val = dsecnd()
```

Description

The functions second/dsecnd return the user time for a process in seconds. These versions get the time from the system function etime. The difference is that dsecnd returns the result with double presision.

Output Parameters

val REAL for second
DOUBLE PRECISION for dsecnd
User time for a process.

xerbla

Error handling routine called by BLAS, LAPACK, VML routines.

Syntax

```
call xerbla( srname, info )
```

Description

The routine `xerbla` is an error handler for the BLAS, LAPACK, and VML routines. It is called by a BLAS, LAPACK, or VML routine if an input parameter has an invalid value.

A message is printed and execution stops.

Installers may consider modifying the `stop` statement in order to call system-specific exception-handling facilities.

Input Parameters

srname	`CHARACTER*6` The name of the routine which called `xerbla`.
info	`INTEGER`. The position of the invalid parameter in the parameter list of the calling routine.

ScaLAPACK Routines

6

This chapter describes the Intel® Math Kernel Library implementation of routines from the ScaLAPACK package for distributed-memory architectures. Routines are supported for both real and complex dense and band matrices to perform the tasks of solving systems of linear equations, solving linear least-squares problems, eigenvalue and singular value problems, as well as performing a number of related computational tasks. All routines are available in both single precision and double precision.

NOTE. ScaLAPACK routines are provided with Intel® Cluster MKL product only which is a superset of Intel MKL.

Sections in this chapter include descriptions of ScaLAPACK computational routines that perform distinct computational tasks, as well as driver routines for solving standard types of problems in one call.

Generally, ScaLAPACK runs on a network of computers using MPI as a message-passing layer and a set of prebuilt communication subprograms (BLACS), as well as a set of BLAS optimized for the target architecture. Intel® Cluster MKL version of ScaLAPACK is optimized for Intel® processors. For the detailed system and environment requirements see Intel MKL Release Notes and Intel MKL User's Guide.

For full reference on ScaLAPACK routines and related information see [SLUG].

Overview

The model of the computing environment for ScaLAPACK is represented as a one-dimensional array of processes (for operations on band or tridiagonal matrices) or also a two-dimensional process grid (for operations on dense matrices). To use ScaLAPACK, all global matrices or vectors should be distributed on this array or grid prior to calling the ScaLAPACK routines.

ScaLAPACK uses the two-dimensional block-cyclic data distribution as a layout for dense matrix computations. This distribution provides good work balance between available processors, as well as gives the opportunity to use BLAS Level 3 routines for optimal local computations. Information about the data distribution that is required to establish the mapping between each global array and its corresponding process and memory location is contained in the so called `array descriptor` associated with each global array. An example of an array descriptor structure is given in Table 6-1.

Table 6-1 Content of the array descriptor for dense matrices

Array Element #	Name	Definition
1	*dtype*	Descriptor type (=1 for dense matrices)
2	*ctxt*	BLACS context handle for the process grid

Array Element #	Name	Definition
3	*m*	Number of rows in the global array
4	*n*	Number of columns in the global array
5	*mb*	Row blocking factor
6	*nb*	Column blocking factor
7	*rsrc*	Process row over which the first row of the global array is distributed
8	*csrc*	Process column over which the first column of the global array is distributed
9	*lld*	Leading dimension of the local array

The number of rows and columns of a global dense matrix that a particular process in a grid receives after data distributing is denoted by $LOC_r()$ and $LOC_c()$, respectively. To compute these numbers, you can use the ScaLAPACK tool routine `numroc`.

After the block-cyclic distribution of global data is done, you may choose to perform an operation on a submatrix of the global matrix *A*, which is contained in the global subarray sub(*A*), defined by the following 6 values (for dense matrices):

m	The number of rows of sub(*A*)
n	The number of columns of sub(*A*)
a	A pointer to the local array containing the entire global array A
ia	The row index of sub(*A*) in the global array
ja	The column index of sub(*A*) in the global array
desca	The array descriptor for the global array

Routine Naming Conventions

For each routine introduced in this chapter, you can use the ScaLAPACK name. The naming convention for ScaLAPACK routines is similar to that used for LAPACK routines (see Routine Naming Conventions in Chapter 4). A general rule is that each routine name in ScaLAPACK, which has an LAPACK equivalent, is simply the LAPACK name prefixed by initial letter `p`.

ScaLAPACK names have the structure `p?yyzzz` or `p?yyzz`, which is described below.

The initial letter `p` is a distinctive prefix of ScaLAPACK routines and is present in each such routine.

The second symbol `?` indicates the data type:

`s`	real, single precision
`d`	real, double precision

c	complex, single precision
z	complex, double precision

The second and third letters yy indicate the matrix type as:

ge	general
gb	general band
gg	a pair of general matrices (for a generalized problem)
dt	general tridiagonal (diagonally dominant-like)
db	general band (diagonally dominant-like)
po	symmetric or Hermitian positive-definite
pb	symmetric or Hermitian positive-definite band
pt	symmetric or Hermitian positive-definite tridiagonal
sy	symmetric
st	symmetric tridiagonal (real)
he	Hermitian
or	orthogonal
tr	triangular (or quasi-triangular)
tz	trapezoidal
un	unitary

For computational routines, the last three letters **zzz** indicate the computation performed and have the same meaning as for LAPACK routines.

For driver routines, the last two letters **zz** or three letters **zzz** have the following meaning:

sv	a *simple* driver for solving a linear system
svx	an *expert* driver for solving a linear system
ls	a driver for solving a linear least squares problem
ev	a simple driver for solving a symmetric eigenvalue problem
evx	an expert driver for solving a symmetric eigenvalue problem
svd	a driver for computing a singular value decomposition
gvx	an expert driver for solving a generalized symmetric definite eigenvalue problem

Simple driver here means that the driver just solves the general problem, whereas an *expert* driver is more versatile and can also optionally perform some related computations (such, for example, as refining the solution and computing error bounds after the linear system is solved).

Computational Routines

In the sections that follow, the descriptions of ScaLAPACK computational routines are given. These routines perform distinct computational tasks that can be used for:

- Solving Systems of Linear Equations
- Orthogonal Factorizations and LLS Problems
- Symmetric Eigenproblems
- Nonsymmetric Eigenproblems
- Singular Value Decomposition
- Generalized Symmetric-Definite Eigenproblems

See also the respective driver routines.

Linear Equations

ScaLAPACK supports routines for the systems of equations with the following types of matrices:

- general
- general banded
- general diagonally dominant-like banded (including general tridiagonal)
- symmetric or Hermitian positive-definite
- symmetric or Hermitian positive-definite banded
- symmetric or Hermitian positive-definite tridiagonal

A `diagonally dominant-like` matrix is defined as a matrix for which it is known in advance that pivoting is not required in the *LU* factorization of this matrix.

For the above matrix types, the library includes routines for performing the following computations: *factoring* the matrix; *equilibrating* the matrix; *solving* a system of linear equations; estimating the condition number of a matrix; *refining* the solution of linear equations and computing its error bounds; *inverting* the matrix. Note that for some of the listed matrix types only part of the computational routines are provided (for example, routines that refine the solution are not provided for band or tridiagonal matrices). See Table 6-2 for full list of available routines.

To solve a particular problem, you can either call two or more computational routines or call a corresponding driver routine that combines several tasks in one call. Thus, to solve a system of linear equations with a general matrix, you can first call `p?getrf`(*LU* factorization) and then

p?getrs(computing the solution). Then, you might wish to call p?gerfs to refine the solution and get the error bounds. Alternatively, you can just use the driver routine p?gesvx which performs all these tasks in one call.

Table 6-2 lists the ScaLAPACK computational routines for factorizing, equilibrating, and inverting matrices, estimating their condition numbers, solving systems of equations with real matrices, refining the solution, and estimating its error.

Table 6-2 Computational Routines for Systems of Linear Equations

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general (partial pivoting)	p?getrf	p?geequ	p?getrs	p?gecon	p?gerfs	p?getri
general band (partial pivoting)	p?gbtrf		p?gbtrs			
general band (no pivoting)	p?dbtrf		p?dbtrs			
general tridiagonal (no pivoting)	p?dttrf		p?dttrs			
symmetric/Hermitian positive-definite	p?potrf	p?poequ	p?potrs	p?pocon	p?porfs	p?potri
symmetric/Hermitian positive-definite, band	p?pbtrf		p?pbtrs			
symmetric/Hermitian positive-definite, tridiagonal	p?pttrf		p?pttrs			
triangular			p?trtrs	p?trcon	p?trrfs	p?trtri

In this table ? stands for s (single precision real), d (double precision real), c (single precision complex), or z (double precision complex).

Routines for Matrix Factorization

This section describes the ScaLAPACK routines for matrix factorization. The following factorizations are supported:

- LU factorization of general matrices
- LU factorization of diagonally dominant-like matrices
- Cholesky factorization of real symmetric or complex Hermitian positive-definite matrices

You can compute the factorizations using full and band storage of matrices.

p?getrf

Computes the LU factorization of a general m-by-n distributed matrix.

Syntax

```
call psgetrf(m, n, a, ia, ja, desca, ipiv, info)
call pdgetrf(m, n, a, ia, ja, desca, ipiv, info)
call pcgetrf(m, n, a, ia, ja, desca, ipiv, info)
call pzgetrf(m, n, a, ia, ja, desca, ipiv, info)
```

Description

The routine forms the *LU* factorization of a general *m*-by-*n* distributed matrix `sub(A) = A(ia:ia+n-1, ja:ja+n-1)` as

```
A = P*L*U
```

where *P* is a permutation matrix, *L* is lower triangular with unit diagonal elements (lower trapezoidal if *m* > *n*) and *U* is upper triangular (upper trapezoidal if *m* < *n*). *L* and *U* are stored in sub(*A*).

The routine uses partial pivoting, with row interchanges.

Input Parameters

m	(global) INTEGER. The number of rows in the distributed submatrix sub(*A*); *m*≥0.
n	(global) INTEGER. The number of columns in the distributed submatrix sub(*A*); *n*≥0.
a	(local) REAL for psgetrf DOUBLE PRECISION for pdgetrf COMPLEX for pcgetrf DOUBLE COMPLEX for pzgetrf. Pointer into the local memory to an array of local dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). Contains the local pieces of the distributed matrix sub(*A*) to be factored.

ia, *ja* (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix *A(ia:ia+n-1, ja:ja+n-1)*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

a Overwritten by local pieces of the factors *L* and *U* from the factorization *A* = *P*L*U*. The unit diagonal elements of *L* are not stored.

ipiv (local) INTEGER array.
The dimension of *ipiv* is (*LOCr(m_a)+ mb_a)*.
This array contains the pivoting information: local row *i* was interchanged with global row *ipiv(i)*. This array is tied to the distributed matrix *A*.

info (global) INTEGER.
If *info*=0, the execution is successful.
info < 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i*100+j)*; if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
If *info* = *i*, u_{ii} is 0. The factorization has been completed, but the factor *U* is exactly singular. Division by zero will occur if you use the factor *U* for solving a system of linear equations.

p?gbtrf

Computes the LU factorization of a general n-by-n banded distributed matrix.

Syntax

```
call psgbtrf(n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork, info)
call pdgbtrf(n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork, info)
call pcgbtrf(n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork, info)
call pzgbtrf(n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork, info)
```

Description

The routine computes the *LU* factorization of a general *n*-by-*n* real/complex banded distributed matrix *A*(*1:n, ja:ja+n-1*) using partial pivoting with row interchanges.

The resulting factorization is not the same factorization as returned from the LAPACK routine ?gbtrf. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form

A(*1:n, ja:ja+n-1*) = *P*L*U*Q*

where *P* and *Q* are permutation matrices, and *L* and *U* are banded lower and upper triangular matrices, respectively. The matrix *Q* represents reordering of columns for the sake of parallelism, while *P* represents reordering of rows for numerical stability using classic partial pivoting.

Input Parameters

n	(global) INTEGER. The number of rows and columns in the distributed submatrix *A*(*1:n, ja:ja+n-1*); *n* ≥ 0.
bwl	(global) INTEGER. The number of sub-diagonals within the band of *A* (0 ≤ *bwl* ≤ *n-1*).
bwu	(global) INTEGER. The number of super-diagonals within the band of *A* (0 ≤ *bwu* ≤ *n-1*).
a	(local) REAL for psgbtrf

DOUBLE PRECISION for pdgbtrf
COMPLEX for pcgbtrf
DOUBLE COMPLEX for pzgbtrf.
Pointer into the local memory to an array of local dimension (*lld_a*, LOC_c(*ja*+*n*−1) where

lld_a ≥ 2**bwl* + 2**bwu* +1.

Contains the local pieces of the *n*-by-*n* distributed banded matrix *A*(*1:n*, *ja:ja+n−1*) to be factored.

ja (global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

If *desca*(*dtype_*) = 501, then *dlen_* ≥ 7;

else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.

laf (local) INTEGER. The dimension of the array *af*.

Must be *laf* ≥
(NB+*bwu*)*(*bwl*+*bwu*)+6*(*bwl*+*bwu*)*(*bwl*+2*bwu*).
If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work (local) Same type as *a*. Workspace array of dimension *lwork* .

lwork (local or global) INTEGER. The size of the *work* array (*lwork* ≥ 1). If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

Output Parameters

a On exit, this array contains details of the factorization. Note that additional permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.

ipiv (local) INTEGER array.

The dimension of *ipiv* must be ≥ *desca*(NB).

	Contains pivot indices for local factorizations. Note that you should not alter the contents of this array between factorization and solve.
af	(local) REAL for psgbtrf DOUBLE PRECISION for pdgbtrf COMPLEX for pcgbtrf DOUBLE COMPLEX for pzgbtrf. Array, dimension (*laf*). Auxiliary Fillin space. Fillin is created during the factorization routine p?gbtrf and this is stored in *af*. Note that if a linear system is to be solved using p?gbtrs after the factorization routine, *af* must not be altered after the factorization.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. If *info*=0, the execution is successful. *info* < 0: If the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = –(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*. *info* > 0: If *info* = k ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not nonsingular, and the factorization was not completed. If *info* = k > NPROCS, the submatrix stored on processor *info*–NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

p?dbtrf

Computes the LU factorization of a n-by-n diagonally dominant-like banded distributed matrix.

Syntax

```
call psdbtrf(n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info)
call pddbtrf(n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info)
call pcdbtrf(n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info)
call pzdbtrf(n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info)
```

Description

The routine computes the *LU* factorization of a *n*-by-*n* real/complex diagonally dominant-like banded distributed matrix *A*(*1:n, ja:ja+n-1*) without pivoting.

Note that the resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

Input Parameters

n — (global) INTEGER. The number of rows and columns in the distributed submatrix *A*(*1:n, ja:ja+n-1*); $n \geq 0$.

bwl — (global) INTEGER. The number of sub-diagonals within the band of *A*

(*0* ≤ *bwl* ≤ *n-1*).

bwu — (global) INTEGER. The number of super-diagonals within the band of *A*

(*0* ≤ *bwu* ≤ *n-1*).

a — (local)
REAL for psdbtrf
DOUBLE PRECISION for pddbtrf
COMPLEX for pcdbtrf
DOUBLE COMPLEX for pzdbtrf.
Pointer into the local memory to an array of local dimension (*lld_a*, LOC_c(*ja+n*-1)).

Contains the local pieces of the n-by-n distributed banded matrix $A(1:n, ja:ja+n-1)$ to be factored.

ja (global) INTEGER. The index in the global array A that points to the start of the matrix to be operated on (which may be either all of A or a submatrix of A).

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix A.

If $desca(dtype\_) = 501$, then $dlen\_ \geq 7$;

else if $desca(dtype\_) = 1$, then $dlen\_ \geq 9$.

laf (local) INTEGER. The dimension of the array *af*.

Must be $laf \geq NB*(bwl+bwu)+*(\max(bwl,bwu))^2$.
If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work (local) Same type as *a*. Workspace array of dimension *lwork* .

lwork (local or global) INTEGER. The size of the *work* array, must be $lwork \geq (\max(bwl,bwu))^2$. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

Output Parameters

a On exit, this array contains details of the factorization. Note that additional permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.

af (local)
REAL for psdbtrf
DOUBLE PRECISION for pddbtrf
COMPLEX for pcdbtrf
DOUBLE COMPLEX for pzdbtrf.
Array, dimension (*laf*).
Auxiliary Fillin space. Fillin is created during the factorization routine p?dbtrf and this is stored in *af*.

Note that if a linear system is to be solved using p?dbtrs after the factorization routine, *af* must not be altered after the factorization.

work(1) — On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
If *info*=0, the execution is successful.
info < 0:
If the *i*th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*. *info* > 0:

If *info* = k ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not diagonally dominant-like, and the factorization was not completed. If *info* = k > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

p?potrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite distributed matrix.

Syntax

```
call pspotrf(uplo, n, a, ia, ja, desca, info)
call pdpotrf(uplo, n, a, ia, ja, desca, info)
call pcpotrf(uplo, n, a, ia, ja, desca, info)
call pzpotrf(uplo, n, a, ia, ja, desca, info)
```

Description

This routine computes the Cholesky factorization of a real symmetric or complex Hermitian positive-definite distributed *n*-by-*n* matrix *A*(*ia:ia+n-1, ja:ja+n-1*), denoted below as sub(*A*).

The factorization has the form

sub(*A*) = U^H*U if *uplo*='U', or

sub(*A*) = *L*L^H if *uplo*='L'

where *L* is a lower triangular matrix and *U* is upper triangular.

Input Parameters

uplo	(global) CHARACTER*1. Indicates whether the upper or lower triangular part of sub(*A*) is stored. Must be 'U' or 'L'. If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix sub(*A*) that is factored as U^H*U. If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix sub(*A*) that is factored as *L*L^H.
n	(global) INTEGER. The order of the distributed submatrix sub(*A*) (*n*≥0).
a	(local) REAL for pspotrf DOUBLE PRECISON for pdpotrf COMPLEX for pcpotrf DOUBLE COMPLEX for pzpotrf. Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, this array contains the local pieces of the *n*-by-*n* symmetric/Hermitian distributed matrix sub(*A*) to be factored. Depending on *uplo*, the array *a* contains either the upper or the lower triangular part of the matrix sub(*A*) (see *uplo*).
ia, *ja*	(global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

a	The upper or lower triangular part of *a* is overwritten by the Cholesky factor *U* or *L*, as specified by *uplo*.
info	(global) INTEGER. If *info*=0, the execution is successful; *info* < 0: if the *i*-th argument is an array, and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*. If *info* = *k* >0, the leading minor of order *k*, *A*(*ia:ia+k-1*, *ja:ja+k-1*), is not positive-definite, and the factorization could not be completed.

p?pbtrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite banded distributed matrix.

Syntax

```
call pspbtrf(uplo, n, bw, a, ja, desca, af, laf, work, lwork, info)
call pdpbtrf(uplo, n, bw, a, ja, desca, af, laf, work, lwork, info)
call pcpbtrf(uplo, n, bw, a, ja, desca, af, laf, work, lwork, info)
call pzpbtrf(uplo, n, bw, a, ja, desca, af, laf, work, lwork, info)
```

Description

This routine computes the Cholesky factorization of an *n*-by-*n* real symmetric or complex Hermitian positive-definite banded distributed matrix *A*(*1:n*, *ja:ja+n-1*).

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

A(*1:n*, *ja:ja+n-1*) = $P*U^H*P^T$, if *uplo*='U', or

A(*1:n*, *ja:ja+n-1*) = $P*L*L^H*P^T$, if *uplo*='L',

where *P* is a permutation matrix and *U* and *L* are banded upper and lower triangular matrices, respectively.

Input Parameters

uplo (global) CHARACTER*1. Must be 'U' or 'L'.
If *uplo* = 'U', upper triangle of *A(1:n, ja:ja+n-1)* is stored;
If *uplo* = 'L', lower triangle of *A(1:n, ja:ja+n-1)* is stored.

n (global) INTEGER. The order of the distributed submatrix *A(1:n, ja:ja+n-1)*.
(*n*≥0).

bw (global) INTEGER.
The number of superdiagonals of the distributed matrix if *uplo* = 'U', or the number of subdiagonals if *uplo* = 'U'
(*bw*≥0).

a (local)
REAL for pspbtrf
DOUBLE PRECISON for pdpbtrf
COMPLEX for pcpbtrf
DOUBLE COMPLEX for pzpbtrf.
Pointer into the local memory to an array of dimension (*lld_a*,*LOCc*(*ja+n*-1)).
On entry, this array contains the local pieces of the upper or lower triangle of the symmetric/Hermitian band distributed matrix *A(1:n, ja:ja+n-1)* to be factored.

ja (global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
If *desca*(*dtype_*) = 501, then *dlen_* ≥ 7;
else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.

laf (local) INTEGER. The dimension of the array *af*.
Must be *laf* ≥ (NB+2*bw*)*bw*.

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work (local) Same type as *a*. Workspace array of dimension *lwork*.

lwork (local or global) INTEGER. The size of the *work* array, must be *lwork* ≥ bw^2.

Output Parameters

a On exit, if *info*=0, contains the permuted triangular factor *U* or *L* from the Cholesky factorization of the band matrix *A*(*1:n, ja:ja+n-1*), as specified by *uplo*.

af (local)
REAL for pspbtrf
DOUBLE PRECISON for pdpbtrf
COMPLEX for pcpbtrf
DOUBLE COMPLEX for pzpbtrf.
Array, dimension (*laf*). Auxiliary Fillin space. Fillin is created during the factorization routine p?pbtrf and this is stored in *af*. Note that if a linear system is to be solved using p?pbtrs after the factorization routine, *af* must not be altered.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info (global) INTEGER.
If *info*=0, the execution is successful.
info < 0:
If the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.
info>0:

If *info* = k ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed.

If *info* = k > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

p?pttrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite tridiagonal distributed matrix.

Syntax

```
call pspttrf(n, d, e, ja, desca, af, laf, work, lwork, info)
call pdpttrf(n, d, e, ja, desca, af, laf, work, lwork, info)
call pcpttrf(n, d, e, ja, desca, af, laf, work, lwork, info)
call pzpttrf(n, d, e, ja, desca, af, laf, work, lwork, info)
```

Description

This routine computes the Cholesky factorization of an *n*-by-*n* real symmetric or complex hermitian positive-definite tridiagonal distributed matrix *A*(*1:n*, *ja:ja+n-1*).

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

A(*1:n*, *ja:ja+n-1*) = $P*L*D*L^H*P^T$, or

A(*1:n*, *ja:ja+n-1*) = $P*U^H*D*U*P^T$,

where *P* is a permutation matrix, and *U* and *L* are tridiagonal upper and lower triangular matrices, respectively.

Input Parameters

n	(global) INTEGER. The order of the distributed submatrix *A*(*1:n*, *ja:ja+n-1*) (*n* ≥ 0).
d, *e*	(local)

REAL for pspttrf
DOUBLE PRECISON for pdpttrf
COMPLEX for pcpttrf
DOUBLE COMPLEX for pzpttrf.
Pointers into the local memory to arrays of dimension (*desca*(*nb_*)) each.
On entry, the array *d* contains the local part of the global vector storing the main diagonal of the distributed matrix *A*.
On entry, the array *e* contains the local part of the global vector storing the upper diagonal of the distributed matrix *A*.

ja — (global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

If *desca*(*dtype_*) = 501, then *dlen_* ≥ 7;

else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.

laf — (local) INTEGER. The dimension of the array *af*.

Must be *laf* ≥ NB+2.
If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work — (local) Same type as *d* and *e*. Workspace array of dimension *lwork* .

lwork — (local or global) INTEGER. The size of the *work* array, must be at least

lwork ≥ 8*NPCOL.

Output Parameters

d, *e* — On exit, overwritten by the details of the factorization.

af — (local)
REAL for pspttrf
DOUBLE PRECISION for pdpttrf
COMPLEX for pcpttrf

	DOUBLE COMPLEX for pzpttrf. Array, dimension (*laf*). Auxiliary Fillin space. Fillin is created during the factorization routine p?pttrf and this is stored in *af*. Note that if a linear system is to be solved using p?pttrs after the factorization routine, *af* must not be altered.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. If *info*=0, the execution is successful. *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*. *info* > 0: If *info* = k ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed. If *info* = k > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

p?dttrf

Computes the LU factorization of a diagonally dominant-like tridiagonal distributed matrix.

Syntax

```
call psdttrf(n, dl, d, du, ja, desca, af, laf, work, lwork, info)
call pddttrf(n, dl, d, du, ja, desca, af, laf, work, lwork, info)
call pcdttrf(n, dl, d, du, ja, desca, af, laf, work, lwork, info)
call pzdttrf(n, dl, d, du, ja, desca, af, laf, work, lwork, info)
```

Description

This routine computes the *LU* factorization of an *n*-by-*n* real/complex diagonally dominant-like tridiagonal distributed matrix *A(1:n, ja:ja+n-1)* without pivoting for stability.

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

$A(1:n, ja:ja+n-1) = P*L*U*P^T$,

where *P* is a permutation matrix, and *L* and *U* are banded lower and upper triangular matrices, respectively.

Input Parameters

n	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix *A(1:n, ja:ja+n-1)* ($n \geq 0$).
dl, *d*, *du*	(local) REAL for pspttrf DOUBLE PRECISON for pdpttrf COMPLEX for pcpttrf DOUBLE COMPLEX for pzpttrf. Pointers to the local arrays of dimension *(desca(nb_))* each. On entry, the array *dl* contains the local part of the global vector storing the subdiagonal elements of the matrix. Globally, *dl*(1) is not referenced, and *dl* must be aligned with *d*. On entry, the array *d* contains the local part of the global vector storing the diagonal elements of the matrix. On entry, the array *du* contains the local part of the global vector storing the super-diagonal elements of the matrix. *du*(*n*) is not referenced, and *du* must be aligned with *d*.
ja	(global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*. If *desca*(*dtype_*) = 501, then *dlen_* ≥ 7; else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.
laf	(local) INTEGER. The dimension of the array *af*. Must be *laf* ≥ 2*(NB+2) . If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).
work	(local) Same type as *d*. Workspace array of dimension *lwork* .
lwork	(local or global) INTEGER. The size of the *work* array, must be at least *lwork* ≥ 8*NPCOL.

Output Parameters

dl, *d*, *du*	On exit, overwritten by the information containing the factors of the matrix.
af	(local) REAL for psdttrf DOUBLE PRECISION for pddttrf COMPLEX for pcdttrf DOUBLE COMPLEX for pzdttrf. Array, dimension (*laf*). Auxiliary Fillin space. Fillin is created during the factorization routine p?dttrf and this is stored in *af*. Note that if a linear system is to be solved using p?dttrs after the factorization routine, *af* must not be altered.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. If *info*=0, the execution is successful. *info* < 0: If the *i*th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

info > 0:

If *info* = k ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not diagonally dominant-like, and the factorization was not completed. If *info* = k > NPROCS, the submatrix stored on processor *info*–NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

Routines for Solving Systems of Linear Equations

This section describes the ScaLAPACK routines for solving systems of linear equations. Before calling most of these routines, you need to factorize the matrix of your system of equations (see Routines for Matrix Factorization in this chapter). However, the factorization is not necessary if your system of equations has a triangular matrix.

p?getrs

Solves a system of distributed linear equations with a general square matrix, using the LU factorization computed by p?getrf.

Syntax

```
call psgetrs(trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pdgetrs(trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pcgetrs(trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pzgetrs(trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
```

Description

This routine solves a system of distributed linear equations with a general *n*-by-*n* distributed matrix sub(*A*) = *A*(*ia:ia+n-1*, *ja:ja+n-1*) using the *LU* factorization computed by p?getrf.

The system has one of the following forms specified by *trans*:

sub(*A*)**X* = sub(*B*) (no transpose),

sub(*A*)^T**X* = sub(*B*) (transpose),

sub(*A*)^H**X* = sub(*B*) (conjugate transpose),

where sub(*B*) = *B*(*ib:ib+n-1*, *jb:jb+nrhs-1*).

Before calling this routine, you must call p?getrf to compute the *LU* factorization of sub(*A*).

Input Parameters

trans	(global) CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If *trans* = 'N', then sub(*A*)**X* = sub(*B*) is solved for *X*. If *trans* = 'T', then sub(*A*)^T**X* = sub(*B*) is solved for *X*. If *trans* = 'C', then sub(*A*)^H **X* = sub(*B*) is solved for *X*.
n	(global) INTEGER. The number of linear equations; the order of the submatrix sub(*A*) (*n*≥0).
nrhs	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix sub(*B*) (*nrhs*≥0).
a, *b*	(global) REAL for psgetrs DOUBLE PRECISION for pdgetrs COMPLEX for pcgetrs DOUBLE COMPLEX for pzgetrs. Pointers into the local memory to arrays of local dimension *a*(*lld_a*, *LOCc*(*ja+n-1*)) and *b*(*lld_b*, *LOCc*(*jb+nrhs-1*)), respectively. On entry, the array *a* contains the local pieces of the factors *L* and *U* from the factorization sub(*A*) = *P*L*U*; the unit diagonal elements of *L* are not stored. On entry, the array *b* contains the right hand sides sub(*B*).
ia, *ja*	(global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
ipiv	(local) INTEGER array.

	The dimension of *ipiv* is (*LOCr*(*m_a*) + *mb_a*). This array contains the pivoting information: local row i of the matrix was interchanged with the global row *ipiv*(i). This array is tied to the distributed matrix *A*.
ib, *jb*	(global) INTEGER. The row and column indices in the global array *B* indicating the first row and the first column of the submatrix sub(*B*), respectively.
descb	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

Output Parameters

b	On exit, overwritten by the solution distributed matrix *X*.
info	INTEGER. If *info*=0, the execution is successful. *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?gbtrs

Solves a system of distributed linear equations with a general band matrix, using the LU factorization computed by p?gbtrf.

Syntax

```
call psgbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, af, laf, work, lwork, info)
```

```
call pdgbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, af, laf, work, lwork, info)
```

```
call pcgbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, af, laf, work, lwork, info)
```

```
call pzgbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, af, laf, work, lwork, info)
```

Description

This routine solves a system of distributed linear equations with a general band distributed matrix sub(*A*) = *A*(*1:n*, *ja:ja+n-1*) using the *LU* factorization computed by p?gbtrf.

The system has one of the following forms specified by *trans*:

sub(*A*)**X* = sub(*B*) (no transpose),

sub(*A*)^T**X* = sub(*B*) (transpose),

sub(*A*)^H**X* = sub(*B*) (conjugate transpose),

where sub(*B*) = *B*(*ib:ib+n-1*, *1:nrhs*) .

Before calling this routine, you must call p?gbtrf to compute the *LU* factorization of sub(*A*).

Input Parameters

trans	(global) CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If *trans* = 'N', then sub(*A*)**X* = sub(*B*) is solved for *X*. If *trans* = 'T', then sub(*A*)^T**X* = sub(*B*) is solved for *X*. If *trans* = 'C', then sub(*A*)^H **X* = sub(*B*) is solved for *X*.
n	(global) INTEGER. The number of linear equations; the order of the distributed submatrix sub(*A*) (*n* ≥ 0).
bwl	(global) INTEGER. The number of sub-diagonals within the band of *A* (0 ≤ *bwl* ≤ *n-1*).
bwu	(global) INTEGER. The number of super-diagonals within the band of *A* (0 ≤ *bwu* ≤ *n-1*).
nrhs	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix sub(*B*) (*nrhs* ≥ 0).
a, *b*	(global) REAL for psgbtrs DOUBLE PRECISION for pdgbtrs COMPLEX for pcgbtrs DOUBLE COMPLEX for pzgbtrs.

Pointers into the local memory to arrays of local dimension *a*(*lld_a*,*LOC*$_c$(*ja*+*n*−1)) and *b*(*lld_b*,*LOC*$_c$(*nrhs*)), respectively.
The array *a* contains details of the *LU* factorization of the distributed band matrix *A*.
On entry, the array *b* contains the local pieces of the right hand sides *B*(*ib*:*ib*+*n*−1, 1:*nrhs*).

ja — (global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

If *desca*(*dtype_*) = 501, then *dlen_* ≥ 7;

else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.

ib — (global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

descb — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

If *desca*(*dtype_*) = 501, then *dlen_* ≥ 7;

else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.

laf — (local) INTEGER. The dimension of the array *af*.

Must be *laf* ≥ NB*(*bwl*+*bwu*)+6*(*bwl*+*bwu*)*(*bwl*+2**bwu*) .

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work — (local) Same type as *a*. Workspace array of dimension *lwork* .

lwork — (local or global) INTEGER. The size of the *work* array, must be at least *lwork* ≥ *nrhs**(NB+2**bwl*+4**bwu*).

Output Parameters

ipiv — (local) INTEGER array.

The dimension of *ipiv* must be ≥ *desca*(NB).

	Contains pivot indices for local factorizations. Note that you should not alter the contents of this array between factorization and solve.
b	On exit, overwritten by the local pieces of the solution distributed matrix *X*.
af	(local) REAL for psgbtrs DOUBLE PRECISION for pdgbtrs COMPLEX for pcgbtrs DOUBLE COMPLEX for pzgbtrs. Array, dimension (*laf*). Auxiliary Fillin space. Fillin is created during the factorization routine p?gbtrf and this is stored in *af*. Note that if a linear system is to be solved using p?gbtrs after the factorization routine, *af* must not be altered after the factorization.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	INTEGER. If *info*=0, the execution is successful. *info* < 0: If the *i*-th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?potrs

Solves a system of linear equations with a Cholesky-factored symmetric/Hermitian distributed positive-definite matrix.

Syntax

```
call pspotrs(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pdpotrs(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pcpotrs(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pzpotrs(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
```

Description

The routine p?potrs solves for X a system of distributed linear equations in the form:

sub(A)*X = sub(B) ,

where sub(A) = $A(ia:ia+n-1, ja:ja+n-1)$ is an n-by-n real symmetric or complex Hermitian positive definite distributed matrix, and sub(B) denotes the distributed matrix $B(ib:ib+n-1, jb:jb+nrhs-1)$.

This routine uses Cholesky factorization

sub(A) = U^H*U, or sub(A) = $L*L^H$

computed by p?potrf.

Input Parameters

uplo	(global) CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', upper triangle of sub(A) is stored; If *uplo* = 'L', lower triangle of sub(A) is stored.
n	(global) INTEGER. The order of the distributed submatrix sub(A) ($n \geq 0$).
nrhs	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix sub(B) ($nrhs \geq 0$).
a, *b*	(local) REAL for pspotrs DOUBLE PRECISION for pdpotrs COMPLEX for pcpotrs DOUBLE COMPLEX for pzpotrs. Pointers into the local memory to arrays of local dimension *a*(*lld_a*,*LOCc*(*ja*+*n*-1)) and *b*(*lld_b*,*LOCc*(*jb*+*nrhs*-1)), respectively. The array *a* contains the factors L or U from the Cholesky factorization sub(A) = $L*L^H$ or sub(A) = U^H*U, as computed by p?potrf. On entry, the array *b* contains the local pieces of the right hand sides sub(B).

ia, *ja*	(global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
ib, *jb*	(global) INTEGER. The row and column indices in the global array *B* indicating the first row and the first column of the submatrix sub(*B*), respectively.
descb	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

Output Parameters

b	Overwritten by the local pieces of the solution matrix *X*.
info	INTEGER. If *info*=0, the execution is successful. *info* < 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?pbtrs

Solves a system of linear equations with a Cholesky-factored symmetric/Hermitian positive-definite band matrix.

Syntax

```
call pspbtrs( uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pdpbtrs( uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pcpbtrs( uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pzpbtrs( uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

Description

The routine p?pbtrs solves for X a system of distributed linear equations in the form:

sub(A)*X = sub(B) ,

where sub(A) = $A(1:n, ja:ja+n-1)$ is an n-by-n real symmetric or complex Hermitian positive definite distributed band matrix, and sub(B) denotes the distributed matrix $B(ib:ib+n-1, 1:nrhs)$.

This routine uses Cholesky factorization

sub(A) = $P*U^H*U*P^T$, or sub(A) = $P*L*L^H*P^T$

computed by p?pbtrf.

Input Parameters

uplo	(global) CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', upper triangle of sub(A) is stored; If *uplo* = 'L', lower triangle of sub(A) is stored.
n	(global) INTEGER. The order of the distributed submatrix sub(A) ($n \geq 0$).
bw	(global) INTEGER. The number of superdiagonals of the distributed matrix if *uplo* = 'U', or the number of subdiagonals if *uplo* = 'L' ($bw \geq 0$).
nrhs	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix sub(B) ($nrhs \geq 0$).
a, *b*	(local) REAL for pspbtrs DOUBLE PRECISION for pdpbtrs COMPLEX for pcpbtrs DOUBLE COMPLEX for pzpbtrs. Pointers into the local memory to arrays of local dimension *a*(*lld_a*,*LOCc*(*ja*+*n*-1)) and *b*(*lld_b*,*LOCc*(*nrhs*-1)), respectively.

The array *a* contains the permuted triangular factor *U* or *L* from the Cholesky factorization sub(*A*) = *P***U*^{H}**U***P*^{T}, or sub(*A*) = *P***L***L*^{H}**P*^{T} of the band matrix *A*, as returned by p?pbtrf.
On entry, the array *b* contains the local pieces of the *n*-by-*nrhs* right hand side distributed matrix sub(*B*).

ja — (global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

If *desca*(*dtype_*) = 501, then *dlen_* ≥ 7;

else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.

ib — (global) INTEGER. The row index in the global array *B* indicating the first row of the submatrix sub(*B*).

descb — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

If *descb*(*dtype_*) = 502, then *dlen_* ≥ 7;

else if *descb*(*dtype_*) = 1, then *dlen_* ≥ 9.

af, *work* — (local) Arrays, same type as *a*.
The array *af* is of dimension (*laf*). It contains auxiliary Fillin space. Fillin is created during the factorization routine p?dbtrf and this is stored in *af*.
The array *work* is a workspace array of dimension *lwork*.

laf — (local) INTEGER. The dimension of the array *af*.

Must be *laf* ≥ *nrhs***bw*.
If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

lwork — (local or global) INTEGER. The size of the array *work*, must be at least *lwork* ≥ *bw*^{2}.

Output Parameters

b — On exit, if *info*=0, this array contains the local pieces of the *n*-by-*nrhs* solution distributed matrix *X*.

work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	INTEGER. If *info*=0, the execution is successful. *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?pttrs

Solves a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal distributed matrix using the factorization computed by p?pttrf.

Syntax

call pspttrs(*n*, *nrhs*, *d*, *e*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

call pdpttrs(*n*, *nrhs*, *d*, *e*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

call pcpttrs(*uplo*, *n*, *nrhs*, *d*, *e*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

call pzpttrs(*uplo*, *n*, *nrhs*, *d*, *e*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

Description

The routine p?pttrs solves for *X* a system of distributed linear equations in the form:

sub(*A*)**X* = sub(*B*) ,

where sub(*A*) = *A*(*1:n*, *ja:ja+n-1*) is an *n*-by-*n* real symmetric or complex Hermitian positive definite tridiagonal distributed matrix, and sub(*B*) denotes the distributed matrix *B*(*ib:ib+n-1*, *1:nrhs*).

This routine uses the factorization

sub(*A*) = $P*L*D*L^H*P^T$, or sub(*A*) = $P*U^H*D*U*P^T$

computed by p?pttrf.

Input Parameters

uplo	(global, used in complex flavors only) CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', upper triangle of sub(*A*) is stored; If *uplo* = 'L', lower triangle of sub(*A*) is stored.
n	(global) INTEGER. The order of the distributed submatrix sub(*A*) (*n*≥0).
nrhs	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix sub(*B*) (*nrhs*≥0).
d, *e*	(local) REAL for pspttrs DOUBLE PRECISON for pdpttrs COMPLEX for pcpttrs DOUBLE COMPLEX for pzpttrs. Pointers into the local memory to arrays of dimension (*desca*(*nb_*)) each. These arrays contain details of the factorization as returned by p?pttrf
ja	(global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*. If *desca*(*dtype_*) = 501 or 502, then *dlen_* ≥ 7; else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.
b	(local) Same type as *d*, *e*. Pointer into the local memory to an array of local dimension *b*(*lld_b*, *LOCc*(*nrhs*)). On entry, the array *b* contains the local pieces of the *n*-by-*nrhs* right hand side distributed matrix sub(*B*).

ib (global) INTEGER. The row index in the global array *B* that points to the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

If *descb*(*dtype_*) = 502, then *dlen_* ≥ 7;

else if *descb*(*dtype_*) = 1, then *dlen_* ≥ 9.

af, *work* (local) REAL for pspttrs
DOUBLE PRECISION for pdpttrs
COMPLEX for pcpttrs
DOUBLE COMPLEX for pzpttrs.
Arrays of dimension (*laf*) and (*lwork*), respectively The array *af* contains auxiliary Fillin space. Fillin is created during the factorization routine p?pttrf and this is stored in *af*.
The array *work* is a workspace array.

laf (local) INTEGER. The dimension of the array *af*.

Must be *laf* ≥ NB+2.
If *laf* is not large enough, an error code is returned and the minimum acceptable size will be returned in *af*(1).

lwork (local or global) INTEGER. The size of the array *work*, must be at least
lwork ≥ (10+2*min(100,*nrhs*))*NPCOL+4**nrhs*.

Output Parameters

b On exit, this array contains the local pieces of the solution distributed matrix *X*.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info INTEGER. If *info*=0, the execution is successful. *info* < 0:
if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.

p?dttrs

Solves a system of linear equations with a diagonally dominant-like tridiagonal distributed matrix using the factorization computed by p?dttrf.

Syntax

call psdttrs(*trans*, *n*, *nrhs*, *dl*, *d*, *du*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

call pddttrs(*trans*, *n*, *nrhs*, *dl*, *d*, *du*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

call pcdttrs(*trans*, *n*, *nrhs*, *dl*, *d*, *du*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

call pzdttrs(*trans*, *n*, *nrhs*, *dl*, *d*, *du*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

Description

The routine p?dttrs solves for X one of the systems of equations:

sub(*A*)**X* = sub(*B*),

$(\text{sub}(A))^{T}*X = \text{sub}(B)$, or

$(\text{sub}(A))^{H}*X = \text{sub}(B)$,

where sub(*A*) = *A*(*1:n*, *ja:ja+n-1*) is a diagonally dominant-like tridiagonal distributed matrix, and sub(*B*) denotes the distributed matrix *B*(*ib:ib+n-1*, *1:nrhs*).

This routine uses the *LU* factorization computed by p?dttrf.

Input Parameters

trans (global) CHARACTER*1. Must be 'N' or 'T' or 'C'.
Indicates the form of the equations:
If *trans* = 'N', then sub(*A*)**X* = sub(*B*) is solved for *X*.
If *trans* = 'T', then $(\text{sub}(A))^{T}*X = \text{sub}(B)$ is solved for *X*.
If *trans* = 'C', then $(\text{sub}(A))^{H}*X = \text{sub}(B)$ is solved for *X*.

n (global) INTEGER. The order of the distributed submatrix sub(*A*) (*n* ≥ 0).

nrhs (global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix sub(*B*) (*nrhs* ≥ 0).

dl, *d*, *du* (local)
REAL for psdttrs
DOUBLE PRECISON for pddttrs
COMPLEX for pcdttrs
DOUBLE COMPLEX for pzdttrs.
Pointers to the local arrays of dimension (*desca*(*nb_*)) each.
On entry, these arrays contain details of the factorization. Globally, *dl*(1) and *du*(*n*) are not referenced; *dl* and *du* must be aligned with *d*.

ja (global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*. If

desca(*dtype_*) = 501 or 502, then *dlen_* ≥ 7;

else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.

b (local) Same type as *d*.
Pointer into the local memory to an array of local dimension *b*(*lld_b*,LOC_c(*nrhs*)).
On entry, the array *b* contains the local pieces of the *n*-by-*nrhs* right hand side distributed matrix sub(*B*).

ib (global) INTEGER. The row index in the global array *B* that points to the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

If *descb*(*dtype_*) = 502, then *dlen_* ≥ 7;

else if *descb*(*dtype_*) = 1, then *dlen_* ≥ 9.

af, *work*	(local) REAL for psdttrs DOUBLE PRECISION for pddttrs COMPLEX for pcdttrs DOUBLE COMPLEX for pzdttrs. Arrays of dimension (*laf*) and (*lwork*), respectively. The array *af* contains auxiliary Fillin space. Fillin is created during the factorization routine p?dttrf and this is stored in *af*. If a linear system is to be solved using p?dttrsafter the factorization routine, *af* must not be altered. The array *work* is a workspace array.
laf	(local) INTEGER. The dimension of the array *af*. Must be *laf* ≥ NB*(*bwl*+*bwu*)+6*(*bwl*+*bwu*)*(*bwl*+2**bwu*) . If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).
lwork	(local or global) INTEGER. The size of the array *work*, must be at least *lwork* ≥ 10*NPCOL+4**nrhs*.

Output Parameters

b	On exit, this array contains the local pieces of the solution distributed matrix *X*.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	INTEGER. If *info*=0, the execution is successful. *info* < 0: if the *i*th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?dbtrs

Solves a system of linear equations with a diagonally dominant-like banded distributed matrix using the factorization computed by `p?dbtrf`.

Syntax

```
call psdbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pddbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pcdbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pzdbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

Description

The routine `p?dbtrs` solves for X one of the systems of equations:

$sub(A)*X = sub(B)$,

$(sub(A))^T*X = sub(B)$, or

$(sub(A))^H*X = sub(B)$,

where $sub(A) = A(1:n, ja:ja+n-1)$ is a diagonally dominant-like banded distributed matrix, and sub(B) denotes the distributed matrix $B(ib:ib+n-1, 1:nrhs)$.

This routine uses the LU factorization computed by `p?dbtrf`.

Input Parameters

trans (global) `CHARACTER*1`. Must be `'N'` or `'T'` or `'C'`.
Indicates the form of the equations:
If *trans* = `'N'`, then $sub(A)*X = sub(B)$ is solved for X.
If *trans* = `'T'`, then $(sub(A))^T*X = sub(B)$ is solved for X.
If *trans* = `'C'`, then $(sub(A))^H*X = sub(B)$ is solved for X.

n	(global) INTEGER. The order of the distributed submatrix sub(*A*) (*n* ≥ 0).
bwl	(global) INTEGER. The number of subdiagonals within the band of *A* (0 ≤ *bwl* ≤ *n-1*).
bwu	(global) INTEGER. The number of superdiagonals within the band of *A* (0 ≤ *bwu* ≤ *n-1*).
nrhs	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix sub(*B*) (*nrhs* ≥ 0).
a, *b*	(local) REAL for psdbtrs DOUBLE PRECISON for pddbtrs COMPLEX for pcdbtrs DOUBLE COMPLEX for pzdbtrs. Pointers into the local memory to arrays of local dimension *a*(*lld_a*,LOC_c(*ja*+*n*-1)) and *b*(*lld_b*,LOC_c(*nrhs*)), respectively. On entry, the array *a* contains details of the *LU* factorization of the band matrix *A*, as computed by p?dbtrf. On entry, the array *b* contains the local pieces of the right hand side distributed matrix sub(*B*).
ja	(global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*. If *desca*(*dtype_*) = 501, then *dlen_* ≥ 7; else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.
ib	(global) INTEGER. The row index in the global array *B* that points to the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

If *descb*(*dtype_*) = 502, then *dlen_* ≥ 7;

else if *descb*(*dtype_*) = 1, then *dlen_* ≥ 9.

af, *work* (local)
REAL for psdbtrs
DOUBLE PRECISION for pddbtrs
COMPLEX for pcdbtrs
DOUBLE COMPLEX for pzdbtrs.
Arrays of dimension (*laf*) and (*lwork*), respectively The array *af* contains auxiliary Fillin space. Fillin is created during the factorization routine p?dbtrf and this is stored in *af*.
The array *work* is a workspace array.

laf (local) INTEGER. The dimension of the array *af*.

Must be *laf* ≥ NB*(*bwl*+*bwu*)+6*(max(*bwl*,*bwu*))^2 .

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

lwork (local or global) INTEGER. The size of the array *work*, must be at least

lwork ≥ (max(*bwl*,*bwu*))^2.

Output Parameters

b On exit, this array contains the local pieces of the solution distributed matrix *X*.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info INTEGER. If *info*=0, the execution is successful. *info* < 0:
if the *i*th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?trtrs

Solves a system of linear equations with a triangular distributed matrix.

Syntax

call pstrtrs(*uplo*, *trans*, *diag*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *info*)

call pdtrtrs(*uplo*, *trans*, *diag*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *info*)

call pctrtrs(*uplo*, *trans*, *diag*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *info*)

call pztrtrs(*uplo*, *trans*, *diag*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *info*)

Description

This routine solves for X one of the following systems of linear equations:

$sub(A)*X = sub(B)$,

$(sub(A))^{T}*X = sub(B)$, or

$(sub(A))^{H}*X = sub(B)$,

where $sub(A) = A(ia:ia+n-1, ja:ja+n-1)$ is a triangular distributed matrix of order n, and sub(B) denotes the distributed matrix $B(ib:ib+n-1, jb:jb+nrhs-1)$.

A check is made to verify that sub(A) is nonsingular.

Input Parameters

uplo (global) CHARACTER*1. Must be 'U' or 'L'.
Indicates whether sub(A) is upper or lower triangular:
If *uplo* = 'U', then sub(A) is upper triangular.
If *uplo* = 'L', then sub(A) is lower triangular.

trans (global) CHARACTER*1. Must be 'N' or 'T' or 'C'.
Indicates the form of the equations:
If *trans* = 'N', then $sub(A)*X = sub(B)$ is solved for X.

If *trans* = 'T', then sub(*A*)^T**X* = sub(*B*) is solved for *X*.
If *trans* = 'C', then sub(*A*)^H**X* = sub(*B*) is solved for *X*.

diag (global) CHARACTER*1. Must be 'N' or 'U'.
If *diag* = 'N', then sub(*A*) is not a unit triangular matrix.
If *diag* = 'U', then sub(*A*) is unit triangular.

n (global) INTEGER. The order of the distributed submatrix sub(*A*) (*n*≥0).

nrhs (global) INTEGER. The number of right-hand sides; i.e., the number of columns of the distributed matrix sub(*B*) (*nrhs*≥0).

a, *b* (local)
REAL for pstrtrs
DOUBLE PRECISION for pdtrtrs
COMPLEX for pctrtrs
DOUBLE COMPLEX for pztrtrs.
Pointers into the local memory to arrays of local dimension *a*(*lld_a*,*LOCc*(*ja*+*n*-1)) and *b*(*lld_b*,*LOCc*(*jb*+*nrhs*-1)), respectively.
The array *a* contains the local pieces of the distributed triangular matrix sub(*A*).
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular matrix, and the strictly lower triangular part of sub(*A*) is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular matrix, and the strictly upper triangular part of sub(*A*) is not referenced.
If *diag* = 'U', the diagonal elements of sub(*A*) are also not referenced and are assumed to be 1.
On entry, the array *b* contains the local pieces of the right hand side distributed matrix sub(*B*).

ia, *ja* (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
ib, *jb*	(global) INTEGER. The row and column indices in the global array *B* indicating the first row and the first column of the submatrix sub(*B*), respectively.
descb	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

Output Parameters

b	On exit, if *info*=0, sub(*B*) is overwritten by the solution matrix *X*.
info	INTEGER. If *info*=0, the execution is successful. *info* < 0: if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*; *info* > 0: if *info* = *i*, the *i*-th diagonal element of sub(*A*) is zero, indicating that the submatrix is singular and the solutions *X* have not been computed.

Routines for Estimating the Condition Number

This section describes the ScaLAPACK routines for estimating the condition number of a matrix. The condition number is used for analyzing the errors in the solution of a system of linear equations. Since the condition number may be arbitrarily large when the matrix is nearly singular, the routines actually compute the *reciprocal* condition number.

p?gecon

Estimates the reciprocal of the condition number of a general distributed matrix in either the 1-norm or the infinity-norm.

Syntax

```
call psgecon( norm, n, a, ia, ja, desca, anorm, rcond, work, lwork, iwork, liwork, info)
call pdgecon( norm, n, a, ia, ja, desca, anorm, rcond, work, lwork, iwork, liwork, info)
call pcgecon( norm, n, a, ia, ja, desca, anorm, rcond, work, lwork, rwork, lrwork, info)
call pzgecon( norm, n, a, ia, ja, desca, anorm, rcond, work, lwork, rwork, lrwork, info)
```

Description

This routine estimates the reciprocal of the condition number of a general distributed real/complex matrix `sub(A) = A(ia:ia+n-1, ja:ja+n-1)` in either the 1-norm or infinity-norm, using the *LU* factorization computed by p?getrf.

An estimate is obtained for $||(sub(A))^{-1}||$, and the reciprocal of the condition number is computed as

$$rcond = \frac{1}{\|sub(A)\| \times \|(sub(A))^{-1}\|}$$

Input Parameters

norm (global) CHARACTER*1. Must be '1' or 'O' or 'I'.
Specifies whether the 1-norm condition number or the infinity-norm condition number is required.
If *norm* = '1' or 'O', then the 1-norm is used;
If *norm* = 'I', then the infinity-norm is used.

n	(global) INTEGER. The order of the distributed submatrix sub(*A*) (*n* ≥ 0).
a	(local) REAL for psgecon DOUBLE PRECISION for pdgecon COMPLEX for pcgecon DOUBLE COMPLEX for pzgecon. Pointer into the local memory to an array of dimension *a*(*lld_a*,*LOCc*(*ja*+*n*-1)). The array *a* contains the local pieces of the factors *L* and *U* from the factorization sub(*A*) = *P***L***U*; the unit diagonal elements of *L* are not stored.
ia, *ja*	(global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
anorm	(global) REAL for single precision flavors, DOUBLE PRECISION for double precision flavors. If *norm* = '1' or 'O', the 1-norm of the original distributed matrix sub(*A*); If *norm* = 'I', the infinity-norm of the original distributed matrix sub(*A*).
work	(local) REAL for psgecon DOUBLE PRECISION for pdgecon COMPLEX for pcgecon DOUBLE COMPLEX for pzgecon. The array *work* of dimension (*lwork*) is a workspace array.
lwork	(local or global) INTEGER. The dimension of the array *work*. For real flavors: *lwork* must be at least

lwork ≥ 2*LOCr(*n*+mod(*ia*-1,*mb_a*))+
2*LOCc(*n*+mod(*ja*-1,*nb_a*))+max(2, max(*nb_a*max(1,
ceil(NPROW-1, NPCOL)),
LOCc(*n*+mod(*ja*-1,*nb_a*))+*nb_a*max(1, ceil(NPCOL-1,
NPROW))).
For complex flavors:
lwork must be at least

lwork ≥ 2*LOCr(*n*+mod(*ia*-1,*mb_a*))+max(2,
max(*nb_a*ceil(NPROW-1, NPCOL),
LOCc(*n*+mod(*ja*-1,*nb_a*))+ *nb_a*ceil(NPCOL-1,
NPROW))).
LOCr and *LOCc* values can be computed using the ScaLAPACK tool function numroc; NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

iwork (local) INTEGER. Workspace array, DIMENSION (*liwork*). Used in real flavors only.

liwork (local or global) INTEGER. The dimension of the array *iwork*; used in real flavors only. Must be at least

liwork ≥ *LOCr*(*n*+mod(*ia*-1,*mb_a*)).

rwork (local) REAL for pcgecon
DOUBLE PRECISION for pzgecon
Workspace array, DIMENSION (*lrwork*). Used in complex flavors only.

lrwork (local or global) INTEGER. The dimension of the array *rwork*; used in complex flavors only. Must be at least

lrwork ≥ 2*LOCc(*n*+mod(*ja*-1,*nb_a*)).

Output Parameters

rcond (global) REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
The reciprocal of the condition number of the distributed matrix sub(*A*). See Description.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

iwork(1)	On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance (for real flavors).
rwork(1)	On exit, *rwork*(1) contains the minimum value of *lrwork* required for optimum performance (for complex flavors).
info	(global) INTEGER. If *info*=0, the execution is successful. *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?pocon

Estimates the reciprocal of the condition number (in the 1 - norm) of a symmetric / Hermitian positive-definite distributed matrix.

Syntax

call pspocon(*uplo*, *n*, *a*, *ia*, *ja*, *desca*, *anorm*, *rcond*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call pdpocon(*uplo*, *n*, *a*, *ia*, *ja*, *desca*, *anorm*, *rcond*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call pcpocon(*uplo*, *n*, *a*, *ia*, *ja*, *desca*, *anorm*, *rcond*, *work*, *lwork*, *rwork*, *lrwork*, *info*)

call pzpocon(*uplo*, *n*, *a*, *ia*, *ja*, *desca*, *anorm*, *rcond*, *work*, *lwork*, *rwork*, *lrwork*, *info*)

Description

This routine estimates the reciprocal of the condition number (in the 1 - norm) of a real symmetric or complex Hermitian positive definite distributed matrix sub(*A*) = *A*(*ia:ia+n-1*, *ja:ja+n-1*), using the Cholesky factorization sub(*A*) = U^H**U* or sub(*A*) = *L**L^H computed by p?potrf.

An estimate is obtained for ||(sub(*A*))$^{-1}$||, and the reciprocal of the condition number is computed as

$$rcond = \frac{1}{\|sub(A)\| \times \|(sub(A))^{-1}\|}$$

Input Parameters

uplo	(global) CHARACTER*1. Must be 'U' or 'L'. Specifies whether the factor stored in sub(*A*) is upper or lower triangular. If *uplo* = 'U', sub(*A*) stores the upper triangular factor *U* of the Cholesky factorization sub(*A*) = U^H*U. If *uplo* = 'L', sub(*A*) stores the lower triangular factor *L* of the Cholesky factorization sub(*A*) = $L*L^H$.
n	(global) INTEGER. The order of the distributed submatrix sub(*A*) (*n*≥0).
a	(local) REAL for pspocon DOUBLE PRECISION for pdpocon COMPLEX for pcpocon DOUBLE COMPLEX for pzpocon. Pointer into the local memory to an array of dimension *a*(*lld_a*,*LOCc*(*ja*+*n*-1)). The array *a* contains the local pieces of the factors *L* or *U* from the Cholesky factorization sub(*A*) = U^H*U, or sub(*A*) = $L*L^H$, as computed by p?potrf.
ia, *ja*	(global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
anorm	(global) REAL for single precision flavors, DOUBLE PRECISION for double precision flavors. The 1-norm of the symmetric/Hermitian distributed matrix sub(*A*).
work	(local)

REAL for pspocon
DOUBLE PRECISION for pdpocon
COMPLEX for pcpocon
DOUBLE COMPLEX for pzpocon.
The array *work* of dimension (*lwork*) is a workspace array.

lwork (local or global) INTEGER. The dimension of the array *work*.
For real flavors:
lwork must be at least

lwork ≥ 2**LOCr*(*n*+mod(*ia*-1,*mb_a*))+
2**LOCc*(*n*+mod(*ja*-1,*nb_a*))+ max(2,
max(*nb_a**ceil(NPROW-1, NPCOL),
LOCc(*n*+mod(*ja*-1,*nb_a*))+
*nb_a**ceil(NPCOL-1, NPROW))).

For complex flavors:
lwork must be at least

lwork ≥ 2**LOCr*(*n*+mod(*ia*-1,*mb_a*))+ max(2,
max(*nb_a**max(1,ceil(NPROW-1, NPCOL)),
LOCc(*n*+mod(*ja*-1,*nb_a*))+ *nb_a**max(1,ceil(NPCOL-1,
NPROW)))).

iwork (local) INTEGER. Workspace array, DIMENSION (*liwork*).
Used in real flavors only.

liwork (local or global) INTEGER. The dimension of the array *iwork*; used in real flavors only. Must be at least *liwork* ≥ *LOCr*(*n*+mod(*ia*-1,*mb_a*)).

rwork (local) REAL for pcpocon
DOUBLE PRECISION for pzpocon
Workspace array, DIMENSION (*lrwork*). Used in complex flavors only.

lrwork (local or global) INTEGER. The dimension of the array *rwork*; used in complex flavors only. Must be at least *lrwork* ≥ 2**LOCc*(*n*+mod(*ja*-1,*nb_a*)).

Output Parameters

rcond (global) REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.

	The reciprocal of the condition number of the distributed matrix sub(*A*).
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
iwork(1)	On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance (for real flavors).
rwork(1)	On exit, *rwork*(1) contains the minimum value of *lrwork* required for optimum performance (for complex flavors).
info	(global) INTEGER. If *info*=0, the execution is successful. *info* < 0: If the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.

p?trcon

Estimates the reciprocal of the condition number of a triangular distributed matrix in either 1-norm or infinity-norm.

Syntax

```
call pstrcon( norm, uplo, diag, n, a, ia, ja, desca, rcond, work, lwork,
iwork, liwork, info)
```

```
call pdtrcon( norm, uplo, diag, n, a, ia, ja, desca, rcond, work, lwork,
iwork, liwork, info)
```

```
call pctrcon( norm, uplo, diag, n, a, ia, ja, desca, rcond, work, lwork,
rwork, lrwork, info)
```

```
call pztrcon( norm, uplo, diag, n, a, ia, ja, desca, rcond, work, lwork,
rwork, lrwork, info)
```

Description

This routine estimates the reciprocal of the condition number of a triangular distributed matrix sub(*A*) = *A*(*ia:ia+n-1*, *ja:ja+n-1*), in either the 1-norm or the infinity-norm.

The norm of sub(*A*) is computed and an estimate is obtained for $||(\text{sub}(A))^{-1}||$, then the reciprocal of the condition number is computed as

$$rcond = \frac{1}{\|sub(A)\| \times \|(sub(A))^{-1}\|}$$

Input Parameters

norm	(global) CHARACTER*1. Must be '1' or 'O' or 'I'. Specifies whether the 1-norm condition number or the infinity-norm condition number is required. If *norm* = '1' or 'O', then the 1-norm is used; If *norm* = 'I', then the infinity-norm is used.
uplo	(global) CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', sub(*A*) is upper triangular. If *uplo* = 'L', sub(*A*) is lower triangular.
diag	(global) CHARACTER*1. Must be 'N' or 'U'. If *diag* = 'N', sub(*A*) is non-unit triangular. If *diag* = 'U', sub(*A*) is unit triangular.
n	(global) INTEGER. The order of the distributed submatrix sub(*A*), (*n*≥0).
a	(local) REAL for pstrcon DOUBLE PRECISION for pdtrcon COMPLEX for pctrcon DOUBLE COMPLEX for pztrcon. Pointer into the local memory to an array of dimension *a*(*lld_a*,*LOCc*(*ja*+*n*-1)). The array *a* contains the local pieces of the triangular distributed matrix sub(*A*). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of this distributed matrix contains the upper triangular matrix, and its strictly lower triangular part is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of this distributed matrix contains the lower triangular matrix, and its strictly upper triangular part is not referenced.

If *diag* = 'U', the diagonal elements of sub(*A*) are also not referenced and are assumed to be 1.

ia, *ja* (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

work (local)
REAL for pstrcon
DOUBLE PRECISION for pdtrcon
COMPLEX for pctrcon
DOUBLE COMPLEX for pztrcon.
The array *work* of dimension (*lwork*) is a workspace array.

lwork (local or global) INTEGER. The dimension of the array *work*.
For real flavors:
lwork must be at least

lwork ≥ 2**LOCr*(*n*+mod(*ia*-1,*mb_a*))+
LOCc(*n*+mod(*ja*-1,*nb_a*))+ max(2,
max(*nb_a**max(1,ceil(NPROW-1, NPCOL)),
LOCc(*n*+mod(*ja*-1,*nb_a*))+ *nb_a**max(1,ceil(NPCOL-1,
NPROW))).

For complex flavors:
lwork must be at least

lwork ≥ 2**LOCr*(*n*+mod(*ia*-1,*mb_a*))+ max(2,
max(*nb_a**ceil(NPROW-1, NPCOL),
LOCc(*n*+mod(*ja*-1,*nb_a*))+ *nb_a**ceil(NPCOL-1,
NPROW))).

iwork (local) INTEGER. Workspace array, DIMENSION (*liwork*). Used in real flavors only.

liwork (local or global) INTEGER. The dimension of the array *iwork*; used in real flavors only. Must be at least

liwork ≥ *LOCr*(*n*+mod(*ia*-1,*mb_a*)).

rwork (local) REAL for pcpocon
DOUBLE PRECISION for pzpocon
Workspace array, DIMENSION (*lrwork*). Used in complex flavors only.

lrwork	(local or global) INTEGER. The dimension of the array *rwork*; used in complex flavors only. Must be at least *lrwork* ≥ *LOCc*(*n*+mod(*ja*-1,*nb_a*)).

Output Parameters

rcond	(global) REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The reciprocal of the condition number of the distributed matrix sub(*A*).
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
iwork(1)	On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance (for real flavors).
rwork(1)	On exit, *rwork*(1) contains the minimum value of *lrwork* required for optimum performance (for complex flavors).
info	(global) INTEGER. If *info*=0, the execution is successful. *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Refining the Solution and Estimating Its Error

This section describes the ScaLAPACK routines for refining the computed solution of a system of linear equations and estimating the solution error. You can call these routines after factorizing the matrix of the system of equations and computing the solution (see Routines for Matrix Factorization and Solving Systems of Linear Equations).

p?gerfs

Improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution.

Syntax

```
call psgerfs(trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv,
b, ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork,
info)
```

```
call pdgerfs(trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv,
b, ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork,
info)
```

```
call pcgerfs(trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv,
b, ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork,
info)
```

```
call pzgerfs(trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv,
b, ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork,
info)
```

Description

This routine improves the computed solution to one of the systems of linear equations

sub(*A*)*sub(*X*) = sub(*B*),

sub(*A*)^T*sub(*X*) = sub(*B*), or

sub(*A*)^T*sub(*X*) = sub(*B*) and provides error bounds and backward error estimates for the solution.

Here sub(*A*) = *A*(*ia:ia+n-1*, *ja:ja+n-1*), sub(*B*) = *B*(*ib:ib+n-1*, *jb:jb+nrhs-1*), and sub(*X*) = *X*(*ix:ix+n-1*, *jx:jx+nrhs-1*).

Input Parameters

trans (global) CHARACTER*1. Must be 'N' or 'T' or 'C'.
Specifies the form of the system of equations:
If *trans* = 'N', the system has the form sub(*A*)*sub(*X*) = sub(*B*) (No transpose);

	If *trans* = 'T', the system has the form sub(*A*)^T*sub(*X*) = sub(*B*) (Transpose); If *trans* = 'C', the system has the form sub(*A*)^H*sub(*X*) = sub(*B*) (Conjugate transpose).
n	(global) INTEGER. The order of the distributed submatrix sub(*A*) (*n* ≥ 0).
nrhs	(global) INTEGER. The number of right-hand sides, i.e., the number of columns of the matrices sub(*B*) and sub(*X*) (*nrhs* ≥ 0).
a, *af*, *b*, *x*	(local) REAL for psgerfs DOUBLE PRECISION for pdgerfs COMPLEX for pcgerfs DOUBLE COMPLEX for pzgerfs. Pointers into the local memory to arrays of local dimension *a*(*lld_a*, *LOCc*(*ja*+*n*-1)), *af*(*lld_af*,*LOCc*(*jaf*+*n*-1)), *b*(*lld_b*,*LOCc*(*jb*+*nrhs*-1)), and *x*(*lld_x*,*LOCc*(*jx*+*nrhs*-1)), respectively. The array *a* contains the local pieces of the distributed matrix sub(*A*). The array *af* contains the local pieces of the distributed factors of the matrix sub(*A*) = *P***L***U* as computed by p?getrf. The array *b* contains the local pieces of the distributed matrix of right hand sides sub(*B*). On entry, the array *x* contains the local pieces of the distributed solution matrix sub(*X*).
ia, *ja*	(global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
iaf, *jaf*	(global) INTEGER. The row and column indices in the global array *AF* indicating the first row and the first column of the submatrix sub(*AF*), respectively.

descaf (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *AF*.

ib, *jb* (global) INTEGER. The row and column indices in the global array *B* indicating the first row and the first column of the submatrix sub(*B*), respectively.

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

ix, *jx* (global) INTEGER. The row and column indices in the global array *X* indicating the first row and the first column of the submatrix sub(*X*), respectively.

descx (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *X*.

ipiv (local) INTEGER.
Array, dimension *LOCr*(*m_af* + *mb_af*.
This array contains pivoting information as computed by p?getrf. If *ipiv*(i)=j, then the local row i was swapped with the global row j.
This array is tied to the distributed matrix *A*.

work (local)
REAL for psgerfs
DOUBLE PRECISION for pdgerfs
COMPLEX for pcgerfs
DOUBLE COMPLEX for pzgerfs.
The array *work* of dimension (*lwork*) is a workspace array.

lwork (local or global) INTEGER. The dimension of the array *work*.
For real flavors:
lwork must be at least

lwork ≥ 3**LOCr*(*n*+mod(*ia*-1,*mb_a*))

For complex flavors:
lwork must be at least

lwork ≥ 2**LOCr*(*n*+mod(*ia*-1,*mb_a*))

iwork (local) INTEGER. Workspace array, DIMENSION (*liwork*).
Used in real flavors only.

liwork (local or global) INTEGER. The dimension of the array *iwork*; used in real flavors only. Must be at least

	liwork ≥ *LOCr*(*n*+mod(*ib*-1,*mb_b*)).
rwork	(local) REAL for pcgerfs DOUBLE PRECISION for pzgerfs Workspace array, DIMENSION (*lrwork*). Used in complex flavors only.
lrwork	(local or global) INTEGER. The dimension of the array *rwork*; used in complex flavors only. Must be at least *lrwork* ≥ *LOCr*(*n*+mod(*ib*-1,*mb_b*))).

Output Parameters

x	On exit, contains the improved solution vectors.
ferr, *berr*	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, dimension *LOCc*(*jb+nrhs-1*) each. The array *ferr* contains the estimated forward error bound for each solution vector of sub(*X*). If XTRUE is the true solution corresponding to sub(*X*), *ferr* is an estimated upper bound for the magnitude of the largest element in (sub(*X*) - XTRUE) divided by the magnitude of the largest element in sub(*X*). The estimate is as reliable as the estimate for rcond, and is almost always a slight overestimate of the true error. This array is tied to the distributed matrix *X*. The array *berr* contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of sub(*A*) or sub(*B*) that makes sub(*X*) an exact solution). This array is tied to the distributed matrix *X*.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
iwork(1)	On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance (for real flavors).
rwork(1)	On exit, *rwork*(1) contains the minimum value of *lrwork* required for optimum performance (for complex flavors).
info	(global) INTEGER. If *info*=0, the execution is successful.

info < 0:
If the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?porfs

Improves the computed solution to a system of linear equations with symmetric/Hermitian positive definite distributed matrix and provides error bounds and backward error estimates for the solution.

Syntax

call psporfs(*uplo*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *af*, *iaf*, *jaf*, *descaf*, *b*, *ib*, *jb*, *descb*, *x*, *ix*, *jx*, *descx*, *ferr*, *berr*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call pdporfs(*uplo*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *af*, *iaf*, *jaf*, *descaf*, *b*, *ib*, *jb*, *descb*, *x*, *ix*, *jx*, *descx*, *ferr*, *berr*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call pcporfs(*uplo*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *af*, *iaf*, *jaf*, *descaf*, *b*, *ib*, *jb*, *descb*, *x*, *ix*, *jx*, *descx*, *ferr*, *berr*, *work*, *lwork*, *rwork*, *lrwork*, *info*)

call pzporfs(*uplo*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *af*, *iaf*, *jaf*, *descaf*, *b*, *ib*, *jb*, *descb*, *x*, *ix*, *jx*, *descx*, *ferr*, *berr*, *work*, *lwork*, *rwork*, *lrwork*, *info*)

Description

The routine p?porfs improves the computed solution to the system of linear equations

sub(*A*)*sub(*X*) = sub(*B*),

where sub(*A*) = *A*(*ia:ia+n-1*, *ja:ja+n-1*) is a real symmetric or complex Hermitian positive definite distributed matrix and

sub(*B*) = *B*(*ib:ib+n-1*, *jb:jb+nrhs-1*),

sub(*X*) = *X*(*ix:ix+n-1*, *jx:jx+nrhs-1*)

are right-hand side and solution submatrices, respectively. This routine also provides error bounds and backward error estimates for the solution.

Input Parameters

uplo	(global) CHARACTER*1. Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix sub(*A*) is stored. If *uplo* = 'U', sub(*A*) is upper triangular. If *uplo* = 'L', sub(*A*) is lower triangular.
n	(global) INTEGER. The order of the distributed matrix sub(*A*) (*n*≥0).
nrhs	(global) INTEGER. The number of right-hand sides, i.e., the number of columns of the matrices sub(*B*) and sub(*X*) (*nrhs*≥0).
a, *af*, *b*, *x*	(local) REAL for psporfs DOUBLE PRECISION for pdporfs COMPLEX for pcporfs DOUBLE COMPLEX for pzporfs. Pointers into the local memory to arrays of local dimension *a*(*lld_a*,*LOCc*(*ja*+*n*-1)), *af*(*lld_af*,*LOCc*(*ja*+*n*-1)), *b*(*lld_b*,*LOCc*(*jb*+*nrhs*-1)), and *x*(*lld_x*,*LOCc*(*jx*+*nrhs*-1)), respectively. The array *a* contains the local pieces of the *n*-by-*n* symmetric/Hermitian distributed matrix sub(*A*). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced. The array *af* contains the factors *L* or *U* from the Cholesky factorization sub(*A*) = $L*L^H$ or sub(*A*) = U^H*U, as computed by p?potrf. On entry, the array *b* contains the local pieces of the distributed matrix of right hand sides sub(*B*). On entry, the array *x* contains the local pieces of the solution vectors sub(*X*).

ia, *ja* (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

iaf, *jaf* (global) INTEGER. The row and column indices in the global array *AF* indicating the first row and the first column of the submatrix sub(*AF*), respectively.

descaf (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *AF*.

ib, *jb* (global) INTEGER. The row and column indices in the global array *B* indicating the first row and the first column of the submatrix sub(*B*), respectively.

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

ix, *jx* (global) INTEGER. The row and column indices in the global array *X* indicating the first row and the first column of the submatrix sub(*X*), respectively.

descx (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *X*.

work (local)
REAL for psporfs
DOUBLE PRECISION for pdporfs
COMPLEX for pcporfs
DOUBLE COMPLEX for pzporfs.
The array *work* of dimension (*lwork*) is a workspace array.

lwork (local) INTEGER. The dimension of the array *work*.
For real flavors:
lwork must be at least

lwork ≥ 3**LOCr*(*n*+mod(*ia*-1,*mb_a*))

For complex flavors:
lwork must be at least

lwork ≥ 2**LOCr*(*n*+mod(*ia*-1,*mb_a*))

iwork (local) INTEGER. Workspace array, DIMENSION (*liwork*).
Used in real flavors only.

liwork	(local or global) INTEGER. The dimension of the array *iwork*; used in real flavors only. Must be at least *liwork* ≥ *LOCr*(*n*+mod(*ib*-1,*mb_b*)).
rwork	(local) REAL for pcporfs DOUBLE PRECISION for pzporfs Workspace array, DIMENSION (*lrwork*). Used in complex flavors only.
lrwork	(local or global) INTEGER. The dimension of the array *rwork*; used in complex flavors only. Must be at least *lrwork* ≥ *LOCr*(*n*+mod(*ib*-1,*mb_b*))).

Output Parameters

x	On exit, contains the improved solution vectors.
ferr, *berr*	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, dimension *LOCc*(*jb+nrhs-1*) each. The array *ferr* contains the estimated forward error bound for each solution vector of sub(*X*). If XTRUE is the true solution corresponding to sub(*X*), *ferr* is an estimated upper bound for the magnitude of the largest element in (sub(*X*) - XTRUE) divided by the magnitude of the largest element in sub(*X*). The estimate is as reliable as the estimate for rcond, and is almost always a slight overestimate of the true error. This array is tied to the distributed matrix *X*. The array *berr* contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of sub(*A*) or sub(*B*) that makes sub(*X*) an exact solution). This array is tied to the distributed matrix *X*.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
iwork(1)	On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance (for real flavors).
rwork(1)	On exit, *rwork*(1) contains the minimum value of *lrwork* required for optimum performance (for complex flavors).

info	(global) INTEGER. If *info*=0, the execution is successful. *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?trrfs

Provides error bounds and backward error estimates for the solution to a system of linear equations with a distributed triangular coefficient matrix.

Syntax

call pstrrfs(*uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork, info*)

call pdtrrfs(*uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork, info*)

call pctrrfs(*uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork, info*)

call pztrrfs(*uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork, info*)

Description

The routine p?trrfs provides error bounds and backward error estimates for the solution to one of the systems of linear equations

sub(*A*)*sub(*X*) = sub(*B*),

sub(*A*)^T*sub(*X*) = sub(*B*), or

sub(*A*)^H*sub(*X*) = sub(*B*) ,

where sub(*A*) = *A*(*ia:ia+n-1, ja:ja+n-1*) is a triangular matrix,

sub(*B*) = *B*(*ib:ib+n-1, jb:jb+nrhs-1*), and

sub(*X*) = *X*(*ix:ix+n-1, jx:jx+nrhs-1*).

The solution matrix *X* must be computed by p?trtrs or some other means before entering this routine. The routine p?trrfs does not do iterative refinement because doing so cannot improve the backward error.

Input Parameters

uplo	(global) CHARACTER*1. Must be 'U' or 'L'. If *uplo* = 'U', sub(*A*) is upper triangular. If *uplo* = 'L', sub(*A*) is lower triangular.
trans	(global) CHARACTER*1. Must be 'N' or 'T' or 'C'. Specifies the form of the system of equations: If *trans* = 'N', the system has the form sub(*A*)*sub(*X*) = sub(*B*) (No transpose); If *trans* = 'T', the system has the form sub((*A*)^T*sub(*X*) = sub(*B*) (Transpose); If *trans* = 'C', the system has the form sub(*A*)^H*sub(*X*) = sub(*B*) (Conjugate transpose).
diag	CHARACTER*1. Must be 'N' or 'U'. If *diag* = 'N', then sub(*A*) is non-unit triangular. If *diag* = 'U', then sub(*A*) is unit triangular.
n	(global) INTEGER. The order of the distributed matrix sub(*A*) (*n*≥0).
nrhs	(global) INTEGER. The number of right-hand sides, that is, the number of columns of the matrices sub(*B*) and sub(*X*) (*nrhs*≥0).
a, *b*, *x*	(local) REAL for pstrrfs DOUBLE PRECISION for pdtrrfs COMPLEX for pctrrfs DOUBLE COMPLEX for pztrrfs. Pointers into the local memory to arrays of local dimension *a*(*lld_a*,*LOCc*(*ja*+*n*-1)), *b*(*lld_b*,*LOCc*(*jb*+*nrhs*-1)), and *x*(*lld_x*,*LOCc*(*jx*+*nrhs*-1)), respectively. The array *a* contains the local pieces of the original triangular distributed matrix sub(*A*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced.
If *diag* = 'U', the diagonal elements of sub(*A*) are also not referenced and are assumed to be 1.
On entry, the array *b* contains the local pieces of the distributed matrix of right hand sides sub(*B*).
On entry, the array *x* contains the local pieces of the solution vectors sub(*X*).

ia, *ja* — (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

ib, *jb* — (global) INTEGER. The row and column indices in the global array *B* indicating the first row and the first column of the submatrix sub(*B*), respectively.

descb — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

ix, *jx* — (global) INTEGER. The row and column indices in the global array *X* indicating the first row and the first column of the submatrix sub(*X*), respectively.

descx — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *X*.

work — (local)
REAL for pstrrfs
DOUBLE PRECISION for pdtrrfs
COMPLEX for pctrrfs
DOUBLE COMPLEX for pztrrfs.
The array *work* of dimension (*lwork*) is a workspace array.

lwork — (local) INTEGER. The dimension of the array *work*.
For real flavors:

	lwork must be at least *lwork* ≥ 3*LOCr(*n*+mod(*ia*-1,*mb_a*)) For complex flavors: *lwork* must be at least *lwork* ≥ 2*LOCr(*n*+mod(*ia*-1,*mb_a*))
iwork	(local) INTEGER. Workspace array, DIMENSION (*liwork*). Used in real flavors only.
liwork	(local or global) INTEGER. The dimension of the array *iwork*; used in real flavors only. Must be at least *liwork* ≥ *LOCr*(*n*+mod(*ib*-1,*mb_b*)).
rwork	(local) REAL for pctrrfs DOUBLE PRECISION for pztrrfs Workspace array, DIMENSION (*lrwork*). Used in complex flavors only.
lrwork	(local or global) INTEGER. The dimension of the array *rwork*; used in complex flavors only. Must be at least *lrwork* ≥ *LOCr*(*n*+mod(*ib*-1,*mb_b*)).

Output Parameters

ferr, *berr*	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, dimension *LOCc*(*jb+nrhs-1*) each. The array *ferr* contains the estimated forward error bound for each solution vector of sub(*X*). If XTRUE is the true solution corresponding to sub(*X*), *ferr* is an estimated upper bound for the magnitude of the largest element in (sub(*X*) - XTRUE) divided by the magnitude of the largest element in sub(*X*). The estimate is as reliable as the estimate for rcond, and is almost always a slight overestimate of the true error. This array is tied to the distributed matrix *X*.

	The array *berr* contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of sub(*A*) or sub(*B*) that makes sub(*X*) an exact solution). This array is tied to the distributed matrix *X*.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
iwork(1)	On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance (for real flavors).
rwork(1)	On exit, *rwork*(1) contains the minimum value of *lrwork* required for optimum performance (for complex flavors).
info	(global) INTEGER. If *info*=0, the execution is successful. *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Routines for Matrix Inversion

This sections describes ScaLAPACK routines that compute the inverse of a matrix based on the previously obtained factorization. Note that it is not recommended to solve a system of equations $Ax = b$ by first computing A^{-1} and then forming the matrix-vector product $x = A^{-1}b$. Call a solver routine instead (see Solving Systems of Linear Equations); this is more efficient and more accurate.

p?getri

Computes the inverse of a LU-factored distributed matrix.

Syntax

```
call psgetri(n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork, info)
call pdgetri(n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork, info)
call pcgetri(n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork, info)
call pzgetri(n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork, info)
```

Description

This routine computes the inverse of a general distributed matrix sub(*A*) = *A*(*ia:ia+n-1*, *ja:ja+n-1*) using the *LU* factorization computed by p?getrf. This method inverts *U* and then computes the inverse of sub(*A*) by solving the system

inv(sub(*A*))**L* = inv(*U*)

for inv(sub(*A*)).

Input Parameters

n	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub(*A*) (*n*≥0).
a	(local) REAL for psgetri DOUBLE PRECISION for pdgetri COMPLEX for pcgetri DOUBLE COMPLEX for pzgetri. Pointer into the local memory to an array of local dimension *a*(*lld_a*,*LOCc*(*ja*+*n*-1)). On entry, the array *a* contains the local pieces of the *L* and *U* obtained by the factorization sub(*A*) = *P*L*U* computed by p?getrf.
ia, *ja*	(global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
work	(local) REAL for psgetri DOUBLE PRECISION for pdgetri COMPLEX for pcgetri DOUBLE COMPLEX for pzgetri. The array *work* of dimension (*lwork*) is a workspace array.
lwork	(local) INTEGER. The dimension of the array *work*. *lwork* must be at least *lwork*≥*LOCr*(*n*+mod(*ia*-1,*mb_a*))**nb_a*.

The array *work* is used to keep at most an entire column block of sub(*A*).

iwork (local) INTEGER. Workspace array used for physically transposing the pivots, DIMENSION (*liwork*).

liwork (local or global) INTEGER. The dimension of the array *iwork*. The minimal value *liwork* of is determined by the following code:

```
if NPROW == NPCOL then
 liwork = LOCc(n_a + mod(ja-1,nb_a))+ nb_a

else

 liwork = LOCc(n_a + mod(ja-1,nb_a)) +

max(ceil(ceil(LOCr(m_a)/mb_a)/(lcm/NPROW)),nb_a)
end if
```

where lcm is the least common multiple of process rows and columns (NPROW and NPCOL).

Output Parameters

ipiv (local) INTEGER.
Array, dimension (*LOCr*(*m_a*)+ *mb_a*).
This array contains the pivoting information.
If *ipiv*(i)=j, then the local row i was swapped with the global row j.
This array is tied to the distributed matrix *A*.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

iwork(1) On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance.

info (global) INTEGER. If *info*=0, the execution is successful.
info < 0:

If the i-th argument is an array and the j-th entry had an illegal value, then $info = -(i*100+j)$; if the i-th argument is a scalar and had an illegal value, then $info = -i$.

$info > 0$:

If $info = i$, $U(i,i)$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, and division by zero will occur if it is used to solve a system of equations.

p?potri

Computes the inverse of a symmetric/Hermitian positive definite distributed matrix.

Syntax

```
call pspotri(uplo, n, a, ia, ja, desca, info)
call pdpotri(uplo, n, a, ia, ja, desca, info)
call pcpotri(uplo, n, a, ia, ja, desca, info)
call pzpotri(uplo, n, a, ia, ja, desca, info)
```

Description

This routine computes the inverse of a real symmetric or complex Hermitian positive definite distributed matrix $sub(A) = A(ia:ia+n-1, ja:ja+n-1)$ using the Cholesky factorization $sub(A) = U^H*U$ or $sub(A) = L*L^H$ computed by p?potrf.

Input Parameters

uplo (global) CHARACTER*1. Must be 'U' or 'L'.
Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix sub(A) is stored.
If $uplo$ = 'U', upper triangle of sub(A) is stored. If $uplo$ = 'L', lower triangle of sub(A) is stored.

n (global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub(A) ($n \geq 0$).

a (local)
REAL for pspotri
DOUBLE PRECISION for pdpotri
COMPLEX for pcpotri
DOUBLE COMPLEX for pzpotri.
Pointer into the local memory to an array of local dimension *a*(*lld_a*,*LOCc*(*ja*+*n*-1)).
On entry, the array *a* contains the local pieces of the triangular factor *U* or *L* from the Cholesky factorization sub(*A*) = U^H*U, or sub(*A*) = *L*L^H, as computed by p?potrf.

ia, *ja* (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

a On exit, overwritten by the local pieces of the upper or lower triangle of the (symmetric/Hermitian) inverse of sub(*A*).

info (global) INTEGER. If *info*=0, the execution is successful.
info < 0:
If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
info > 0:
If *info* = *i*, the (*i*, *i*) element of the factor *U* or *L* is zero, and the inverse could not be computed.

p?trtri

Computes the inverse of a triangular distributed matrix.

Syntax

```
call pstrtri(uplo, diag, n, a, ia, ja, desca, info)
call pdtrtri(uplo, diag, n, a, ia, ja, desca, info)
call pctrtri(uplo, diag, n, a, ia, ja, desca, info)
call pztrtri(uplo, diag, n, a, ia, ja, desca, info)
```

Description

This routine computes the inverse of a real or complex upper or lower triangular distributed matrix `sub(A) = A(ia:ia+n-1, ja:ja+n-1)`.

Input Parameters

uplo
(global) `CHARACTER*1`. Must be `'U'` or `'L'`.
Specifies whether the distributed matrix sub(*A*) is upper or lower triangular.
If *uplo* = `'U'`, sub(*A*) is upper triangular.
If *uplo* = `'L'`, sub(*A*) is lower triangular.

diag
`CHARACTER*1`. Must be `'N'` or `'U'`.
Specifies whether or not the distributed matrix sub(*A*) is unit triangular.
If *diag* = `'N'`, then sub(*A*) is non-unit triangular.
If *diag* = `'U'`, then sub(*A*) is unit triangular.

n
(global) `INTEGER`. The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub(*A*) (*n*≥0).

a
(local)
`REAL` for `pstrtri`
`DOUBLE PRECISION` for `pdtrtri`
`COMPLEX` for `pctrtri`
`DOUBLE COMPLEX` for `pztrtri`.

Pointer into the local memory to an array of local dimension *a*(*lld_a*,*LOCc*(*ja*+*n*-1)).
The array *a* contains the local pieces of the triangular distributed matrix sub(*A*).
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular matrix to be inverted, and the strictly lower triangular part of sub(*A*) is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular matrix, and the strictly upper triangular part of sub(*A*) is not referenced.

ia, *ja*
(global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca
(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

a
On exit, overwritten by the (triangular) inverse of the original matrix.

info
(global) INTEGER. If *info*=0, the execution is successful.
info < 0:
If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
info > 0:
If *info* = *k*, *A*(*ia*+*k*-1, *ja*+*k*-1) is exactly zero. The triangular matrix sub(*A*) is singular and its inverse can not be computed.

Routines for Matrix Equilibration

ScaLAPACK routines described in this section are used to compute scaling factors needed to equilibrate a matrix. Note that these routines do not actually scale the matrices.

p?geequ

Computes row and column scaling factors intended to equilibrate a general rectangular distributed matrix and reduce its condition number.

Syntax

```
call psgeequ(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
call pdgeequ(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
call pcgeequ(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
call pzgeequ(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
```

Description

This routine computes row and column scalings intended to equilibrate an *m*-by-*n* distributed matrix sub(*A*) = *A*(*ia:ia+m-1, ja:ja+n-1*) and reduce its condition number. The output array *r* returns the row scale factors and the array *c* the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix *B* with elements b_{ij}=*r*(i)**a*$_{ij}$**c*(*j*) have absolute value 1.

r(i) and *c*(j) are restricted to be between *SMLNUM* = smallest safe number and *BIGNUM* = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of sub(*A*) but works well in practice.

The auxiliary function p?laqge uses scaling factors computed by p?geequ to scale a general rectangular matrix.

Input Parameters

m	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(*A*) (*m* ≥ 0).
n	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*) (*n* ≥ 0).
a	(local) REAL for psgeequ DOUBLE PRECISION for pdgeequ

COMPLEX for pcgeequ
DOUBLE COMPLEX for pzgeequ .
Pointer into the local memory to an array of local dimension *a*(*lld_a*,*LOCc*(*ja*+*n*-1)).
The array *a* contains the local pieces of the *m*-by-*n* distributed matrix whose equilibration factors are to be computed.

ia, *ja* — (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

r, *c* — (local) REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
Arrays, dimension *LOCr*(*m_a*) and *LOCc*(*n_a*), respectively.
If *info* = 0, or *info* > *ia*+*m*-*1*, the array *r* (*ia*:*ia*+*m*-*1*) contains the row scale factors for sub(*A*). *r* is aligned with the distributed matrix *A*, and replicated across every process column. *r* is tied to the distributed matrix *A*.
If *info* = 0, the array *c* (*ja*:*ja*+*n*-*1*) contains the column scale factors for sub(*A*). *c* is aligned with the distributed matrix *A*, and replicated down every process row. *c* is tied to the distributed matrix *A*.

rowcnd, *colcnd* — (global) REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
If *info* = 0 or *info* > *ia*+*m*-*1*, *rowcnd* contains the ratio of the smallest *r*(i) to the largest *r*(i) (*ia* ≤ i ≤ *ia*+*m*-*1*). If *rowcnd* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *r* (*ia*:*ia*+*m*-*1*).
If *info* = 0, *colcnd* contains the ratio of the smallest *c*(j) to the largest *c*(j) (*ja* ≤ j ≤ *ja*+*n*-*1*).
If *colcnd* ≥ 0.1, it is not worth scaling by *c*(*ja*:*ja*+*n*-*1*).

amax — (global) REAL for single precision flavors;

	DOUBLE PRECISION for double precision flavors. Absolute value of the largest matrix element. If *amax* is very close to overflow or very close to underflow, the matrix should be scaled.
info	(global) INTEGER. If *info*=0, the execution is successful. *info* < 0: If the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*. *info* > 0: If *info* = *i* and *i* ≤ *m*, the *i*th row of the distributed matrix sub(*A*) is exactly zero; *i* > *m*, the (*i*-*m*)th column of the distributed matrix sub(*A*) is exactly zero.

p?poequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite distributed matrix and reduce its condition number.

Syntax

```
call pspoequ(n, a, ia, ja, desca, sr, sc, scond, amax, info)
call pdpoequ(n, a, ia, ja, desca, sr, sc, scond, amax, info)
call pcpoequ(n, a, ia, ja, desca, sr, sc, scond, amax, info)
call pzpoequ(n, a, ia, ja, desca, sr, sc, scond, amax, info)
```

Description

This routine computes row and column scalings intended to equilibrate a real symmetric or complex Hermitian positive definite distributed matrix sub(*A*) = *A*(*ia:ia+n-1*, *ja:ja+n-1*) and reduce its condition number (with respect to the two-norm). The output arrays *sr* and *sc* return the row and column scale factors

$$s(i) = \frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled distributed matrix *B* with elements b_{ij}=*s*(i)**a*$_{ij}$**s*(j) has ones on the diagonal.

This choice of *sr* and *sc* puts the condition number of *B* within a factor *n* of the smallest possible condition number over all possible diagonal scalings.

The auxiliary function p?laqsy uses scaling factors computed by p?geequ to scale a general rectangular mtrix.

Input Parameters

n	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub(*A*) (*n*≥0).
a	(local) REAL for pspoequ DOUBLE PRECISION for pdpoequ COMPLEX for pcpoequ DOUBLE COMPLEX for pzpoequ. Pointer into the local memory to an array of local dimension *a*(*lld_a*,*LOCc*(*ja*+*n*-1)). The array *a* contains the *n*-by-*n* symmetric/Hermitian positive definite distributed matrix sub(*A*) whose scaling factors are to be computed. Only the diagonal elements of sub(*A*) are referenced.
ia, *ja*	(global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

sr, *sc*	(local)

REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
Arrays, dimension *LOCr(m_a)* and *LOCc(n_a)*, respectively.
If *info* = 0, the array *sr(ia:ia+n-1)* contains the row scale factors for sub(*A*). *sr* is aligned with the distributed matrix *A*, and replicated across every process column. *sr* is tied to the distributed matrix *A*.
If *info* = 0, the array *sc (ja:ja+n-1)* contains the column scale factors for sub(*A*). *sc* is aligned with the distributed matrix *A*, and replicated down every process row. *sc* is tied to the distributed matrix *A*.

scond (global)
REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
If *info* = 0, *scond* contains the ratio of the smallest *sr*(i) (or *sc*(j)) to the largest *sr*(i) (or *sc*(j)), with *ia*≤i≤*ia+n-1* and *ja*≤j≤*ja+n-1*.
If *scond* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *sr* (or *sc*).

amax (global)
REAL for single precision flavors;
DOUBLE PRECISION for double precision flavors.
Absolute value of the largest matrix element. If *amax* is very close to overflow or very close to underflow, the matrix should be scaled.

info (global) INTEGER.
If *info*=0, the execution is successful.
info < 0:
If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
info > 0:
If *info* = *k*, the *k*-th diagonal entry of sub(*A*) is nonpositive.

Orthogonal Factorizations

This section describes the ScaLAPACK routines for the *QR (RQ)* and *LQ (QL)* factorization of matrices. Routines for the *RZ* factorization as well as for generalized *QR* and *RQ* factorizations are also included. For the mathematical definition of the factorizations, see the respective LAPACK sections or refer to [SLUG].

Table 6-3 lists ScaLAPACK routines that perform orthogonal factorization of matrices.

Table 6-3 Computational Routines for Orthogonal Factorizations

Matrix type, factorization	Factorize without pivoting	Factorize with pivoting	Generate matrix Q	Apply matrix Q
general matrices, QR factorization	p?geqrf	p?geqpf	p?orgqr p?ungqr	p?ormqrp?unmqr
general matrices, RQ factorization	p?gerqf		p?orgrqp?ungrq	p?ormrqp?unmrq
general matrices, LQ factorization	p?gelqf		p?orglqp?unglq	p?ormlqp?unmlq
general matrices, QL factorization	p?geqlf		p?orgqlp?ungql	p?ormqlp?unmql
trapezoidal matrices, RZ factorization	p?tzrzf			p?ormrzp?unmrz
pair of matrices, generalized QR factorization	p?ggqrf			
pair of matrices, generalized RQ factorization	p?ggrqf			

p?geqrf

Computes the QR factorization of a general m-by-n matrix.

Syntax

```
call psgeqrf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgeqrf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgeqrf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgeqrf( m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine forms the *QR* factorization of a general *m*-by-*n* distributed matrix sub(*A*) = *A*(*ia:ia*+*m*-1,*ja:ja*+*n*-1) as

A=*Q***R*

Input Parameters

m	(global) INTEGER. The number of rows in the distributed submatrix sub(*A*); (*m* ≥ 0).
n	(global) INTEGER. The number of columns in the distributed submatrix sub(*A*); (*n* ≥ 0).
a	(local) REAL for psgeqrf DOUBLE PRECISION for pdgeqrf COMPLEX for pcgeqrf DOUBLE COMPLEX for pzgeqrf. Pointer into the local memory to an array of local dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). Contains the local pieces of the distributed matrix sub(*A*) to be factored.
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*(*ia:ia*+*m*-*1*,*ja:ja*+*n*-*1*), respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*
work	(local). REAL for psgeqrf DOUBLE PRECISION for pdgeqrf. COMPLEX for pcgeqrf. DOUBLE COMPLEX for pzgeqrf Workspace array of dimension *lwork*.
lwork	(local or global) INTEGER, dimension of *work*, must be at least *lwork* ≥ *nb_a* * (*mp*0+*nq*0+*nb_a*), where *iroff* = mod(*ia*-1, *mb_a*), *icoff* = mod(*ja*-1, *nb_a*), *iarow* = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
*mp*0 = numroc(*m*+*iroff*, *mb_a*, MYROW, *iarow*, NPROW),
*nq*0 = numroc(*n*+*icoff*, *nb_a*, MYCOL, *iacol*, NPCOL),
and numroc, indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = –1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a
The elements on and above the diagonal of sub(*A*) contain the min(*m*,*n*)-by-*n* upper trapezoidal matrix *R* (*R* is upper triangular if *m* ≥ *n*); the elements below the diagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see Application Notes below).

tau
(local)
REAL for psgeqrf
DOUBLE PRECISION for pdgeqrf
COMPLEX for pcgeqrf
DOUBLE COMPLEX for pzgeqrf.
Array, DIMENSION *LOCc*(*ja*+min(*m*,*n*)–1).
Contains the scalar factor *tau* of elementary reflectors. *tau* is tied to the distributed matrix *A*.

work(1)
On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info
(global) INTEGER.
= 0, the execution is successful.
< 0, if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = – (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = –*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = H(ja) H(ja+1)... H(ja+k-1),

where *k = min(m,n)*.

Each *H*(*i*) has the form

*H(j) = I - tau * v * v'*

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1:*i*-1) = 0 and *v*(*i*) = 1; *v*(*i*+1:*m*) is stored on exit in *A*(*ia*+*i*:*ia*+*m*-1, *ja*+*i*-1), and *tau* in *tau*(*ja*+*i*-1).

p?geqpf

Computes the QR factorization of a general m-by-n matrix with pivoting.

Syntax

```
call psgeqpf( m, n, a, ia, ja, desca, ipiv, tau, work, lwork, info)
call pdgeqpf( m, n, a, ia, ja, desca, ipiv, tau, work, lwork, info)
call pcgeqpf( m, n, a, ia, ja, desca, ipiv, tau, work, lwork, info)
call pzgeqpf( m, n, a, ia, ja, desca, ipiv, tau, work, lwork, info)
```

Description

The routine forms the *QR* factorization with column pivoting of a general *m*-by-*n* distributed matrix sub(*A*)= *A*(*ia:ia*+*m*-1,*ja:ja*+*n*-1) as

sub(*A*)**P*=*Q***R*

Input Parameters

m	(global) INTEGER. The number of rows in the submatrix sub(*A*) (*m* ≥ 0).
n	(global) INTEGER. The number of columns in the submatrix sub(*A*) (*n* ≥ 0).
a	(local) REAL for psgeqpf

DOUBLE PRECISION for pdgeqpf
COMPLEX for pcgeqpf
DOUBLE COMPLEX for pzgeqpf.
Pointer into the local memory to an array of local dimension (*lld_a*, *LOCc*(*ja*+*n*-1)).
Contains the local pieces of the distributed matrix sub(*A*) to be factored.

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*(*ia:ia*+*m*-1,*ja:ja*+*n*-1), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

work (local).
REAL for psgeqpf
DOUBLE PRECISION for pdgeqpf.
COMPLEX for pcgeqpf.
DOUBLE COMPLEX for pzgeqpf
Workspace array of dimension *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least
For real flavors:

lwork ≥ max(3,*mp*0+*nq*0) + *LOCc* (*ja*+*n*-*1*) + *nq*0.

For complex flavors:

lwork ≥ max(3,*mp*0+*nq*0) .

Here
iroff = mod(*ia*-1, *mb_a*), *icoff* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),
iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
*mp*0 = numroc(*m*+*iroff*, *mb_a*, MYROW, *iarow*, NPROW),
*nq*0 = numroc(*n*+*icoff*, *nb_a*, MYCOL, *iacol*, NPCOL),
LOCc (*ja*+*n*-*1*) = numroc(*ja*+*n*-*1*, *nb_a*, MYCOL,*csrc_a*, NPCOL), and numroc, indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — The elements on and above the diagonal of sub(*A*)contain the min(*m*, *n*)-by-*n* upper trapezoidal matrix *R* (*R* is upper triangular if *m* ≥ *n*); the elements below the diagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see Application Notes below)

ipiv — (local) INTEGER. Array, DIMENSION *LOCc*(*ja*+*n*-1). *ipiv*(*i*) = *k*, the local *i*-th column of sub(*A*)*P* was the global *k*-th column of sub(*A*). *ipiv* is tied to the distributed matrix *A*.

tau — (local)
REAL for psgeqpf
DOUBLE PRECISION for pdgeqpf
COMPLEX for pcgeqpf
DOUBLE COMPLEX for pzgeqpf.
Array, DIMENSION *LOCc*(*ja*+min(*m*, *n*)-1)).
Contains the scalar factor *tau* of elementary reflectors. *tau* is tied to the distributed matrix *A*.

work(1) — On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0, the execution is successful.
< 0, if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = H(1) H(2)... H(n),

Each H(i) has the form

H = I - tau * v * v'

where tau is a real/complex scalar, and v is a real/complex vector with v(1:i-1) = 0 and v(i) = 1; v(i+1:m) is stored on exit in A(ia+i-1:ia+m-1,ja+i-1).

The matrix P is represented in jpvt as follows: if jpvt(j)= i then the j-th column of P is the i-th canonical unit vector.

p?orgqr

Generates the orthogonal matrix Q of the QR factorization formed by p?geqrf.

Syntax

call psorgqr(m, n, k, a, ia, ja, desca, tau, work, lwork, info)

call pdorgqr(m, n, k, a, ia, ja, desca, tau, work, lwork, info)

Description

The routine generates the whole or part of m-by-n real distributed matrix Q denoting A(ia:ia+m-1, ja:ja+n-1) with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m

Q= H(1) H(2)...H(k)

as returned by p?geqrf.

Input Parameters

m	(global) INTEGER. The number of rows in the submatrix sub(Q) (m ≥ 0).
n	(global) INTEGER. The number of columns in the submatrix sub(Q) (m ≥ n ≥ 0).
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q (n ≥ k ≥ 0).
a	(local) REAL for psorgqr

DOUBLE PRECISION for pdorgqr
Pointer into the local memory to an array of local dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). The *j*-th column must contain the vector which defines the elementary reflector *H*(*j*), *ja*≤*j*≤*ja* +*k*-1, as returned by p?geqrf in the *k* columns of its distributed matrix argument *A*(*ia*:*, *ja*:*ja*+*k*-1).

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*(*ia:ia*+*m-1*,*ja:ja*+*n-1*), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)
REAL for psorgqr
DOUBLE PRECISION for pdorgqr
Array, DIMENSION *LOCc*(*ja*+*k*-1)).
Contains the scalar factor *tau* (*j*) of elementary reflectors *H*(*j*) as returned by p?geqrf. *tau* is tied to the distributed matrix *A*.

work (local)
REAL for psorgqr
DOUBLE PRECISION for pdorgqr
Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*.

Must be at least *lwork* ≥ *nb_a**(*nqa*0 + *mpa*0 + *nb_a*), where
iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),
iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),
*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL);
indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a	Contains the local pieces of the *m*-by-*n* distributed matrix Q.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?ungqr

Generates the complex unitary matrix Q of the QR factorization formed by p?geqrf.

Syntax

```
call pcungqr( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungqr( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine generates the whole or part of *m*-by-*n* complex distributed matrix *Q* denoting *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) with orthonormal columns, which is defined as the first *n* columns of a product of *k* elementary reflectors of order m

Q = *H*(1) *H*(2)... *H*(*k*)

as returned by p?geqrf.

Input Parameters

m	(global) INTEGER. The number of rows in the submatrix sub(*Q*); (*m*≥0).
n	(global) INTEGER. The number of columns in the submatrix sub(*Q*) (*m*≥*n*≥0).
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q* (*n*≥*k*≥0).
a	(local) COMPLEX for pcungqr DOUBLE COMPLEX for pzungqr Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)).The *j*-th column must contain the vector which defines the elementary reflector *H*(*j*), *ja*≤ *j*≤ *ja* +*k*-1, as returned by p?geqrf in the *k* columns of its distributed matrix argument *A*(*ia*:*, ja*:*ja*+*k*-1).
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
tau	(local) COMPLEX for pcungqr DOUBLE COMPLEX for pzungqr Array, DIMENSION *LOCc*(*ja*+*k*-1)). Contains the scalar factor *tau* (*j*) of elementary reflectors *H*(*j*) as returned by p?geqrf. *tau* is tied to the distributed matrix *A*.
work	(local) COMPLEX for pcungqr DOUBLE COMPLEX for pzungqr Workspace array of dimension of *lwork*.
lwork	(local or global) INTEGER, dimension of *work*, must be at least *lwork* ≥ *nb_a**(*nqa*0 + *mpa*0 + *nb_a*), where *iroffa* = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),
iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),
*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL)
indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a	Contains the local pieces of the *m*-by-*n* distributed matrix *Q*.
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?ormqr

Multiplies a general matrix by the orthogonal matrix Q of the QR factorization formed by p?geqrf.

Syntax

```
call psormqr( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

```
call pdormqr( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

Description

The routine overwrites the general real m-by-n distributed matrix sub(C) = $C(ic:ic+m-1, jc:jc+n-1)$ with

	side ='L'	*side* ='R'
trans = 'N':	Q*sub(C)	sub(C)*Q
trans = 'T':	Q^T*sub(C)	sub(C)*Q^T

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$Q = H(1)\ H(2) \ldots\ H(k)$

as returned by p?geqrf. Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

side (global) CHARACTER
='L': Q or Q^T is applied from the left.
='R': Q or Q^T is applied from the right.

trans (global) CHARACTER
='N', no transpose, Q is applied.
='T', transpose, Q^T is applied.

m (global) INTEGER. The number of rows in the distributed matrix sub(C) ($m \geq 0$).

n (global) INTEGER. The number of columns in the distributed matrix sub(C) ($n \geq 0$).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints:
If *side* = 'L', $m \geq k \geq 0$
If *side* = 'R', $n \geq k \geq 0$.

a (local)
REAL for psormqr
DOUBLE PRECISION for pdormqr.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*($ja+k-1$)). The j-th column must contain the vector which defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by p?geqrf in the k columns of

its distributed matrix argument *A*(*ia*:*, *ja*:*ja*+*k*−1). *A*(*ia*:*, *ja*:*ja*+*k*−1) is modified by the routine but restored on exit.

If *side* = 'L', *lld_a* ≥ max(1, *LOCr*(*ia*+*m*−1)

If *side* ='R', *lld_a* ≥ max(1, *LOCr*(*ia*+*n*−1)

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)
REAL for psormqr
DOUBLE PRECISION for pdormqr
Array, DIMENSION *LOCc*(*ja*+*k*−1).
Contains the scalar factor *tau* (*j*) of elementary reflectors *H*(*j*) as returned by p?geqrf. *tau* is tied to the distributed matrix *A*.

c (local)
REAL for psormqr
DOUBLE PRECISION for pdormqr
Pointer into the local memory to an array of local dimension (*lld_c*, *LOCc*(*jc*+*n*−1)).
Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic, *jc* (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)
REAL for psormqr
DOUBLE PRECISION for pdormqr.
Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:
if *side* = 'L',

lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0+*mpc*0)**nb_a*) + *nb_a***nb_a*
else if *side* = 'R',
lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0+max(*npa*0+numroc(numroc(*n*+*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a***nb_a*
end if
where
lcmq = *lcm*/NPCOL with *lcm* = ilcm(NPROW, NPCOL),
iroffa = mod(*ia*-1, *mb_a*),
icoffa = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),
*npa*0= numroc(*n*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),
iroffc = mod(*ic*-1, *mb_c*),
icoffc = mod(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),
iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),
*mpc*0= numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),
*nqc*0= numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — Overwritten by the product *Q**sub(*C*), or Q^T*sub(*C*), or sub(*C*)*Q^T, or sub(*C*)**Q*.

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?unmqr

Multiplies a complex matrix by the unitary matrix Q of the QR factorization formed by p?geqrf.

Syntax

```
call pcunmqr( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info )
```

```
call pzunmqr( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info )
```

Description

The routine overwrites the general complex *m*-by-*n* distributed matrix sub (*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'T':	Q^H*sub(*C*)	sub(*C*)*Q^H

where *Q* is a complex unitary distributed matrix defined as the product of *k* elementary reflectors

Q = *H*(1) *H*(2)... *H*(*k*) as returned by p?geqrf. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* ='R'.

Input Parameters

side	(global) CHARACTER ='L': *Q* or Q^H is applied from the left. ='R': *Q* or Q^H is applied from the right.
trans	(global) CHARACTER ='N', no transpose, *Q* is applied.

	='C', conjugate transpose, Q^H is applied.
m	(global) INTEGER. The number of rows in the distributed matrix sub(*C*) (*m*≥0).
n	(global) INTEGER. The number of columns in the distributed matrix sub(*C*) (*n*≥0).
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints: If *side* = 'L', *m*≥*k*≥0 If *side* = 'R', *n*≥*k*≥0.
a	(local) COMPLEX for pcunmqr DOUBLE COMPLEX for pzunmqr. Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*k*-1)). The *j*-th column must contain the vector which defines the elementary reflector *H*(*j*), *ja*≤*j*≤*ja* +*k*-1, as returned by p?geqrf in the *k* columns of its distributed matrix argument *A*(*ia*:*, *ja*:*ja*+*k*-1). *A*(*ia*:*, *ja*:*ja*+*k*-1) is modified by the routine but restored on exit. If *side*= 'L', *lld_a*≥ max(1, *LOCr*(*ia*+*m*-1) If *side* = 'R', *lld_a*≥ max(1, *LOCr*(*ia*+*n*-1)
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
tau	(local) COMPLEX for pcunmqr DOUBLE COMPLEX for pzunmqr Array, DIMENSION *LOCc*(*ja*+*k*-1)). Contains the scalar factor *tau* (*j*) of elementary reflectors *H*(*j*) as returned by p?geqrf. *tau* is tied to the distributed matrix *A*.
c	(local)

COMPLEX for pcunmqr
DOUBLE COMPLEX for pzunmqr.
Pointer into the local memory to an array of local dimension (*lld_c*, *LOCc*(*jc*+*n*-1)).
Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic, *jc* — (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work — (local)
COMPLEX for pcunmqr
DOUBLE COMPLEX for pzunmqr.
Workspace array of dimension of *lwork*.

lwork — (local or global) INTEGER, dimension of *work*, must be at least:
If *side* = 'L',

lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + *mpc*0)**nb_a*) + *nb_a***nb_a*

else if *side* = 'R',

lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + max(*npa*0 + numroc(numroc(*n*+*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a***nb_a*

end if
where
lcmq = *lcm*/NPCOL with *lcm* = ilcm (NPROW, NPCOL),
iroffa = mod(*ia*-1, *mb_a*),
icoffa = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),
*npa*0 = numroc(*n*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),
iroffc = mod(*ic*-1, *mb_c*),
icoffc = *mod*(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),
iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),

*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),
*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c	Overwritten by the product *Q**sub(*C*), or Q^H*sub(*C*), or sub(*C*)*Q^H, or sub(*C*)**Q*.
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?gelqf

Computes the LQ factorization of a general rectangular matrix.

Syntax

```
call psgelqf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgelqf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgelqf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgelqf( m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine computes the *LQ* factorization of a real/complex distributed *m*-by-*n* matrix `sub(A) = A(ia:ia+m-1, ia:ia+n-1) = L*Q`.

Input Parameters

m	(global) `INTEGER`. The number of rows in the submatrix sub(*Q*) (*m* ≥ 0).
n	(global) `INTEGER`. The number of columns in the submatrix sub(*Q*) (*n* ≥ 0).
k	(global) `INTEGER`. The number of elementary reflectors whose product defines the matrix *Q* (*n* ≥ *k* ≥ 0).

a	(local)
	`REAL` for `psgelqf`
	`DOUBLE PRECISION` for `pdgelqf`
	`COMPLEX` for `pcgelqf`
	`DOUBLE COMPLEX` for `pzgelqf`
	Pointer into the local memory to an array of local dimension (*lld_a*, *LOCc*(*ja*+*n*-1)).
	Contains the local pieces of the distributed matrix sub(*A*) to be factored.

ia, *ja*	(global) `INTEGER`. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*((*ia*:*ia*+*m*-1, *ia*:*ia*+*n*-1), respectively.
desca	(global and local) `INTEGER` array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

work	(local)
	`REAL` for `psgelqf`
	`DOUBLE PRECISION` for `pdgelqf`
	`COMPLEX` for `pcgelqf`
	`DOUBLE COMPLEX` for `pzgelqf`
	Workspace array of dimension of *lwork*.

lwork	(local or global) `INTEGER`, dimension of *work*, must be at least *lwork* ≥ *mb_a**(*mp*0 + *nq*0 + *mb_a*), where *iroff* = mod(*ia*-1, *mb_a*),

icoff = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),
iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
*mp*0 = numroc(*m*+*iroff*, *mb_a*, MYROW, *iarow*, NPROW),
*nq*0 = numroc(*n*+*icoff*, *nb_a*, MYCOL, *iacol*, NPCOL)
indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a
The elements on and below the diagonal of sub(*A*) contain the *m* by min(*m*,*n*) lower trapezoidal matrix *L* (*L* is lower trapezoidal if $m \le n$); the elements above the diagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see Application Notes below)

tau
(local)
REAL for psgelqf
DOUBLE PRECISION for pdgelqf
COMPLEX for pcgelqf
DOUBLE COMPLEX for pzgelqf
Array, DIMENSION *LOCr*(*ia*+min(*m*, *n*)-1)).
Contains the scalar factors of elementary reflectors. *tau* is tied to the distributed matrix *A*.

work(1)
On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info
(global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*ia*+*k*-1) *H*(*ia*+*k*-2)... *H*(*ia*),

where *k* = min(*m*,*n*)

Each *H*(*i*) has the form

H(*i*) = *I* - *tau* * *v* * *v*'

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1:*i*-1) = 0 and *v*(*i*) = 1; *v*(*i*+1:*n*) is stored on exit in *A*(*ia*+*i*-1:*ia*+*i*-1,*ja*+*n*-1), and *tau* in *tau* (*ia*+*i*-1).

p?orglq

Generates the real orthogonal matrix Q of the LQ factorization formed by p?gelqf.

Syntax

```
call psorglq( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorglq( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine generates the whole or part of *m*-by-*n* real distributed matrix *Q* denoting *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) with orthonormal rows, which is defined as the first *m* rows of a product of *k* elementary reflectors of order *n*

Q = *H*(*k*)... *H*(2) *H*(1)

as returned by p?gelqf.

Input Parameters

m	(global) INTEGER. The number of rows in the submatrix sub(*Q*); (*m*≥0).
n	(global) INTEGER. The number of columns in the submatrix sub(*Q*) (*n*≥*m*≥0).

k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q* (*m*≥*k*≥0).
a	(local) REAL for psorglq DOUBLE PRECISION for pdorglq Pointer into the local memory to an array of local dimension (*lld_a*, *LOCc* (*ja*+*n*-1)). On entry, the *i*-th row must contain the vector which defines the elementary reflector *H*(*i*), *ia*≤*i*≤*ia*+*k*-1, as returned by p?gelqf in the *k* rows of its distributed matrix argument *A*(*ia*:*ia*+*k* -1, *ja*:*).
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*((*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1), respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
work	(local) REAL for psorglq DOUBLE PRECISION for pdorglq Workspace array of dimension of *lwork*.
lwork	(local or global) INTEGER, dimension of *work*, must be at least *lwork* ≥ *mb_a**(*mpa*0 + *nqa*0 + *mb_a*), where *iroffa* = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*), *iarow* = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW), *iacol* = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL), *mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW), *nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL) indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo. If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a	Contains the local pieces of the *m*-by-*n* distributed matrix *Q* to be factored.
tau	(local) REAL for psorglq DOUBLE PRECISION for pdorglq Array, DIMENSION *LOCr*(*ia*+k–1). Contains the scalar factors *tau* of elementary reflectors *H*(*i*). *tau* is tied to the distributed matrix *A*.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = – (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = –*i*.

p?unglq

Generates the unitary matrix Q of the LQ factorization formed by p?gelqf.

Syntax

```
call pcunglq( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzunglq( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine generates the whole or part of *m*-by-*n* complex distributed matrix *Q* denoting *A*(*ia*:*ia*+*m*–1, *ja*:*ja*+*n*–1) with orthonormal rows, which is defined as the first *m* rows of a product of *k* elementary reflectors of order *n*

Q = *H*(*k*)... *H*(2)' *H*(1)'

as returned by p?gelqf.

Input Parameters

m	(global) INTEGER. The number of rows in the submatrix sub(*Q*) (*m*≥0).
n	(global) INTEGER. The number of columns in the submatrix sub(*Q*) (*n*≥*m*≥0).
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q* (*m*≥*k*≥0).
a	(local) COMPLEX for pcunglq DOUBLE COMPLEX for pzunglq Pointer into the local memory to an array of local dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, the *i*-th row must contain the vector which defines the elementary reflector *H*(*i*), *ia*≤*i*≤*ia*+*k*-1, as returned by p?gelqf in the *k* rows of its distributed matrix argument *A*(*ia*:*ia*+*k*-1, *ja*:*).
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1), respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
tau	(local) COMPLEX for pcunglq DOUBLE COMPLEX for pzunglq Array, DIMENSION *LOCr*(*ia*+k-1)). Contains the scalar factors *tau* of elementary reflectors *H*(*i*). *tau* is tied to the distributed matrix *A*.
work	(local) COMPLEX for pcunglq DOUBLE COMPLEX for pzunglq Workspace array of dimension of *lwork*.
lwork	(local or global) INTEGER, dimension of *work*, must be at least *lwork* ≥ *mb_a**(*mpa*0 + *nqa*0 + *mb_a*), where *iroffa* = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),
iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),
*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL)
indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a	Contains the local pieces of the *m*-by-*n* distributed matrix *Q* to be factored.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?ormlq

Multiplies a general matrix by the orthogonal matrix Q of the LQ factorization formed by p?gelqf.

Syntax

call psormlq(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *work*, *lwork*, *info*)

call pdormlq(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *work*, *lwork*, *info*)

Description

The routine overwrites the general real m-by-n distributed matrix sub(C) = $C(ic:ic+m-1, jc:jc+n-1)$ with

	side ='L'	*side* ='R'
trans = 'N':	Q*sub(C)	sub(C)*Q
trans = 'T':	Q^T*sub(C)	sub(C)*Q^T

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$Q = H(k)...H(2) H(1)$

as returned by p?gelqf. Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

side (global) CHARACTER
='L': Q or Q^T is applied from the left.
='R': Q or Q^T is applied from the right.

trans (global) CHARACTER
='N', no transpose, Q is applied.
='T', transpose, Q^T is applied.

m (global) INTEGER. The number of rows in the distributed matrix sub(C) ($m \geq 0$).

n (global) INTEGER. The number of columns in the distributed matrix sub(C) ($n \geq 0$).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints:
If *side* = 'L', $m \geq k \geq 0$
If *side* = 'R', $n \geq k \geq 0$.

a (local)
REAL for psormlq
DOUBLE PRECISION for pdormlq.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*m*-1)), if *side* = 'L' and (*lld_a*, *LOCc*(*ja*+*n*-1)), if *side* = 'R'.The i-th row must contain

	the vector which defines the elementary reflector *H*(*i*), *ia*≤*i*≤*ia*+*k*–1, as returned by p?gelqf in the *k* rows of its distributed matrix argument *A*(*ia*:*ia*+*k*–1, *ja*:*). *A*(*ia*:*ia*+*k*–1, *ja*:*) is modified by the routine but restored on exit.
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
tau	(local) REAL for psormlq DOUBLE PRECISION for pdormlq Array, DIMENSION *LOCc*(*ja*+*k*–1)). Contains the scalar factor *tau* (*i*) of elementary reflectors *H*(*i*) as returned by p?gelqf. *tau* is tied to the distributed matrix *A*.
c	(local) REAL for psormlq DOUBLE PRECISION for pdormlq Pointer into the local memory to an array of local dimension (*lld_c*, *LOCc*(*jc*+*n*–1)). Contains the local pieces of the distributed matrix sub(*C*) to be factored.
ic, *jc*	(global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.
descc	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.
work	(local) REAL for psormlq DOUBLE PRECISION for pdormlq. Workspace array of dimension of *lwork*.
lwork	(local or global) INTEGER, dimension of the array *work*; must be at least: If *side* = 'L',

lwork ≥ max((*mb_a**(*mb_a*-1))/2, (*mpc*0+max *mqa*0)+ numroc(numroc(*m* + *iroffc*, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0))* *mb_a*) + *mb_a***mb_a*
else if *side* = 'R',
lwork ≥ max((*mb_a** (*mb_a*-1))/2, (*mpc*0+nqc0)**mb_a* + *mb_a***mb_a*
end if
where
lcmp = *lcm*/NPROW with *lcm* = ilcm (NPROW, NPCOL),
iroffa = mod(*ia*-1, *mb_a*),
icoffa = mod(*ja*-1, *nb_a*),
iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
*mqa*0 = *numroc*(*m*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),
iroffc = mod(*ic*-1, *mb_c*),
icoffc = *mod*(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),
iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),
*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),
*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c	Overwritten by the product *Q**sub(*C*), or *Q*' *sub (*C*), or sub(*C*)**Q*', or sub(*C*)**Q*
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER.

= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?unmlq

Multiplies a general matrix by the unitary matrix Q of the LQ factorization formed by p?gelqf.

Syntax

```
call pcunmlq( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info )
```

```
call pzunmlq( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info )
```

Description

The routine overwrites the general complex *m*-by-*n* distributed matrix sub (*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'T':	Q^H*sub(*C*)	sub(*C*)*Q^H

where *Q* is a complex unitary distributed matrix defined as the product of *k* elementary reflectors

Q = *H*(*k*)' ... *H*(2)' *H*(1)'

as returned by p?gelqf. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side (global) CHARACTER
='L': *Q* or Q^H is applied from the left.
='R': *Q* or Q^H is applied from the right.

trans (global) CHARACTER
='N', no transpose, *Q* is applied.
='C', conjugate transpose, Q^H is applied.

m	(global) INTEGER. The number of rows in the distributed matrix sub(*C*) (*m*≥0).
n	(global) INTEGER. The number of columns in the distributed matrix sub(*C*) (*n*≥0).
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints: If *side* = 'L', *m*≥*k*≥0 If *side* = 'R', *n*≥*k*≥0.
a	(local) COMPLEX for pcunmlq DOUBLE COMPLEX for pzunmlq. Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*m*-1)), if *side* = 'L', and (*lld_a*, *LOCc*(*ja*+*n*-1)), if *side* = 'R', where *lld_a* ≥ max(1, *LOCr* (*ia*+*k*-1)). The *i*-th column must contain the vector which defines the elementary reflector *H*(*i*), *ia*≤*i*≤*ia*+*k*-1, as returned by p?gelqf in the *k* rows of its distributed matrix argument *A*(*ia*:*ia*+*k*-1, *ja*:*). *A*(*ia*:*ia*+*k*-1, *ja*:*) is modified by the routine but restored on exit.
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
tau	(local) COMPLEX for pcunmlq DOUBLE COMPLEX for pzunmlq Array, DIMENSION *LOCc*(*ia*+*k*-1)). Contains the scalar factor *tau* (*i*) of elementary reflectors *H*(*i*) as returned by p?gelqf. *tau* is tied to the distributed matrix *A*.
c	(local) COMPLEX for pcunmlq DOUBLE COMPLEX for pzunmlq.

Pointer into the local memory to an array of local dimension (*lld_c*, *LOCc*(*jc*+*n*-1)).
Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic, *jc* (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)
COMPLEX for pcunmlq
DOUBLE COMPLEX for pzunmlq.
Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of the array *work*; must be at least:
If *side* = 'L',

lwork ≥ max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + max *mqa*0)+ numroc(numroc(*m* + *iroffc*, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0))**mb_a*) + *mb_a***mb_a*
else if *side* = 'R',

lwork ≥ max((*mb_a** (*mb_a*-1))/2, (*mpc*0 + nqc0)**mb_a* + *mb_a***mb_a*
end if
where
lcmp = *lcm*/NPROW with *lcm* = ilcm (NPROW, NPCOL),
iroffa = mod(*ia*-1, *mb_a*),
icoffa = mod(*ja*-1, *nb_a*),
iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
*mqa*0 = *numroc*(m + *icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),
iroffc = mod(*ic*-1, *mb_c*),
icoffc = *mod*(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),
iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),
*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c	Overwritten by the product *Q**sub(*C*), or *Q*'*sub (*C*), or sub(*C*)**Q*', or sub(*C*)**Q*
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?geqlf

Computes the QL factorization of a general matrix.

Syntax

```
call psgeqlf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgeqlf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgeqlf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgeqlf(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine forms the *QL* factorization of a real/complex distributed *m*-by-*n* matrix sub(A) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) = *Q***L*.

Input Parameters

m — (global) INTEGER. The number of rows in the submatrix sub(*Q*); (*m* ≥ 0).

n — (global) INTEGER. The number of columns in the submatrix sub(*Q*) (*n* ≥ 0).

a — (local)
REAL for psgeqlf
DOUBLE PRECISION for pdgeqlf
COMPLEX for pcgeqlf
DOUBLE COMPLEX for pzgeqlf
Pointer into the local memory to an array of local dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). Contains the local pieces of the distributed matrix sub(*A*) to be factored.

ia, *ja* — (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*((*ia*:*ia*+*m*-1, *ia*:*ia*+*n*-1), respectively.

desca — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

work — (local)
REAL for psgeqlf
DOUBLE PRECISION for pdgeqlf
COMPLEX for pcgeqlf
DOUBLE COMPLEX for pzgeqlf
Workspace array of dimension of *lwork*.

lwork — (local or global) INTEGER, dimension of *work*, must be at least *lwork* ≥ *nb_a**(*mp*0 + *nq*0 + *nb_a*), where
iroff = mod(*ia*-1, *mb_a*),
icoff = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),
iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
*mp*0 = numroc(*m*+*iroff*, *mb_a*, MYROW, *iarow*, NPROW),

*nq*0 = numroc(*n*+*icoff*, *nb_a*, MYCOL, *iacol*, NPCOL)
numroc and indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a
On exit, if $m \geq n$, the lower triangle of the distributed submatrix *A*(*ia*+*m*-*n*:*ia*+*m*-1, *ja*:*ja*+n-1) contains the n-by-n lower triangular matrix *L*; if $m \leq n$, the elements on and below the (n-m)-th superdiagonal contain the m-by-n lower trapezoidal matrix *L*; the remaining elements, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see Application Notes below)

tau
(local)
REAL for psgeqlf
DOUBLE PRECISION for pdgeqlf
COMPLEX for pcgeqlf
DOUBLE COMPLEX for pzgeqlf
Array, DIMENSION *LOCc*(*ja*+*n*-1)).
Contains the scalar factors of elementary reflectors. *tau* is tied to the distributed matrix *A*.

work(1)
On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info
(global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix Q is represented as a product of elementary reflectors

Q = H(ja+k-1)... H(ja+1) H(ja),

where k = min(m,n)

Each H(i) has the form

H(i) = I - tau * v * v'

where tau is a real/complex scalar, and v is a real/complex vector with v(m-k+i+1:m) = 0 and v(m-k+i) = 1; v(m-k+i-1) is stored on exit in A(ia+ia+m-k+i-2, ja+n-k+i-1), and tau in tau (ja+n-k+i-1).

p?orgql

Generates the orthogonal matrix Q of the QL factorization formed by p?geqlf.

Syntax

call psorgql(m, n, k, a, ia, ja, desca, tau, work, lwork, info)

call pdorgql(m, n, k, a, ia, ja, desca, tau, work, lwork, info)

Description

The routine generates the whole or part of m-by-n real distributed matrix Q denoting A(ia:ia+m-1, ja:ja+n-1) with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

Q = H(k)... H(2) H(1)

as returned by p?geqlf.

Input Parameters

m	(global) INTEGER. The number of rows in the submatrix sub(Q); (m≥0).
n	(global) INTEGER. The number of columns in the submatrix sub(Q) (m≥n≥0).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q* (*n*≥*k*≥0).

a (local)
REAL for psorgql
DOUBLE PRECISION for pdorgql
Pointer into the local memory to an array of local dimension (*lld_a*, *LOCc* (*ja*+*n*-1)). On entry, the *j*-th column must contain the vector which defines the elementary reflector *H*(*j*), *ja*+*n*-*k*≤*j*≤*ja*+*n*-1, as returned by p?geqlf in the *k* columns of its distributed matrix argument *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1).

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*((*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)
REAL for psorgql
DOUBLE PRECISION for pdorgql
Array, DIMENSION *LOCc*(*ja*+n-1)).
Contains the scalar factors *tau*(*j*) of elementary reflectors *H*(*j*). *tau* is tied to the distributed matrix *A*.

work (local)
REAL for psorgql
DOUBLE PRECISION for pdorgql
Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least *lwork* ≥ *nb_a** (*nqa*0 + *mpa*0 + *nb_a*), where
iroffa = mod(*ia*-1, *mb_a*),
icoffa = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),
iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),
*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL)

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — Contains the local pieces of the *m*-by-*n* distributed matrix *Q* to be factored.

work(1) — On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?ungql

Generates the unitary matrix Q of the QL factorization formed by p?geqlf.

Syntax

```
call pcungql( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungql( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine generates the whole or part of *m*-by-*n* complex distributed matrix *Q* denoting *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) with orthonormal rows, which is defined as the first *n* columns of a product of *k* elementary reflectors of order *m*

Q = *H*(*k*)... *H*(2)' *H*(1)'

as returned by p?geqlf.

Input Parameters

m	(global) INTEGER. The number of rows in the submatrix sub(*Q*) (*m*≥0).
n	(global) INTEGER. The number of columns in the submatrix sub(*Q*) (*m*≥*n*≥0).
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q* (*n*≥*k*≥0).
a	(local) COMPLEX for pcungql DOUBLE COMPLEX for pzungql Pointer into the local memory to an array of local dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, the *j*-th column must contain the vector which defines the elementary reflector *H*(*j*), *ja*+*n*-*k*≤ *j*≤ *ja*+*n*-1, as returned by p?geqlf in the *k* columns of its distributed matrix argument *A*(*ia*:*, *ja*+*n*-*k*: *ja*+*n*-1).
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1), respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
tau	(local) COMPLEX for pcungql DOUBLE COMPLEX for pzungql Array, DIMENSION *LOCr*(*ia*+n-1)). Contains the scalar factors *tau* (*j*) of elementary reflectors *H*(*j*). *tau* is tied to the distributed matrix *A*.
work	(local) COMPLEX for pcungql DOUBLE COMPLEX for pzungql Workspace array of dimension of *lwork*.
lwork	(local or global) INTEGER, dimension of *work*, must be at least *lwork* ≥ *nb_a**(*nqa*0 + *mpa*0 + *nb_a*), where

iroffa = mod(*ia*-1, *mb_a*),
icoffa = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),
iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),
*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL)
indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — Contains the local pieces of the *m*-by-*n* distributed matrix *Q* to be factored.

work(1) — On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?ormql

Multiplies a general matrix by the orthogonal matrix Q of the QL factorization formed by p?geqlf.

Syntax

call psormql(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pdormql(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Description

The routine overwrites the general real m-by-n distributed matrix sub(C) = C(ic:ic+m-1,jc:jc+n-1) with

	side ='L'	*side* ='R'
trans = 'N':	Q*sub(C)	sub(C)*Q
trans = 'T':	Q^T*sub(C)	sub(C)*Q^T

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$Q = H(k)' \ldots H(2)' H(1)'$

as returned by p?geqlf. Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

side (global) CHARACTER
='L': Q or Q^T is applied from the left.
='R': Q or Q^T is applied from the right.

trans (global) CHARACTER
='N', no transpose, Q is applied.
='T', transpose, Q^T is applied.

m (global) INTEGER. The number of rows in the distributed matrix sub(C) (m≥0).

n (global) INTEGER. The number of columns in the distributed matrix sub(C) (n≥0).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:

If *side* = 'L', *m*≥*k*≥0

If *side* = 'R', *n*≥*k*≥0.

a (local)
REAL for psormql
DOUBLE PRECISION for pdormql.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*k*-1)). The *j*-th column must contain the vector which defines the elementary reflector *H*(*j*),

ja≤*j*≤*ja*+*k*-1, as returned by p?gelqf in the *k* columns of its distributed matrix argument *A*(*ia*:*, *ja*:*ja*+*k*-1).*A*(*ia*:*, *ja*:*ja*+*k*-1) is modified by the routine but restored on exit.

If *side* = 'L',*lld_a* ≥ max(1, *LOCr*(*ia*+*m*-1)),

If *side* = 'R', *lld_a* ≥ max(1, *LOCr*(*ia*+*n*-1)).

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)
REAL for psormql
DOUBLE PRECISION for pdormql.
Array, DIMENSION *LOCc*(*ja*+*n*-1)).
Contains the scalar factor *tau* (*j*) of elementary reflectors *H*(*j*) as returned by p?geqlf. *tau* is tied to the distributed matrix *A*.

c (local)
REAL for psormql
DOUBLE PRECISION for pdormql.
Pointer into the local memory to an array of local dimension (*lld_c*, *LOCc*(*jc*+*n*-1)).
Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic, *jc*
(global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc
(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work
(local)
REAL for psormql
DOUBLE PRECISION for pdormql.
Workspace array of dimension of *lwork*.

lwork
(local or global) INTEGER, dimension of *work*, must be at least:
If *side* = 'L',

lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0+mpc0)**nb_a* + *nb_a***nb_a*

else if *side* ='R',

lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0+max *npa*0)+ numroc(numroc(*n*+*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a***nb_a*

end if
where
lcmp = *lcm*/NPCOL with *lcm* = ilcm (NPROW, NPCOL),
iroffa = mod(*ia*-1, *mb_a*),
icoffa = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),
*npa*0= *numroc*(*n* + *iroffa*, *mb_a*, MYROW, *iarow*, NPROW),
iroffc = mod(*ic*-1, *mb_c*),
icoffc = *mod*(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),
iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),
*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),
*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c	Overwritten by the product *Q** sub(*C*), or *Q*'*sub (*C*), or sub(*C*)* *Q*', or sub(*C*)* *Q*
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?unmql

Multiplies a general matrix by the unitary matrix Q of the QL factorization formed by p?geqlf.

Syntax

```
call pcunmql( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)
```

```
call pzunmql( side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)
```

Description

The routine overwrites the general complex *m*-by-*n* distributed matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'C':	Q^H*sub(*C*)	sub(*C*)*Q^H

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

$Q = H(k)' \ldots H(2)' H(1)'$

as returned by p?geqlf. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

side	(global) CHARACTER ='L': Q or Q^H is applied from the left. ='R': Q or Q^H is applied from the right.
trans	(global) CHARACTER ='N', no transpose, Q is applied. ='C', conjugate transpose, Q^H is applied.
m	(global) INTEGER. The number of rows in the distributed matrix sub(C) ($m \geq 0$).
n	(global) INTEGER. The number of columns in the distributed matrix sub(C) ($n \geq 0$).
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints: If $side = 'L'$, $m \geq k \geq 0$ If $side = 'R'$, $n \geq k \geq 0$.
a	(local) COMPLEX for pcunmql DOUBLE COMPLEX for pzunmql. Pointer into the local memory to an array of dimension (lld_a, $LOCc(ja+k-1)$). The j-th column must contain the vector which defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by p?geqlf in the k columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$. $A(ia:*, ja:ja+k-1)$ is modified by the routine but restored on exit. If $side = 'L'$, $lld\_a \geq \max(1, LOCr(ia+m-1))$, If $side = 'R'$, $lld\_a \geq \max(1, LOCr(ia+n-1))$.

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)
COMPLEX for pcunmql
DOUBLE COMPLEX for pzunmql
Array, DIMENSION *LOCc*(*ia*+*n*-1)).
Contains the scalar factor *tau* (*j*) of elementary reflectors *H*(*j*) as returned by p?geqlf. *tau* is tied to the distributed matrix *A*.

c (local)
COMPLEX for pcunmql
DOUBLE COMPLEX for pzunmql.
Pointer into the local memory to an array of local dimension (*lld_c*, *LOCc*(*jc*+*n*-1)).
Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic, *jc* (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)
COMPLEX for pcunmql
DOUBLE COMPLEX for pzunmql.
Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:
If *side* = 'L',

lwork ≥ max((*nb_a** (*nb_a*-1))/2, (*nqc*0+mpc0)**nb_a* + *nb_a***nb_a*
else if *side* ='R',

lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0+max *npa*0)+
numroc(numroc(*n*+*icoffc*, *nb_a*, 0, 0, NPCOL),
nb_a, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a***nb_a*
end if
where
lcmp = *lcm*/NPCOL with *lcm* = ilcm (NPROW, NPCOL),
iroffa = mod(*ia*-1, *mb_a*),
icoffa = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),
*npa*0 = *numroc* (*n* + *iroffa*, *mb_a*, MYROW, *iarow*,
NPROW),
iroffc = mod(*ic*-1, *mb_c*),
icoffc = *mod*(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),
iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),
*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*,
NPROW),
*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*,
NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — Overwritten by the product *Q** sub(*C*), or *Q*' sub (*C*), or sub(*C*)* *Q*', or sub(*C*)* *Q*

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?gerqf

Computes the RQ factorization of a general rectangular matrix.

Syntax

```
call psgerqf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgerqf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgerqf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgerqf( m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine forms the *QR* factorization of a general *m*-by-*n* distributed matrix sub(*A*) = *A*(*ia:ia*+*m*-1,*ja:ja*+*n*-1) as

A= *R***Q*

Input Parameters

m (global) INTEGER. The number of rows in the distributed submatrix sub(*A*); (*m*≥0).

n (global) INTEGER. The number of columns in the distributed submatrix sub(*A*); (*n*≥0).

a (local)
REAL for psgeqrf
DOUBLE PRECISION for pdgeqrf
COMPLEX for pcgeqrf
DOUBLE COMPLEX for pzgeqrf.
Pointer into the local memory to an array of local dimension (*lld_a*, *LOCc*(*ja*+*n*-1)).

Contains the local pieces of the distributed matrix sub(*A*) to be factored.

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*(*ia:ia*+*m-1*,*ja:ja*+*n-1*), respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*

work (local).
REAL for psgeqrf
DOUBLE PRECISION for pdgeqrf.
COMPLEX for pcgeqrf.
DOUBLE COMPLEX for pzgeqrf
Workspace array of dimension *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least *lwork* ≥ *mb_a**(*mp*0+*nq*0+*mb_a*), where
iroff = mod(*ia*-1, *mb_a*),
icoff = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),
iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
*mp*0 = numroc(*m*+*iroff*, *mb_a*, MYROW, *iarow*, NPROW),
*nq*0 = numroc(*n*+*icoff*, *nb_a*, MYCOL, *iacol*, NPCOL)
and numroc, indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit, if *m*≤*n*, the upper triangle of *A*(*ia:ia*+*m-1*, *ja:ja*+*n-1*) contains the *m*-by-*m* upper triangular matrix *R*; if *m*≥*n*, the elements on and above the (*m* - *n*)-th subdiagonal contain the *m*-by-*n* upper trapezoidal matrix *R*; the remaining

	elements, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see Application Notes below)
tau	(local) REAL for psgeqrf DOUBLE PRECISION for pdgeqrf COMPLEX for pcgeqrf DOUBLE COMPLEX for pzgeqrf. Array, DIMENSION *LOCr*(*ia*+*m*-1). Contains the scalar factor *tau* of elementary reflectors. *tau* is tied to the distributed matrix *A*.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0, the execution is successful. < 0, if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*ia*) *H*(*ia*+*1*)... *H*(*ia*+*k*-1),

where *k* = min(*m*,*n*).

Each *H*(*i*) has the form

H(*i*) = *I* - *tau* * *v* * *v*'

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(*n*-*k*+*i*+1:*n*) = 0 and *v*(*n*-*k*+*i*) = 1; *v*(1:*n*-*k*+*i*-1)/conjg (*v*(1:*n*-*k*+*i*-1)) is stored on exit in *A*(*ia*+*m*-*k*+*i*-1,*ja*:*ja*+*n*-*k*+*i*-2), and *tau* in *tau*(*ia*+*m*-*k*+*i*-1).

p?orgrq

Generates the orthogonal matrix Q of the RQ factorization formed by p?gerqf.

Syntax

```
call psorgrq( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorgrq( m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine generates the whole or part of *m*-by-*n* real distributed matrix *Q* denoting *A(ia:ia+m-1, ja:ja+n-1)* with orthonormal columns, which is defined as the last *m* rows of a product of *k* elementary reflectors of order *m*

Q= H(1) H(2)...H(k)

as returned by p?gerqf.

Input Parameters

m	(global) INTEGER. The number of rows in the submatrix sub(*Q*); *(m≥0)*.
n	(global) INTEGER. The number of columns in the submatrix sub(*Q*) *(n≥m≥0)*.
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q (m≥k≥0)*.
a	(local) REAL for psorgrq DOUBLE PRECISION for pdorgrq Pointer into the local memory to an array of local dimension *(lld_a, LOCc(ja+n-1))*. The *i*-th column must contain the vector which defines the elementary reflector *H(i)*, *ja≤j≤ja+k-1*, as returned by p?geqrf in the *k* columns of its distributed matrix argument *A(ia:*, ja:ja+k-1)*.
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A(ia:ia+m-1,ja:ja+n-1)*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)
REAL for psorgrq
DOUBLE PRECISION for pdorgrq
Array, DIMENSION *LOCc*(*ja*+*k*-1)).
Contains the scalar factor *tau* (*i*) of elementary reflectors *H*(*i*) as returned by p?gerqf. *tau* is tied to the distributed matrix *A*.

work (local)
REAL for psorgrq
DOUBLE PRECISION for pdorgrq
Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least *lwork*≥*mb_a**(*mpa*0 + *nqa*0 + *mb_a*), where
iroffa = mod(*ia*-1, *mb_a*),
icoffa = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),
iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),
*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL)
indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a Contains the local pieces of the *m*-by-*n* distributed matrix Q.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?ungrq

Generates the unitary matrix Q of the RQ factorization formed by p?gerqf.

Syntax

call pcungrq(*m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *work*, *lwork*, *info*)

call pzungrq(*m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *work*, *lwork*, *info*)

Description

The routine generates the *m*-by-*n* complex distributed matrix *Q* denoting *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1) with orthonormal rows, which is defined as the last *m* rows of a product of *k* elementary reflectors of order n

Q = *H*(1)' *H*(2)'... *H*(*k*)'

as returned by p?gerqf.

Input Parameters

m	(global) INTEGER. The number of rows in the submatrix sub(*Q*); (*m*≥0).
n	(global) INTEGER. The number of columns in the submatrix sub(*Q*) (*n*≥*m*≥0).
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q* (*m*≥*k*≥0).
a	(local) COMPLEX for pcungrq DOUBLE COMPLEX for pzungrqc

Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). The *i*-th row must contain the vector which defines the elementary reflector *H*(*i*), *ia*+*m*-*k*≤*i*≤*ia*+*m*-1, as returned by p?gerqf in the *k* rows of its distributed matrix argument *A*(*ia*+*m*-*k*:*ia*+*m*-1, *ja*:*).

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)
COMPLEX for pcungrq
DOUBLE COMPLEX for pzungrq
Array, DIMENSION *LOCr*(*ia*+*m*-1)).
Contains the scalar factor *tau* (*i*) of elementary reflectors *H*(*i*) as returned by p?gerqf. *tau* is tied to the distributed matrix *A*.

work (local)
COMPLEX for pcungrq
DOUBLE COMPLEX for pzungrq
Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least *lwork* ≥ *mb_a*(*mpa*0 +*nqa*0+*mb_a*), where
iroffa = mod(*ia*-1, *mb_a*),
icoffa = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),
iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),
*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL)
indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a	Contains the local pieces of the *m*-by-*n* distributed matrix *Q*.
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?ormrq

Multiplies a general matrix by the orthogonal matrix Q of the RQ factorization formed by p?gerqf.

Syntax

call psormrq(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pdormrq(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Description

The routine overwrites the general real *m*-by-*n* distributed matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side = 'L'	*side* = 'R'
trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'T':	Q^T*sub(*C*)	sub(*C*)*Q^T

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$Q = H(1)\ H(2) \ldots\ H(k)$

as returned by p?gerqf. Q is of order m if $side$ = 'L' and of order n if $side$ = 'R'.

Input Parameters

side	(global) CHARACTER ='L': Q or Q^T is applied from the left. ='R': Q or Q^T is applied from the right.
trans	(global) CHARACTER ='N', no transpose, Q is applied. ='T', transpose, Q^T is applied.
m	(global) INTEGER. The number of rows in the distributed matrix sub(C) ($m \geq 0$).
n	(global) INTEGER. The number of columns in the distributed matrix sub(C) ($n \geq 0$).
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints: If $side$ = 'L', $m \geq k \geq 0$ If $side$ = 'R', $n \geq k \geq 0$.
a	(local) REAL for psormqr DOUBLE PRECISION for pdormqr. Pointer into the local memory to an array of dimension (lld_a, $LOCc(ja+m-1)$) if $side$ = 'L', and (lld_a, $LOCc(ja+n-1)$) if $side$ = 'R'. The i-th row must contain the vector which defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gerqf in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$. $A(ia:ia+k-1, ja:*)$ is modified by the routine but restored on exit.
ia, *ja*	(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)
REAL for psormqr
DOUBLE PRECISION for pdormqr
Array, DIMENSION *LOCc*(*ja*+*k*-1)).
Contains the scalar factor *tau* (*i*) of elementary reflectors *H*(*i*) as returned by p?gerqf. *tau* is tied to the distributed matrix *A*.

c (local)
REAL for psormrq
DOUBLE PRECISION for pdormrq
Pointer into the local memory to an array of local dimension (*lld_c*, *LOCc*(*jc*+*n*-1)).
Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic, *jc* (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)
REAL for psormrq
DOUBLE PRECISION for pdormrq.
Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:
If *side* = 'L',

lwork ≥ max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + max(*mqa*0 + numroc(numroc(*n*+*iroffc*, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0))**mb_a*) + *mb_a***mb_a*

else if *side* ='R',

lwork ≥ max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + *nqc*0)**mb_a*) + *mb_a***mb_a*

end if
where

lcmp = *lcm*/NPROW with *lcm* = ilcm (NPROW, NPCOL),
iroffa = mod(*ia*-1, *mb_a*),
icoffa = mod(*ja*-1, *nb_a*),
iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
*mqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),
iroffc = mod(*ic*-1, *mb_c*),
icoffc = *mod*(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),
iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),
*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),
*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c
Overwritten by the product *Q** sub(*C*), or *Q*'*sub (*C*), or sub(*C*)* *Q*', or sub(*C*)* *Q*

work(1)
On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info
(global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?unmrq

Multiplies a general matrix by the unitary matrix Q of the RQ factorization formed by p?gerqf.

Syntax

call pcunmrq(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pzunmrq(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Description

The routine overwrites the general complex *m*-by-*n* distributed matrix sub (*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'C':	Q^H*sub(*C*)	sub(*C*)*Q^H

where *Q* is a complex unitary distributed matrix defined as the product of *k* elementary reflectors

Q = *H*(1)' *H*(2)'... *H*(*k*)'

as returned by p?gerqf. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side	(global) CHARACTER ='L': *Q* or Q^H is applied from the left. ='R': *Q* or Q^H is applied from the right.
trans	(global) CHARACTER ='N', no transpose, *Q* is applied. ='C', conjugate transpose, Q^H is applied.
m	(global) INTEGER. The number of rows in the distributed matrix sub(*C*) (*m*≥0).
n	(global) INTEGER. The number of columns in the distributed matrix sub(*C*) (*n*≥0).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:

If *side* = 'L', *m*≥*k*≥0

If *side* = 'R', *n*≥*k*≥0.

a (local)
COMPLEX for pcunmrq
DOUBLE COMPLEX for pzunmrq.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*m*-1)) if *side* = 'L', and (*lld_a*, *LOCc*(*ja*+*n*-1)) if *side* = 'R'. The *i*-th row must contain the vector which defines the elementary reflector *H*(*i*),

ia≤*i*≤*ia*+*k*-1, as returned by p?gerqf in the *k* rows of its distributed matrix argument *A*(*ia*:*ia* +*k*-1, *ja**).*A*(*ia*:*ia* +*k*-1, *ja**) is modified by the routine but restored on exit.

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)
COMPLEX for pcunmrq
DOUBLE COMPLEX for pzunmrq
Array, DIMENSION *LOCc*(*ja*+*k*-1)).
Contains the scalar factor *tau* (*i*) of elementary reflectors *H*(*i*) as returned by p?gerqf. *tau* is tied to the distributed matrix *A*.

c (local)
COMPLEX for pcunmrq
DOUBLE COMPLEX for pzunmrq.
Pointer into the local memory to an array of local dimension (*lld_c*, *LOCc*(*jc*+*n*-1)).
Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic, *jc* (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)
COMPLEX for pcunmrq
DOUBLE COMPLEX for pzunmrq.
Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:
If *side* = 'L',

lwork ≥ max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + max(*mqa*0+numroc(numroc(*n*+*iroffc*, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0))**mb_a*) + *mb_a***mb_a*

else if *side* = 'R',

lwork ≥ max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + *nqc*0)**mb_a*) + *mb_a***mb_a*

end if
where
lcmp = *lcm*/NPROW with *lcm* = ilcm(NPROW, NPCOL),
iroffa = mod(*ia*-1, *mb_a*),
icoffa = mod(*ja*-1, *nb_a*),
iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
*mqa*0 = numroc(*m*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),
iroffc = mod(*ic*-1, *mb_c*),
icoffc = *mod*(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),
iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),
*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),
*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c	Overwritten by the product *Q** sub(*C*) or *Q*'*sub (*C*), or sub(*C*)* *Q*', or sub(*C*)* *Q*
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?tzrzf

Reduces the upper trapezoidal matrix A to upper triangular form.

Syntax

```
call pstzrzf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdtzrzf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pctzrzf( m, n, a, ia, ja, desca, tau, work, lwork, info)
call pztzrzf( m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

This routine reduces the m-by-n ($m \le n$) real/complex upper trapezoidal matrix sub(A)=(ia:ia+m-1,ja:ja+n-1) to upper triangular form by means of orthogonal/unitary transformations. The upper trapezoidal matrix A is factored as

A = (R 0)*Z,

where Z is an n-by-n orthogonal/unitary matrix and R is an m-by-m upper triangular matrix.

Input Parameters

m	(global) INTEGER. The number of rows in the submatrix sub(A); (m≥0).
n	(global) INTEGER. The number of columns in the submatrix sub(A) (n≥0).
a	(local) REAL for pstzrzf DOUBLE PRECISION for pdtzrzf. COMPLEX for pctzrzf. DOUBLE COMPLEX for pztzrzf. Pointer into the local memory to an array of dimension (lld_a, LOCc(ja+n-1)). Contains the local pieces of the m-by-n distributed matrix sub (A) to be factored.
ia, *ja*	(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A, respectively.
desca	(global and local) INTEGER array, dimension (dlen_). The array descriptor for the distributed matrix A.
work	(local) REAL for pstzrzf DOUBLE PRECISION for pdtzrzf. COMPLEX for pctzrzf. DOUBLE COMPLEX for pztzrzf. Workspace array of dimension of lwork.
lwork	(local or global) INTEGER, dimension of work, must be at least lwork ≥ mb_a*(mp0+nq0+mb_a), where iroff = mod(ia-1, mb_a),

icoff = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),
iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
*mp*0 = numroc (*m*+*iroff*, *mb_a*, MYROW, *iarow*, NPROW),
*nq*0 = numroc (*n*+*icoff*, *nb_a*, MYCOL, *iacol*, NPCOL)
indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a
On exit, the leading *m*-by-m upper triangular part of sub(*A*) contains the upper triangular matrix *R*, and elements *m*+1 to *n* of the first *m* rows of sub (*A*), with the array *tau*, represent the orthogonal/unitary matrix *Z* as a product of *m* elementary reflectors.

work(1)
On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

tau
(local)
REAL for pstzrzf
DOUBLE PRECISION for pdtzrzf.
COMPLEX for pctzrzf.
DOUBLE COMPLEX for pztzrzf.
Array, DIMENSION *LOCr*(*ia*+*m*-1)).
Contains the scalar factor of elementary reflectors. *tau* is tied to the distributed matrix *A*.

info
(global) INTEGER.
= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The factorization is obtained by the Householder's method. The *k*-th transformation matrix, *Z*(*k*), which is or whose conjugate transpose is used to introduce zeros into the (*m* - *k* +1)-th row of sub(*A*), is given in the form

$$Z(k) = \begin{bmatrix} i & 0 \\ 0 & \mathrm{T}(k) \end{bmatrix}$$

where

T(*k*) = *i* - *tau***u*(*k*)**u*(*k*)',

$$u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

tau is a scalar and *Z*(*k*) is an (*n* - *m*) element vector. *tau* and *Z*(*k*) are chosen to annihilate the elements of the *k*-th row of sub(*A*). The scalar *tau* is returned in the *k*-th element of *tau* and the vector *u*(*k*) in the *k*-th row of sub(*A*), such that the elements of *Z*(*k*) are in *a*(*k*, *m* + 1),..., *a*(*k*, *n*). The elements of *R* are returned in the upper triangular part of sub(*A*). *Z* is given by

Z = *Z*(1) * *Z*(2) *... * *Z*(*m*).

p?ormrz

Multiplies a general matrix by the orthogonal matrix from a reduction to upper triangular form formed by p?tzrzf*.*

Syntax

```
call psormrz( side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)
```

```
call pdormrz( side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)
```

Description

The routine overwrites the general real m-by-n distributed matrix sub(C) = C(ic:ic+m-1, jc:jc+n-1) with

	side ='L'	*side* ='R'
trans = 'N':	Q*sub(C)	sub(C)*Q
trans = 'T':	Q^T*sub(C)	sub(C)*Q^T

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$Q = H(1)\ H(2) \ldots\ H(k)$

as returned by p?tzrzf. Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

side	(global) CHARACTER ='L': Q or Q^T is applied from the left. ='R': Q or Q^T is applied from the right.
trans	(global) CHARACTER ='N', no transpose, Q is applied. ='T', transpose, Q^T is applied.
m	(global) INTEGER. The number of rows in the distributed matrix sub(C) (m≥0).
n	(global) INTEGER. The number of columns in the distributed matrix sub(C) (n≥0).

k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints: If *side* = 'L', *m* ≥ *k* ≥0 If *side* = 'R', *n* ≥ *k* ≥0.
l	(global) The columns of the distributed submatrix sub(*A*) containing the meaningful part of the Householder reflectors. If *side* = 'L', *m* ≥ *l* ≥0 If *side* = 'R', *n* ≥ *l* ≥0.
a	(local) REAL for psormrz DOUBLE PRECISION for pdormrz. Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*m*–1)) if *side* = 'L', and (*lld_a*, *LOCc*(*ja*+*n*–1)) if *side* = 'R', where *lld_a* ≥ max(1,*LOCr*(*ia*+*k*–1). The *i*-th row must contain the vector which defines the elementary reflector *H*(*i*), *ia*≤*i*≤*ia*+*k*–1, as returned by p?tzrzf in the *k* rows of its distributed matrix argument *A*(*ia*:*ia*+*k*–1, *ja*:*).A(*ia*:*ia*+*k*–1, *ja*:*) is modified by the routine but restored on exit.
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
tau	(local) REAL for psormrz DOUBLE PRECISION for pdormrz Array, DIMENSION *LOCc*(*ia*+*k*–1)). Contains the scalar factor *tau* (*i*) of elementary reflectors *H*(*i*) as returned by p?tzrzf. *tau* is tied to the distributed matrix *A*.
c	(local) REAL for psormrz

DOUBLE PRECISION for pdormrz
Pointer into the local memory to an array of local dimension (*lld_c*, *LOCc*(*jc*+*n*-1)).
Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic, *jc* (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)
REAL for psormrz
DOUBLE PRECISION for pdormrz.
Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:
If *side* = 'L',

lwork ≥ max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + max(*mqa*0 + numroc(numroc(*n*+*iroffc*, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0))**mb_a*) + *mb_a***mb_a*

else if *side* ='R',

lwork ≥ max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + *nqc*0)**mb_a*) + *mb_a***mb_a*

end if
where
lcmp = *lcm*/NPROW with *lcm* = ilcm (NPROW, NPCOL),
iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),
iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
*mqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),
iroffc = mod(*ic*-1, *mb_c*),
icoffc = *mod*(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),
iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),
*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c	Overwritten by the product *Q**sub(*C*), or *Q*'*sub (*C*), or sub(*C*)**Q*', or sub(*C*)**Q*
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?unmrz

Multiplies a general matrix by the unitary transformation matrix from a reduction to upper triangular form determined by p?tzrzf.

Syntax

call pcunmrz(*side*, *trans*, *m*, *n*, *k*, *l*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pzunmrz(*side*, *trans*, *m*, *n*, *k*, *l*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Description

The routine overwrites the general complex *m*-by-*n* distributed matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'C':	Q^H*sub(*C*)	sub(*C*)*Q^H

where *Q* is a complex unitary distributed matrix defined as the product of *k* elementary reflectors

Q = *H*(1)' *H*(2)'... *H*(*k*)'

as returned by pctzrzf/pztzrzf. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side	(global) CHARACTER ='L': *Q* or Q^H is applied from the left. ='R': *Q* or Q^H is applied from the right.
trans	(global) CHARACTER ='N', no transpose, *Q* is applied. ='C', conjugate transpose, Q^H is applied.
m	(global) INTEGER. The number of rows in the distributed matrix sub(*C*), (*m*≥0).
n	(global) INTEGER. The number of columns in the distributed matrix sub(*C*), (*n*≥0).
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints: If *side* = 'L', *m*≥*k*≥0 If *side* = 'R', *n*≥*k*≥0.
l	(global) INTEGER. The columns of the distributed submatrix sub(*A*) containing the meaningful part of the Householder reflectors. If *side* = 'L', *m*≥*l*≥0 If *side* = 'R', *n*≥*l*≥0.
a	(local)

	COMPLEX for pcunmrz DOUBLE COMPLEX for pzunmrz. Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*m*-1)) if *side* = 'L', and (*lld_a*, *LOCc*(*ja*+*n*-1)) if *side* = 'R', where *lld_a*≥max(1, *LOCr*(*ja*+*k*-1). The *i*-th row must contain the vector which defines the elementary reflector *H*(*i*), *ia*≤*i*≤*ia*+*k*-1, as returned by p?gerqf in the *k* rows of its distributed matrix argument *A*(*ia*:*ia* +*k*-1, *ja**). *A*(*ia*:*ia* +*k*-1, *ja**) is modified by the routine but restored on exit.
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
tau	(local) COMPLEX for pcunmrz DOUBLE COMPLEX for pzunmrz Array, DIMENSION *LOCc*(*ia*+*k*-1)). Contains the scalar factor *tau* (*i*) of elementary reflectors *H*(*i*) as returned by p?gerqf. *tau* is tied to the distributed matrix *A*.
c	(local) COMPLEX for pcunmrz DOUBLE COMPLEX for pzunmrz. Pointer into the local memory to an array of local dimension (*lld_c*, *LOCc*(*jc*+*n*-1)). Contains the local pieces of the distributed matrix sub(*C*) to be factored.
ic, *jc*	(global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.
descc	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.
work	(local) COMPLEX for pcunmrz

DOUBLE COMPLEX for pzunmrz.
Workspace array of dimension *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:
If *side* = 'L',

lwork ≥ max((*mb_a**(*mb_a*-1))/2, (*mpc*0+max(*mqa*0+numroc(numroc(*n*+*iroffc*, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0))**mb_a*) + *mb_a***mb_a*

else if *side* ='R',

lwork ≥ max((*mb_a**(*mb_a*-1))/2, (*mpc*0+*nqc*0)**mb_a*) + *mb_a***mb_a*

end if
where
lcmp = *lcm*/NPROW with *lcm* = ilcm(NPROW, NPCOL),
iroffa = mod(*ia*-1, *mb_a*),
icoffa = mod(*ja*-1, *nb_a*),
iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
*mqa*0 = numroc(*m*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),
iroffc = mod(*ic*-1, *mb_c*),
icoffc = *mod*(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),
iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),
*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),
*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = –1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c	Overwritten by the product *Q** sub(*C*), or *Q*'*sub (*C*), or sub(*C*)**Q*', or sub(*C*)**Q*
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = – (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?ggqrf

Computes the generalized QR factorization.

Syntax

call psggqrf(*n*, *m*, *p*, *a*, *ia*, *ja*, *desca*, *taua*, *b*, *ib*, *jb*, *descb*, *taub*, *work*, *lwork*, *info*)

call pdggqrf(*n*, *m*, *p*, *a*, *ia*, *ja*, *desca*, *taua*, *b*, *ib*, *jb*, *descb*, *taub*, *work*, *lwork*, *info*)

call pcggqrf(*n*, *m*, *p*, *a*, *ia*, *ja*, *desca*, *taua*, *b*, *ib*, *jb*, *descb*, *taub*, *work*, *lwork*, *info*)

call pzggqrf(*n*, *m*, *p*, *a*, *ia*, *ja*, *desca*, *taua*, *b*, *ib*, *jb*, *descb*, *taub*, *work*, *lwork*, *info*)

Description

The routine forms the generalized *QR* factorization of an *n*-by-*m* matrix

sub(*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*m*-1)

and an *n*-by-*p* matrix

sub(*B*) = *B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*p*-1):

as

sub(A) = *Q***R*, sub(*B*) = *Q***T***Z*,

where Q is an n-by-n orthogonal/unitary matrix, Z is a p-by-p orthogonal/unitary matrix, and R and T assume one of the forms:

If $n \geq m$

$$R = \begin{matrix} \begin{pmatrix} R_{11} \\ 0 \end{pmatrix} & \begin{matrix} m \\ n - m \end{matrix} \\ m & \end{matrix}$$

or if $n < m$

$$R = \begin{matrix} \begin{pmatrix} R_{11} & R_{12} \end{pmatrix} & n \\ n \quad m - n & \end{matrix}$$

where R_{11} is upper triangular, and

$$T = \begin{matrix} \begin{pmatrix} 0 & T_{12} \end{pmatrix} & n, \text{ if } n \leq p, \\ p - n \quad n & \end{matrix}$$

$$\text{or } T = \begin{matrix} \begin{pmatrix} T_{11} \\ T_{21} \end{pmatrix} & \begin{pmatrix} n - p \\ p \end{pmatrix}, \text{ if } n > p, \\ p & \end{matrix}$$

where T_{12} or T_{21} is an upper triangular matrix.

In particular, if sub(B) is square and nonsingular, the GQR factorization of sub(A) and sub(B) implicitly gives the QR factorization of inv (sub(B))* sub (A):

`inv(sub(B))*sub(A) =` Z^H`*(inv(T)*R`

Input Parameters

n	(global) INTEGER. The number of rows in the distributed matrices sub (*A*) and sub(*B*) (*n*≥0).
m	(global) INTEGER. The number of columns in the distributed matrix sub(*A*) (*m*≥0).
p	INTEGER. The number of columns in the distributed matrix sub(*B*) (*p*≥0).
a	(local) REAL for psggqrf DOUBLE PRECISION for pdggqrf COMPLEX for pcggqrf DOUBLE COMPLEX for pzggqrf. Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*m*-1)). Contains the local pieces of the *n*-by-*m* matrix sub(*A*) to be factored.
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
b	(local) REAL for psggqrf DOUBLE PRECISION for pdggqrf COMPLEX for pcggqrf DOUBLE COMPLEX for pzggqrf. Pointer into the local memory to an array of dimension (*lld_b*, *LOCc*(*jb*+*p*-1)). Contains the local pieces of the *n*-by-*p* matrix sub(*B*) to be factored.
ib, *jb*	(global) INTEGER. The row and column indices in the global array *b* indicating the first row and the first column of the submatrix *B*, respectively.
descb	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix B.
work	(local)

REAL for psggqrf
DOUBLE PRECISION for pdggqrf
COMPLEX for pcggqrf
DOUBLE COMPLEX for pzggqrf.
Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER. Dimension of *work*, must be at least

lwork ≥ max(*nb_a**(*npa*0+*mqa*0+*nb_a*),
max((*nb_a**(*nb_a*-1))/2,
(*pqb*0+*npb*0)**nb_a*)+*nb_a***nb_a*,
*mb_b**(*npb*0+*pqb*0+*mb_b*)),
where
iroffa = mod(*ia-1*, *mb_A*),
icoffa = mod(*ja-1*, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),
iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
npa0 = numroc (*n*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),
mqa0 = numroc (*m*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL)
iroffb = mod(*ib-1*, *mb_b*),
icoffb = mod(*jb-1*, *nb_b*),
ibrow = indxg2p(*ib*, *mb_b*, MYROW, *rsrc_b*, NPROW),
ibcol = indxg2p(*jb*, *nb_b*, MYCOL, *csrc_b*, NPCOL),
npb0 = numroc (*n*+*iroffa*, *mb_b*, MYROW, *Ibrow*, NPROW),
pqb0 = numroc(*m*+*icoffb*, *nb_b*, MYCOL, *ibcol*, NPCOL)
and numroc, indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a	On exit, the elements on and above the diagonal of sub (*A*) contain the min(*n*, *m*)-by-*m* upper trapezoidal matrix *R* (*R* is upper triangular if *n* ≥ *m*); the elements below the diagonal, with the array *taua*, represent the orthogonal/unitary matrix *Q* as a product of min(*n*,*m*) elementary reflectors. (See Application Notes below).
taua, *taub*	(local) REAL for psggqrf DOUBLE PRECISION for pdggqrf COMPLEX for pcggqrf DOUBLE COMPLEX for pzggqrf. Arrays, DIMENSION *LOCc*(*ja*+min(*n*,*m*)-1) for *taua* and *LOCr*(*ib*+*n*-1) for *taub*. The array *taua* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Q*. *taua* is tied to the distributed matrix *A*. (See Application Notes below). The array *taub* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Z*. *taub* is tied to the distributed matrix *B*. (See Application Notes below).
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*ja*) *H*(*ja*+1)... *H*(*ja*+*k*-1),

where *k* = min(*n*,*m*).

Each $H(i)$ has the form

$H(i) = i - taua * v * v'$

where $taua$ is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:n)$ is stored on exit in $A(ia+i:ia+n-1,ja+i-1)$, and $taua$ in $taua(ja+i-1)$.To form Q explicitly, use ScaLAPACK subroutine p?orgqr/p?ungqr. To use Q to update another matrix, use ScaLAPACK subroutine p?ormqr/p?unmqr.

The matrix Z is represented as a product of elementary reflectors

$Z = H(ib) H(ib+1) . . . H(ib+k-1)$, where $k = \min(n,p)$.

Each $H(i)$ has the form

$H(i) = i - taub * v * v'$

where $taub$ is a real/complex scalar, and v is a real/complex vector with $v(p-k+i+1:p) = 0$ and $v(p-k+i) = 1$; $v(1:p-k+i-1)$ is stored on exit in $B(ib+n-k+i-1,jb:jb+p-k+i-2)$, and $taub$ in $taub(ib+n-k+i-1)$. To form Z explicitly, use ScaLAPACK subroutine p?orgrq/p?ungrq. To use Z to update another matrix, use ScaLAPACK subroutine p?ormrq/p?unmrq.

p?ggrqf

Computes the generalized RQ factorization.

Syntax

```
call psggrqf(m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
```

```
call pdggrqf(m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
```

```
call pcggrqf(m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
```

```
call pzggrqf(m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
```

Description

The routine forms the generalized RQ factorization of an m-by-n matrix $\text{sub}(A)=(ia:ia+m-1, ja:ja+n-1)$ and a p-by-n matrix $\text{sub}(B)=(ib:ib+p-1, ja:ja+n-1)$:

sub(*A*) = *R***Q*, sub(*B*) = *Z***T***Q*,

where *Q* is an *n*-by-*n* orthogonal/unitary matrix, *Z* is a *p*-by-*p* orthogonal/unitary matrix, and *R* and *T* assume one of the forms:

$$R = \underset{n-m}{m}\begin{pmatrix} 0 & \underset{m}{R_{12}} \end{pmatrix}, \text{ if } m \le n,$$

or

$$R = \begin{pmatrix} R_{11} \\ R_{12} \end{pmatrix}\begin{matrix} m-n \\ n \end{matrix}, \text{ if } m > n$$
$$\underset{n}{}$$

where R_{11} or R_{21} is upper triangular, and

$$T = \begin{pmatrix} T_{11} \\ 0 \end{pmatrix}\begin{matrix} n \\ p-n \end{matrix}, \text{ if } p \ge n$$
$$\underset{n}{}$$

or

$$T = \underset{p}{p}\begin{pmatrix} \underset{p}{T_{11}} & \underset{n-p}{T_{12}} \end{pmatrix}, \text{ if } p < n,$$

where T^{11} is upper triangular.

In particular, if sub(*B*) is square and nonsingular, the *GRQ* factorization of sub(*A*) and sub(*B*) implicitly gives the *RQ* factorization of sub (*A*)*inv(sub(*B*)):

```
sub(A)*inv(sub(B))= (R*inv(T))*Z'
```

where inv(sub(*B*)) denotes the inverse of the matrix sub(*B*), and *Z'* denotes the transpose of matrix *Z*.

Input Parameters

m	(global) INTEGER. The number of rows in the distributed matrices sub (*A*) (*m*≥0).
p	INTEGER. The number of rows in the distributed matrix sub(*B*) (*p*≥0).
n	(global) INTEGER. The number of columns in the distributed matrices sub(*A*) and sub(*B*) (*n*≥0).
a	(local) REAL for psggrqf DOUBLE PRECISION for pdggrqf COMPLEX for pcggrqf DOUBLE COMPLEX for pzggrqf. Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). Contains the local pieces of the *m*-by-*n* distributed matrix sub(*A*) to be factored.
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
b	(local) REAL for psggrqf DOUBLE PRECISION for pdggrqf COMPLEX for pcggrqf DOUBLE COMPLEX for pzggrqf. Pointer into the local memory to an array of dimension (*lld_b*, *LOCc*(*jb*+*n*-1)). Contains the local pieces of the *p*-by-*n* matrix sub(*B*) to be factored.

ib, *jb*
(global) INTEGER. The row and column indices in the global array *b* indicating the first row and the first column of the submatrix *B*, respectively.

descb
(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

work
(local)
REAL for psggrqf
DOUBLE PRECISION for pdggrqf
COMPLEX for pcggrqf
DOUBLE COMPLEX for pzggrqf.
Workspace array of dimension of *lwork*.

lwork
(local or global) INTEGER.
Dimension of *work*, must be at least *lwork* ≥
max(*mb_a**(*mpa*0+*nqa*0+*mb_a*),
max((*mb_a**(*mb_a*-1))/2, (*ppb*0+*nqb*0)**mb_a*) +
*mb_a***mb_a*, *nb_b**(*ppb*0+*nqb*0+*nb_b*)), where
iroffa = mod(*ia*-1, *mb_A*),
icoffa = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),
iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
mpa0 = *numroc* (*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),
nqa0 = *numroc* (*m*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL)
iroffb = mod(*ib-1*, *mb_b*),
icoffb = mod(*jb-1*, *nb_b*),
ibrow = indxg2p(*ib*, *mb_b*, MYROW, *rsrc_b*, NPROW),
ibcol = indxg2p(*jb*, *nb_b*, MYCOL, *csrc_b*, NPCOL),
ppb0 = numroc (*p*+*iroffb*, *mb_b*, MYROW, *ibrow*, NPROW),
nqb0 = numroc (*n*+*icoffb*, *nb_b*, MYCOL, *ibcol*, NPCOL)
and numroc, indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — On exit, if $m \le n$, the upper triangle of *A*(*ia*:*ia*+*m*-1, *ja*+*n*-*m*:*ja*+*n*-1) contains the *m*-by-*m* upper triangular matrix *R*; if $m \ge n$, the elements on and above the (*m*-*n*)-th subdiagonal contain the *m*-by-*n* upper trapezoidal matrix *R*; the remaining elements, with the array *taua*, represent the orthogonal/unitary matrix *Q* as a product of min(*n*, *m*) elementary reflectors. (See Application Notes below).

taua, *taub* — (local)
REAL for psggqrf
DOUBLE PRECISION for pdggqrf
COMPLEX for pcggqrf
DOUBLE COMPLEX for pzggqrf.
Arrays, DIMENSION *LOCr*(*ia*+*m*-1)for *taua* and *LOCc*(*jb*+min(p,*n*)-1) for *taub*.
The array *taua* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Q*. *taua* is tied to the distributed matrix *A*.(See Application Notes below).
The array *taub* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Z*. *taub* is tied to the distributed matrix *B*. (See Application Notes below).

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$Q = H(ia) H(ia+1) \ldots H(ia+k-1)$,

where $k = \min(m,n)$.

Each $H(i)$ has the form

$H(i) = i - taua * v * v'$

where $taua$ is a real/complex scalar, and v is a real/complex vector with $v(n-k+i+1:n) = 0$ and $v(n-k+i) = 1$; $v(1:n-k+i-1)$ is stored on exit in $A(ia+m-k+i-1, ja:ja+n-k+i-2)$, and $taua$ in $taua(ia+m-k+i-1)$. To form Q explicitly, use ScaLAPACK subroutine p?orgrq/p?ungrq. To use Q to update another matrix, use ScaLAPACK subroutine p?ormrq/p?unmrq.

The matrix Z is represented as a product of elementary reflectors

$Z = H(jb) H(jb+1) \ldots H(jb+k-1)$, where $k = \min(p,n)$.

Each $H(i)$ has the form

$H(i) = i - taub * v * v'$

where $taub$ is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:p)$ is stored on exit in $B(ib+i:ib+p-1, jb+i-1)$, and $taub$ in $taub(jb+i-1)$. To form Z explicitly, use ScaLAPACK subroutine p?orgqr/p?ungqr. To use Z to update another matrix, use ScaLAPACK subroutine p?ormqr/p?unmqr.

Symmetric Eigenproblems

To solve a symmetric eigenproblem with ScaLAPACK, you usually need to reduce the matrix to real tridiagonal form T and then find the eigenvalues and eigenvectors of the tridiagonal matrix T. ScaLAPACK includes routines for reducing the matrix to a tridiagonal form by an orthogonal (or unitary) similarity transformation $A = QTQ^H$ as well as for solving tridiagonal symmetric eigenvalue problems. These routines are listed in Table 6-4 .

There are different routines for symmetric eigenproblems, depending on whether you need eigenvalues only or eigenvectors as well, and on the algorithm used (either the QTQ algorithm, or bisection followed by inverse iteration).

Table 6-4 Computational Routines for Solving Symmetric Eigenproblems

Operation	Dense symmetric/Hermitian matrix	Orthogonal/unitary matrix	Symmetric tridiagonal matrix
Reduce to tridiagonal form $A = QTQ^H$	p?sytrd/p?hetrd		
Multiply matrix after reduction		p?ormtr/p?unmtr	
Find all eigenvalues and eigenvectors of a tridiagonal matrix T by a QTQ method			steqr2*)
Find selected eigenvalues of a tridiagonal matrix T via bisection			p?stebz
Find selected eigenvectors of a tridiagonal matrix T by inverse iteration			p?stein

*) This routine is described as part of auxiliary ScaLAPACK routines.

p?sytrd

Reduces a symmetric matrix to real symmetric tridiagonal form by an orthogonal similarity transformation.

Syntax

```
call pssytrd( uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pdsytrd( uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
```

Description

This routine reduces a real symmetric matrix sub(*A*) to symmetric tridiagonal form *T* by an orthogonal similarity transformation:

Q'*sub(*A*)**Q* = *T*,

where sub(*A*) = *A*(*ia*:*ia*+*n*-1,*ja*:*ja*+*n*-1).

Input Parameters

uplo (global) CHARACTER.
Specifies whether the upper or lower triangular part of the symmetric matrix sub(*A*) is stored:
If *uplo* = 'U', upper triangular

	If *uplo* = 'L', lower triangular
n	(global) INTEGER. The order of the distributed matrix sub(*A*) (*n*≥0).
a	(local) REAL for pssytrd DOUBLE PRECISION for pdsytrd. Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, this array contains the local pieces of the symmetric distributed matrix sub(*A*). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced. See Application Notes below.
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
work	(local) REAL for pssytrd DOUBLE PRECISION for pdsytrd. Workspace array of dimension *lwork*.
lwork	(local or global) INTEGER, dimension of *work*, must be at least: *lwork* ≥ max(NB*(*np* +1), 3*NB), where NB = *mb_a* = *nb_a*, *np* = numroc(*n*, NB, MYROW, *iarow*, NPROW), *iarow* = indxg2p(*ia*, NB, MYROW, *rsrc_a*, NPROW). indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — On exit, if *uplo* = 'U', the diagonal and first superdiagonal of sub(*A*) are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements above the first superdiagonal, with the array *tau*, represent the orthogonal matrix *Q* as a product of elementary reflectors; if *uplo* = 'L', the diagonal and first subdiagonal of sub(*A*) are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements below the first subdiagonal, with the array *tau*, represent the orthogonal matrix *Q* as a product of elementary reflectors. See Application Notes below.

d — (local)
REAL for pssytrd
DOUBLE PRECISION for pdsytrd.
Arrays, DIMENSION *LOCc*(*ja*+*n*-1) .The diagonal elements of the tridiagonal matrix *T*:
d(*i*)= *A*(*i*,*i*).
d is tied to the distributed matrix *A*.

e — (local)
REAL for pssytrd
DOUBLE PRECISION for pdsytrd.
Arrays, DIMENSION *LOCc*(*ja*+*n*-1) if *uplo* = 'U', *LOCc*(*ja*+*n*-2) otherwise.
The off-diagonal elements of the tridiagonal matrix *T*:
e(*i*)= *A*(*i*,*i*+1) if *uplo* = 'U',
e(*i*) = *A*(*i*+1,*i*) if *uplo* = 'L'.
e is tied to the distributed matrix *A*.

tau — (local)
REAL for pssytrd
DOUBLE PRECISION for pdsytrd.

	Arrays, DIMENSION *LOCc*(*ja*+*n*-1). This array contains the scalar factors *tau* of the elementary reflectors. *tau* is tied to the distributed matrix *A*.
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

If *uplo* = 'U', the matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*n*-1)... *H*(2) *H*(1).

Each *H*(*i*) has the form

H(*i*) = *i* - *tau* * *v* * *v*',

where *tau* is a real scalar, and *v* is a real vector with *v*(*i*+1:*n*) = 0 and *v*(*i*) = 1; *v*(1:*i*-1) is stored on exit in *A*(*ia*:*ia*+*i*-2, *ja*+*i*), and *tau* in *tau* (*ja*+*i*-1).

If *uplo* = 'L', the matrix *Q* is represented as a product of elementary reflectors

Q = *H*(1) *H*(2)... *H*(*n*-1).

Each *H*(*i*) has the form

H(*i*) = *i* - *tau* * *v* * *v*',

where *tau* is a real scalar, and *v* is a real vector with *v*(1:*i*) = 0 and *v*(*i*+1) = 1; *v*(*i*+2:*n*) is stored on exit in *A*(*ia*+*i*+1:*ia*+*n*-1,*ja*+*i*-1), and *tau* in *tau*(*ja*+*i*-1).

The contents of sub(*A*) on exit are illustrated by the following examples with *n* = 5:

If *uplo* = 'U':

$$\begin{bmatrix} d & e & v2 & v3 & v4 \\ & d & e & v3 & v4 \\ & & d & e & v4 \\ & & & d & e \\ & & & & d \end{bmatrix}$$

If *uplo* = 'L':

$$\begin{bmatrix} d & & & & \\ e & d & & & \\ v1 & e & d & & \\ v1 & v2 & e & d & \\ v1 & v2 & v3 & e & d \end{bmatrix}$$

where *d* and *e* denote diagonal and off-diagonal elements of *T*, and *v i* denotes an element of the vector defining *H*(*i*).

p?ormtr

Multiplies a general matrix by the orthogonal transformation matrix from a reduction to tridiagonal form determined by p?sytrd*.*

Syntax

call psormtr(*side*, *uplo*, *trans*, *m*, *n*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pdormtr(*side*, *uplo*, *trans*, *m*, *n*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Description

The routine overwrites the general real distributed *m*-by-*n* matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'T':	Q^T*sub(*C*)	sub(*C*)*Q^T

where *Q* is a real orthogonal distributed matrix of order *nq*, with *nq* = *m* if *side* = 'L' and *nq* = *n* if *side* = 'R'.

Q is defined as the product of *nq* elementary reflectors, as returned by p?sytrd.

If *uplo* = 'U', *Q* = *H*(*nq*-1)... *H*(2) *H*(1);

If *uplo* = 'L', *Q* = *H*(1) *H*(2)... *H*(*nq*-1).

Input Parameters

side — (global) CHARACTER
='L': *Q* or Q^T is applied from the left.
='R': *Q* or Q^T is applied from the right.

trans — (global) CHARACTER
='N', no transpose, *Q* is applied.
='T', transpose, Q^T is applied.

uplo — (global) CHARACTER.
= 'U': Upper triangle of *A*(*ia*:*, *ja*:*) contains elementary reflectors from p?sytrd;
= 'L': Lower triangle of *A*(*ia*:*,*ja*:*) contains elementary reflectors from p?sytrd

m — (global) INTEGER. The number of rows in the distributed matrix sub(*C*) (*m*≥0).

n — (global) INTEGER. The number of columns in the distributed matrix sub(*C*) (*n*≥0).

a — (local)
REAL for psormtr
DOUBLE PRECISION for pdormtr.

Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*m*-1)) if *side*='L', or (*lld_a*, *LOCc*(*ja*+*n*-1)) if *side* = 'R'.
Contains the vectors which define the elementary reflectors, as returned by p?sytrd.

If *side*='L', *lld_a* ≥ max(1,*LOCr*(*ia*+*m*-1));

If *side* ='R', *lld_a* ≥ max(1, *LOCr*(*ia*+*n*-1)).

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)
REAL for psormtr
DOUBLE PRECISION for pdormtr.
Array, DIMENSION of *ltau* where
if *side* = 'L' and *uplo* = 'U', *ltau* = *LOCc*(*m_a*),
if *side* = 'L' and *uplo* = 'L', *ltau* = *LOCc*(*ja*+*m*-2),
if *side* = 'R' and *uplo* = 'U', *ltau* = *LOCc*(*n_a*),
if *side* = 'R' and *uplo* = 'L', *ltau* = *LOCc*(*ja*+*n*-2).
tau(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by p?sytrd. *tau* is tied to the distributed matrix *A*.

c (local) REAL for psormtr
DOUBLE PRECISION for pdormtr.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc* (*ja*+*n*-1)). Contains the local pieces of the distributed matrix sub (*C*).

ic, *jc* (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)
REAL for psormtr
DOUBLE PRECISION for pdormtr.

Workspace array of dimension *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:

if *uplo* = 'U',
iaa= *ia*; *jaa*= *ja*+1, *icc*= *ic*; *jcc*= *jc*;
else *uplo* = 'L',
iaa= *ia*+1, *jaa*= *ja*;
If *side* = 'L',
icc= *ic*+1; *jcc*= *jc*;
else *icc*= *ic*; *jcc*= *jc*+1;
end if
end if
If *side* = 'L',
mi= *m*-1; *ni*= *n*

lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + *mpc*0)**nb_a*) + *nb_a***nb_a*

else
If *side* = 'R',
mi= *m*; *mi* = *n*-1;

lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + max(*npa*0+numroc(numroc(*ni*+*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*)+ *nb_a***nb_a*

end if
where *lcmq* = *lcm*/NPCOL with *lcm* = ilcm(NPROW, NPCOL),
iroffa = mod(*iaa*-1, *mb_a*),
icoffa = mod(*jaa*-1, *nb_a*),
iarow = indxg2p(*iaa*, *mb_a*, MYROW, *rsrc_a*, NPROW),
*npa*0 = numroc(*ni*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),
iroffc = mod(*icc*-1, *mb_c*),
icoffc = mod(*jcc*-1, *nb_c*),
icrow = indxg2p(*icc*, *mb_c*, MYROW, *rsrc_c*, NPROW),
iccol = indxg2p(*jcc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),
*mpc*0 = numroc(*mi*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

*nqc*0 = numroc(*ni*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo. If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c	Overwritten by the product *Q**sub(*C*), or *Q'**sub(*C*), or sub(*C*)**Q'*, or sub(*C*)**Q*.
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?hetrd

Reduces a Hermitian matrix to Hermitian tridiagonal form by a unitary similarity transformation.

Syntax

```
call pchetrd( uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pzhetrd( uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
```

Description

This routine reduces a complex Hermitian matrix sub(*A*) to Hermitian tridiagonal form *T* by a unitary similarity transformation:

Q'*sub(*A*)**Q* = *T*

where sub(*A*) = *A*(*ia*:*ia*+*n*-1,*ja*:*ja*+*n*-1).

Input Parameters

uplo	(global) CHARACTER. Specifies whether the upper or lower triangular part of the Hermitian matrix sub(*A*) is stored: If *uplo* = 'U', upper triangular If *uplo* = 'L', lower triangular
n	(global) INTEGER. The order of the distributed matrix sub(*A*) (*n*≥0).
a	(local) COMPLEX for pchetrd DOUBLE COMPLEX for pzhetrd. Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, this array contains the local pieces of the Hermitian distributed matrix sub(*A*). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.(See Application Notes below).
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
work	(local) COMPLEX for pchetrd DOUBLE COMPLEX for pzhetrd. Workspace array of dimension *lwork*.
lwork	(local or global) INTEGER, dimension of *work*, must be at least: *lwork*≥max(NB*(*np* +1), 3*NB)

where NB = *mb_a* = *nb_a*,
np = numroc(*n*, NB, MYROW, *iarow*, NPROW),
iarow = indxg2p(*ia*, NB, MYROW, *rsrc_a*, NPROW).
indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a

On exit,
If *uplo* = 'U', the diagonal and first superdiagonal of sub(*A*) are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements above the first superdiagonal, with the array *tau*, represent the unitary matrix *Q* as a product of elementary reflectors;if *uplo* = 'L', the diagonal and first subdiagonal of sub(*A*) are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements below the first subdiagonal, with the array *tau*, represent the unitary matrix *Q* as a product of elementary reflectors. (See Application Notes below).

d

(local)
REAL for pchetrd
DOUBLE PRECISION for pzhetrd.
Arrays, DIMENSION *LOCc*(*ja*+*n*-1). The diagonal elements of the tridiagonal matrix *T*:
d(*i*)= *A*(*i*,*i*).
d is tied to the distributed matrix *A*.

e

(local)
REAL for pchetrd
DOUBLE PRECISION for pzhetrd.
Arrays, DIMENSION *LOCc*(*ja*+*n*-1) if *uplo* = 'U';
LOCc(*ja*+*n*-2) - otherwise.
The off-diagonal elements of the tridiagonal matrix *T*:

	e(*i*)= *A*(*i*,*i*+1) if *uplo* = 'U', *e*(*i*)= *A*(*i*+1,*i*) if *uplo* = 'L'. *e* is tied to the distributed matrix *A*.
tau	(local) COMPLEX for pchetrd DOUBLE COMPLEX for pzhetrd. Arrays, DIMENSION *LOCc*(*ja*+*n*-1). This array contains the scalar factors *tau* of the elementary reflectors. *tau* is tied to the distributed matrix *A*.
work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

If *uplo* = 'U', the matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*n*-1)... *H*(2) *H*(1).

Each *H*(*i*) has the form

H(*i*) = *i* - *tau* * *v* * *v*',

where *tau* is a complex scalar, and *v* is a complex vector with *v*(*i*+1:*n*) = 0 and *v*(*i*) = 1; *v*(1:*i*-1) is stored on exit in *A*(*ia*:*ia*+*i*-2, *ja*+*i*), and *tau* in *tau* (*ja*+*i*-1).

If *uplo* = 'L', the matrix *Q* is represented as a product of elementary reflectors

Q = *H*(1) *H*(2)... *H*(*n*-1).

Each *H*(*i*) has the form

H(*i*) = *i* - *tau* * *v* * *v*',

where *tau* is a complex scalar, and *v* is a complex vector with *v*(1:*i*) = 0 and *v*(*i*+1) = 1; *v*(*i*+2:*n*) is stored on exit in *A*(*ia*+*i*+1:*ia*+*n*-1,*ja*+*i*-1), and *tau* in *tau*(*ja*+*i*-1).

The contents of sub(*A*) on exit are illustrated by the following examples with *n* = 5:

If *uplo* = 'U':

$$\begin{bmatrix} d & e & v2 & v3 & v4 \\ & d & e & v3 & v4 \\ & & d & e & v4 \\ & & & d & e \\ & & & & d \end{bmatrix}$$

If *uplo* = 'L':

$$\begin{bmatrix} d & & & & \\ e & d & & & \\ v1 & e & d & & \\ v1 & v2 & e & d & \\ v1 & v2 & v3 & e & d \end{bmatrix}$$

where *d* and *e* denote diagonal and off-diagonal elements of *T*, and *v i* denotes an element of the vector defining *H*(*i*).

p?unmtr

Multiplies a general matrix by the unitary transformation matrix from a reduction to tridiagonal form determined by p?hetrd.

Syntax

call pcunmtr(*side*, *uplo*, *trans*, *m*, *n*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pzunmtr(*side*, *uplo*, *trans*, *m*, *n*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Description

The routine overwrites the general complex distributed *m*-by-*n* matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'C':	Q^H*sub(*C*)	sub(*C*)*Q^H

where *Q* is a complex unitary distributed matrix of order *nq*, with *nq* = *m* if *side* = 'L' and *nq* = *n* if *side* = 'R'.

Q is defined as the product of *nq*-1 elementary reflectors, as returned by p?hetrd.

If *uplo* = 'U', *Q* = *H*(*nq*-1)... *H*(2) *H*(1);

If *uplo* = 'L', *Q* = *H*(1) *H*(2)... *H*(*nq*-1).

Input Parameters

side	(global) CHARACTER ='L': *Q* or Q^H is applied from the left. ='R': *Q* or Q^H is applied from the right.
trans	(global) CHARACTER ='N', no transpose, *Q* is applied. ='C', conjugate transpose, Q^H is applied.
uplo	(global) CHARACTER.

= 'U': Upper triangle of *A*(*ia*:*, *ja*:*) contains elementary reflectors from p?hetrd;
= 'L': Lower triangle of *A*(*ia*:*,*ja*:*) contains elementary reflectors from p?hetrd

m (global) INTEGER. The number of rows in the distributed matrix sub(*C*) (*m*≥0).

n (global) INTEGER. The number of columns in the distributed matrix sub(*C*) (*n*≥0).

a (local)
REAL for pcunmtr
DOUBLE PRECISION for pzunmtr.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*m*-1)) if *side*='L', or (*lld_a*, *LOCc*(*ja*+*n*-1)) if *side* = 'R'.
Contains the vectors which define the elementary reflectors, as returned by p?hetrd.

If *side*='L', *lld_a* ≥ max(1,*LOCr*(*ia*+*m*-1));

If *side* ='R', *lld_a* ≥ max(1, *LOCr*(*ia*+*n*-1)).

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)
COMPLEX for pcunmtr
DOUBLE COMPLEX for pzunmtr.
Array, DIMENSION of *ltau* where
If *side* = 'L' and *uplo* = 'U', *ltau* = *LOCc*(*m_a*),
if *side* = 'L' and *uplo* = 'L', *ltau* = *LOCc*(*ja*+*m*-2),
if *side* = 'R' and *uplo* = 'U', *ltau* = *LOCc*(*n_a*),
if *side* = 'R' and *uplo* = 'L', *ltau* = *LOCc*(*ja*+*n*-2).
tau(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by p?hetrd. *tau* is tied to the distributed matrix *A*.

c (local) COMPLEX for pcunmtr

DOUBLE COMPLEX for pzunmtr.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc* (*ja*+*n*-1)). Contains the local pieces of the distributed matrix sub (*C*).

ic, *jc* — (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work — (local)
COMPLEX for pcunmtr
DOUBLE COMPLEX for pzunmtr.
Workspace array of dimension *lwork*.

lwork — (local or global) INTEGER, dimension of *work*, must be at least:
If *uplo* = 'U',
iaa= *ia*; *jaa*= *ja*+1, *icc*= *ic*; *jcc*= *jc*;
else *uplo* = 'L',
iaa= *ia*+1, *jaa*= *ja*;
If *side* = 'L',
icc= *ic*+1; *jcc*= *jc*;
else *icc*= *ic*; *jcc*= *jc*+1;
end if
end if
If *side* = 'L',
mi= *m*-1; *ni*= *n*
lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + *mpc*0)**nb_a*) + *nb_a***nb_a*
else
If *side* = 'R',
mi= *m*; *mi* = *n*-1;
lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + max(*npa*0+numroc(numroc(*ni*+*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a***nb_a*
end if

where *lcmq* = *lcm*/NPCOL with *lcm* = ilcm(NPROW, NPCOL),
iroffa = mod(*iaa*-1, *mb_a*),
icoffa = mod(*jaa*-1, *nb_a*),
iarow = indxg2p(*iaa*, *mb_a*, MYROW, *rsrc_a*, NPROW),
*npa*0 = numroc(*ni*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),
iroffc = mod(*icc*-1, *mb_c*),
icoffc = mod(*jcc*-1, *nb_c*),
icrow = indxg2p(*icc*, *mb_c*, MYROW, *rsrc_c*, NPROW),
iccol = indxg2p(*jcc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),
*mpc*0 = numroc(*mi*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),
*nqc*0 = numroc(*ni*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo. If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — Overwritten by the product *Q**sub(*C*), or *Q*'*sub(*C*), or sub(*C*)**Q*', or sub(*C*)**Q*.

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - *i*

p?stebz

Computes the eigenvalues of a symmetric tridiagonal matrix by bisection.

Syntax

call psstebz(*ictxt*, *range*, *order*, *n*, *vl*, *vu*, *il*, *iu*, *abstol*, *d*, *e*, *m*, *nsplit*, *w*, *iblock*, *isplit*, *work*, *iwork*, *liwork*, *info*)

call pdstebz(*ictxt*, *range*, *order*, *n*, *vl*, *vu*, *il*, *iu*, *abstol*, *d*, *e*, *m*, *nsplit*, *w*, *iblock*, *isplit*, *work*, *iwork*, *liwork*, *info*)

Description

This routine computes the eigenvalues of a symmetric tridiagonal matrix in parallel. These may be all eigenvalues, all eigenvalues in the interval [*vl vu*], or the eigenvalues indexed *il* through *iu*. A static partitioning of work is done at the beginning of p?stebz which results in all processes finding an (almost) equal number of eigenvalues.

Input Parameters

ictxt	(global) INTEGER. The BLACS context handle.
range	(global) CHARACTER. Must be 'A' or 'V' or 'I'. If *range* = 'A', the routine computes all eigenvalues. If *range* = 'V', the routine computes eigenvalues in the interval [*vl vu*]. If *range* ='I', the routine computes eigenvalues with indices *il* to *iu*.
order	(global) CHARACTER. Must be 'B' or 'E'. If *order* = 'B', the eigenvalues are to be ordered from smallest to largest within each split-off block. If *order* = 'E', the eigenvalues for the entire matrix are to be ordered from smallest to largest.
n	(global) INTEGER. The order of the tridiagonal matrix *T* (*n*≥0).
vl, *vu*	(global) REAL for psstebz DOUBLE PRECISION for pdstebz.

If *range* = 'V', the routine computes the lower and the upper bounds for the eigenvalues on the interval [*1*, *vu*]. If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu* (global)

INTEGER. Constraint: 1≤*il*≤*iu*≤*n*.

If *range* = 'I', the index of the smallest eigenvalue is returned for *il* and of the largest eigenvalue for *iu* (assuming that the eigenvalues are in ascending order) must be returned. *il* must be at least 1. *iu* must be at least *il* and no greater than *n*.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol (global)

REAL for psstebz

DOUBLE PRECISION for pdstebz.

The absolute tolerance to which each eigenvalue is required. An eigenvalue (or cluster) is considered to have converged if it lies in an interval of width *abstol*. If *abstol*≤0, then the tolerance is taken as *ulp*||*T*||, where *ulp* is the machine precision, and ||*T*|| means the 1-norm of *T*

Eigenvalues will be computed most accurately when *abstol* is set to the underflow threshold slamch('U'), not 0. Note that if eigenvectors are desired later by inverse iteration (p?stein), *abstol* should be set to 2*p?lamch('S').

d (global)

REAL for psstebz

DOUBLE PRECISION for pdstebz.

Array, DIMENSION (*n*).

Contains *n* diagonal elements of the tridiagonal matrix *T*. To avoid overflow, the matrix must be scaled so that its largest entry is no greater than the $overflow^{(1/2)} * underflow^{(1/4)}$ in absolute value, and for greatest accuracy, it should not be much smaller than that.

e (global)

REAL for psstebz

DOUBLE PRECISION for pdstebz.

Array, DIMENSION (*n* – 1).

Contains (*n*-1) off-diagonal elements of the tridiagonal matrix *T*. To avoid overflow, the matrix must be scaled so that its largest entry is no greater than $overflow^{(1/2)} * underflow^{(1/4)}$ in absolute value, and for greatest accuracy, it should not be much smaller than that.

work (local)
REAL for psstebz
DOUBLE PRECISION for pdstebz.
Array, DIMENSION max(5*n*, 7). This is a workspace array.

lwork (local) INTEGER. The size of the *work* array must be ≥ max(5*n*, 7).
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

iwork (local) INTEGER. Array, DIMENSION max(4*n*, 14). This is a workspace array.

liwork (local) INTEGER. the size of the *iwork* array must ≥ max(4*n*, 14, NPROCS).
If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

m (global) INTEGER. The actual number of eigenvalues found. $0 \le m \le n$

nsplit (global) INTEGER. The number of diagonal blocks detected in *T*. $1 \le nsplit \le n$

w (global)
REAL for psstebz
DOUBLE PRECISION for pdstebz.

Array, DIMENSION (*n*). On exit, the first *m* elements of *w* contain the eigenvalues on all processes.

iblock (global) INTEGER.
Array, DIMENSION (*n*). At each row/column *j* where *e*(*j*) is zero or small, the matrix *T* is considered to split into a block diagonal matrix. On exit *iblock*(*i*) specifies which block (from 1 to the number of blocks) the eigenvalue *w*(*i*) belongs to.

NOTE. In the (theoretically impossible) event that bisection does not converge for some or all eigenvalues, *info* is set to 1 and the ones for which it did not are identified by a negative block number.

isplit (global) INTEGER.
Array, DIMENSION (*n*).
Contains the splitting points, at which *T* breaks up into submatrices. The first submatrix consists of rows/columns 1 to *isplit*(1), the second of rows/columns *isplit*(1)+1 through *isplit*(2), etc., and the *nsplit*-th consists of rows/columns *isplit*(*nsplit*-1)+1 through *isplit*(*nsplit*)=*n*. (Only the first *nsplit* elements are used, but since the *nsplit* values are not known, *n* words must be reserved for *isplit*.)

info (global) INTEGER.
If *info* = 0, the execution is successful.
If *info* < 0, if *info* = -*i*, the *i*-th argument has an illegal value.
If *info* > 0, some or all of the eigenvalues fail to converge or not computed.
If *info* = 1, bisection fails to converge for some eigenvalues; these eigenvalues are flagged by a negative block number. The effect is that the eigenvalues may not be as accurate as the absolute and relative tolerances.
If *info* = 2, mismatch between the number of eigenvalues output and the number desired.

If *info* = 3: *range*='*i*', and the Gershgorin interval initially used is incorrect. No eigenvalues are computed. Probable cause: the machine has a sloppy floating point arithmetic. Increase the *fudge* parameter, recompile, and try again.

p?stein

Computes the eigenvectors of a tridiagonal matrix using inverse iteration.

Syntax

call psstein(*n*, *d*, *e*, *m*, *w*, *iblock*, *isplit*, *orfac*, *z*, *iz*, *jz*, *descz*, *work*, *lwork*, *iwork*, *liwork*, *ifail*, *iclustr*, *gap*, *info*)

call pdstein(*n*, *d*, *e*, *m*, *w*, *iblock*, *isplit*, *orfac*, *z*, *iz*, *jz*, *descz*, *work*, *lwork*, *iwork*, *liwork*, *ifail*, *iclustr*, *gap*, *info*)

call pcstein(*n*, *d*, *e*, *m*, *w*, *iblock*, *isplit*, *orfac*, *z*, *iz*, *jz*, *descz*, *work*, *lwork*, *iwork*, *liwork*, *ifail*, *iclustr*, *gap*, *info*)

call pzstein(*n*, *d*, *e*, *m*, *w*, *iblock*, *isplit*, *orfac*, *z*, *iz*, *jz*, *descz*, *work*, *lwork*, *iwork*, *liwork*, *ifail*, *iclustr*, *gap*, *info*)

Description

This routine computes the eigenvectors of a symmetric tridiagonal matrix *T* corresponding to specified eigenvalues, by inverse iteration. p?stein does not orthogonalize vectors that are on different processes. The extent of orthogonalization is controlled by the input parameter *lwork*. Eigenvectors that are to be orthogonalized are computed by the same process. p?stein decides on the allocation of work among the processes and then calls ?stein2 (modified LAPACK routine) on each individual process. If insufficient workspace is allocated, the expected orthogonalization may not be done.

NOTE. If the eigenvectors obtained are not orthogonal, increase *lwork* and run the code again.

p = NPROW * NPCOL is the total number of processes.

Input Parameters

n	(global) INTEGER. The order of the matrix *T* (*n* ≥ 0).
m	(global) INTEGER. The number of eigenvectors to be returned.
d, *e*, *w*	(global) REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: *d*(*) contains the diagonal elements of *T*. DIMENSION (*n*). *e*(*) contains the off-diagonal elements of *T*. DIMENSION (*n*–1). *w*(*) contains all the eigenvalues grouped by split-off block.The eigenvalues are supplied from smallest to largest within the block. (Here the output array *w* from p?stebz with order = 'B' is expected. The array should be replicated in all processes.) DIMENSION(*m*)
iblock	(global) INTEGER. Array, DIMENSION (*n*). The submatrix indices associated with the corresponding eigenvalues in *w*--1 for eigenvalues belonging to the first submatrix from the top, 2 for those belonging to the second submatrix, etc. (The output array *iblock* from p?stebz is expected here).
isplit	(global) INTEGER. Array, DIMENSION (*n*). The splitting points, at which *T* breaks up into submatrices. The first submatrix consists of rows/columns 1 to *isplit*(1), the second of rows/columns *isplit*(1)+1 through *isplit*(2), etc., and the *nsplit*-th consists of rows/columns *isplit*(*nsplit*–1)+1 through *isplit*(*nsplit*)=*n* . (The output array *isplit* from p?stebz is expected here.)
orfac	(global) REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors.

orfac specifies which eigenvectors should be orthogonalized. Eigenvectors that correspond to eigenvalues within *orfac**||T|| of each other are to be orthogonalized. However, if the workspace is insufficient (see *lwork*), this tolerance may be decreased until all eigenvectors can be stored in one process. No orthogonalization is done if *orfac* is equal to zero. A default value of 1000 is used if *orfac* is negative. *orfac* should be identical on all processes

iz, *jz*

(global) INTEGER. The row and column indices in the global array *z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz

(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *Z*.

work

(local). REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
Workspace array, DIMENSION (*lwork*).

lwork

(local) INTEGER.
lwork controls the extent of orthogonalization which can be done. The number of eigenvectors for which storage is allocated on each process is *nvec* = floor((*lwork*-max(5**n*,*np*00**mq*00))/*n*). Eigenvectors corresponding to eigenvalue clusters of size *nvec* - ceil(*m*/*p*) + 1 are guaranteed to be orthogonal (the orthogonality is similar to that obtained from ?stein2).

NOTE. *lwork* must be no smaller than max(5**n*,*np*00**mq*00) + ceil(*m*/*p*)**n* and should have the same input value on all processes.

It is the minimum value of *lwork* input on different processes that is significant.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

iwork (local) INTEGER.
Workspace array, DIMENSION (3*n*+*p*+1).

liwork (local) INTEGER. The size of the array *iwork*. It must be ≥ 3**n* + *p* + 1.
If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

z (local)
REAL for psstein
DOUBLE PRECISION for pdstein
COMPLEX for pcstein
DOUBLE COMPLEX for pzstein.
Array, DIMENSION (*descz*(*dlen_*), *n*/NPCOL + NB). *z* contains the computed eigenvectors associated with the specified eigenvalues. Any vector which fails to converge is set to its current iterate after MAXIT iterations (See ?stein2). On output, *z* is distributed across the *p* processes in block cyclic format.

work(1) On exit, *work*(1) gives a lower bound on the workspace (*lwork*) that guarantees the user desired orthogonalization (see *orfac*). Note that this may overestimate the minimum workspace needed.

iwork On exit, *iwork*(1) contains the amount of integer workspace required.
On exit, the *iwork*(2) through *iwork*(*p*+2) indicate the eigenvectors computed by each process. Process *i* computes eigenvectors indexed *iwork*(*i*+2)+1 through *iwork*(*i*+3).

ifail (global). INTEGER. Array, DIMENSION (*m*). On normal exit, all elements of *ifail* are zero. If one or more eigenvectors fail to converge after MAXIT iterations (as in ?stein), then *info* > 0 is returned. If mod(*info*, *m*+1)>0, then for *i*=1

to mod(*info*,*m*+1), the eigenvector corresponding to the eigenvalue *w*(*ifail*(*i*)) failed to converge (*w* refers to the array of eigenvalues on output).

iclustr (global) INTEGER. Array, DIMENSION (2**p*)
This output array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be orthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*(2*I-1) to *iclustr*(2*I), *i* = 1 to *info*/(*m*+1), could not be orthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these * clusters may not be orthogonal. *iclustr* is a zero terminated array
---(*iclustr*(2**k*).*ne*.0.*and*.*iclustr*(2**k*+1).*eq*.*0*)
if and only if *k* is the number of clusters.

gap (global)
REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
This output array contains the gap between eigenvalues whose eigenvectors could not be orthogonalized. The *info*/*m* output values in this array correspond to the *info*/(*m*+1) clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the *i-th* cluster may be as high as (*O*(*n*)**macheps*)/*gap*(*i*).

info (global) INTEGER.
If *info* = 0, the execution is successful.
If *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*),
If the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
If *info* < 0: if *info* = -*i*, the *i*-th argument had an illegal value.
If *info* > 0: if mod(*info*, *m*+1) = *i*, then *i* eigenvectors failed to converge in MAXIT iterations. Their indices are stored in the array *ifail*. If *info*/(*m*+1) = *i*, then eigenvectors corresponding to *i* clusters of eigenvalues could not be orthogonalized due to insufficient workspace. The indices of the clusters are stored in the array *iclustr*.

Nonsymmetric Eigenvalue Problems

This section describes ScaLAPACK routines for solving nonsymmetric eigenvalue problems, computing the Schur factorization of general matrices, as well as performing a number of related computational tasks.

To solve a nonsymmetric eigenvalue problem with ScaLAPACK, you usually need to reduce the matrix to the upper Hessenberg form and then solve the eigenvalue problem with the Hessenberg matrix obtained.

Table 6-5 lists ScaLAPACK routines for reducing the matrix to the upper Hessenberg form by an orthogonal (or unitary) similarity transformation $A = QHQ^H$, as well as routines for solving eigenproblems with Hessenberg matrices, and multiplying the matrix after reduction.

Table 6-5 Computational Routines for Solving Nonsymmetric Eigenproblems

Operation performed	General matrix	Orthogonal/Unitary matrix	Hessenberg matrix
Reduce to Hessenberg form $A = QHQ^H$	p?gehrd		
Multiply the matrix after reduction		p?ormhr/p?unmhr	
Find eigenvalues and Schur factorization			p?lahqr

p?gehrd

Reduces a general matrix to upper Hessenberg form.

Syntax

```
call psgehrd( n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pdgehrd( n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pcgehrd( n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pzgehrd( n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine reduces a real/complex general distributed matrix sub (*A*) to upper Hessenberg form *H* by an orthogonal or unitary similarity transformation

Q'*sub(*A*)**Q* = *H*,

where sub(*A*) = *A*(*ia*+*n*-1:*ia*+*n*-1, *ja*+*n*-1:*ja*+*n*-1).

Input Parameters

n	(global) INTEGER. The order of the distributed matrix sub(*A*) (*n*≥0).
ilo, *ihi*	(global) INTEGER. It is assumed that sub(*A*) is already upper triangular in rows *ia*:*ia*+*ilo*-2 and *ia*+*ihi*:*ia*+*n*-1 and columns *ja*:*ja*+*ilo*-2 and *ja*+*ihi*:*ja*+*n*-1. (See Application Notes below). If *n* > 0, 1≤*ilo*≤*ihi*≤*n*; otherwise set *ilo* = 1, *ihi* = *n*.
a	(local) REAL for psgehrd DOUBLE PRECISION for pdgehrd COMPLEX for pcgehrd DOUBLE COMPLEX for pzgehrd. Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, this array contains the local pieces of the *n*-by-*n* general distributed matrix sub(*A*) to be reduced.
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
work	(local) REAL for psgehrd DOUBLE PRECISION for pdgehrd COMPLEX for pcgehrd DOUBLE COMPLEX for pzgehrd. Workspace array of dimension *lwork*.
lwork	(local or global) INTEGER, dimension of the array *work*. *lwork* is local input and must be at least *lwork*≥NB*NB + NB*max(*ihip*+1, *ihlp*+*inlq*) where NB = *mb_a* = *nb_a*, *iroffa* = mod(*ia*-1, NB),

icoffa = mod(*ja*-1, NB),
ioff = mod(*ia*+*ilo*-2, NB), *iarow* = indxg2p(*ia*, NB, MYROW, *rsrc_a*, NPROW), *ihip* = numroc(*ihi*+*iroffa*, NB, MYROW, *iarow*, NPROW),
ilrow = indxg2p(*ia*+*ilo*-1, NB, MYROW, *rsrc_a*, NPROW),
ihlp = numroc(*ihi*-*ilo*+*ioff*+1, NB, MYROW, *ilrow*, NPROW),
ilcol = indxg2p(*ja*+*ilo*-1, NB, MYCOL, *csrc_a*, NPCOL),
inlq = numroc(*n*-*ilo*+*ioff*+1, NB, MYCOL, *ilcol*, NPCOL),
indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a

On exit, the upper triangle and the first subdiagonal of sub(*A*)are overwritten with the upper Hessenberg matrix *H*, and the elements below the first subdiagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors. (See Application Notes below).

tau

(local). REAL for psgehrd
DOUBLE PRECISION for pdgehrd
COMPLEX for pcgehrd
DOUBLE COMPLEX for pzgehrd.
Array, DIMENSION at least max(*ja*+*n*-2).
The scalar factors of the elementary reflectors (see Application Notes below). Elements *ja*:*ja*+*ilo*-2 and *ja*+*ihi*:*ja*+*n*-2 of *tau* are set to zero. *tau* is tied to the distributed matrix *A*.

work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = – (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of (*ihi*-*ilo*) elementary reflectors

Q = *H*(*ilo*) *H*(*ilo*+1)... *H*(*ihi*-1).

Each *H*(*i*) has the form

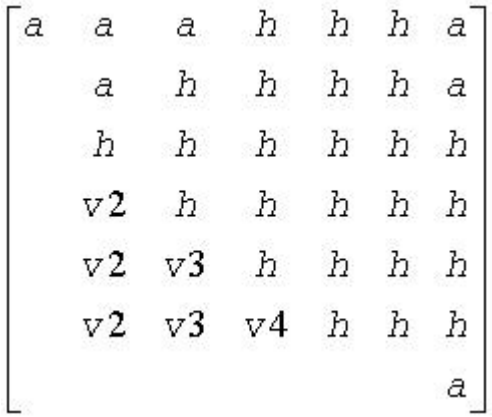
H(*i*)= *i* – *tau* * *v* * *v*'

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1:*i*)= 0, *v*(*i*+1)= 1 and *v*(*ihi*+1:*n*)= 0; *v*(*i*+2:*ihi*) is stored on exit in *a*(*ia*+*ilo*+*i*:*ia*+*ihi*-1,*ja*+*ilo*+*i*-2), and *tau* in *tau*(*ja*+*ilo*+*i*-2). The contents of *a*(*ia*:*ia*+*n*-1,*ja*:*ja*+*n*-1) are illustrated by the following example, with *n* = 7, *ilo* = 2 and *ihi* = 6:

on entry

$$\begin{bmatrix} a & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & & & & & & a \end{bmatrix}$$

on exit



where *a* denotes an element of the original matrix sub(*A*), *H* denotes a modified element of the upper Hessenberg matrix *H*, and *vi* denotes an element of the vector defining *H*(*ja*+*ilo*+*i*-2).

p?ormhr

Multiplies a general matrix by the orthogonal transformation matrix from a reduction to Hessenberg form determined by p?gehrd.

Syntax

call psormhr(*side*, *trans*, *m*, *n*, *ilo*, *ihi*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pdormhr(*side*, *trans*, *m*, *n*, *ilo*, *ihi*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Description

The routine overwrites the general real distributed *m*-by-*n* matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1) with

	side = 'L'	*side* = 'R'
trans = 'N':	Q*sub(*C*)	sub(*C*)*Q
trans = 'T':	Q^T*sub(*C*)	sub(*C*)*Q^T

where Q is a real orthogonal distributed matrix of order nq, with $nq = m$ if $side = 'L'$ and $nq = n$ if $side = 'R'$.

Q is defined as the product of $ihi-ilo$ elementary reflectors, as returned by p?gehrd.

$Q = H(ilo)\ H(ilo+1)\ldots\ H(ihi-1)$.

Input Parameters

side	(global) CHARACTER ='L': Q or Q^T is applied from the left. ='R': Q or Q^T is applied from the right.
trans	(global) CHARACTER ='N', no transpose, Q is applied. ='T', transpose, Q^T is applied.
m	(global) INTEGER. The number of rows in the distributed matrix sub (C) ($m \geq 0$).
n	(global) INTEGER. The number of columns in he distributed matrix sub (C) ($n \geq 0$).
ilo, *ihi*	(global) INTEGER. *ilo* and *ihi* must have the same values as in the previous call of p?gehrd. Q is equal to the unit matrix except for the distributed submatrix $Q(ia+ilo:ia+ihi-1, ia+ilo:ja+ihi-1)$. If $side = 'L'$, $1 \leq ilo \leq ihi \leq \max(1,m)$; If $side = 'R'$, $1 \leq ilo \leq ihi \leq \max(1,n)$; *ilo* and *ihi* are relative indexes.
a	(local) REAL for psormhr DOUBLE PRECISION for pdormhr Pointer into the local memory to an array of dimension $(lld\_a,\ LOCc(ja+m-1))$ if $side='L'$, and $(lld\_a,\ LOCc(ja+n-1))$ if $side = 'R'$. Contains the vectors which define the elementary reflectors, as returned by p?gehrd.

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)
REAL for psormhr
DOUBLE PRECISION for pdormhr
Array, DIMENSION *LOCc*(*ja*+*m*-2), if *side* = 'L', and *LOCc*(*ja*+*n*-2) if *side* = 'R'.
This array contains the scalar factors *tau*(*j*) of the elementary reflectors *H*(*j*) as returned by p?gehrd. *tau* is tied to the distributed matrix *A*.

c (local)
REAL for psormhr
DOUBLE PRECISION for pdormhr
Pointer into the local memory to an array of dimension (*lld_c*,*LOCc*(*jc*+*n*-1)).
Contains the local pieces of the distributed matrix sub(*C*).

ic, *jc* (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)
REAL for psormhr
DOUBLE PRECISION for pdormhr
Workspace array with dimension *lwork*.

lwork (local or global) INTEGER.
The dimension of the array *work*.
lwork must be at least *iaa* = *ia* + *ilo*; *jaa* = *ja*+*ilo*-1;
If *side* = 'L',
mi = *ihi*-*ilo*; *ni* = *n*; *icc* = *ic* + *ilo*; *jcc* = *jc*;
lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0+*mpc*0)**nb_a*) + *nb_a***nb_a*

else if *side* = 'R',
mi = *m*; *ni* = *ihi*-*ilo*; *icc* = *ic*; *jcc* = *jc* + *ilo*;
lwork ≥ max((*nb_a**(*nb_a*-1))/2,
(*nqc*0+max(*npa*0+numroc(numroc(*ni*+*icoffc*, *nb_a*,
0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) +
*nb_a***nb_a*
end if
where *lcmq* = *lcm*/NPCOL with *lcm* = ilcm(NPROW,
NPCOL),
iroffa = mod(*iaa*-1, *mb_a*),
icoffa = mod(*jaa*-1, *nb_a*),
iarow = indxg2p(*iaa*, *mb_a*, MYROW, *rsrc_a*, NPROW),
*npa*0 = numroc(*ni*+*iroffa*, *mb_a*, MYROW, *iarow*,
NPROW),
iroffc = mod(*icc*-1, *mb_c*), *icoffc* = mod(*jcc*-1,
nb_c),
icrow = indxg2p(*icc*, *mb_c*, MYROW, *rsrc_c*, NPROW),
iccol = indxg2p(*jcc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),
*mpc*0 = numroc(*mi*+*iroffc*, *mb_c*, MYROW, *icrow*,
NPROW),
*nqc*0 = numroc(*ni*+*icoffc*, *nb_c*, MYCOL, *iccol*,
NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — sub(*C*) is overwritten by *Q**sub(*C*), or *Q'**sub(*C*), or sub(*C*)**Q'*, or sub(*C*)**Q*.

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.

= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?unmhr

Multiplies a general matrix by the unitary transformation matrix from a reduction to Hessenberg form determined by p?gehrd.

Syntax

call pcunmhr(*side*, *trans*, *m*, *n*, *ilo*, *ihi*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pzunmhr(*side*, *trans*, *m*, *n*, *ilo*, *ihi*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Description

The routine overwrites the general complex distributed *m*-by-*n* matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'H':	Q^H*sub(*C*)	sub(*C*)*Q^H

where *Q* is a complex unitary distributed matrix of order *nq*, with *nq* = *m* if *side* = 'L' and *nq* = *n* if *side* = 'R'.

Q is defined as the product of *ihi*-*ilo* elementary reflectors, as returned by p?gehrd.

Q = *H*(*ilo*) *H*(*ilo*+1)... *H*(*ihi*-1).

Input Parameters

side (global) CHARACTER
='L': *Q* or Q^H is applied from the left.
='R': *Q* or Q^H is applied from the right.

trans (global) CHARACTER

	='N', no transpose, Q is applied. ='C', conjugate transpose, Q^H is applied.
m	(global) INTEGER. The number of rows in the distributed submatrix sub (C) ($m \geq 0$).
n	(global) INTEGER. The number of columns in the distributed submatrix sub (C) ($n \geq 0$).
ilo, *ihi*	(global) INTEGER These must be the same parameters *ilo* and *ihi*, respectively, as supplied to p?gehrd. Q is equal to the unit matrix except in the distributed submatrix Q (*ia*+*ilo*:*ia*+*ihi*-1,*ia*+*ilo*:*ja*+*ihi*-1). If *side* ='L', then $1 \leq ilo \leq ihi \leq \max(1,m)$. If *side* = 'R', then $1 \leq ilo \leq ihi \leq \max(1,n)$ *ilo* and *ihi* are relative indexes.
a	(local) COMPLEX for pcunmhr DOUBLE COMPLEX for pzunmhr. Pointer into the local memory to an array of dimension (*lld_a*, *LOC c*(*ja*+*m*-1)) if *side*='L', and (*lld_a*, *LOCc*(*ja*+*n*-1)) if *side* = 'R'. Contains the vectors which define the elementary reflectors, as returned by p?gehrd.
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix A, respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix A.
tau	(local) COMPLEX for pcunmhr DOUBLE COMPLEX for pzunmhr. Array, DIMENSION *LOCc*(*ja*+*m*-2), if *side* = 'L', and *LOCc*(*ja*+*n*-2) if *side* = 'R'. This array contains the scalar factors *tau*(j) of the elementary reflectors $H(j)$ as returned by p?gehrd. *tau* is tied to the distributed matrix A.

c (local)
COMPLEX for pcunmhr
DOUBLE COMPLEX for pzunmhr.
Pointer into the local memory to an array of dimension (*lld_c*, *LOCc*(*jc*+*n*-1)).
Contains the local pieces of the distributed matrix sub(*C*).

ic, *jc* (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)
COMPLEX for pcunmhr
DOUBLE COMPLEX for pzunmhr.
Workspace array with dimension *lwork*.

lwork (local or global)
The dimension of the array *work*.
lwork must be at least *iaa* = *ia* + *ilo*; *jaa* = *ja*+*ilo*-1;
If *side* = 'L', *mi* = *ihi*-*ilo*; *ni* = *n*; *icc* = *ic* + *ilo*;
jcc = *jc*; *lwork* ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0+*mpc*0)**nb_a*) + *nb_a***nb_a*
else if *side* = 'R',
mi = *m*; *ni* = *ihi*-*ilo*; *icc* = *ic*; *jcc* = *jc* + *ilo*;
lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + max(*npa*0+numroc(numroc(*ni*+*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a***nb_a*
end if
where *lcmq* = *lcm*/NPCOL with *lcm* = ilcm(NPROW, NPCOL),
iroffa = mod(*iaa*-1, *mb_a*),
icoffa = mod(*jaa*-1, *nb_a*),
iarow = indxg2p(*iaa*, *mb_a*, MYROW, *rsrc_a*, NPROW),
*npa*0 = numroc(*ni*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),
iroffc = mod(*icc*-1, *mb_c*),

icoffc = mod(*jcc*-1, *nb_c*),
icrow = indxg2p(*icc*, *mb_c*, MYROW, *rsrc_c*, NPROW),
iccol = indxg2p(*jcc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),
*mpc*0 = numroc(*mi*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),
*nqc*0 = numroc(*ni*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — *C* is overwritten by *Q** sub(*C*) or *Q*'*sub(*C*) or sub(*C*)**Q*' or sub(*C*)**Q*.

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?lahqr

Computes the Schur decomposition and/or eigenvalues of a matrix already in Hessenberg form.

Syntax

```
call pslahqr(wantt, wantz, n, ilo, ihi, a, desca, wr, wi, iloz, ihiz, z, descz, work, lwork, iwork, ilwork, info)
```

```
call pdlahqr(wantt, wantz, n, ilo, ihi, a, desca, wr, wi, iloz, ihiz, z, descz, work, lwork, iwork, ilwork, info)
```

Description

This is an auxiliary routine used to find the Schur decomposition and/or eigenvalues of a matrix already in Hessenberg form from columns *ilo* to *ihi*.

Input Parameters

wantt — (global) LOGICAL
If *wantt* = .TRUE., the full Schur form T is required;
If *wantt* = .FALSE., only eigenvalues are required.

wantz — (global) LOGICAL.
If *wantz* = .TRUE., the matrix of Schur vectors *z* is required;
If *wantz* = .FALSE., Schur vectors are not required.

n — (global) INTEGER. The order of the Hessenberg matrix *A* (and *z* if *wantz*). (*n*≥0).

ilo, *ihi* — (global) INTEGER.
It is assumed that *A* is already upper quasi-triangular in rows and columns *ihi*+1:*n*, and that *A*(*ilo*, *ilo*–1) = 0 (unless *ilo* = 1). p?lahqr works primarily with the Hessenberg submatrix in rows and columns *ilo* to *ihi*, but applies transformations to all of *h* if *wantt* is .TRUE..
1≤*ilo*≤max(1,*ihi*); *ihi* ≤ *n*.

a — (global)
REAL for pslahqr

DOUBLE PRECISION for pdlahqr
Array, DIMENSION (*desca*(*lld_*),*). On entry, the upper Hessenberg matrix *A*.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

iloz, *ihiz* (global) INTEGER. Specify the rows of *z* to which transformations must be applied if *wantz* is .TRUE..
1≤*iloz*≤*ilo*; *ihi*≤*ihiz*≤*n*.

z (global) REAL for pslahqr
DOUBLE PRECISION for pdlahqr
Array. If *wantz* is .TRUE., on entry *z* must contain the current matrix *Z* of transformations accumulated by pdhseqr. If *wantz* is .FALSE., *z* is not referenced.

descz (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix Z.

work (local)
REAL for pslahqr
DOUBLE PRECISION for pdlahqr
Workspace array with dimension *lwork*.

lwork (local) INTEGER. The dimension of *work*. *lwork* is assumed big enough so that *lwork*≥3**n* + max(2*max(*descz*(*lld_*),*desca*(*lld_*)) + 2**LOCq*(*n*), 7*ceil(*n*/*hbl*)/*lcm*(NPROW,NPCOL))).
If *lwork* = −1, then *work*(1) gets set to the above number and the code returns immediately.

iwork (global and local) INTEGER array of size *ilwork*.

ilwork (local) INTEGER This holds some of the *iblk* integer arrays.

Output Parameters

a On exit, if *wantt* is .TRUE., *A* is upper quasi-triangular in rows and columns *ilo*:*ihi*, with any 2-by-2 or larger diagonal blocks not yet in standard form. If *wantt* is .FALSE., the contents of *A* are unspecified on exit.

work(1) On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

wr, *wi*	(global replicated output) REAL for pslahqr DOUBLE PRECISION for pdlahqr Arrays, DIMENSION(*n*) each. The real and imaginary parts, respectively, of the computed eigenvalues *ilo* to *ihi* are stored in the corresponding elements of *wr* and *wi*. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of *wr* and *wi*, say the *i*-th and (*i*+1)-th, with *wi*(*i*)> 0 and *wi*(*i*+1) < 0. If *wantt* is .TRUE. , the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *A*. *A* may be returned with larger diagonal blocks until the next release.
z	On exit *z* has been updated; transformations are applied only to the submatrix *z*(*iloz*:*ihiz*, *ilo*:*ihi*).
info	(global) INTEGER. = 0: the execution is successful. < 0: parameter number -*info* incorrect or inconsistent > 0: p?lahqr failed to compute all the eigenvalues *ilo* to *ihi* in a total of 30*(*ihi*-*ilo*+1) iterations; if *info* = *i*, elements *i*+1:*ihi* of *wr* and *wi* contain those eigenvalues which have been successfully computed.

Singular Value Decomposition

This section describes ScaLAPACK routines for computing the singular value decomposition (SVD) of a general *m*-by-*n* matrix *A* (see "Singular Value Decomposition" in LAPACK chapter).

To find the SVD of a general matrix *A*, this matrix is first reduced to a bidiagonal matrix *B* by a unitary (orthogonal) transformation, and then SVD of the bidiagonal matrix is computed. Note that the SVD of *B* is computed using the LAPACK routine ?bdsqr .

Table 6-6 lists ScaLAPACK computational routines for performing this decomposition.

Table 6-6 Computational Routines for Singular Value Decomposition (SVD)

Operation	General matrix	Orthogonal/unitary matrix
Reduce *A* to a bidiagonal matrix	p?gebrd	
Multiply matrix after reduction		p?ormbr/p?unmbr

p?gebrd

Reduces a general matrix to bidiagonal form.

Syntax

```
call psgebrd(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pdgebrd(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pcgebrd(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pzgebrd(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
```

Description

The routine reduces a real/complex general *m*-by-*n* distributed matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1) to upper or lower bidiagonal form *B* by an orthogonal/unitary transformation:

Q'*sub(*A*)**P* = *B*.

If *m*≥ *n*, *B* is upper bidiagonal; if *m* < *n*, *B* is lower bidiagonal.

Input Parameters

m	(global) INTEGER. The number of rows in the distributed matrix sub(*A*) (*m*≥0).
n	(global) INTEGER. The number of columns in the distributed matrix sub(*A*) (*n*≥0).
a	(local) REAL for psgebrd DOUBLE PRECISION for pdgebrd COMPLEX for pcgebrd DOUBLE COMPLEX for pzgebrd. Real pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, this array contains the distributed matrix sub (*A*).
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

work (local)
REAL for psgebrd
DOUBLE PRECISION for pdgebrd
COMPLEX for pcgebrd
DOUBLE COMPLEX for pzgebrd. Workspace array of dimension *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:

lwork ≥ *nb**(*mpa*0 + nqa0+1)+ *nqa*0
where *nb* = *mb_a* = *nb_a*,
iroffa = mod(*ia*-1, *nb*),
icoffa = mod(*ja*-1, nb),
iarow = indxg2p(*ia*, *nb*, MYROW, *rsrc_a*, NPROW),
iacol = indxg2p (*ja*, nb, MYCOL, *csrc_a*, NPCOL),
*mpa*0 = numroc(*m* +*iroffa*, nb, MYROW, iarow, NPROW),
*nqa*0 = numroc(*n* +*icoffa*, nb, MYCOL, *iacol*, NPCOL),
indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit, if *m*≥*n*, the diagonal and the first superdiagonal of sub(*A*) are overwritten with the upper bidiagonal matrix *B*; the elements below the diagonal, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors, and the elements above the first superdiagonal, with the array *taup*, represent the orthogonal matrix *P* as a product of elementary reflectors. If *m* < *n*,

the diagonal and the first subdiagonal are overwritten with the lower bidiagonal matrix *B*; the elements below the first subdiagonal, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors, and the elements above the diagonal, with the array *taup*, represent the orthogonal matrix *P* as a product of elementary reflectors. (See Application Notes below)

d (local)
REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors. Array,
DIMENSION *LOCc*(*ja*+min(*m*,*n*)−1) if *m*≥*n*;
LOCr(*ia*+min(*m*,*n*)−1) otherwise. The distributed diagonal elements of the bidiagonal matrix *B*: *d*(*i*) = *a*(*i*,*i*). *d* is tied to the distributed matrix *A*.

e (local)
REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors. Array,
DIMENSION *LOCr*(*ia*+min(*m*,*n*)−1) if *m*≥*n*;
LOCc(*ja*+min(*m*,*n*)−2) otherwise. The distributed off-diagonal elements of the bidiagonal distributed matrix *B*:
If *m*≥*n*, *e*(*i*) = *a*(*i*,*i*+1) for *i* = 1,2,..., *n*−1; if *m* < *n*, *e*(*i*) = *a*(*i*+1, *i*) for *i* = 1,2,...,*m*−1. *e* is tied to the distributed matrix *A*.

tauq, *taup* (local)
REAL for psgebrd
DOUBLE PRECISION for pdgebrd
COMPLEX for pcgebrd
DOUBLE COMPLEX for pzgebrd.
Arrays, DIMENSION *LOCc*(*ja*+min(*m*,*n*)−1) for *tauq* and *LOCr*(*ia*+min(*m*,*n*)−1) for *taup*. Contain the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrices *Q* and *P*, respectively. *tauq* and *taup* are tied to the distributed matrix *A*. (See Application Notes below)

work(1)	On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = – (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrices *Q* and *P* are represented as products of elementary reflectors:

If *m* ≥ *n*,

Q = *H*(1) *H*(2)... *H*(*n*) and *P* = *G*(1) *G*(2)... *G*(*n*-1).

Each *H*(*i*) and *G*(*i*) has the form:

H(*i*)= *i* – *tauq* * *v* * *v*′ and *G*(*i*) = *i* – *taup* * *u* * *u*′

where *tauq* and *taup* are real/complex scalars, and *v* and *u* are real/complex vectors;

v(1:*i*-1) = 0, *v*(*i*) = 1, and *v*(*i*+1:*m*) is stored on exit in *A*(*ia*+*i*:*ia*+*m*-1,*ja*+*i*-1);

u(1:*i*) = 0, *u*(*i*+1) = 1, and *u*(*i*+2:*n*) is stored on exit in *A* (*ia*+*i*-1,*ja*+*i*+1:*ja*+*n*-1);

tauq is stored in *tauq*(*ja*+*i*-1) and *taup* in *taup*(*ia*+*i*-1).

If *m* < *n*,

Q = *H*(1) *H*(2)... *H* (*m-1) and P* = *G (1) G(2)... G* (*m*)

Each *H* (*i*) and *G*(*i*) has the form:

H(*i*)= *i* - *tauq* * *v* * *v*′ and *G*(*i*)= *i* - *taup* * *u* * *u*′

here *tauq* and *taup* are real/complex scalars, and *v* and *u* are real/complex vectors;

v(1:*i*) = 0, *v*(*i*+1) = 1, and *v*(*i*+2:*m*) is stored on exit in *A* (*ia*+*i*:*ia*+*m*-1,*ja*+*i*-1);
u(1:*i*-1) = 0, *u*(*i*) = 1, and *u*(*i*+1:*n*) is stored on exit in *A*(*ia*+*i*-1,*ja*+*i*+1:*ja*+*n*-1);

tauq is stored in *tauq*(*ja*+*i*-1) and *taup* in *taup*(*ia*+*i*-1).

The contents of sub(*A*) on exit are illustrated by the following examples:

m = 6 and *n* = 5 (*m* > *n*):

$$\begin{bmatrix} d & e & u1 & u1 & u1 \\ v1 & d & e & u2 & u2 \\ v1 & v2 & d & e & u3 \\ v1 & v2 & v3 & d & e \\ v1 & v2 & v3 & v4 & d \\ v1 & v2 & v3 & v4 & v5 \end{bmatrix}$$

$m = 5$ and $n = 6$ ($m < n$):

$$\begin{bmatrix} d & u1 & u1 & u1 & u1 & u1 \\ e & d & u2 & u2 & u2 & u2 \\ v1 & e & d & u3 & u3 & u3 \\ v1 & v2 & e & d & u4 & u4 \\ v1 & v2 & v3 & e & d & u5 \end{bmatrix}$$

where d and e denote diagonal and off-diagonal elements of B, vi denotes an element of the vector defining $H(i)$, and ui an element of the vector defining $G(i)$.

p?ormbr

Multiplies a general matrix by one of the orthogonal matrices from a reduction to bidiagonal form determined by p?gebrd.

Syntax

```
call psormbr(vect, side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)
```

```
call pdormbr(vect, side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)
```

Description

If *vect* = '*Q*', the routine overwrites the general real distributed *m*-by-*n* matrix sub(*C*) = *C*(*c*:*ic*+*m*-1,*jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	*Q* sub(*C*)	sub(*C*) *Q*
trans = 'T':	Q^T sub(*C*)	sub(*C*) Q^T

If *vect* = 'P', the routine overwrites sub(*C*) with

	side ='L'	*side* ='R'
trans = 'N':	*P* sub(*C*)	sub(*C*) *P*
trans = 'T':	P^T sub(*C*)	sub(*C*) P^T

Here *Q* and P^T are the orthogonal distributed matrices determined by p?gebrd when reducing a real distributed matrix *A*(*ia*:*, *ja*:*) to bidiagonal form: *A*(*ia*:*,*ja*:*) = *Q***B***P*^{T}. *Q* and P^T are defined as products of elementary reflectors *H*(*i*) and *G*(*i*) respectively.

Let *nq* = *m* if *side* = 'L' and *nq* = *n* if *side* = 'R'. Thus *nq* is the order of the orthogonal matrix *Q* or P^T that is applied.

If *vect* = '*Q*', *A*(*ia*:*, *ja*:*) is assumed to have been an *nq*-by-*k* matrix:

If *nq* ≥ *k*, *Q* = *H*(1) *H*(2)...*H*(*k*);

If *nq* < *k*, *Q* = *H*(1) *H*(2)...*H*(*nq*-1).

If *vect* = '*P*', *A*(*ia*:*, *ja*:*) is assumed to have been a *k*-by-*nq* matrix:

If $k < nq$, $P = G(1)\ G(2)...G(k)$;

If $k \geq nq$, $P = G(1)\ G(2)...G(nq-1)$.

Input Parameters

vect	(global) CHARACTER. If *vect* ='Q', then Q or Q^T is applied. If *vect* ='P', then P or P^T is applied.
side	(global) CHARACTER. If *side* ='L', then Q or Q^T, P or P^T is applied from the left. If *side* ='R', then Q or Q^T, P or P^T is applied from the right.
trans	(global) CHARACTER. If *trans* = 'N', no transpose, Q or P is applied. If *trans* = 'T', then Q^T or P^T is applied.
m	(global) INTEGER. The number of rows in the distributed matrix sub (C).
n	(global) INTEGER. The number of columns in the distributed matrix sub (C).
k	(global) INTEGER. If *vect* = 'Q', the number of columns in the original distributed matrix reduced by p?gebrd; If *vect* = 'P', the number of rows in the original distributed matrix reduced by p?gebrd. Constraints: $k \geq 0$.
a	(local) REAL for psormbr DOUBLE PRECISION for pdormbr. Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+min(*nq*,*k*)-1)) If *vect*='Q', and (*lld_a*, *LOCc*(*ja*+*nq*-1)) If *vect* = 'P'. *nq* = *m* if *side* = 'L', and *nq* = *n* otherwise. The vectors which define the elementary reflectors $H(i)$ and $G(i)$, whose products determine the matrices Q and P, as returned by p?gebrd. If *vect* = 'Q', *lld_a*≥max(1, *LOCr*(*ia*+*nq*-1));

If *vect* = 'P', *lld_a*≥max(1, *LOCr*(*ia*+min(*nq*, *k*)-1)).

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)
REAL for psormbr
DOUBLE PRECISION for pdormbr.
Array, DIMENSION *LOCc*(*ja*+min(*nq*, *k*)-1), if *vect* = 'Q', and *LOCr*(*ia*+min(*nq*, *k*)-1), if *vect* = 'P'.
tau(*i*) must contain the scalar factor of the elementary reflector *H*(*i*) or *G*(*i*), which determines *Q* or *P*, as returned by pdgebrd in its array argument *tauq* or *taup*. *tau* is tied to the distributed matrix *A*.

c (local) REAL for psormbr
DOUBLE PRECISION for pdormbr
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc* (*jc*+*n*-1)).
Contains the local pieces of the distributed matrix sub (*C*).

ic, *jc* (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descc (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)
REAL for psormbr
DOUBLE PRECISION for pdormbr.
Workspace array of dimension *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:
If *side* = 'L'
nq = *m*;
if ((*vect* = 'Q' and *nq*≥*k*) or (*vect* is not equal to 'Q' and *nq*>*k*)), *iaa*=*ia*; *jaa*=*ja*; *mi*=*m*; *ni*=*n*; *icc*=*ic*; *jcc*=*jc*;

else
iaa= *ia*+1; *jaa*=*ja*; *mi*=*m*-1; *ni*=*n*; *icc*=*ic*+1; *jcc*= *jc*;
end if
else
If *side* = 'R', *nq* = *n*;
if((*vect* = 'Q' and *nq*≥*k*) or (*vect* is not equal to 'Q' and *nq*>*k*)),
iaa=*ia*; *jaa*=*ja*; *mi*=*m*; *ni*=*n*; *icc*=*ic*; *jcc*=*jc*;
else
iaa= *ia*; *jaa*= *ja*+1; *mi*= *m*; *ni*= *n*-1; *icc*= *ic*; *jcc*= *jc*+1;
end if
end if
If *vect* = 'Q',
If *side* = 'L', *lwork*≥max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + *mpc*0)**nb_a*) + *nb_a* * *nb_a*
else if *side* = 'R',
lwork≥max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + max(*npa*0 + numroc(numroc(*ni*+*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a***nb_a*
end if
else if *vect* is not equal to 'Q', if *side* = 'L',
lwork≥max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + max(*mqa*0 + numroc(numroc(*mi*+iroffc, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0))**mb_a*) + *mb_a***mb_a*
else if *side* = 'R',
lwork≥max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + *nqc*0)**mb_a*) + *mb_a***mb_a*
end if
end if
where *lcmp* = *lcm*/NPROW, *lcmq* = *lcm*/NPCOL, with
lcm = ilcm(NPROW, NPCOL),
iroffa = mod(*iaa*-1, *mb_a*),
icoffa = mod(*jaa*-1, *nb_a*),
iarow = indxg2p(*iaa*, *mb_a*, MYROW, *rsrc_a*, NPROW),
iacol = indxg2p(*jaa*, *nb_a*, MYCOL, *csrc_a*, NPCOL),

*mqa*0 = numroc(*mi*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),
*npa*0 = numroc(*ni*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),
iroffc = mod(*icc*-1, *mb_c*),
icoffc = mod(*jcc*-1, *nb_c*),
icrow = indxg2p(*icc*, *mb_c*, MYROW, *rsrc_c*, NPROW),
iccol = indxg2p(*jcc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),
*mpc*0 = numroc(*mi*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),
*nqc*0 = numroc(*ni*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),
indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c
On exit, if *vect*='Q', sub(*C*) is overwritten by *Q**sub(*C*), or *Q*'*sub(*C*), or sub(*C*)**Q*', or sub(*C*)**Q*; if *vect*='P', sub(*C*) is overwritten by *P**sub(*C*), or *P*'*sub(*C*), or sub(*C*)**P*, or sub(*C*)**P*'.

work(1)
On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info
(global) INTEGER.
= 0: the execution is successful.
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?unmbr

Multiplies a general matrix by one of the unitary transformation matrices from a reduction to bidiagonal form determined by p?gebrd.

Syntax

call pcunmbr(*vect*, *side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pzunmbr(*vect*, *side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Description

If *vect* = 'Q', the routine overwrites the general complex distributed *m*-by-*n* matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1) with

	side ='L'	*side* ='R'
trans = 'N':	*Q**sub(*C*)	sub(*C*)**Q*
trans = 'C':	Q^H*sub(*C*)	sub(*C*)*Q^H

If *vect* = 'P', the routine overwrites sub(*C*) with

	side ='L'	*side* ='R'
trans = 'N':	*P**sub(*C*)	sub(*C*)**P*
trans = 'C':	P^H*sub(*C*)	sub(*C*)*P^H

Here *Q* and P^H are the unitary distributed matrices determined by p?gebrd when reducing a complex distributed matrix *A*(*ia*:*, *ja*:*) to bidiagonal form: *A*(*ia*:*,*ja*:*) = *Q***B**P^H.

Q and P^H are defined as products of elementary reflectors *H*(*i*) and *G*(*i*) respectively.

Let *nq* = *m* if *side* = 'L' and *nq* = *n* if *side* = 'R'. Thus *nq* is the order of the unitary matrix *Q* or P^H that is applied.

If *vect* = *'Q'*, *A*(*ia*:*, *ja*:*) is assumed to have been an *nq*-by-*k* matrix:

If *nq* ≥ *k*, *Q* = *H*(1) *H*(2)... *H*(*k*);

If *nq* < *k*, *Q* = *H*(1) *H*(2)... *H*(*nq*-1).

If *vect* = *'P'*, *A*(*ia*:*, *ja*:*) is assumed to have been a *k*-by-*nq* matrix:

If $k < nq$, $P = G(1)\ G(2) \ldots\ G(k)$;

If $k \geq nq$, $P = G(1)\ G(2) \ldots\ G(nq-1)$.

Input Parameters

vect (global) CHARACTER.
If *vect* ='Q', then Q or Q^H is applied.
If *vect* ='P', then P or P^H is applied.

side (global) CHARACTER.
If *side* ='L', then Q or Q^H, P or P^H is applied from the left.
If *side* ='R', then Q or Q^H, P or P^H is applied from the right.

trans (global) CHARACTER.
If *trans* = 'N', no transpose, Q or P is applied.
If *trans* = 'C', conjugate transpose, Q^H or P^H is applied.

m (global) INTEGER. The number of rows in the distributed matrix sub (C) $m \geq 0$.

n (global) INTEGER. The number of columns in the distributed matrix sub (C) $n \geq 0$.

k (global) INTEGER.
If *vect* = 'Q', the number of columns in the original distributed matrix reduced by p?gebrd;
If *vect* = 'P', the number of rows in the original distributed matrix reduced by p?gebrd.
Constraints: $k \geq 0$.

a (local)
COMPLEX for psormbr
DOUBLE COMPLEX for pdormbr.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+min(*nq*,*k*)-1)) if *vect*='Q', and (*lld_a*, *LOCc*(*ja*+*nq*-1)) if *vect* = 'P'.
nq = *m* if *side* = 'L', and *nq* = *n* otherwise.
The vectors which define the elementary reflectors $H(i)$ and $G(i)$, whose products determine the matrices Q and P, as returned by p?gebrd.
If *vect* = 'Q', *lld_a* ≥ max(1, *LOCr*(*ia*+*nq*-1));

	If *vect* = 'P', *lld_a* ≥ max(1, *LOCr*(*ia*+min(*nq*, *k*)-1)).
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
tau	(local) COMPLEX for pcunmbr DOUBLE COMPLEX for pzunmbr. Array, DIMENSION *LOCc*(*ja*+min(*nq*, *k*)-1), if *vect* = 'Q', and *LOCr*(*ia*+min(*nq*, *k*)-1), if *vect* = 'P'. *tau*(*i*) must contain the scalar factor of the elementary reflector *H*(*i*) or *G*(*i*), which determines *Q* or *P*, as returned by p?gebrd in its array argument *tauq* or *taup*. *tau* is tied to the distributed matrix *A*.
c	(local) COMPLEX for pcunmbr DOUBLE COMPLEX for pzunmbr Pointer into the local memory to an array of dimension (*lld_a*, *LOCc* (*jc*+*n*-1)). Contains the local pieces of the distributed matrix sub (*C*).
ic, *jc*	(global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.
descc	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.
work	(local) COMPLEX for pcunmbr DOUBLE COMPLEX for pzunmbr. Workspace array of dimension *lwork*.
lwork	(local or global) INTEGER, dimension of *work*, must be at least: If *side* = 'L' *nq* = *m*;

if ((*vect* = 'Q' and *nq* ≥ *k*) or (*vect* is not equal to 'Q' and *nq* > *k*)), *iaa*= *ia*; *jaa*= *ja*; *mi*= *m*; *ni*= *n*; *icc*= *ic*; *jcc*= *jc*;
else
iaa= *ia*+1; *jaa*= *ja*; *mi*= *m*-1; *ni*= *n*; *icc*= *ic*+1; *jcc*= *jc*;
end if
else
If *side* = 'R', *nq* = *n*;
if ((*vect* = 'Q' and *nq* ≥ *k*) or (*vect* is not equal to 'Q' and *nq* ≥ *k*)),
iaa= *ia*; *jaa*= *ja*; *mi*= *m*; *ni*= *n*; *icc*= *ic*; *jcc*= *jc*;
else
iaa= *ia*; *jaa*= *ja*+1; *mi*= *m*; *ni*= *n*-1; *icc*= *ic*; *jcc*= *jc*+1;
end if
end if
If *vect* = 'Q',
If *side* = 'L', *lwork* ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0+*mpc*0)**nb_a*) + *nb_a***nb_a*
else if *side* = 'R',
lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc*0 + max(*npa*0+numroc(numroc(*ni*+*icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc*0))**nb_a*) + *nb_a***nb_a*
end if
else if *vect* is not equal to 'Q',
if *side* = 'L',
lwork ≥ max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + max(*mqa*0+numroc(numroc(*mi*+iroffc, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nqc*0))**mb_a*) + *mb_a***mb_a*
else if *side* = 'R',
lwork ≥ max((*mb_a**(*mb_a*-1))/2, (*mpc*0 + *nqc*0)**mb_a*) + *mb_a***mb_a*
end if

end if
where *lcmp* = *lcm*/NPROW, *lcmq* = *lcm*/NPCOL, with *lcm* = ilcm(NPROW, NPCOL),
iroffa = mod(*iaa*-1, *mb_a*),
icoffa = mod(*jaa*-1, *nb_a*),
iarow = indxg2p(*iaa*, *mb_a*, MYROW, *rsrc_a*, NPROW),
iacol = indxg2p(*jaa*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
*mqa*0 = numroc(*mi*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),
*npa*0 = numroc(*ni*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),
iroffc = mod(*icc*-1, *mb_c*),
icoffc = mod(*jcc*-1, *nb_c*),
icrow = indxg2p(*icc*, *mb_c*, MYROW, *rsrc_c*, NPROW),
iccol = indxg2p(*jcc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),
*mpc*0 = numroc(*mi*+*iroffc*, *mb_c*, MYROW, *icrow*, NPROW),
*nqc*0 = numroc(*ni*+*icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),
indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — On exit, if *vect*='Q', sub(*C*) is overwritten by *Q**sub(*C*), or *Q*'*sub(*C*), or sub(*C*)**Q*', or sub(*C*)**Q*; if *vect*='P', sub(*C*) is overwritten by *P**sub(*C*), or *P*'*sub(*C*), or sub(*C*)**P*, or sub(*C*)**P*'.

work(1) — On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info — (global) INTEGER.
= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Generalized Symmetric-Definite Eigen Problems

This section describes ScaLAPACK routines that allow you to reduce the generalized symmetric-definite eigenvalue problems (see Generalized Symmetric-Definite Eigenvalue Problems in LAPACK chapters) to standard symmetric eigenvalue problem $Cy = \lambda y$, which you can solve by calling ScaLAPACK routines described earlier in this chapter (see Symmetric Eigenproblems).

Table 6-7 lists these routines.

Table 6-7 Computational Routines for Reducing Generalized Eigenproblems to Standard Problems

Operation	Real symmetric matrices	Complex Hermitian matrices
Reduce to standard problems	p?sygst	p?hegst

p?sygst

Reduces a real symmetric-definite generalized eigenvalue problem to the standard form.

Syntax

```
call pssygst( ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale, info )
```

```
call pdsygst( ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale, info )
```

Description

This routine reduces real symmetric-definite generalized eigenproblems to the standard form.

In the following sub(*A*) denotes *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) and sub(*B*) denotes *B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*n*-1).

If *ibtype* = 1, the problem is

sub(*A*)**x* = λ*sub(*B*)**x*,

and sub(*A*) is overwritten by inv(U^T)*sub(*A*)*inv(*U*), or inv(*L*)*sub(*A*)*inv(L^T).

If *ibtype* = 2 or 3, the problem is

sub(*A*)*sub(*B*)**x* = λ**x*, or sub(*B*)*sub(*A*)**x* = λ**x*,

and sub(*A*) is overwritten by *U**sub(*A*)*U^T, or L^T*sub(*A*)**L*.

sub(*B*) must have been previously factorized as U^T**U* or *L**L^T by p?potrf.

Input Parameters

ibtype	(global) INTEGER. Must be 1 or 2 or 3. If *itype* = 1, compute inv(U^T)*sub(*A*)*inv(*U*), or inv(*L*)*sub(*A*)*inv(L^T); If *itype* = 2 or 3, compute *U**sub(*A*)*U^T, or L^T*sub(*A*)**L*.
uplo	(global) CHARACTER. Must be 'U' or 'L'. If *uplo* = 'U', the upper triangle of sub(*A*) is stored and sub (*B*) is factored as U^T**U*. If *uplo* = 'L', the lower triangle of sub(*A*) is stored and sub (*B*) is factored as *L**L^T.
n	(global) INTEGER. The order of the matrices sub (*A*) and sub (*B*) (*n* ≥ 0).
a	(local) REAL for pssygst DOUBLE PRECISION for pdsygst. Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, the array contains the local pieces of the *n*-by-*n* symmetric distributed matrix sub(*A*). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

b (local)
REAL for pssygst
DOUBLE PRECISION for pdsygst.
Pointer into the local memory to an array of dimension (*lld_b*, *LOCc*(*jb*+*n*-1)). On entry, the array contains the local pieces of the triangular factor from the Cholesky factorization of sub (*B*) as returned by p?potrf.

ib, *jb* (global) INTEGER. The row and column indices in the global array *b* indicating the first row and the first column of the submatrix *B*, respectively.

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix B.

Output Parameters

a On exit, if *info* = 0, the transformed matrix, stored in the same format as sub(*A*).

scale (global)
REAL for pssygst
DOUBLE PRECISION for pdsygst.
Amount by which the eigenvalues should be scaled to compensate for the scaling performed in this routine. At present, *scale* is always returned as 1.0, it is returned here to allow for future enhancement.

info (global) INTEGER.
If *info* = 0, the execution is successful. If *info* < 0, if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?hegst

Reduces a Hermitian-definite generalized eigenvalue problem to the standard form.

Syntax

```
call pchegst( ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale, info)
```

```
call pzhegst( ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale, info)
```

Description

This routine reduces complex Hermitian-definite generalized eigenproblems to the standard form.

In the following sub(*A*) denotes *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) and sub(*B*) denotes *B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*n*-1).

If *ibtype* = 1, the problem is

sub(*A*)**x* = λ*sub(*B*)**x*,

and sub(*A*) is overwritten by inv(U^H)*sub(*A*)*inv(*U*), or inv(*L*)*sub(*A*)*inv(L^H).

If *ibtype* = 2 or 3, the problem is

sub(*A*)*sub(*B*)**x* = λ**x*, or sub(*B*)*sub(*A*)**x* = λ**x*,

and sub(*A*) is overwritten by *U**sub(*A*)*U^H, or L^H*sub(*A*)**L*.

sub(*B*) must have been previously factorized as U^H**U* or *L**L^H by p?potrf.

Input Parameters

ibtype	(global) INTEGER. Must be 1 or 2 or 3. If *itype* = 1, compute inv(U^H)*sub(*A*)*inv(*U*), or inv(*L*)*sub(*A*)*inv(L^H); If *itype* = 2 or 3, compute *U**sub(*A*)*U^H, or L^H*sub(*A*)**L*.
uplo	(global) CHARACTER. Must be 'U' or 'L'. If *uplo* = 'U', the upper triangle of sub(*A*) is stored and sub (*B*) is factored as U^H**U*.

If *uplo* = 'L', the lower triangle of sub(*A*) is stored and sub (*B*) is factored as $L*L^H$.

n (global) INTEGER. The order of the matrices sub (*A*) and sub (*B*) (*n*≥0).

a (local)
COMPLEX for pchegst
DOUBLE COMPLEX for pzhegst.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, the array contains the local pieces of the *n*-by-*n* Hermitian distributed matrix sub(*A*). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

b (local)
COMPLEX for pchegst
DOUBLE COMPLEX for pzhegst.
Pointer into the local memory to an array of dimension (*lld_b*, *LOCc*(*jb*+*n*-1)). On entry, the array contains the local pieces of the triangular factor from the Cholesky factorization of sub (*B*) as returned by p?potrf.

ib, *jb* (global) INTEGER. The row and column indices in the global array *b* indicating the first row and the first column of the submatrix *B*, respectively.

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix B.

Output Parameters

a	On exit, if *info* = 0, the transformed matrix, stored in the same format as sub(*A*).
scale	(global) REAL for pchegst DOUBLE PRECISION for pzhegst. Amount by which the eigenvalues should be scaled to compensate for the scaling performed in this routine. At present, *scale* is always returned as 1.0, it is returned here to allow for future enhancement.
info	(global) INTEGER. If *info* = 0, the execution is successful. If *info* <0, if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i*100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Driver Routines

Table 6-8 lists ScaLAPACK driver routines available for solving systems of linear equations, linear least-squares problems, standard eigenvalue and singular value problems, and generalized symmetric definite eigenproblems.

Table 6-8 ScaLAPACK Driver Routines

Type of Problem	Matrix type, storage scheme	Driver
Linear equations	general (partial pivoting)	p?gesv (simple driver)p?gesvx (expert driver)
	general band (partial pivoting)	p?gbsv (simple driver)
	general band (no pivoting)	p?dbsv (simple driver)
	general tridiagonal (no pivoting)	p?dtsv (simple driver)
	symmetric/Hermitian positive-definite	p?posv (simple driver)p?posvx (expert driver)
	symmetric/Hermitian positive-definite, band	p?pbsv (simple driver)
	symmetric/Hermitian positive-definite, tridiagonal	p?ptsv (simple driver)
Linear least squares problem	general *m*-by-*n*	p?gels
Symmetric eigenvalue problem	symmetric/Hermitian	p?syev (simple driver) p?syevx / p?heevx (expert driver)

Type of Problem	Matrix type, storage scheme	Driver
Singular value decomposition	general *m*-by-*n*	p?gesvd
Generalized symmetric definite eigenvalue problem	symmetric/Hermitian, one matrix also positive-definite	p?sygvx / p?hegvx (expert driver)

p?gesv

Computes the solution to the system of linear equations with a square distributed matrix and multiple right-hand sides.

Syntax

```
call psgesv(n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pdgesv(n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pcgesv(n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pzgesv(n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
```

Description

The routine p?gesv computes the solution to a real or complex system of linear equations sub(*A*)***X* = sub(*B*), where sub(*A*) = *A*(*ia:ia+n-1*, *ja:ja+n-1*) is an *n*-by-*n* distributed matrix and *X* and sub(*B*) = *B*(*ib:ib+n-1*, *jb:jb+nrhs-1*) are *n*-by-*nrhs* distributed matrices.

The *LU* decomposition with partial pivoting and row interchanges is used to factor sub(*A*) as sub(*A*) = *P*L*U*, where *P* is a permutation matrix, *L* is unit lower triangular, and *U* is upper triangular. *L* and *U* are stored in sub(*A*). The factored form of sub(*A*) is then used to solve the system of equations sub(*A*)***X* = sub(*B*).

Input Parameters

n — (global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub(*A*) (*n* ≥ 0).

nrhs — (global) INTEGER. The number of right hand sides, that is, the number of columns of the distributed submatrices *B* and *X* (*nrhs* ≥ 0).

a, *b* — (local)
REAL for psgesv

	DOUBLE PRECISION for pdgesv COMPLEX for pcgesv DOUBLE COMPLEX for pzgesv. Pointers into the local memory to arrays of local dimension *a*(*lld_a*,*LOCc*(*ja*+*n*-1)) and *b*(*lld_b*,*LOCc*(*jb*+*nrhs*-*1*)), respectively. On entry, the array *a* contains the local pieces of the *n*-by-*n* distributed matrix sub(*A*) to be factored. On entry, the array *b* contains the right hand side distributed matrix sub(*B*).
ia, *ja*	(global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of sub(*A*), respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
ib, *jb*	(global) INTEGER. The row and column indices in the global array *B* indicating the first row and the first column of sub(*B*), respectively.
descb	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

Output Parameters

a	Overwritten by the factors *L* and *U* from the factorization sub(*A*) = *P***L***U*; the unit diagonal elements of *L* are not stored .
b	Overwritten by the solution distributed matrix *X*.
ipiv	(local) INTEGER array. The dimension of *ipiv* is (*LOCr*(*m_a*)+*mb_a*). This array contains the pivoting information. The (local) row i of the matrix was interchanged with the (global) row *ipiv*(i). This array is tied to the distributed matrix *A*.
info	(global) INTEGER. If *info*=0, the execution is successful. *info* < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

info > 0:
If *info* = *k*, *U*(*ia+k-1*,*ja+k-1*) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution could not be computed.

p?gesvx

Uses the LU factorization to compute the solution to the system of linear equations with a square matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax

```
call psgesvx(fact, trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf,
ipiv, equed, r, c, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr,
work, lwork, iwork, liwork, info)
```

```
call pdgesvx(fact, trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf,
ipiv, equed, r, c, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr,
work, lwork, iwork, liwork, info)
```

```
call pcgesvx(fact, trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf,
ipiv, equed, r, c, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr,
work, lwork, rwork, lrwork, info)
```

```
call pzgesvx(fact, trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf,
ipiv, equed, r, c, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr,
work, lwork, rwork, lrwork, info)
```

Description

This routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations *AX = B*, where *A* denotes the *n*-by-*n* submatrix *A(ia:ia+n-1, ja:ja+n-1)*, *B* denotes the *n*-by-*nrhs* submatrix *B(ib:ib+n-1, jb:jb+nrhs-1)* and *X* denotes the *n*-by-*nrhs* submatrix *X(ix:ix+n-1, jx:jx+nrhs-1)*.

Error bounds on the solution and a condition estimate are also provided.

In the following description, *af* stands for the subarray *af*(*iaf:iaf+n-1*, *jaf:jaf+n-1*).

The routine p?gesvx performs the following steps:

1. If *fact* = 'E', real scaling factors *R* and *C* are computed to equilibrate the system:

   trans = 'N': diag(*R*)*A*diag(*C*) *diag(*C*)-1*X* = diag(*R*)*B

   trans = 'T': (diag(*R*)*A*diag(*C*))*T* *diag(*R*)-1*X* = diag(*C*)*B

   trans = 'C': (diag(*R*)*A*diag(*C*))*H* *diag(*R*)-1*X* = diag(*C*)*B

   Whether or not the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by diag(*R*)*A*diag(*C*) and *B* by diag(*R*)*B* (if *trans*='N') or diag(*c*)*B* (if *trans* = 'T' or 'C').
2. If *fact* = 'N' or 'E', the *LU* decomposition is used to factor the matrix *A* (after equilibration if *fact* = 'E') as *A* = *P* *L* *U*, where *P* is a permutation matrix, *L* is a unit lower triangular matrix, and *U* is upper triangular.
3. The factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than relative machine precision, steps 4 - 6 are skipped.
4. The system of equations is solved for *X* using the factored form of *A*.
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix *X* is premultiplied by diag(*C*) (if *trans* = 'N') or diag(*R*) (if *trans* = 'T' or 'C') so that it solves the original system before equilibration.

Input Parameters

fact — (global) CHARACTER*1. Must be 'F', 'N', or 'E'.
Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.
If *fact* = 'F' then, on entry, *af* and *ipiv* contain the factored form of *A*. If *equed* is not 'N', the matrix *A* has been equilibrated with scaling factors given by *r* and *c*. Arrays *a*, *af*, and *ipiv* are not modified.
If *fact* = 'N', the matrix *A* is copied to *af* and factored.
If *fact* = 'E', the matrix *A* is equilibrated if necessary, then copied to *af* and factored.

trans (global) CHARACTER*1. Must be 'N', 'T', or 'C'.
Specifies the form of the system of equations:
If *trans* = 'N', the system has the form *A*X* = *B* (No transpose);
If *trans* = 'T', the system has the form A^T**X* = *B* (Transpose);
If *trans* = 'C', the system has the form A^H**X* = *B* (Conjugate transpose);

n (global) INTEGER. The number of linear equations; the order of the submatrix *A* (*n* ≥ 0).

nrhs (global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrices *B* and *X* (*nrhs* ≥ 0).

a, *af*, *b*, *work* (local)
REAL for psgesvx
DOUBLE PRECISION for pdgesvx
COMPLEX for pcgesvx
DOUBLE COMPLEX for pzgesvx.
Pointers into the local memory to arrays of local dimension *a*(*lld_a*,*LOCc*(*ja*+*n*-1)), *af*(*lld_af*,*LOCc*(*ja*+*n*-1)), *b*(*lld_b*,*LOCc*(*jb*+*nrhs*-1)), *work*(*lwork*), respectively. The array *a* contains the matrix *A*. If *fact* = 'F' and *equed* is not 'N', then *A* must have been equilibrated by the scaling factors in *r* and/or *c*.
The array *af* is an input argument if *fact* = 'F'. In this case it contains on entry the factored form of the matrix *A*, that is, the factors *L* and *U* from the factorization *A* = *P***L***U* as computed by p?getrf. If *equed* is not 'N', then *af* is the factored form of the equilibrated matrix *A*.
The array *b* contains on entry the matrix *B* whose columns are the right-hand sides for the systems of equations.
work(*) is a workspace array. The dimension of *work* is (*lwork*).

ia, *ja* (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1), respectively.

desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
iaf, *jaf*	(global) INTEGER. The row and column indices in the global array *af* indicating the first row and the first column of the subarray *af*(*iaf:iaf+n-1*, *jaf:jaf+n-1*), respectively.
descaf	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *AF*.
ib, *jb*	(global) INTEGER. The row and column indices in the global array B indicating the first row and the first column of the submatrix *B*(*ib:ib+n-1*, *jb:jb+nrhs-1*), respectively.
descb	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.
ipiv	(local) INTEGER array. The dimension of *ipiv* is (*LOCr*(*m_a*)+*mb_a*). The array *ipiv* is an input argument if *fact* = 'F'. On entry, it contains the pivot indices from the factorization *A* = *P***L***U* as computed by p?getrf; (local) row i of the matrix was interchanged with the (global) row *ipiv*(i). This array must be aligned with *A*(*ia:ia+n-1*, ***).
equed	(global) CHARACTER*1. Must be 'N', 'R', 'C', or 'B'. *equed* is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done: If *equed* = 'N', no equilibration was done (always true if *fact* = 'N'); If *equed* = 'R', row equilibration was done, that is, *A* has been premultiplied by diag(*r*); If *equed* = 'C', column equilibration was done, that is, *A* has been postmultiplied by diag(*c*); If *equed* = 'B', both row and column equilibration was done; *A* has been replaced by diag(*r*)**A**diag(*c*).
r, *c*	(local) REAL for single precision flavors; DOUBLE PRECISION for double precision flavors. Arrays, dimension *LOCr*(m_a) and *LOCc*(n_a), respectively. The array *r* contains the row scale factors for *A*, and the array *c* contains the column scale factors for *A*. These arrays are input arguments if *fact* = 'F' only; otherwise they

are output arguments. If *equed* = 'R' or 'B', *A* is multiplied on the left by diag(*r*); if *equed* = 'N' or 'C', *r* is not accessed.
If *fact* = 'F' and *equed* = 'R' or 'B', each element of *r* must be positive.
If *equed* = 'C' or 'B', *A* is multiplied on the right by diag(*c*); if *equed* = 'N' or 'R', *c* is not accessed.
If *fact* = 'F' and *equed* = 'C' or 'B', each element of *c* must be positive. Array *r* is replicated in every process column, and is aligned with the distributed matrix *A*. Array *c* is replicated in every process row, and is aligned with the distributed matrix *A*.

ix, *jx* — (global) INTEGER. The row and column indices in the global array *X* indicating the first row and the first column of the submatrix *X*(*ix:ix+n-1*, *jx:jx+nrhs-1*), respectively.

descx — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *X*.

lwork — (local or global) INTEGER. The dimension of the array *work* ; must be at least max(p?gecon(*lwork*), p?gerfs(*lwork*))+*LOCr*(n_a) .

iwork — (local, psgesvx/pdgesvx only) INTEGER. Workspace array. The dimension of *iwork* is (*liwork*).

liwork — (local, psgesvx/pdgesvx only) INTEGER. The dimension of the array *iwork* , must be at least *LOCr*(n_a) .

rwork — (local) REAL for pcgesvx
DOUBLE PRECISION for pzgesvx.
Workspace array, used in complex flavors only.
The dimension of *rwork* is (*lrwork*).

lrwork — (local or global, pcgesvx/pzgesvx only) INTEGER. The dimension of the array *rwork*;must be at least 2**LOCc*(n_a) .

Output Parameters

x — (local)
REAL for psgesvx
DOUBLE PRECISION for pdgesvx

COMPLEX for pcgesvx
DOUBLE COMPLEX for pzgesvx.
Pointer into the local memory to an array of local dimension *x*(lld_x,*LOCc*(*jx+nrhs-1*)).
If *info* = 0, the array *x* contains the solution matrix *X* to the original system of equations. Note that *A* and *B* are modified on exit if *equed* ≠ 'N', and the solution to the equilibrated system is:
diag(*C*)-1**X*, if *trans* = 'N' and *equed* = 'C' or 'B'; and diag(*R*)-1**X*, if *trans* = 'T' or 'C' and *equed* = 'R' or 'B'.

a

Array *a* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'.
If *equed* ≠ 'N', *A* is scaled on exit as follows:
equed = 'R': *A* = diag(*R*)**A*
equed = 'C': *A* = *A**diag(*c*)
equed = 'B': *A* = diag(*R*)**A**diag(*c*)

af

If *fact* = 'N' or 'E', then *af* is an output argument and on exit returns the factors *L* and *U* from the factorization *A* = *P*L*U* of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *a* for the form of the equilibrated matrix.

b

Overwritten by diag(*R*)**B* if *trans* = 'N' and *equed* = 'R' or 'B';
overwritten by diag(*c*)**B* if *trans* = 'T' and *equed* = 'C' or 'B'; not changed if *equed* = 'N'.

r, *c*

These arrays are output arguments if *fact* ≠ 'F'.
See the description of *r*, *c* in Input Arguments section.

rcond

(global) REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). The routine sets *rcond* =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

ferr, *berr*
(local) REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION *LOCc*(n_b) each. Contain the component-wise forward and relative backward errors, respectively, for each solution vector.
Arrays *ferr* and *berr* are both replicated in every process row, and are aligned with the matrices *B* and *X*.

ipiv
If *fact* = 'N' or 'E', then *ipiv* is an output argument and on exit contains the pivot indices from the factorization *A* = *P*L*U* of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E').

equed
If *fact* ≠ 'F' , then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in Input Arguments section).

work(1)
If *info*=0, on exit *work*(1) returns the minimum value of *lwork* required for optimum performance.

iwork(1)
If *info*=0, on exit *iwork*(1) returns the minimum value of *liwork* required for optimum performance.

rwork(1)
If *info*=0, on exit *rwork*(1) returns the minimum value of *lrwork* required for optimum performance.

info
INTEGER. If *info*=0, the execution is successful.
info < 0: if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*. If *info* = *i*, and *i* ≤ *n*, then *U*(*i*,*i*) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed. If *info* = *i*, and *i* = *n* +1, then *U* is nonsingular, but *rcond* is less than machine precision. The factorization has been completed, but the matrix is singular to working precision and the solution and error bounds have not been computed.

p?gbsv

Computes the solution to the system of linear equations with a general banded distributed matrix and multiple right-hand sides.

Syntax

call psgbsv(*n*, *bwl*, *bwu*, *nrhs*, *a*, *ja*, *desca*, *ipiv*, *b*, *ib*, *descb*, *work*, *lwork*, *info*)

call pdgbsv(*n*, *bwl*, *bwu*, *nrhs*, *a*, *ja*, *desca*, *ipiv*, *b*, *ib*, *descb*, *work*, *lwork*, *info*)

call pcgbsv(*n*, *bwl*, *bwu*, *nrhs*, *a*, *ja*, *desca*, *ipiv*, *b*, *ib*, *descb*, *work*, *lwork*, *info*)

call pzgbsv(*n*, *bwl*, *bwu*, *nrhs*, *a*, *ja*, *desca*, *ipiv*, *b*, *ib*, *descb*, *work*, *lwork*, *info*)

Description

The routine p?gbsv computes the solution to a real or complex system of linear equations

sub(*A*)**X* = sub(*B*),

where sub(*A*) = *A*(*1:n*, *ja:ja+n-1*) is an *n*-by-*n* real/complex general banded distributed matrix with *bwl* subdiagonals and *bwu* superdiagonals, and *X* and sub(*B*)= *B*(*ib:ib+n-1*, *1:rhs*) are *n*-by-*nrhs* distributed matrices.

The *LU* decomposition with partial pivoting and row interchanges is used to factor sub(*A*) as sub(*A*) = *P***L***U***Q*, where *P* and *Q* are permutation matrices, and *L* and *U* are banded lower and upper triangular matrices, respectively. The matrix *Q* represents reordering of columns for the sake of parallelism, while *P* represents reordering of rows for numerical stability using classic partial pivoting.

Input Parameters

n	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub(*A*) (*n* ≥ 0).
bwl	(global) INTEGER. The number of subdiagonals within the band of *A* (0≤ *bwl* ≤ *n-1*).

bwu (global) INTEGER. The number of superdiagonals within the band of *A* (0≤ *bwu* ≤ *n-1*).

nrhs (global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix sub(*B*)

(*nrhs* ≥ 0).

a, *b* (local)
REAL for psgbsv
DOUBLE PRECISON for pdgbsv
COMPLEX for pcgbsv
DOUBLE COMPLEX for pzgbsv.
Pointers into the local memory to arrays of local dimension *a*(*lld_a*,LOC_c(*ja*+*n*-1)) and *b*(*lld_b*,LOC_c(*nrhs*)), respectively.
On entry, the array *a* contains the local pieces of the global array *A*.
On entry, the array *b* contains the right hand side distributed matrix sub(*B*).

ja (global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

If *desca*(*dtype_*) = 501, then *dlen_* ≥ 7;

else if *desca*(*dtype_*) = 1, then *dlen_* ≥ 9.

ib (global) INTEGER. The row index in the global array *B* that points to the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

If *descb*(*dtype_*) = 502, then *dlen_* ≥ 7;

else if *descb*(*dtype_*) = 1, then *dlen_* ≥ 9.

work (local)
REAL for psgbsv
DOUBLE PRECISON for pdgbsv
COMPLEX for pcgbsv

DOUBLE COMPLEX for pzgbsv.
Workspace array of dimension (*lwork*).

lwork (local or global) INTEGER. The size of the array *work*, must be at least *lwork* ≥ (NB+*bwu*)*(*bwl*+*bwu*)+6*(*bwl*+*bwu*)*(*bwl*+2*bwu*) + + max(*nrhs* *(NB+2*bwl*+4*bwu*), 1).

Output Parameters

a On exit, contains details of the factorization. Note that the resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

b On exit, this array contains the local pieces of the solution distributed matrix *X*.

ipiv (local) INTEGER array.
The dimension of *ipiv* must be at least *desca*(NB). This array contains pivot indices for local factorizations. You should not alter the contents between factorization and solve.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info INTEGER. If *info*=0, the execution is successful. *info* < 0:
If the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.
info > 0:

If *info* = k ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not nonsingular, and the factorization was not completed. If *info* = k > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

p?dbsv

Solves a general band system of linear equations.

Syntax

```
call psdbsv(n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pddbsv(n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pcdbsv(n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pzdbsv(n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
```

Description

This routine solves the system of linear equations

`A(1:n, ja:ja+n-1)* X = B(ib:ib+n-1, 1:nrhs),`

where `A(1:n, ja:ja+n-1)` is an `n`-by-`n` real/complex banded diagonally dominant-like distributed matrix with bandwidth `bwl`, `bwu`.

Gaussian elimination without pivoting is used to factor a reordering of the matrix into `LU`.

Input Parameters

`n`	(global) `INTEGER`. The order of the distributed submatrix `A`, `(n ≥ 0)`.
`bwl`	(global) `INTEGER`. Number of subdiagonals. `0 ≤ bwl ≤ n-1`.
`bwu`	(global) `INTEGER`. Number of subdiagonals. `0 ≤ bwu ≤ n-1`.
`nrhs`	(global) `INTEGER`. The number of right-hand sides; the number of columns of the distributed submatrix `B`, `(nrhs ≥ 0)`.

`a`	(local). `REAL` for `psdbsv`
	`DOUBLE PRECISION` for `pddbsv`
	`COMPLEX` for `pcdbsv`
	`DOUBLE COMPLEX` for `pzdbsv`.

	Pointer into the local memory to an array with first dimension *lld_a* ≥ (*bwl*+*bwu*+1) (stored in *desca*). On entry, this array contains the local pieces of the distributed matrix.
ja	(global) INTEGER. The index in the global array *a* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).
desca	(global and local) INTEGER array of dimension *dlen*. If 1*d* type (*dtype_a*=501 or 502), *dlen* ≥ 7; If 2*d* type (*dtype_a*=1), *dlen* ≥ 9. The array descriptor for the distributed matrix *A*. Contains information of mapping of *A* to memory.
b	(local) REAL for psdbsv DOUBLE PRECISON for pddbsv COMPLEX for pcdbsv DOUBLE COMPLEX for pzdbsv. Pointer into the local memory to an array of local lead dimension *lld_b* ≥ *nb*. On entry, this array contains the local pieces of the right hand sides *B*(*ib*:*ib*+*n*–1, 1:*nrhs*).
ib	(global) INTEGER. The row index in the global array *b* that points to the first row of the matrix to be operated on (which may be either all of *b* or a submatrix of *B*).
descb	(global and local) INTEGER array of dimension *dlen*. If 1*d* type (*dtype_b* =502), *dlen* ≥ 7; If 2*d* type (*dtype_b* =1), *dlen* ≥ 9. The array descriptor for the distributed matrix *B*. Contains information of mapping of *B* to memory.
work	(local). REAL for psdbsv DOUBLE PRECISON for pddbsv COMPLEX for pcdbsv DOUBLE COMPLEX for pzdbsv.

Temporary workspace. This space may be overwritten in between calls to routines. *work* must be the size given in *lwork*.

lwork (local or global) INTEGER. Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

lwork ≥

nb(*bwl*+*bwu*)+6max(*bwl*,*bwu*)*max(*bwl*,*bwu*)+max((max(*bwl*,*bwu*)*nrhs*), max(*bwl*,*bwu*)*max(*bwl*,*bwu*))

Output Parameters

a On exit, this array contains information containing details of the factorization.
Note that permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.

b On exit, this contains the local piece of the solutions distributed matrix *X*.

work On exit, *work*(1) contains the minimal *lwork*.

info (local) INTEGER. If *info*=0, the execution is successful.
< 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
> 0: If *info* = *k* < NPROCS, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed.
If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

p?dtsv

Solves a general tridiagonal system of linear equations.

Syntax

```
call psdtsv(n, nrhs, dl, d, du, ja, desca, b, ib, descb, work, lwork, info)
call pddtsv(n, nrhs, dl, d, du, ja, desca, b, ib, descb, work, lwork, info)
call pcdtsv(n, nrhs, dl, d, du, ja, desca, b, ib, descb, work, lwork, info)
call pzdtsv(n, nrhs, dl, d, du, ja, desca, b, ib, descb, work, lwork, info)
```

Description

This routine solves a system of linear equations

`A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs),`

where `A(1:n, ja:ja+n-1)` is an `n`-by-`n` complex tridiagonal diagonally dominant-like distributed matrix.

Gaussian elimination without pivoting is used to factor a reordering of the matrix into `L U`.

Input Parameters

n	(global) INTEGER. The order of the distributed submatrix `A` (`n` ≥ 0).
nrhs	INTEGER. The number of right hand sides; the number of columns of the distributed matrix `B` (`nrhs` ≥ 0).
dl	(local). REAL for psdtsv DOUBLE PRECISION for pddtsv COMPLEX for pcdtsv DOUBLE COMPLEX for pzdtsv. Pointer to local part of global vector storing the lower diagonal of the matrix. Globally, `dl`(1)is not referenced, and `dl` must be aligned with `d`. Must be of size > `desca`(`nb_`).
d	(local). REAL for psdtsv DOUBLE PRECISION for pddtsv COMPLEX for pcdtsv

	DOUBLE COMPLEX for pzdtsv. Pointer to local part of global vector storing the main diagonal of the matrix.
du	(local). REAL for psdtsv DOUBLE PRECISION for pddtsv COMPLEX for pcdtsv DOUBLE COMPLEX for pzdtsv. Pointer to local part of global vector storing the upper diagonal of the matrix. Globally, *du*(*n*) is not referenced, and *du* must be aligned with *d*.
ja	(global) INTEGER. The index in the global array *a* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).
desca	(global and local) INTEGER array of dimension *dlen*. If 1*d* type (*dtype_a*=501 or 502), *dlen* ≥ 7; If 2*d* type (*dtype_a*=1), *dlen* ≥ 9. The array descriptor for the distributed matrix *A*. Contains information of mapping of *A* to memory.
b	(local) REAL for psdtsv DOUBLE PRECISONfor pddtsv COMPLEX for pcdtsv DOUBLE COMPLEX for pzdtsv. Pointer into the local memory to an array of local lead dimension *lld_b* > *nb*. On entry, this array contains the local pieces of the right hand sides *B*(*ib*:*ib*+*n*-1, 1:*nrhs*).
ib	(global) INTEGER. The row index in the global array *b* that points to the first row of the matrix to be operated on (which may be either all of *b* or a submatrix of *B*).
descb	(global and local) INTEGER array of dimension *dlen*. If 1*d* type (*dtype_b* =502), *dlen* ≥ 7; If 2*d* type (*dtype_b* =1), *dlen* ≥ 9. The array descriptor for the distributed matrix *B*. Contains information of mapping of *B* to memory.
work	(local).

REAL for psdtsv
DOUBLE PRECISON for pddtsv
COMPLEX for pcdtsv
DOUBLE COMPLEX for pzdtsv. Temporary workspace. This space may be overwritten in between calls to routines. *work* must be the size given in *lwork*.

lwork (local or global) INTEGER. Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned. *lwork* > (12*NPCOL+3**nb*)+max((10+2*min(100, *nrhs*))*NPCOL+4**nrhs*, 8*NPCOL)

Output Parameters

dl On exit, this array contains information containing the * factors of the matrix.

d On exit, this array contains information containing the * factors of the matrix. Must be of size > *desca*(*nb_*).

du On exit, this array contains information containing the * factors of the matrix. Must be of size > *desca*(*nb_*).

b On exit, this contains the local piece of the solutions distributed matrix *X*.

work On exit, *work*(1) contains the minimal *lwork*.

info (local) INTEGER. If *info*=0, the execution is successful.
< 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
> 0: If *info* = *k* < NPROCS, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed.
If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

p?posv

Solves a symmetric positive definite system of linear equations.

Syntax

```
call psposv(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pdposv(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pcposv(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pzposv(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
```

Description

This routine computes the solution to a real/complex system of linear equations

sub(*A*)**X* = sub(*B*),

where sub(*A*) denotes *A*(*ia*:*ia*+*n*-1,*ja*:*ja*+*n*-1) and is an *n*-by-*n* symmetric/Hermitian distributed positive definite matrix and *X* and sub(*B*) denoting *B*(*ib*:*ib*+*n*-1,*jb*:*jb*+*nrhs*-1) are *n*-by-*nrhs* distributed matrices. The Cholesky decomposition is used to factor sub(*A*) as

sub(*A*) = U^T**U*, if *uplo* = 'U', or

sub(*A*) = *L**L^T, if *uplo* = 'L',

where *U* is an upper triangular matrix and *L* is a lower triangular matrix. The factored form of sub(*A*) is then used to solve the system of equations.

Input Parameters

uplo	(global). CHARACTER. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of sub(*A*) is stored.
n	(global) INTEGER. The order of the distributed submatrix sub(*A*) (*n* ≥ 0).
nrhs	INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix sub(*B*) (*nrhs* ≥ 0).
a	(local) REAL for psposv

	DOUBLE PRECISION for pdposv COMPLEX for pcposv COMPLEX*16 for pzposv. Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*–1)). On entry, this array contains the local pieces of the *n*-by-*n* symmetric distributed matrix sub(*A*) to be factored. If *uplo* = '*U*', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If *uplo* = '*L*', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced.
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
b	(local) REAL for psposv DOUBLE PRECISON for pdposv COMPLEX for pcposv COMPLEX*16 for pzposv. Pointer into the local memory to an array of dimension (*lld_b*,*LOC*(*jb*+*nrhs*–1)). On entry, the local pieces of the right hand sides distributed matrix sub(*B*).
ib, *jb*	(global) INTEGER. The row and column indices in the global array *b* indicating the first row and the first column of the submatrix *B*, respectively.
descb	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

Output Parameters

a	On exit, if *info* = 0, this array contains the local pieces of the factor *U* or *L* from the Cholesky factorization sub(*A*) = U^H*U, or $L*L^H$.
b	On exit, if *info* = 0, sub(*B*) is overwritten by the solution distributed matrix *X*.
info	(global) INTEGER. If *info* =0, the execution is successful. If *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*. If *info* > 0: If *info* = *k*, the leading minor of order *k*, *A*(*ia*:*ia*+*k*-1, *ja*:*ja*+*k*-1) is not positive definite, and the factorization could not be completed, and the solution has not been computed.

p?posvx

Solves a symmetric or Hermitian positive definite system of linear equations.

Syntax

call psposvx(*fact*, *uplo*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *af*, *iaf*, *jaf*, *descaf*, *equed*, *sr*, *sc*, *b*, *ib*, *jb*, *descb*, *x*, *ix*, *jx*, *descx*, *rcond*, *ferr*, *berr*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call pdposvx(*fact*, *uplo*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *af*, *iaf*, *jaf*, *descaf*, *equed*, *sr*, *sc*, *b*, *ib*, *jb*, *descb*, *x*, *ix*, *jx*, *descx*, *rcond*, *ferr*, *berr*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call pcposvx(*fact*, *uplo*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *af*, *iaf*, *jaf*, *descaf*, *equed*, *sr*, *sc*, *b*, *ib*, *jb*, *descb*, *x*, *ix*, *jx*, *descx*, *rcond*, *ferr*, *berr*, *work*, *lwork*, *iwork*, *liwork*, *info*)

call pzposvx(*fact*, *uplo*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *af*, *iaf*, *jaf*, *descaf*, *equed*, *sr*, *sc*, *b*, *ib*, *jb*, *descb*, *x*, *ix*, *jx*, *descx*, *rcond*, *ferr*, *berr*, *work*, *lwork*, *iwork*, *liwork*, *info*)

Description

This routine uses the Cholesky factorization $A=U^T*U$ or $A=L*L^T$ to compute the solution to a real or complex system of linear equations

$A(ia:ia+n-1, ja:ja+n-1)*X = B(ib:ib+n-1, jb:jb+nrhs-1)$,

where $A(ia:ia+n-1, ja:ja+n-1)$ is a n-by-n matrix and X and $B(ib:ib+n-1,jb:jb+nrhs-1)$ are n-by-$nrhs$ matrices.

Error bounds on the solution and a condition estimate are also provided.

In the following comments y denotes $Y(iy:iy+m-1, jy:jy+k-1)$ a m-by-k matrix where y can be a, af, b and x.

The routine p?posvx performs the following steps:

1. If $fact = 'E'$, real scaling factors s are computed to equilibrate the system:

   $diag(sr)*A*diag(sc)*inv(diag(sc))*X = diag(sr)*B$

   Whether or not the system will be equilibrated depends on the scaling of the matrix A, but if equilibration is used, A is overwritten by $diag(sr)*A*diag(sc)$ and B by $diag(sr)*B$.
2. If $fact = 'N'$ or $'E'$, the Cholesky decomposition is used to factor the matrix A (after equilibration if $fact = 'E'$) as

   $A = U^T*U$, if $uplo = 'U'$, or

   $A = L*L^T$, if $uplo = 'L'$,

   where U is an upper triangular matrix and L is a lower triangular matrix.
3. The factored form of A is used to estimate the condition number of the matrix A. If the reciprocal of the condition number is less than machine precision, steps 4-6 are skipped
4. The system of equations is solved for X using the factored form of A.
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix X is premultiplied by diag(sr) so that it solves the original system before equilibration.

Input Parameters

fact	(global) CHARACTER. Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.
If *fact* = 'F': on entry, *af* contains the factored form of *A*. If *equed* = 'Y', the matrix *A* has been equilibrated with scaling factors given by *s*. *a* and *af* will not be modified.
If *fact* = 'N', the matrix *A* will be copied to *af* and factored.
If *fact* = 'E', the matrix *A* will be equilibrated if necessary, then copied to *af* and factored.

uplo (global) CHARACTER. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of *A* is stored.

n (global) INTEGER. The order of the distributed submatrix sub(*A*) (*n* ≥ 0).

nrhs (global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrices *B* and *X*. (*nrhs* ≥ 0).

a (local)
REAL for psposvx
DOUBLE PRECISION for pdposvx
COMPLEX for pcposvx
DOUBLE COMPLEX for pzposvx.
Pointer into the local memory to an array of local dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, the symmetric/Hermitian matrix *A*, except if *fact* = 'F' and *equed* = 'Y', then *A* must contain the equilibrated matrix diag(*sr*)*A*diag(*sc*).
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *A* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *A* is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *A* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *A* is not referenced. *A* is not modified if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N' on exit.

ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
af	(local) REAL for psposvx DOUBLE PRECISION for pdposvx COMPLEX for pcposvx DOUBLE COMPLEX for pzposvx. Pointer into the local memory to an array of local dimension (*lld_af*, *LOCc*(*ja*+*n*-1)). If *fact* = 'F', then *af* is an input argument and on entry contains the triangular factor *U* or *L* from the Cholesky factorization $A = U^T*U$ or $A = L*L^T$, in the same storage format as *A*. If *equed* ≠ 'N', then *af* is the factored form of the equilibrated matrix diag(*sr*)*A*diag(*sc*).
iaf, *jaf*	(global) INTEGER. The row and column indices in the global array *af* indicating the first row and the first column of the submatrix *AF*, respectively.
descaf	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *AF*.
equed	(global). CHARACTER. Must be 'N' or 'Y'. *equed* is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done: If *equed* = 'N', no equilibration was done (always true if *fact* = 'N'); If *equed* = 'Y', equilibration was done and *A* has been replaced by diag(*sr*)*A*diag(*sc*).
sr	(local) REAL for psposvx DOUBLE PRECISION for pdposvx COMPLEX for pcposvx DOUBLE COMPLEX for pzposvx. Array, DIMENSION (*lld_a*).

The array *s* contains the scale factors for *A*. This array is an input argument if *fact* = 'F' only; otherwise it is an output argument.
If *equed* = 'N', *s* is not accessed.
If *fact* = 'F' and *equed* = 'Y', each element of *s* must be positive.

b (local)
REAL for psposvx
DOUBLE PRECISION for pdposvx
COMPLEX for pcposvx
DOUBLE COMPLEX for pzposvx.
Pointer into the local memory to an array of local dimension (*lld_b*, *LOCc*(*jb*+*nrhs*-1)). On entry, the *n*-by-*nrhs* right-hand side matrix *B*.

ib, *jb* (global) INTEGER. The row and column indices in the global array *b* indicating the first row and the first column of the submatrix *B*, respectively.

descb (global and local) INTEGER. Array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

x (local)
REAL for psposvx
DOUBLE PRECISION for pdposvx
COMPLEX for pcposvx
DOUBLE COMPLEX for pzposvx.
Pointer into the local memory to an array of local dimension (*lld_x*, *LOCc*(*jx*+*nrhs*-1)).

ix, *jx* (global) INTEGER. The row and column indices in the global array *x* indicating the first row and the first column of the submatrix *X*, respectively.

descx (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *X*.

work (local)
REAL for psposvx
DOUBLE PRECISION for pdposvx
COMPLEX for pcposvx
DOUBLE COMPLEX for pzposvx.
Workspace array, DIMENSION (*lwork*).

lwork (local or global) INTEGER.
The dimension of the array *work*. *lwork* is local input and must be at least *lwork* = max(p?pocon(*lwork*), p?porfs(*lwork*)) + *LOCr*(*n_a*).
lwork = 3**desca*(*lld_*).
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

iwork (local) INTEGER. Workspace array, dimension (*liwork*).

liwork (local or global)
INTEGER. The dimension of the array *iwork*. *liwork* is local input and must be at least *liwork* = *desca*(*lld_*) *liwork* = *LOCr*(*n_a*).
If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit, if *fact* = 'E' and *equed* = 'Y', *a* is overwritten by diag(*sr*)**a**diag(*sc*).

af If *fact* = 'N', then *af* is an output argument and on exit returns the triangular factor *U* or *L* from the Cholesky factorization $A = U^T*U$ or $A = L*L^T$ of the original matrix *A*.
If *fact* = 'E', then *af* is an output argument and on exit returns the triangular factor *U* or *L* from the Cholesky factorization $A = U^T*U$ or $A = L*L^T$ of the equilibrated matrix *A* (see the description of *A* for the form of the equilibrated matrix).

equed If *fact* ≠ 'F' , then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in Input Arguments section).

sr
This array is an output argument if *fact* ≠ 'F'.
See the description of *sr* in Input Arguments section.

sc
This array is an output argument if *fact* ≠ 'F'.
See the description of *sc* in Input Arguments section.

b
On exit, if *equed* = 'N', *b* is not modified; if *trans* = 'N' and *equed* = 'R' or 'B', *b* is overwritten by diag(*r*)*b; if *trans* = 'T' or 'C' and *equed* = 'C' or 'B', *b* is overwritten by diag(*c*)*b.

x
(local)
REAL for psposvx
DOUBLE PRECISION for pdposvx
COMPLEX for pcposvx
DOUBLE COMPLEX for pzposvx.
If *info* = 0 the *n*-by-*nrhs* solution matrix *X* to the original system of equations.
Note that *A* and *B* are modified on exit if *equed* ≠ 'N', and the solution to the equilibrated system is
inv(diag(*sc*))*X if *trans* = 'N' and *equed* = 'C' or 'B', or
inv(diag(*sr*))*X if *trans* = 'T' or 'C' and *equed* = 'R' or 'B'.

rcond
(global)
REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision (in particular, if *rcond*=0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr
REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
Arrays, DIMENSION at least max(*LOC*, *n_b*). The estimated forward error bounds for each solution vector *X*(*j*) (the *j*-th column of the solution matrix *X*). If *xtrue* is the true solution, *ferr*(*j*) bounds the magnitude of the largest entry in (*X*(*j*) – *xtrue*) divided by the magnitude of the largest

	entry in *X*(*j*). The quality of the error bound depends on the quality of the estimate of `norm(inv(A))` computed in the code; if the estimate of `norm(inv(A))` is accurate, the error bound is guaranteed.
berr	(local) `REAL` for single precision flavors. `DOUBLE PRECISION` for double precision flavors. Arrays, `DIMENSION` at least `max(LOC, n_b)`. The componentwise relative backward error of each solution vector *X*(*j*) (the smallest relative change in any entry of *A* or *B* that makes *X*(*j*) an exact solution).
work(1)	(local) On exit, *work*(1) returns the minimal and optimal *liwork*.
info	(global) `INTEGER`. If *info*=0, the execution is successful. < 0: if *info* = −*i*, the *i*-th argument had an illegal value > 0: if *info* = *i*, and *i* is ≤ *n*: if *info* = *i*, the leading minor of order *i* of *a* is not positive definite, so the factorization could not be completed, and the solution and error bounds could not be computed. = *n*+1: *rcond* is less than machine precision. The factorization has been completed, but the matrix is singular to working precision, and the solution and error bounds have not been computed.

p?pbsv

Solves a symmetric/Hermitian positive definite banded system of linear equations.

Syntax

```
call pspbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pdpbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pcpbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pzpbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
```

Description

This routine solves a system of linear equations

`A(1:n, ja:ja+n-1)*X = B(ib:ib+n-1, 1:nrhs),`

where `A(1:n, ja:ja+n-1)` is an `n`-by-`n` real/complex banded symmetric positive definite distributed matrix with bandwidth `bw`.

Cholesky factorization is used to factor a reordering of the matrix into `L*L'`.

Input Parameters

uplo	(global) CHARACTER. Must be 'U' or 'L'. Indicates whether the upper or lower triangular of *A* is stored. If *uplo* = 'U', the upper triangular *A* is stored If *uplo* = 'L', the lower triangular of *A* is stored.
n	(global) INTEGER. The order of the distributed matrix *A* (*n* ≥ 0).
bw	(global) INTEGER. The number of subdiagonals in *L* or *U*. 0 ≤ *bw* ≤ *n*-1.
nrhs	(global) INTEGER. The number of right-hand sides; the number of columns in *B* (*nrhs* ≥ 0).
a	(local). REAL for pspbsv DOUBLE PRECISON for pdpbsv COMPLEX for pcpbsv DOUBLE COMPLEX for pzpbsv. Pointer into the local memory to an array with first dimension *lld_a* ≥ (*bw*+1) (stored in *desca*). On entry, this array contains the local pieces of the distributed matrix sub(*A*) to be factored.
ja	(global) INTEGER. The index in the global array *a* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
b	(local)

	REAL for pspbsv DOUBLE PRECISON for pdpbsv COMPLEX for pcpbsv DOUBLE COMPLEX for pzpbsv. Pointer into the local memory to an array of local lead dimension *lld_b* ≥ *nb*. On entry, this array contains the local pieces of the right hand sides *B*(*ib*:*ib*+*n*-1, 1:*nrhs*).
ib	(global) INTEGER. The row index in the global array *b* that points to the first row of the matrix to be operated on (which may be either all of *b* or a submatrix of *B*).
descb	(global and local) INTEGER array of dimension *dlen*. If 1D type (*dtype_b* =502), *dlen* ≥ 7; If 2D type (*dtype_b* =1), *dlen* ≥ 9. The array descriptor for the distributed matrix *B*. Contains information of mapping of *B* to memory.
work	(local). REAL for pspbsv DOUBLE PRECISON for pdpbsv COMPLEX for pcpbsv DOUBLE COMPLEX for pzpbsv. Temporary workspace. This space may be overwritten in between calls to routines. *work* must be the size given in *lwork*.
lwork	(local or global) INTEGER. Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned. *lwork* ≥ (*nb*+2**bw*)**bw* +max((*bw***nrhs*), *bw***bw*)

Output Parameters

a	On exit, this array contains information containing details of the factorization. Note that permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.
b	On exit, contains the local piece of the solutions distributed matrix *X*.

work — On exit, *work*(1) contains the minimal *lwork*.

info — (global). INTEGER. If *info*=0, the execution is successful.
< 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
> 0: If *info* = *k* ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed.
If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

p?ptsv

Solves a symmetric or Hermitian positive definite tridiagonal system of linear equations.

Syntax

```
call psptsv(n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
call pdptsv(n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
call pcptsv(n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
call pzptsv(n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
```

Description

This routine solves a system of linear equations

A(1:*n*, *ja*:*ja*+*n*-1)**X* = *B*(*ib*:*ib*+*n*-1, 1:*nrhs*),

where *A*(1:*n*, *ja*:*ja*+*n*-1) is an *n*-by-*n* real tridiagonal symmetric positive definite distributed matrix.

Cholesky factorization is used to factor a reordering of the matrix into *L***L*'.

Input Parameters

n — (global) INTEGER. The order of matrix *A* (*n* ≥ 0).

nrhs	(global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix *B* (*nrhs* ≥ 0).
d	(local) REAL for psptsv DOUBLE PRECISON for pdptsv COMPLEX for pcptsv DOUBLE COMPLEX for pzptsv. Pointer to local part of global vector storing the main diagonal of the matrix.
e	(local) REAL for psptsv DOUBLE PRECISON for pdptsv COMPLEX for pcptsv DOUBLE COMPLEX for pzptsv. Pointer to local part of global vector storing the upper diagonal of the matrix. Globally, *du*(n) is not referenced, and *du* must be aligned with *d*.
ja	(global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).
desca	(global and local) INTEGER array of dimension *dlen*. If 1*d* type (*dtype_a*=501 or 502), *dlen* ≥ 7; If 2*d* type (*dtype_a*=1), *dlen* ≥ 9. The array descriptor for the distributed matrix *A*. Contains information of mapping of *A* to memory.
b	(local) REAL for psptsv DOUBLE PRECISON for pdptsv COMPLEX for pcptsv DOUBLE COMPLEX for pzptsv. Pointer into the local memory to an array of local lead dimension *lld_b* ≥ *nb*. On entry, this array contains the local pieces of the right hand sides *B*(*ib*:*ib*+*n*-1, 1:*nrhs*).

ib (global) INTEGER. The row index in the global array *b* that points to the first row of the matrix to be operated on (which may be either all of *b* or a submatrix of *B*).

descb (global and local) INTEGER array of dimension *dlen*.

If 1*d* type (*dtype_b* = 502), *dlen* ≥ 7;

If 2*d* type (*dtype_b* = 1), *dlen* ≥ 9.

The array descriptor for the distributed matrix *B*. Contains information of mapping of *B* to memory.

work (local).
REAL for psptsv
DOUBLE PRECISON for pdptsv
COMPLEX for pcptsv
DOUBLE COMPLEX for pzptsv.
Temporary workspace. This space may be overwritten in between calls to routines. *work* must be the size given in *lwork*.

lwork (local or global) INTEGER. Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned. *lwork* > (12*NPCOL+3**nb*)+max((10+2*min(100, *nrhs*))*NPCOL+4**nrhs*, 8*NPCOL).

Output Parameters

d On exit, this array contains information containing the factors of the matrix. Must be of size greater than or equal to *desca*(*nb_*).

e On exit, this array contains information containing the factors of the matrix. Must be of size greater than or equal to *desca*(*nb_*).

b On exit, this contains the local piece of the solutions distributed matrix *X*.

work On exit, *work*(1) contains the minimal *lwork*.

info (local) INTEGER. If *info*=0, the execution is successful.

< 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

> 0: If *info* = *k* ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed.
If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

p?gels

Solves overdetermined or underdetermined linear systems involving a matrix of full rank.

Syntax

call psgels(*trans*, *m*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *work*, *lwork*, *info*)

call pdgels(*trans*, *m*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *work*, *lwork*, *info*)

call pcgels(*trans*, *m*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *work*, *lwork*, *info*)

call pzgels(*trans*, *m*, *n*, *nrhs*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *work*, *lwork*, *info*)

Description

This routine solves overdetermined or underdetermined real/ complex linear systems involving an *m*-by-*n* matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1), or its transpose/ conjugate-transpose, using a *QTQ* or *LQ* factorization of sub(*A*). It is assumed that sub(*A*) has full rank.

The following options are provided:

1. If *trans* = 'N' and *m* ≥ *n*: find the least squares solution of an overdetermined system, that is, solve the least squares problem

   minimize ||sub(*B*) - sub(*A*)**X*||

2. If *trans* = 'N' and *m* < *n*: find the minimum norm solution of an underdetermined system sub(*A*)**X* = sub(*B*).
3. If *trans* = 'T' and *m* ≥ *n*: find the minimum norm solution of an undetermined system sub(*A*)^T**X* = sub(*B*).
4. If *trans* = 'T' and *m* < *n*: find the least squares solution of an overdetermined system, that is, solve the least squares problem

   minimize ||sub(*B*) - sub(*A*)^T**X*||,

   where sub(*B*) denotes *B*(*ib*:*ib*+*m*-1, *jb*:*jb*+*nrhs*-1) when *trans* = 'N' and *B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*nrhs*-1) otherwise. Several right hand side vectors *b* and solution vectors *x* can be handled in a single call; when *trans* = 'N', the solution vectors are stored as the columns of the *n*-by-*nrhs* right hand side matrix sub(*B*) and the *m*-by-*nrhs* right hand side matrix sub(*B*) otherwise.

Input Parameters

trans	(global) CHARACTER. Must be 'N', or 'T'. If *trans* = 'N', the linear system involves matrix sub(*A*); If *trans* = 'T', the linear system involves the transposed matrix A^T (for real flavors only).
m	(global) INTEGER. The number of rows in the distributed submatrix sub (*A*) (*m* ≥ 0).
n	(global) INTEGER. The number of columns in the distributed submatrix sub (*A*) (*n* ≥ 0).
nrhs	(global) INTEGER. The number of right-hand sides; the number of columns in the distributed submatrices sub(*B*) and *X*. (*nrhs* ≥ 0).
a	(local) REAL for psgels DOUBLE PRECISION for pdgels COMPLEX for pcgels DOUBLE COMPLEX for pzgels. Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, contains the *m*-by-*n* matrix *A*.

ia, *ja*
(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca
(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

b
(local)
REAL for psgels
DOUBLE PRECISION for pdgels
COMPLEX for pcgels
DOUBLE COMPLEX for pzgels.
Pointer into the local memory to an array of local dimension (*lld_b*, *LOCc*(*jb*+*nrhs*-1)). On entry, this array contains the local pieces of the distributed matrix *B* of right-hand side vectors, stored columnwise; sub(*B*) is *m*-by-*nrhs* if *trans*='N', and *n*-by-*nrhs* otherwise.

ib, *jb*
(global) INTEGER. The row and column indices in the global array *b* indicating the first row and the first column of the submatrix *B*, respectively.

descb
(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

work
(local)
REAL for psgels
DOUBLE PRECISION for pdgels
COMPLEX for pcgels
DOUBLE COMPLEX for pzgels.
Workspace array with dimension *lwork*.

lwork
(local or global) INTEGER.
The dimension of the array *work* *lwork* is local input and must be at least *lwork* ≥ *ltau* + max(*lwf*, *lws*), where if *m* > *n*, then
ltau = numroc(*ja*+min(*m*,*n*)-1, *nb_a*, MYCOL, *csrc_a*, NPCOL),
lwf = *nb_a**(*mpa*0 + *nqa*0 + *nb_a*)
lws = max((*nb_a**(*nb_a*-1))/2, (*nrhsqb*0 + *mpb*0)**nb_a*) + *nb_a***nb_a*
else

ltau = numroc(*ia*+min(*m*,*n*)-1, *mb_a*, MYROW, *rsrc_a*, NPROW),
lwf = *mb_a* * (*mpa*0 + *nqa*0 + *mb_a*)
lws = max((*mb_a**(*mb_a*-1))/2, (*npb*0 + max(*nqa*0 + numroc(numroc(*n*+*iroffb*, *mb_a*, 0, 0, NPROW), *mb_a*, 0, 0, *lcmp*), *nrhsqb*0))**mb_a*) + *mb_a***mb_a*
end if,
where *lcmp* = *lcm*/NPROW with *lcm* = ilcm(NPROW, NPCOL),
iroffa = mod(*ia*-1, *mb_a*),
icoffa = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, MYROW, *rsrc_a*, NPROW),
iacol= indxg2p(*ja*, *nb_a*, MYROW, *rsrc_a*, NPROW)
*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, MYROW, *iarow*, NPROW),
*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),
iroffb = mod(*ib*-1, *mb_b*),
icoffb = mod(*jb*-1, *nb_b*),
ibrow = indxg2p(*ib*, *mb_b*, MYROW, *rsrc_b*, NPROW),
ibcol = indxg2p(*jb*, *nb_b*, MYCOL, *csrc_b*, NPCOL),
*mpb*0 = numroc(*m*+*iroffb*, *mb_b*, MYROW, *icrow*, NPROW),
*nqb*0 = numroc(*n*+*icoffb*, *nb_b*, MYCOL, *ibcol*, NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW, and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a	On exit, If $m \geq n$, sub(*A*) is overwritten by the details of its *QR* factorization as returned by p?geqrf; if $m < n$, sub(*A*) is overwritten by details of its *LQ* factorization as returned by p?gelqf.
b	On exit, sub(*B*) is overwritten by the solution vectors, stored columnwise: if *trans* = 'N' and $m \geq n$, rows 1 to *n* of sub(*B*) contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements *n*+1 to *m* in that column; If *trans* = 'N' and $m < n$, rows 1 to *n* of sub(*B*) contain the minimum norm solution vectors; If *trans* = 'T' and $m \geq n$, rows 1 to *m* of sub(*B*) contain the minimum norm solution vectors; if *trans* = 'T' and $m < n$, rows 1 to *m* of sub(*B*) contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements *m*+1 to *n* in that column.
work(1)	On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.
info	(global) INTEGER. = 0: the execution is successful. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?syev

Computes selected eigenvalues and eigenvectors of a symmetric matrix.

Syntax

```
call pssyev( jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork, info )
```

```
call pdsyev( jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork, info )
```

Description

This routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A* by calling the recommended sequence of ScaLAPACK routines.

In its present form, the routine assumes a homogeneous system and makes no checks for consistency of the eigenvalues or eigenvectors across the different processes. Because of this, it is possible that a heterogeneous system may return incorrect results without any error messages.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

jobz	(global). CHARACTER. Must be 'N' or 'V'. Specifies if it is necessary to compute the eigenvectors: If *jobz* ='N', then only eigenvalues are computed. If *jobz* ='V', then eigenvalues and eigenvectors are computed.
uplo	(global). CHARACTER. Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored: If *uplo* = 'U', *a* stores the upper triangular part of *A*. If *uplo* = 'L', *a* stores the lower triangular part of *A*.
n	(global) INTEGER. The number of rows and columns of the matrix *A* (*n* ≥ 0).
a	(local)

REAL for pssyev.
DOUBLE PRECISION for pdsyev.
Block cyclic array of global dimension (*n*, *n*) and local dimension (*lld_a*, *LOC c*(*ja*+*n*-1)). On entry, the symmetric matrix *A*.
If *uplo* = 'U', only the upper triangular part of *A* is used to define the elements of the symmetric matrix.
If *uplo* = 'L', only the lower triangular part of *A* is used to define the elements of the symmetric matrix.

ia, *ja* — (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

iz, *jz* — (global) INTEGER. The row and column indices in the global array *z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *Z*.

work — (local)
REAL for pssyev.
DOUBLE PRECISION for pdsyev.
Array, DIMENSION (*lwork*).

lwork — (local) INTEGER. See below for definitions of variables used to define *lwork*.
If no eigenvectors are requested (*jobz* = 'N'), then *lwork* ≥ 5**n* + *sizesytrd* + 1,
where *sizesytrd* is the workspace for p?sytrd and is max(NB*(*np* +1), 3*NB).
If eigenvectors are requested (*jobz* = 'V') then the amount of workspace required to guarantee that all eigenvectors are computed is:
qrmem = 2**n*-2
lwmin = 5**n* + *n***ldc* + max(*sizemqrleft*, *qrmem*) + 1
Variable definitions:

nb = *desca*(*mb_*) = *desca*(*nb_*) = *descz*(*mb_*) = *descz*(*nb_*);
nn = max(*n*, *nb*, 2);
desca(*rsrc_*) = *desca*(*rsrc_*) = *descz*(*rsrc_*) = *descz*(*csrc_*) = 0
np = numroc(*nn*, *nb*, 0, 0, NPROW)
nq = numroc(max(*n*, *nb*, 2), *nb*, 0, 0, NPCOL)
nrc = numroc(*n*, *nb*, *myprowc*, 0, NPROCS)
ldc = max(1, *nrc*)
sizemqrleft is the workspace for p?ormtr when its *side* argument is 'L'.
myprowc is defined when a new context is created as follows:
call blacs_get(*desca*(*ctxt_*), 0, *contextc*)
call blacs_gridinit(*contextc*, 'R', NPROCS, 1)
call blacs_gridinfo(*contextc*, *nprowc*, *npcolc*, *myprowc*, *mypcolc*)
If *lwork* = –1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — On exit, the lower triangle (if *uplo*='L') or the upper triangle (if *uplo*='U') of *A*, including the diagonal, is destroyed.

w — (global). REAL for pssyev
DOUBLE PRECISION for pdsyev
Array, DIMENSION (*n*).
On normal exit, the first *m* entries contain the selected eigenvalues in ascending order.

z — (local). REAL for pssyev
DOUBLE PRECISION for pdsyev
Array, global dimension (*n*, *n*), local dimension (*lld_z*, *LOCc*(*jz*+*n*–1)). If *jobz* = 'V', then on normal exit the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues.

	If *jobz* = 'N', then *z* is not referenced.
work(1)	On output, *work*(1) returns the workspace needed to guarantee completion. If the input parameters are incorrect, *work*(1) may also be incorrect. If *jobz* = 'N' *work*(1) = minimal (optimal) amount of workspace If *jobz* = 'V' *work*(1) = minimal workspace required to generate all the eigenvectors.
info	(global) INTEGER. If *info* = 0, the execution is successful. If *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*. If *info* > 0: If *info*= 1 through *n*, the *i*-th eigenvalue did not converge in ?steqr2 after a total of 30*n* iterations. If *info*= *n*+1, then p?syev has detected heterogeneity by finding that eigenvalues were not identical across the process grid. In this case, the accuracy of the results from p?syev cannot be guaranteed.

p?syevx

Computes selected eigenvalues and, optionally, eigenvectors of a symmetric matrix.

Syntax

call pssyevx(*jobz*, *range*, *uplo*, *n*, *a*, *ia*, *ja*, *desca*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *nz*, *w*, *orfac*, *z*, *iz*, *jz*, *descz*, *work*, *lwork*, *iwork*, *liwork*, *ifail*, *iclustr*, *gap*, *info*)

call pdsyevx(*jobz*, *range*, *uplo*, *n*, *a*, *ia*, *ja*, *desca*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *nz*, *w*, *orfac*, *z*, *iz*, *jz*, *descz*, *work*, *lwork*, *iwork*, *liwork*, *ifail*, *iclustr*, *gap*, *info*)

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A* by calling the recommended sequence of ScaLAPACK routines. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

jobz (global). CHARACTER*1. Must be 'N' or 'V'. Specifies if it is necessary to compute the eigenvectors:
If *jobz* ='N', then only eigenvalues are computed.
If *jobz* ='V', then eigenvalues and eigenvectors are computed.

range (global). CHARACTER*1. Must be 'A', 'V', or 'I'.
If *range* = 'A', all eigenvalues will be found.
If *range* = 'V', all eigenvalues in the half-open interval [*vl*, *vu*] will be found.
If *range* = 'I', the eigenvalues with indices *il* through *iu* will be found.

uplo (global). CHARACTER*1. Must be 'U' or 'L'.
Specifies whether the upper or lower triangular part of the symmetric matrix *A* is stored:
If *uplo* = 'U', *a* stores the upper triangular part of *A*.
If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n (global) INTEGER. The number of rows and columns of the matrix *A* (*n* ≥ 0).

a (local). REAL for pssyevx
DOUBLE PRECISION for pdsyevx.
Block cyclic array of global dimension (*n*, *n*) and local dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, the symmetric matrix *A*.
If *uplo* = 'U', only the upper triangular part of *A* is used to define the elements of the symmetric matrix.
If *uplo* = 'L', only the lower triangular part of *A* is used to define the elements of the symmetric matrix.

ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
vl, *vu*	(global) REAL for pssyevx DOUBLE PRECISION for pdsyevx. If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues; *vl* ≤ *vu*. Not referenced if *range* = 'A' or 'I'.
il, *iu*	(global) INTEGER. If *range* ='I', the indices of the smallest and largest eigenvalues to be returned. Constraints: *il* ≥ 1 min(*il*,*n*) ≤ *iu* ≤ *n* Not referenced if *range* = 'A' or 'V'.
abstol	(global). REAL for pssyevx DOUBLE PRECISION for pdsyevx. If *jobz*='V', setting *abstol* to p?lamch(*context*, 'U') yields the most orthogonal eigenvectors. The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [*a*, *b*] of width less than or equal to *abstol* + *eps* * max(|*a*|,|*b*|), where *eps* is the machine precision. If *abstol* is less than or equal to zero, then *eps**norm(T) will be used in its place, where norm(T) is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*p?lamch('S') not zero. If this routine returns with ((mod(*info*,2).*ne*.0).*or*. * (mod(*info*/8,2).*ne*.0)), indicating that some eigenvalues or eigenvectors did not converge, try setting *abstol* to 2*p?lamch('S').

orfac (global). REAL for pssyevx
DOUBLE PRECISION for pdsyevx.
Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within *tol*=*orfac**norm(*A*) of each other are to be reorthogonalized. However, if the workspace is insufficient (see *lwork*), *tol* may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if *orfac* equals zero. A default value of 1.0e-3 is used if *orfac* is negative. *orfac* should be identical on all processes.

iz, *jz* (global) INTEGER. The row and column indices in the global array *z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *Z*. *descz*(*ctxt_*) must equal *desca*(*ctxt_*).

work (local)
REAL for pssyevx.
DOUBLE PRECISION for pdsyevx.
Array, DIMENSION (*lwork*).

lwork (local) INTEGER. The dimension of the array *work*.
See below for definitions of variables used to define *lwork*.
If no eigenvectors are requested (*jobz* = 'N'), then *lwork* ≥ 5**n* + max(5**nn*, NB*(*np*0 + 1)).
If eigenvectors are requested (*jobz* = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is:
lwork ≥ 5**n* + max(5**nn*, *np*0**mq*0 + 2*NB*NB) + iceil(*neig*, NPROW*NPCOL)**nn*
The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality (at the cost of potentially poor performance) you should add the following to *lwork*:
(*clustersize*-1)**n*,

where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:

{*w*(*k*),..., *w*(*k*+*clustersize*-1)| *w*(*j*+1) ≤ *w*(*j*)) + *orfac**2*norm(*A*)},

where

neig = number of eigenvectors requested

nb = *desca*(*mb_*) = *desca*(*nb_*) = *descz*(*mb_*) = *descz*(*nb_*);

nn = max(*n*, *nb*, 2);

desca(*rsrc_*) = *desca*(*nb_*) = *descz*(*rsrc_*) = *descz*(*csrc_*) = 0;

*np*0 = numroc(*nn*, *nb*, 0, 0, NPROW);

*mq*0 = numroc(max(*neig*, *nb*, 2), *nb*, 0, 0, NPCOL)

iceil(*x*, *y*) is a ScaLAPACK function returning ceiling(*x*/*y*)

If *lwork* is too small to guarantee orthogonality, p?syevx attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.

If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -23 is returned.

Note that when *range*='V', number of requested eigenvectors are not known until the eigenvalues are computed. In this case and if *lwork* is large enough to compute the eigenvalues, p?sygvx computes the eigenvalues and as many eigenvectors as possible.

Relationship between workspace, orthogonality & performance:

Greater performance can be achieved if adequate workspace is provided. In some situations, performance can decrease as the provided workspace increases above the workspace amount shown below:

lwork ≥ max(*lwork*, 5**n* + *nsytrd_lwopt*),

where *lwork*, as defined previously, depends upon the number of eigenvectors requested, and

nsytrd_lwopt = *n* + 2*(*anb*+1)*(4**nps*+2) + (*nps* + 3)**nps*;

anb = pjlaenv(*desca*(*ctxt_*), 3, 'p?syttrd', 'L', 0, 0, 0, 0);
sqnpc = int(*sqrt*(*dble*(NPROW * NPCOL)));
nps = max(numroc(*n*, 1, 0, 0, *sqnpc*), 2**anb*);
numroc is a ScaLAPACK tool functions;
pjlaenv is a ScaLAPACK environmental inquiry function
MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
For large n, no extra workspace is needed, however the biggest boost in performance comes for small n, so it is wise to provide the extra workspace (typically less than a megabyte per process).
If *clustersize* > *n*/*sqrt*(NPROW*NPCOL), then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. At the limit (that is, *clustersize* = *n*–1) p?stein will perform no better than ?stein on single processor.
For *clustersize* = *n*/*sqrt*(NPROW*NPCOL) reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.
For *clustersize* > *n*/*sqrt*(NPROW*NPCOL) execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.
If *lwork* = –1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxerbla.

iwork (local) INTEGER. Workspace array.

liwork (local) INTEGER, dimension of *iwork*. *liwork* ≥ 6**nnp*
Where: *nnp* = max(*n*, NPROW*NPCOL + 1, 4)
If *liwork* = –1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of

these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — On exit, the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U')of *A*, including the diagonal, is overwritten.

m — (global) INTEGER. The total number of eigenvalues found; $0 \le m \le n$.

nz — (global) INTEGER. Total number of eigenvectors computed. $0 \le nz \le m$.
The number of columns of *z* that are filled.
If *jobz* ≠ 'V', *nz* is not referenced.
If *jobz* = 'V', *nz* = *m* unless the user supplies insufficient space and p?syevx is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in *z* (*m*, *le*.*descz*(*n_*)) and sufficient workspace to compute them. (See *lwork*). p?syevx is always able to detect insufficient space without computation unless *range*.*eq*.'V'.

w — (global). REAL for pssyevx
DOUBLE PRECISION for pdsyevx.
Array, DIMENSION (*n*). The first *m* elements contain the selected eigenvalues in ascending order.

z — (local). REAL for pssyevx
DOUBLE PRECISION for pdsyevx.
Array, global dimension (*n*, *n*), local dimension (*lld_z*, *LOCc*(*jz*+*n*-1)).
If *jobz* = 'V', then on normal exit the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

	If *jobz* = 'N', then *z* is not referenced.
work(1)	On exit, returns workspace adequate workspace to allow optimal performance.
iwork(1)	On return, *iwork*(1) contains the amount of integer workspace required.
ifail	(global) INTEGER. Array, DIMENSION (*n*). If *jobz* = 'V', then on normal exit, the first *m* elements of *ifail* are zero. If (mod(*info*,2). *ne*.0) on exit, then *ifail* contains the indices of the eigenvectors that failed to converge. If *jobz* = 'N', then *ifail* is not referenced.
iclustr	(global) INTEGER. Array, DIMENSION (2*NPROW*NPCOL) This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*(2*i*-1) to *iclustr*(2*i*), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. *iclustr*() is a zero terminated array. (*iclustr*(2*k*).*ne*.0. *and*. *iclustr*(2*k*+1).*eq*.0) if and only if *k* is the number of clusters. *iclustr* is not referenced if *jobz* = 'N'.
gap	(global) REAL for pssyevx DOUBLE PRECISION for pdsyevx. Array, DIMENSION (NPROW*NPCOL) This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the *i*th cluster may be as high as (*C**n*)/*gap*(*i*) where *C* is a small constant.
info	(global) INTEGER. If *info* = 0, the execution is successful. If *info* < 0:

If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
If *info* > 0: if (mod(*info*,2).*ne*.0), then one or more eigenvectors failed to converge. Their indices are stored in *ifail*. Ensure *abstol*=2.0*p?lamch('U').
If (mod(*info*/2,2).*ne*.0), then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace.The indices of the clusters are stored in the array *iclustr*.
If (mod(*info*/4,2).*ne*.0), then space limit prevented p?syevxf rom computing all of the eigenvectors between *vl* and *vu*. The number of eigenvectors computed is returned in *nz*.
If (mod(*info*/8,2).*ne*.0), then p?stebz failed to compute eigenvalues. Ensure *abstol*=2.0*p?lamch('U').

p?heevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

Syntax

call pcheevx(*jobz*, *range*, *uplo*, *n*, *a*, *ia*, *ja*, *desca*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *nz*, *w*, *orfac*, *z*, *iz*, *jz*, *descz*, *work*, *lwork*, *rwork*, *lrwork*, *iwork*, *liwork*, *ifail*, *iclustr*, *gap*, *info*)

call pzheevx(*jobz*, *range*, *uplo*, *n*, *a*, *ia*, *ja*, *desca*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *nz*, *w*, *orfac*, *z*, *iz*, *jz*, *descz*, *work*, *lwork*, *rwork*, *lrwork*, *iwork*, *liwork*, *ifail*, *iclustr*, *gap*, *info*)

Description

This routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A* by calling the recommended sequence of ScaLAPACK routines. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

jobz (global). CHARACTER*1. Must be 'N' or 'V'.
Specifies if it is necessary to compute the eigenvectors:
If *jobz* ='N', then only eigenvalues are computed.
If *jobz* ='V', then eigenvalues and eigenvectors are computed.

range (global). CHARACTER*1. Must be 'A', 'V', or 'I'.
If *range* = 'A', all eigenvalues will be found.
If *range* = 'V', all eigenvalues in the half-open interval [*vl*, *vu*] will be found.
If *range* = 'I', the eigenvalues with indices *il* through *iu* will be found.

uplo (global). CHARACTER*1. Must be 'U' or 'L'.
Specifies whether the upper or lower triangular part of the Hermitian matrix *A* is stored:
If *uplo* = 'U', *a* stores the upper triangular part of *A*.
If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n (global) INTEGER. The number of rows and columns of the matrix *A* (*n* ≥ 0).

a (local).
COMPLEX for pcheevx
DOUBLE COMPLEX for pzheevx.
Block cyclic array of global dimension (*n*, *n*) and local dimension (*lld_a*, *LOC c*(*ja*+*n*−1)). On entry, the Hermitian matrix *A*.
If *uplo* = 'U', only the upper triangular part of *A* is used to define the elements of the symmetric matrix.
If *uplo* = 'L', only the lower triangular part of *A* is used to define the elements of the Hermitian matrix.

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*. If *desca*(*ctxt_*) is incorrect, p?heevx cannot guarantee correct error reporting
vl, *vu*	(global) REAL for pcheevx DOUBLE PRECISION for pzheevx. If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues; not referenced if *range* = 'A' or 'I'.
il, *iu*	(global) INTEGER. If *range* ='I', the indices of the smallest and largest eigenvalues to be returned. Constraints: $1 \le il \le iu \le n$, if $n > 0$; $il = 1$ and $iu = 0$ if $n = 0$. Not referenced if *range* = 'A' or 'V'.
abstol	(global). REAL for pcheevx DOUBLE PRECISION for pzheevx. If *jobz*='V', setting *abstol* to p?lamch(*context*, 'U') yields the most orthogonal eigenvectors. The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to $abstol + eps * \max(|a|, |b|)$, where *eps* is the machine precision. If *abstol* is less than or equal to zero, then *eps**norm(T) will be used in its place, where norm(T) is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues are computed most accurately when *abstol* is set to twice the underflow threshold 2*p?lamch('S'), not zero. If this routine returns with ((mod(*info*,2).*ne*.0).*or*.(mod(*info*/8,2).*ne*.0)), indicating that some eigenvalues or eigenvectors did not converge, try setting *abstol* to 2*p?lamch('S').
orfac	(global). REAL for pcheevx DOUBLE PRECISION for pzheevx.

Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within *tol*=*orfac**norm(*A*) of each other are to be reorthogonalized. However, if the workspace is insufficient (see *lwork*), *tol* may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if *orfac* equals zero. A default value of 1.0e-3 is used if *orfac* is negative. *orfac* should be identical on all processes.

iz, *jz* — (global) INTEGER. The row and column indices in the global array *z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz — (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *Z*. *descz*(*ctxt_*) must equal *desca*(*ctxt_*).

work — (local)
COMPLEX for pcheevx
DOUBLE COMPLEX for pzheevx.
Array, DIMENSION (*lwork*).

lwork — (local). INTEGER. The dimension of the array *work*.
If only eigenvalues are requested:

lwork ≥ *n* + max(*nb**(*np*0 + 1), 3)

If eigenvectors are requested:

lwork ≥ *n* + (*np*0+*mq*0+*nb*)**nb*

with *nq*0 = numroc(*nn*, *nb*, 0, 0, NPCOL).

lwork ≥ 5**n* + max(5**nn*, *np*0**mq*0+2**nb***nb*) + iceil(*neig*, NPROW*NPCOL)**nn*

For optimal performance, greater workspace is needed, that is

lwork ≥ max(*lwork*, *nhetrd_lwork*)

where *lwork* is as defined above, and *nhetrd_lwork* = *n* + 2*(*anb*+1)*(4**nps*+2) + (*nps*+1)**nps*
ictxt = *desca*(*ctxt_*)
anb = pjlaenv(*ictxt*, 3, 'pchettrd', 'L', 0, 0, 0, 0)
sqnpc = *sqrt*(*dble*(NPROW * NPCOL))

nps = max(numroc(*n*, 1, 0, 0, *sqnpc*), 2**anb*)
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxerbla.

rwork (local)
REAL for pcheevx
DOUBLE PRECISION for pzheevx.
Workspace array, DIMENSION (*lrwork*).

lrwork (local) INTEGER. The dimension of the array *work*.
See below for definitions of variables used to define *lwork*.
If no eigenvectors are requested (*jobz* = 'N'), then
lrwork ≥ 5**nn*+4**n*.
If eigenvectors are requested (*jobz* = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is:
lrwork ≥ 4**n* + max(5**nn*, *np*0**mq*0+2**nb***nb*) + iceil(*neig*, NPROW*NPCOL)**nn*
The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality (at the cost of potentially poor performance) you should add the following values to *lrwork*:
(*clustersize*-1)**n*,
where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:
{*w*(*k*),..., *w*(*k*+*clustersize*-1)|*w*(*j*+1) ≤ *w*(*j*)+*orfac**2*norm*(*A*)}.
Variable definitions:
neig = number of eigenvectors requested;
nb = *desca*(*mb_*) = *desca*(*nb_*) = *descz*(*mb_*) = *descz*(*nb_*);
nn = max(*n*, NB, 2);

desca(*rsrc_*) = *desca*(*nb_*) = *descz*(*rsrc_*) = *descz*(*csrc_*) = 0;
*np*0 = numroc(*nn*, *nb*, 0, 0, NPROW);
*mq*0 = numroc(max(*neig*, *nb*, 2), *nb*, 0, 0, NPCOL);
iceil(*x*, *y*) is a ScaLAPACK function returning ceiling(*x*/*y*)
When *lrwork* is too small:
If *lwork* is too small to guarantee orthogonality, p?heevx attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues. If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -23 is returned. Note that when *range*='V', p?heevx does not know how many eigenvectors are requested until the eigenvalues are computed. Therefore, when *range*='V' and as long as *lwork* is large enough to allow p?heevx to compute the eigenvalues, p?heevx will compute the eigenvalues and as many eigenvectors as it can.
Relationship between workspace, orthogonality and performance:
If *clustersize* ≥ *n*/*sqrt*(NPROW*NPCOL), then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. In the limit (that is, *clustersize* = *n*-1) p?stein will perform no better than ?stein on 1 processor.
For *clustersize* = *n*/*sqrt*(NPROW*NPCOL) reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.
For *clustersize* > *n*/*sqrt*(NPROW*NPCOL) execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxerbla.

iwork (local) INTEGER. Workspace array.

liwork (local) INTEGER, dimension of *iwork*.

liwork ≥ 6*nnp*

Where: *nnp* = max(*n*, NPROW*NPCOL+1, 4)

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit, the lower triangle (if *uplo* = 'L'), or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

m (global) INTEGER. The total number of eigenvalues found;

0 ≤ *m* ≤ *n*.

nz (global) INTEGER. Total number of eigenvectors computed.

0 ≤ *nz* ≤ *m*.

The number of columns of *z* that are filled.

If *jobz* ≠ 'V', *nz* is not referenced.

If *jobz* = 'V', *nz* = *m* unless the user supplies insufficient space and p?heevx is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in *z* (*m*.*le*.*descz*(*n*_)) and sufficient workspace to compute them. (See *lwork*). p?heevx is always able to detect insufficient space without computation unless *range*.*eq*.'V'.

w (global).

REAL for pcheevx

DOUBLE PRECISION for pzheevx.

Array, DIMENSION (*n*). The first *m* elements contain the selected eigenvalues in ascending order.

z (local).

COMPLEX for pcheevx
DOUBLE COMPLEX for pzheevx.
Array, global dimension (*n*, *n*), local dimension (*lld_z*, *LOCc*(*jz*+*n*-1)).
If *jobz* ='V', then on normal exit the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
If *jobz* = 'N', then *z* is not referenced.

work(1) — On exit, returns workspace adequate workspace to allow optimal performance.

rwork — (local).
REAL for pcheevx
DOUBLE PRECISION for pzheevx.
Array, DIMENSION (*lrwork*). On return, *rwork(1)* contains the optimal amount of workspace required for efficient execution.
If *jobz*='N' *rwork(1)* = optimal amount of workspace required to compute eigenvalues efficiently.
If *jobz*='V' *rwork(1)* = optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality.
If *range*='V', it is assumed that all eigenvectors may be required.

iwork(1) — (local)
On return, *iwork*(1) contains the amount of integer workspace required.

ifail — (global) INTEGER.
Array, DIMENSION (*n*).
If *jobz* ='V', then on normal exit, the first *m* elements of *ifail* are zero. If (mod(*info*,2).*ne*.0) on exit, then *ifail* contains the indices of the eigenvectors that failed to converge.
If *jobz* = 'N', then *ifail* is not referenced.

iclustr — (global) INTEGER.
Array, DIMENSION (2*NPROW*NPCOL).

This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*(2*i*-1) to *iclustr*(2*i*), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. *iclustr*() is a zero terminated array. (*iclustr*(2*k*).*ne*.0. *and*. *iclustr*(2*k*+1).*eq*.0) if and only if *k* is the number of clusters. *iclustr* is not referenced if *jobz* = 'N'.

gap

(global)
REAL for pcheevx
DOUBLE PRECISION for pzheevx.
Array, DIMENSION (NPROW*NPCOL)
This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the *i*-th cluster may be as high as (*C**n*)/*gap*(*i*) where *C* is a small constant.

info

(global) INTEGER.
If *info* = 0, the execution is successful.
If *info* < 0:
If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
If *info* > 0:
If (mod(*info*,2).*ne*.0), then one or more eigenvectors failed to converge. Their indices are stored in *ifail*. Ensure *abstol*=2.0*p?lamch('U')
If (mod(*info*/2,2).*ne*.0), then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace.The indices of the clusters are stored in the array *iclustr*.

If (mod(*info*/4,2).*ne*.0), then space limit prevented p?syevx from computing all of the eigenvectors between *vl* and *vu*. The number of eigenvectors computed is returned in *nz*.
If (mod(*info*/8,2).*ne*.0), then p?stebz failed to compute eigenvalues. Ensure *abstol*=2.0*p?lamch('U').

p?gesvd

Computes the singular value decomposition of a general matrix, optionally computing the left and/or right singular vectors.

Syntax

```
call psgesvd( jobu, jobvt, m, n, a, ia, ja, desca, s, u, iu, ju, descu, vt,
ivt, jvt, descvt, work, lwork, info )
```

```
call pdgesvd( jobu, jobvt, m, n, a, ia, ja, desca, s, u, iu, ju, descu, vt,
ivt, jvt, descvt, work, lwork, info )
```

```
call pcgesvd( jobu, jobvt, m, n, a, ia, ja, desca, s, u, iu, ju, descu, vt,
ivt, jvt, descvt, work, lwork, rwork, info )
```

```
call pzgesvd( jobu, jobvt, m, n, a, ia, ja, desca, s, u, iu, ju, descu, vt,
ivt, jvt, descvt, work, lwork, rwork, info )
```

Description

This routine computes the singular value decomposition (SVD) of an m-by-n matrix A, optionally computing the left and/or right singular vectors. The SVD is written

$A = U*\Sigma*V^T$,

where Σ is an m-by-n matrix that is zero except for its min(m, n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of Σ are the singular values of A and the columns of U and V are the corresponding right and left singular vectors, respectively. The singular values are returned in array s in decreasing order and only the first min(m,n) columns of U and rows of $vt = V^T$ are computed.

Input Parameters

mp = number of local rows in A and U

nq = number of local columns in *A* and *VT*

size = min(*m*, *n*)

sizeq = number of local columns in *U*

sizep = number of local rows in *VT*

jobu	(global). CHARACTER*1. Specifies options for computing all or part of the matrix *U*. If *jobu* = 'V', the first *size* columns of *U* (the left singular vectors) are returned in the array *u*; If *jobu* ='N', no columns of *U* (no left singular vectors)are computed.
jobvt	(global) CHARACTER*1. Specifies options for computing all or part of the matrix V^T. If *jobvt* = 'V', the first *size* rows of V^T (the right singular vectors) are returned in the array *vt*; If *jobvt* = 'N', no rows of V^T(no right singular vectors) are computed.
m	(global) INTEGER. The number of rows of the matrix *A* (*m* ≥ 0).
n	(global) INTEGER. The number of columns in *A* (*n* ≥ 0).
a	(local). REAL for psgesvd DOUBLE PRECISION for pdgesvd COMPLEX for pcgesvd COMPLEX*16 for pzgesvd Block cyclic array, global dimension (*m*, *n*), local dimension (*mp*, *nq*). *work*(*lwork*) is a workspace array.
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.
desca	(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.
iu, *ju*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *U*, respectively.

descu (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *U*.

ivt, *jvt* (global) INTEGER. The row and column indices in the global array *vt* indicating the first row and the first column of the submatrix *VT*, respectively.

descvt (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *VT*.

work (local). REAL for psgesvd
DOUBLE PRECISION for pdgesvd
COMPLEX for pcgesvd
COMPLEX*16 for pzgesvd
Workspace array, dimension (*lwork*)

lwork (local) INTEGER. The dimension of the array *work*;
lwork > 2 + 6**sizeb* + max(*watobd*, *wbdtosvd*),
where *sizeb* = max(*m*, *n*), and *watobd* and *wbdtosvd* refer, respectively, to the workspace required to bidiagonalize the matrix *A* and to go from the bidiagonal matrix to the singular value decomposition *U S VT*.
For *watobd*, the following holds:
watobd = max(max(*wp?lange*,*wp?gebrd*),
max(*wp?lared2d*, *wp?lared1d*)),
where *wp?lange*, *wp?lared1d*, *wp?lared2d*, *wp?gebrd* are the workspaces required respectively for the subprograms p?lange, p?lared1d, p?lared2d, p?gebrd. Using the standard notation
mp = numroc(*m*, *mb*, MYROW, *desca*(*ctxt_*), NPROW),
nq = numroc(*n*, *nb*, MYCOL, *desca*(*lld_*), NPCOL),
the workspaces required for the above subprograms are
wp?lange = *mp*,
wp?lared1d = *nq*0,
wp?lared2d = *mp*0,
wp?gebrd = *nb**(*mp* + *nq* + 1) + *nq*,
where *nq*0 and *mp*0 refer, respectively, to the values obtained at MYCOL = 0 and MYROW = 0. In general, the upper limit for the workspace is given by a workspace required on processor (0,0):
watobd ≤ *nb**(*mp*0 + *nq*0 + 1) + *nq*0.

In case of a homogeneous process grid this upper limit can be used as an estimate of the minimum workspace for every processor.
For *wbdtosvd*, the following holds:
*wbdtosvd = size**(*wantu***nru* + *wantvt***ncvt*) + max(*w?bdsqr*, max(*wantu***wp?ormbrqln*, *wantvt***wp?ormbrprt*)),
where
wantu(*wantvt*) = 1, if left/right singular vectors are wanted, and *wantu*(*wantvt*) = 0, otherwise. *w?bdsqr*, *wp?ormbrqln*, and *wp?ormbrprt* refer respectively to the workspace required for the subprograms ?bdsqr, p?ormbr(*qln*), and p?ormbr(*prt*), where *qln* and *prt* are the values of the arguments *vect*, *side*, and *trans* in the call to p?ormbr. *nru* is equal to the local number of rows of the matrix *U* when distributed 1-dimensional "column" of processes. Analogously, *ncvt* is equal to the local number of columns of the matrix *VT* when distributed across 1-dimensional "row" of processes. Calling the LAPACK procedure ?bdsqr requires
w?bdsqr = max(1, 2**size* + (2**size* - 4)* max(*wantu*, *wantvt*))
on every processor. Finally,
wp?ormbrqln = max((*nb**(*nb*-1))/2, (*sizeq*+*mp*)**nb*)+*nb***nb*,
wp?ormbrprt = max((*mb**(*mb*-1))/2, (*sizep*+*nq*)**mb*)+*mb***mb*,
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum size for the work array. The required workspace is returned as the first element of *work* and no error message is issued by pxerbla.

rwork

REAL for psgesvd
DOUBLE PRECISION for pdgesvd
COMPLEX for pcgesvd
COMPLEX*16 for pzgesvd
Workspace array, dimension (1 + 4**sizeb*)

Output Parameters

a — On exit, the contents of *a* are destroyed.

s — (global). REAL for psgesvd
DOUBLE PRECISION for pdgesvd
COMPLEX for pcgesvd
COMPLEX*16 for pzgesvd
Array, DIMENSION (*size*).
Contains the singular values of *A* sorted so that *s*(i) ≥ *s*(i+1).

u — (local). REAL for psgesvd
DOUBLE PRECISION for pdgesvd
COMPLEX for pcgesvd
COMPLEX*16 for pzgesvd
local dimension (*mp*, *sizeq*), global dimension (*m*, *size*)
If *jobu* = 'V', *u* contains the first min(*m*, *n*) columns of *U*.
If *jobu* = 'N' or 'O', *u* is not referenced.

vt — (local). REAL for psgesvd
DOUBLE PRECISION for pdgesvd
COMPLEX for pcgesvd
COMPLEX*16 for pzgesvd
local dimension (*sizep*, *nq*), global dimension (*size*, *n*)
If *jobvt* = 'V', *vt* contains the first *size* rows of V^T if *jobu* = 'N', *vt* is not referenced.

work — On exit, if *info* = 0, then *work*(1) returns the required minimal size of *lwork*.

rwork — On exit, if *info* = 0, then *rwork*(1) returns the required size of *rwork*.

info — (global) INTEGER.
If *info* = 0, the execution is successful.
If *info* < 0, If *info* = −*i*, the *i*th parameter had an illegal value.
If *info* > 0 *i*, then if ?bdsqr did not converge,
If *info* = min(*m*, *n*) + 1, then p?gesvd has detected heterogeneity by finding that eigenvalues were not identical across the process grid. In this case, the accuracy of the results from p?gesvd cannot be guaranteed.

p?sygvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.

Syntax

call pssygvx(*ibtype*, *jobz*, *range*, *uplo*, *n*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *nz*, *w*, *orfac*, *z*, *iz*, *jz*, *descz*, *work*, *lwork*, *iwork*, *liwork*, *ifail*, *iclustr*, *gap*, *info*)

call pdsygvx(*ibtype*, *jobz*, *range*, *uplo*, *n*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *nz*, *w*, *orfac*, *z*, *iz*, *jz*, *descz*, *work*, *lwork*, *iwork*, *liwork*, *ifail*, *iclustr*, *gap*, *info*)

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

sub(*A*)**x* = λ*sub(*B*)**x*, sub(*A*) sub(*B*)**x* = λ**x*, or sub(*B*)*sub(*A*)**x* = λ**x*.

Here *x* denotes eigen vectors, λ (*lambda*) denotes eigenvalues, sub(*A*) denoting A(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) is assumed to symmetric, and sub(*B*) denoting *B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*n*-1) is also positive definite.

Input Parameters

ibtype (global) INTEGER. Must be 1 or 2 or 3.
Specifies the problem type to be solved:
If *ibtype* = 1, the problem type is sub(*A*)**x* = *lambda**sub(*B*)**x*;
If *ibtype* = 2, the problem type is sub(*A*)*sub(*B*)**x* = *lambda***x*;
If *ibtype* = 3, the problem type is sub(*B*)*sub(*A*)**x* = *lambda***x*.

jobz (global). CHARACTER*1. Must be 'N' or 'V'.
If *jobz* ='N', then compute eigenvalues only.
If *jobz* ='V', then compute eigenvalues and eigenvectors.

range (global). CHARACTER*1. Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.
If *range* = 'V', the routine computes eigenvalues in the interval: [*vl*, *vu*]
If *range* = 'I', the routine computes eigenvalues with indices *il* through *iu*.

uplo (global). CHARACTER*1. Must be 'U' or 'L'.
If *uplo* = 'U', arrays *a* and *b* store the upper triangles of sub(*A*) and sub (*B*);
If *uplo* = 'L', arrays *a* and *b* store the lower triangles of sub(*A*) and sub (*B*).

n (global). INTEGER. The order of the matrices sub(*A*) and sub (*B*), $n \geq 0$.

a (local)
REAL for pssygvx
DOUBLE PRECISION for pdsygvx.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, this array contains the local pieces of the *n*-by-*n* symmetric distributed matrix sub(*A*).
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix.

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*. If *desca*(*ctxt_*) is incorrect, p?sygvx cannot guarantee correct error reporting.

b (local). REAL for pssygvx
DOUBLE PRECISION for pdsygvx.
Pointer into the local memory to an array of dimension (*lld_b*, *LOCc*(*jb*+*n*-1)). On entry, this array contains the local pieces of the *n*-by-*n* symmetric distributed matrix sub(*B*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*B*) contains the upper triangular part of the matrix.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix.

ib, *jb*
(global) INTEGER. The row and column indices in the global array *b* indicating the first row and the first column of the submatrix *B*, respectively.

descb
(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*. *descb*(*ctxt_*) must be equal to *desca*(*ctxt_*).

vl, *vu*
(global)
REAL for pssygvx
DOUBLE PRECISION for pdsygvx.
If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu*
(global)
INTEGER.
If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint:
$il \geq 1$, $\min(il, n) \leq iu \leq n$
If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol
(global)
REAL for pssygvx
DOUBLE PRECISION for pdsygvx.
If *jobz*='V', setting *abstol* to p?lamch(*context*, 'U') yields the most orthogonal eigenvectors.
The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [*a*,*b*] of width less than or equal to
abstol + *eps**max(|*a*|,|*b*|),
where *eps* is the machine precision. If *abstol* is less than or equal to zero, then *eps**norm(T) will be used in its place, where norm(T) is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form.

Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*p?lamch('S') not zero. If this routine returns with ((mod(*info*,2).*ne*.0).*or*.*(*mod*(*info*/8,2).*ne*.0)), indicating that some eigenvalues or eigenvectors did not converge, try setting *abstol* to 2*p?lamch('S').

orfac (global).
REAL for pssygvx
DOUBLE PRECISION for pdsygvx.
Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within *tol*=*orfac**norm(*A*) of each other are to be reorthogonalized. However, if the workspace is insufficient (see *lwork*), *tol* may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if *orfac* equals zero. A default value of 1.0e-3 is used if *orfac* is negative. *orfac* should be identical on all processes.

iz, *jz* (global) INTEGER. The row and column indices in the global array *z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *Z*. *descz*(*ctxt_*) must equal *desca*(*ctxt_*).

work (local)
REAL for pssygvx
DOUBLE PRECISION for pdsygvx.
Workspace array, dimension (*lwork*)

lwork (local) INTEGER.
Dimension of the array *work*. See below for definitions of variables used to define *lwork*.
If no eigenvectors are requested (*jobz* = 'N'), then *lwork* ≥ 5**n* + max(5**nn*, NB*(*np*0 + 1)).
If eigenvectors are requested (*jobz* = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is:

lwork ≥ 5*n + max(5*nn, np0*mq0 + 2*nb*nb) + iceil(neig, NPROW*NPCOL)*nn.

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality at the cost of potentially poor performance you should add the following to *lwork*:

(*clustersize*-1)*n,

where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:

{w(k),..., w(k+clustersize-1)|w(j+1) ≤ w(j) + orfac*2*norm(A)}

Variable definitions:

neig = number of eigenvectors requested,

nb = desca(*mb_*) = desca(*nb_*) = descz(*mb_*) = descz(*nb_*),

nn = max(*n*, *nb*, 2),

desca(*rsrc_*) = desca(*nb_*) = descz(*rsrc_*) = descz(*csrc_*) = 0,

*np*0 = numroc(*nn*, *nb*, 0, 0, NPROW),

*mq*0 = numroc(max(*neig*, *nb*, 2), *nb*, 0, 0, NPCOL)

iceil(*x*, *y*) is a ScaLAPACK function returning ceiling(*x*/*y*)

If *lwork* is too small to guarantee orthogonality, p?syevx attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.

If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -23 is returned.

Note that when *range*='V', number of requested eigenvectors are not known until the eigenvalues are computed. In this case and if *lwork* is large enough to compute the eigenvalues, p?sygvx computes the eigenvalues and as many eigenvectors as possible.

Greater performance can be achieved if adequate workspace is provided. In some situations, performance can decrease as the provided workspace increases above the workspace amount shown below:

lwork ≥ max(*lwork*, 5**n* + *nsytrd_lwopt*, *nsygst_lwopt*), where
lwork, as defined previously, depends upon the number of eigenvectors requested, and
nsytrd_lwopt = *n* + 2*(*anb*+1)*(4**nps*+2) + (*nps*+3)**nps*
nsygst_lwopt = 2**np*0**nb* + *nq*0**nb* + *nb***nb*
anb = pjlaenv(*desca*(*ctxt_*), 3, p?syttrd ', 'L', 0, 0, 0, 0)
sqnpc = int(*sqrt*(*dble*(NPROW * NPCOL)))
nps = max(numroc(*n*, 1, 0, 0, *sqnpc*), 2**anb*)
NB = *desca*(*mb_*)
*np*0 = numroc(*n*, *nb*, 0, 0, NPROW)
*nq*0 = numroc(*n*, *nb*, 0, 0, NPCOL)
numroc is a ScaLAPACK tool functions;
pjlaenv is a ScaLAPACK environmental inquiry function
MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
For large n, no extra workspace is needed, however the biggest boost in performance comes for small n, so it is wise to provide the extra workspace (typically less than a Megabyte per process).

If *clustersize* ≥ *n*/*sqrt*(NPROW*NPCOL), then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. At the limit (that is, *clustersize* = *n*–1) p?stein will perform no better than ?stein on a single processor.
For *clustersize* = *n*/*sqrt*(NPROW*NPCOL) reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.
For *clustersize* > *n*/*sqrt*(NPROW*NPCOL) execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.
If *lwork* = –1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each

of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxerbla.

iwork (local) INTEGER. Workspace array.

liwork (local) INTEGER, dimension of *iwork*.

liwork ≥ 6*nnp

Where:

nnp = max(*n*, NPROW*NPCOL + 1, 4)

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit,

If *jobz* = 'V', and if *info* = 0, sub(*A*) contains the distributed matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:

for *ibtype* = 1 or 2, Z^T*sub(*B*)**Z* = *i*;

for *ibtype* = 3, Z^T*inv(sub(*B*))**Z* = *i*.

If *jobz* = 'N', then on exit the upper triangle (if *uplo*='U') or the lower triangle (if *uplo*='L') of sub(*A*), including the diagonal, is destroyed.

b On exit, if *info* ≤ *n*, the part of sub(*B*) containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization sub(*B*) = U^T**U* or sub(*B*) = *L***L*^{T}.

m (global) INTEGER. The total number of eigenvalues found, 0 ≤ *m* ≤ *n*.

nz (global) INTEGER.

Total number of eigenvectors computed. 0 ≤ *nz* ≤ *m*. The number of columns of *z* that are filled.

If *jobz* ≠ 'V', *nz* is not referenced.

If *jobz* = 'V', *nz* = *m* unless the user supplies insufficient space and p?sygvx is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in *z* (*m*.*le*.*descz*(*n_*)) and sufficient workspace to compute them. (See *lwork* below.) p?sygvx is always able to detect insufficient space without computation unless *range*.*eq*.'V'.

w (global)
REAL for pssygvx
DOUBLE PRECISION for pdsygvx.
Array, DIMENSION (*n*). On normal exit, the first *m* entries contain the selected eigenvalues in ascending order.

z (local).
REAL for pssygvx
DOUBLE PRECISION for pdsygvx.
global dimension (*n*, *n*), local dimension (*lld_z*, *LOCc*(*jz*+*n*-1)).
If *jobz* = 'V', then on normal exit the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
If *jobz* = 'N', then *z* is not referenced.

work If *jobz*='N' *work*(1) = optimal amount of workspace required to compute eigenvalues efficiently
If *jobz* = 'V' *work*(1) = optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality.
If *range*='V', it is assumed that all eigenvectors may be required.

ifail (global) INTEGER.
Array, DIMENSION (*n*).
ifail provides additional information when *info*.*ne*.0

If (*mod*(*info*/16,2).*ne*.0) then *ifail*(1) indicates the order of the smallest minor which is not positive definite. If (*mod*(*info*,2).*ne*.0) on exit, then *ifail* contains the indices of the eigenvectors that failed to converge.
If neither of the above error conditions hold and *jobz* = 'V', then the first *m* elements of *ifail* are set to zero.

iclustr (global) INTEGER.
Array, DIMENSION (2*NPROW*NPCOL). This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*(2**i*-1) to *iclustr*(2**i*), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. *iclustr*() is a zero terminated array. (*iclustr*(2**k*).*ne*.0.*and*. *iclustr*(2**k*+1).*eq*.0) if and only if *k* is the number of clusters *iclustr* is not referenced if *jobz* = 'N'.

gap (global)
REAL for pssygvx
DOUBLE PRECISION for pdsygvx.
Array, DIMENSION (NPROW*NPCOL). This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the *i*-th cluster may be as high as (*C***n*)/*gap*(*i*), where *C* is a small constant.

info (global) INTEGER.
If *info* = 0, the execution is successful.
If *info* <0: the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
If *info* > 0:
If (mod(*info*,2).*ne*.0), then one or more eigenvectors failed to converge. Their indices are stored in *ifail*.

If (mod(*info*,2,2).*ne*.0), then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array *iclustr*.
If (mod(*info*/4,2).*ne*.0), then space limit prevented p?sygvx from computing all of the eigenvectors between *vl* and *vu*. The number of eigenvectors computed is returned in *nz*.
If (mod(*info*/8,2).*ne*.0), then p?stebz failed to compute eigenvalues.
If (mod(*info*/16,2).*ne*.0), then *B* was not positive definite. *ifail(1)* indicates the order of the smallest minor which is not positive definite.

p?hegvx

Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian definite eigenproblem.

Syntax

call pchegvx(*ibtype*, *jobz*, *range*, *uplo*, *n*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *nz*, *w*, *orfac*, *z*, *iz*, *jz*, *descz*, *work*, *lwork*, *rwork*, *lrwork*, *iwork*, *liwork*, *ifail*, *iclustr*, *gap*, *info*)

call pzhegvx(*ibtype*, *jobz*, *range*, *uplo*, *n*, *a*, *ia*, *ja*, *desca*, *b*, *ib*, *jb*, *descb*, *vl*, *vu*, *il*, *iu*, *abstol*, *m*, *nz*, *w*, *orfac*, *z*, *iz*, *jz*, *descz*, *work*, *lwork*, *rwork*, *lrwork*, *iwork*, *liwork*, *ifail*, *iclustr*, *gap*, *info*)

Description

This routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian-definite eigenproblem, of the form

sub(*A*)**x* = λ*sub(*B*)**x*, sub(*A*)*sub(*B*)**x* = λ**x*, or sub(*B*)*sub(*A*)**x* = λ**x*.

Here sub (*A*) denoting *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) and sub(*B*) are assumed to be Hermitian and sub(*B*) denoting *B*(*ib*:*ib*+*n*-1, *jb*:*jb*+*n*-1) is also positive definite.

Input Parameters

ibtype
(global) INTEGER. Must be 1 or 2 or 3.
Specifies the problem type to be solved:
If *ibtype* = 1, the problem type is sub(*A*)**x* = *lambda**sub(*B*)**x*;
If *ibtype* = 2, the problem type is sub(*A*)*sub(*B*)**x* = *lambda***x*;
If *ibtype* = 3, the problem type is sub(*B*)*sub(*A*)**x* = *lambda***x*.

jobz
(global). CHARACTER*1. Must be 'N' or 'V'.
If *jobz* ='N', then compute eigenvalues only.
If *jobz* ='V', then compute eigenvalues and eigenvectors.

range
(global). CHARACTER*1. Must be 'A' or 'V' or 'I'.
If *range* = 'A', the routine computes all eigenvalues.
If *range* = 'V', the routine computes eigenvalues in the interval: [*vl*, *vu*]
If *range* = 'I', the routine computes eigenvalues with indices *il* through *iu*.

uplo
(global). CHARACTER*1. Must be 'U' or 'L'.
If *uplo* = 'U', arrays *a* and *b* store the upper triangles of sub(*A*) and sub (*B*);
If *uplo* = 'L', arrays *a* and *b* store the lower triangles of sub(*A*) and sub (*B*).

n
(global). INTEGER.
The order of the matrices sub(*A*) and sub (*B*) (*n* ≥ 0).

a
(local)
COMPLEX for pchegvx
DOUBLE COMPLEX for pzhegvx.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, this array contains the local pieces of the *n*-by-*n* Hermitian distributed matrix sub(*A*). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix.

ia, *ja*
(global) INTEGER.
The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca
(global and local) INTEGER array, dimension (*dlen_*).
The array descriptor for the distributed matrix *A*. If *desca*(*ctxt_*) is incorrect, p?hegvx cannot guarantee correct error reporting.

b
(local).
COMPLEX for pchegvx
DOUBLE COMPLEX for pzhegvx.
Pointer into the local memory to an array of dimension (*lld_b*, *LOCc*(*jb*+*n*–1)). On entry, this array contains the local pieces of the *n*-by-*n* Hermitian distributed matrix sub(*B*).
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*B*) contains the upper triangular part of the matrix.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*B*) contains the lower triangular part of the matrix.

ib, *jb*
(global) INTEGER.
The row and column indices in the global array *b* indicating the first row and the first column of the submatrix *B*, respectively.

descb
(global and local) INTEGER array, dimension (*dlen_*).
The array descriptor for the distributed matrix *B*.
descb(*ctxt_*) must be equal to *desca*(*ctxt_*).

vl, *vu*
(global)
REAL for pchegvx
DOUBLE PRECISION for pzhegvx.
If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu*
(global)
INTEGER.
If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint:
il ≥ 1, min(*il*, *n*) ≤ *iu* ≤ *n*

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol (global)
REAL for pchegvx
DOUBLE PRECISION for pzhegvx.
If *jobz*='V', setting *abstol* to p?lamch(*context*, 'U') yields the most orthogonal eigenvectors.
The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [*a*,*b*] of width less than or equal to
abstol + *eps**max(|*a*|,|*b*|),
where *eps* is the machine precision. If *abstol* is less than or equal to zero, then *eps**norm(T) will be used in its place, where norm(T) is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form.
Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold 2*p?lamch('S') not zero. If this routine returns with
((mod(*info*,2).*ne*.0).*or*. * (*mod*(*info*/8,2).*ne*.0)),
indicating that some eigenvalues or eigenvectors did not converge, try setting *abstol* to 2*p?lamch('S').

orfac (global).
REAL for pchegvx
DOUBLE PRECISION for pzhegvx.
Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within *tol*=*orfac**norm(*A*) of each other are to be reorthogonalized. However, if the workspace is insufficient (see *lwork*), *tol* may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if *orfac* equals zero. A default value of 1.0E-3 is used if *orfac* is negative. *orfac* should be identical on all processes.

iz, *jz* (global) INTEGER. The row and column indices in the global array *z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *Z*. *descz*(*ctxt_*) must equal *desca*(*ctxt_*).

work (local)
COMPLEX for pchegvx
DOUBLE COMPLEX for pzhegvx.
Workspace array, dimension (*lwork*)

lwork (local).
INTEGER. The dimension of the array *work*.
If only eigenvalues are requested:

lwork ≥ *n*+ max(NB*(*np*0 + 1), 3)

If eigenvectors are requested:

lwork ≥ *n* + (*np*0+ *mq*0 + NB)*NB

with *nq*0 = numroc(*nn*, NB, 0, 0, NPCOL).
For optimal performance, greater workspace is needed, that is

lwork ≥ max(*lwork*, *n*, *nhetrd_lwopt*, *nhegst_lwopt*)

where *lwork* is as defined above, and

nhetrd_lwork = 2*(*anb*+1)*(4**nps*+2) + (*nps* + 1)**nps*;
nhegst_lwopt = 2**np*0**nb* + *nq*0**nb* + *nb***nb*
nb = *desca*(*mb_*)
*np*0 = numroc(*n*, *nb*, 0, 0, NPROW)
*nq*0 = numroc(*n*, *nb*, 0, 0, NPCOL)
ictxt = *desca*(*ctxt_*)
anb = pjlaenv(*ictxt*, 3, 'p?hettrd', 'L', 0, 0, 0, 0)
sqnpc = *sqrt*(*dble*(NPROW * NPCOL))
nps = max(numroc(*n*, 1, 0, 0, *sqnpc*), 2**anb*)

numroc is a ScaLAPACK tool functions;
pjlaenv is a ScaLAPACK environmental inquiry function
MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = –1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each

of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxerbla.

rwork (local)
REAL for pchegvx
DOUBLE PRECISION for pzhegvx.
Workspace array, DIMENSION (*lrwork*).

lrwork (local) INTEGER. The dimension of the array *rwork*.
See below for definitions of variables used to define *lrwork*.
If no eigenvectors are requested (*jobz* = 'N'), then

lrwork ≥ 5*nn*+4*n*

If eigenvectors are requested (*jobz* = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is:

lrwork ≥ 4*n* + max(5*nn*, *np*0*mq*0)+ iceil(*neig*, NPROW*NPCOL)**nn*

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality (at the cost of potentially poor performance) you should add the following value to *lrwork*:
(*clustersize*-1)**n*,
where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:

{*w*(*k*),..., *w*(*k*+*clustersize*-1)| *w*(*j*+1) ≤ *w*(*j*)+*orfac**2*norm*(*A*)}

Variable definitions:
neig = number of eigenvectors requested;
nb = *desca*(*mb_*) = *desca*(*nb_*) = *descz*(*mb_*) = *descz*(*nb_*);
nn = max(*n*, *nb*, 2);
desca(*rsrc_*) = *desca*(*nb_*) = *descz*(*rsrc_*) = *descz*(*csrc_*) = 0 ;
*np*0 = numroc(*nn*, *nb*, 0, 0, NPROW);
*mq*0 = numroc(max(*neig*, *nb*, 2), *nb*, 0, 0, NPCOL);
iceil(*x*, *y*) is a ScaLAPACK function returning ceiling(*x*/*y*).

When *lrwork* is too small:
If *lwork* is too small to guarantee orthogonality, p?hegvx attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.
If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -25 is returned. Note that when *range*='V', p?hegvx does not know how many eigenvectors are requested until the eigenvalues are computed. Therefore, when *range*='V' and as long as *lwork* is large enough to allow p?hegvx to compute the eigenvalues, p?hegvx will compute the eigenvalues and as many eigenvectors as it can.
Relationship between workspace, orthogonality & performance:
If *clustersize* > *n*/*sqrt*(NPROW*NPCOL), then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. In the limit (that is, *clustersize* = *n*-1) p?stein will perform no better than ?stein on 1 processor.
For *clustersize* = *n*/*sqrt*(NPROW*NPCOL) reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.
For *clustersize* > *n*/*sqrt*(NPROW*NPCOL) execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.
If *lwork* = -1, then *lrwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxerbla.

iwork (local) INTEGER. Workspace array.

liwork (local) INTEGER, dimension of *iwork*.

liwork ≥ 6**nnp*
Where: *nnp* = max(*n*, NPROW*NPCOL + 1, 4)

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a

On exit, if *jobz* = 'V', then if *info* = 0, sub(*A*) contains the distributed matrix *Z* of eigenvectors.
The eigenvectors are normalized as follows:
If *ibtype* = 1 or 2, then Z^H*sub(*B*)**Z* = *i*;
If *ibtype* = 3, then Z^H*inv(sub(*B*))**Z* = *i*.
If *jobz* = 'N', then on exit the upper triangle (if *uplo*='U') or the lower triangle (if *uplo*='L') of sub(*A*), including the diagonal, is destroyed.

b

On exit, if *info* ≤ *n*, the part of sub(*B*) containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization sub(*B*) = U^H**U*, or sub(*B*) = *L***L*^{H}.

m

(global) INTEGER. The total number of eigenvalues found, 0 ≤ *m* ≤ *n*.

nz

(global) INTEGER. Total number of eigenvectors computed. 0 < *nz* < *m*. The number of columns of *z* that are filled.
If *jobz* ≠ 'V', *nz* is not referenced.
If *jobz* = 'V', *nz* = *m* unless the user supplies insufficient space and p?hegvx is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in *z* (*m*, *le*, *descz*(*n_*)) and sufficient workspace to compute them. (See *lwork* below.) The routine p?hegvx is always able to detect insufficient space without computation unless *range* = 'V'.

w

(global)
REAL for pchegvx
DOUBLE PRECISION for pzhegvx.

	Array, DIMENSION (*n*). On normal exit, the first *m* entries contain the selected eigenvalues in ascending order.
z	(local). COMPLEX for pchegvx DOUBLE COMPLEX for pzhegvx. global dimension (*n*, *n*), local dimension (*lld_z*, *LOCc*(*jz*+*n*-1)). If *jobz* = 'V', then on normal exit the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*. If *jobz* = 'N', then *z* is not referenced.
work	On exit, *work*(1) returns the optimal amount of workspace.
rwork	On exit, *rwork*(1) contains the amount of workspace required for optimal efficiency If *jobz*='N' *rwork*(1) = optimal amount of workspace required to compute eigenvalues efficiently If *jobz*='V' *rwork*(1) = optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality. If *range*='V', it is assumed that all eigenvectors may be required when computing optimal workspace.

ifail	(global) INTEGER.
	Array, DIMENSION (*n*).
	ifail provides additional information when *info*.*ne*.0
	If (*mod*(*info*/16,2).*ne*.0), then *ifail*(1) indicates the order of the smallest minor which is not positive definite.
	If (*mod*(*info*,2).*ne*.0) on exit, then *ifail*(1) contains the indices of the eigenvectors that failed to converge.
	If neither of the above error conditions are held, and *jobz* = 'V', then the first *m* elements of *ifail* are set to zero.

iclustr	(global) INTEGER. Array, DIMENSION (2*NPROW*NPCOL). This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to

insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*(2*i*–1) to *iclustr*(2*i*), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal.
iclustr() is a zero terminated array.
(*iclustr*(2**k*).*ne*.0.*and*.*clustr*(2**k*+1).*eq*.0) if and only if *k* is the number of clusters.
iclustr is not referenced if *jobz* = 'N'.

gap (global)
REAL for pchegvx
DOUBLE PRECISION for pzhegvx.
Array, DIMENSION (NPROW*NPCOL).
This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the *i*-th cluster may be as high as (*C***n*)/*gap*(*i*), where *C* is a small constant.

info (global) INTEGER.
If *info* = 0, the execution is successful.
If *info* <0: the *i*–th argument is an array and the *j*–entry had an illegal value, then *info* = –(*i*100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = –*i*.
If *info* > 0:
If (mod(*info*,2).*ne*.0), then one or more eigenvectors failed to converge. Their indices are stored in *ifail*.
If (mod(*info*,2,2).*ne*.0), then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array *iclustr*.
If (mod(*info*/4,2).*ne*.0), then space limit prevented p?sygvx from computing all of the eigenvectors between *vl* and *vu*. The number of eigenvectors computed is returned in *nz*.

If (mod(*info*/8,2).*ne*.0), then p?stebz failed to compute eigenvalues.
If (mod(*info*/16,2).*ne*.0), then *B* was not positive definite. *ifail(1*) indicates the order of the smallest minor which is not positive definite.

ScaLAPACK Auxiliary and Utility Routines 7

This chapter describes the Intel® Math Kernel Library implementation of ScaLAPACK Auxiliary Routines and Utility Functions and Routines. The library includes routines for both real and complex data.

NOTE. ScaLAPACK routines are provided with Intel® Cluster MKL product only which is a superset of Intel MKL.

Routine naming conventions, mathematical notation, and matrix storage schemes used for ScaLAPACK auxiliary and utility routines are the same as described in previous chapters. Some routines and functions may have combined character codes, such as `sc` or `dz`. For example, the routine `pscsum1` uses a complex input array and returns a real value.

Auxiliary Routines

Table 7-1 ScaLAPACK Auxiliary Routines

Routine Name	Data Types	Description
p?lacgv	c,z	Conjugates a complex vector.
p?max1	c,z	Finds the index of the element whose real part has maximum absolute value (similar to the Level 1 PBLAS p?amax, but using the absolute value to the real part).
?combamax1	c,z	Finds the element with maximum real part absolute value and its corresponding global index.
p?sum1	sc,dz	Forms the 1-norm of a complex vector similar to Level 1 PBLAS p?asum, but using the true absolute value.
p?dbtrsv	s,d,c,z	Computes an *LU* factorization of a general tridiagonal matrix with no pivoting. The routine is called by p?dbtrs.
p?dttrsv	s,d,c,z	Computes an *LU* factorization of a general band matrix, using partial pivoting with row interchanges. The routine is called by p?dttrs.

Routine Name	Data Types	Description
p?gebd2	s,d,c,z	Reduces a general rectangular matrix to real bidiagonal form by an orthogonal/unitary transformation (unblocked algorithm).
p?gehd2	s,d,c,z	Reduces a general matrix to upper Hessenberg form by an orthogonal/unitary similarity transformation (unblocked algorithm).
p?gelq2	s,d,c,z	Computes an *LQ* factorization of a general rectangular matrix (unblocked algorithm).
p?geql2	s,d,c,z	Computes a *QL* factorization of a general rectangular matrix (unblocked algorithm).
p?geqr2	s,d,c,z	Computes a *QR* factorization of a general rectangular matrix (unblocked algorithm).
p?gerq2	s,d,c,z	Computes an *RQ* factorization of a general rectangular matrix (unblocked algorithm).
p?getf2	s,d,c,z	Computes an *LU* factorization of a general matrix, using partial pivoting with row interchanges (local blocked algorithm).
p?labrd	s,d,c,z	Reduces the first *nb* rows and columns of a general rectangular matrix A to real bidiagonal form by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A.
p?lacon	s,d,c,z	Estimates the 1-norm of a square matrix, using the reverse communication for evaluating matrix-vector products.
p?laconsb	s,d	Looks for two consecutive small subdiagonal elements.
p?lacp2	s,d,c,z	Copies all or part of a distributed matrix to another distributed matrix.
p?lacp3	s,d	Copies from a global parallel array into a local replicated array or vice versa.

Routine Name	Data Types	Description
p?lacpy	s,d,c,z	Copies all or part of one two-dimensional array to another.
p?laevswp	s,d,c,z	Moves the eigenvectors from where they are computed to ScaLAPACK standard block cyclic array.
p?lahrd	s,d,c,z	Reduces the first nb columns of a general rectangular matrix A so that elements below the k^{th} subdiagonal are zero, by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A.
p?laiect	s,d,c,z	Exploits IEEE arithmetic to accelerate the computations of eigenvalues. (C interface function).
p?lange	s,d,c,z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a general rectangular matrix.
p?lanhs	s,d,c,z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of an upper Hessenberg matrix.
p?lansy, p?lanhe	s,d,c,z/c,z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a real symmetric or complex Hermitian matrix.
p?lantr	s,d,c,z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a triangular matrix.
p?lapiv	s,d,c,z	Applies a permutation matrix to a general distributed matrix, resulting in row or column pivoting.
p?laqge	s,d,c,z	Scales a general rectangular matrix, using row and column scaling factors computed by p?geequ.
p?laqsy	s,d,c,z	Scales a symmetric/Hermitian matrix, using scaling factors computed by p?poequ.

Routine Name	Data Types	Description
p?lared1d	s,d	Redistributes an array assuming that the input array *bycol* is distributed across rows and that all process columns contain the same copy of *bycol*.
p?lared2d	s,d	Redistributes an array assuming that the input array *byrow* is distributed across columns and that all process rows contain the same copy of *byrow* .
p?larf	s,d,c,z	Applies an elementary reflector to a general rectangular matrix.
p?larfb	s,d,c,z	Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.
p?larfc	c,z	Applies the conjugate transpose of an elementary reflector to a general matrix.
p?larfg	s,d,c,z	Generates an elementary reflector (Householder matrix).
p?larft	s,d,c,z	Forms the triangular vector *T* of a block reflector $H=I-VTV^H$
p?larz	s,d,c,z	Applies an elementary reflector as returned by p?tzrzf to a general matrix.
p?larzb	s,d,c,z	Applies a block reflector or its transpose/conjugate-transpose as returned by p?tzrzf to a general matrix.
p?larzc	c,z	Applies (multiplies by) the conjugate transpose of an elementary reflector as returned by p?tzrzf to a general matrix.
p?larzt	s,d,c,z	Forms the triangular factor *T* of a block reflector $H=I-VTV^H$ as returned by p?tzrzf.
p?lascl	s,d,c,z	Multiplies a general rectangular matrix by a real scalar defined as C_{to}/C_{from}.
p?laset	s,d,c,z	Initializes the off-diagonal elements of a matrix to α and the diagonal elements to β.

Routine Name	Data Types	Description
p?lasmsub	s,d	Looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero.
p?lassq	s,d,c,z	Updates a sum of squares represented in scaled form.
p?laswp	s,d,c,z	Performs a series of row interchanges on a general rectangular matrix.
p?latra	s,d,c,z	Computes the trace of a general square distributed matrix.
p?latrd	s,d,c,z	Reduces the first *nb* rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.
p?latrz	s,d,c,z	Reduces an upper trapezoidal matrix to upper triangular form by means of orthogonal/unitary transformations.
p?lauu2	s,d,c,z	Computes the product UU^H or L^HL, where U and L are upper or lower triangular matrices (local unblocked algorithm).
p?lauum	s,d,c,z	Computes the product UU^H or L^HL, where U and L are upper or lower triangular matrices.
p?lawil	s,d	Forms the Wilkinson transform.
p?org2l/p?ung2l	s,d,c,z	Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by p?geqlf (unblocked algorithm).
p?org2r/p?ung2r	s,d,c,z	Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by p?geqrf (unblocked algorithm).
p?orgl2/p?ungl2	s,d,c,z	Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by p?gelqf (unblocked algorithm).
p?orgr2/p?ungr2	s,d,c,z	Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by p?gerqf (unblocked algorithm).

Routine Name	Data Types	Description
p?orm2l/p?unm2l	s,d,c,z	Multiplies a general matrix by the orthogonal/unitary matrix from a *QL* factorization determined by p?geqlf (unblocked algorithm).
p?orm2r/p?unm2r	s,d,c,z	Multiplies a general matrix by the orthogonal/unitary matrix from a *QR* factorization determined by p?geqrf (unblocked algorithm).
p?orml2/p?unml2	s,d,c,z	Multiplies a general matrix by the orthogonal/unitary matrix from an *LQ* factorization determined by p?gelqf (unblocked algorithm).
p?ormr2/p?unmr2	s,d,c,z	Multiplies a general matrix by the orthogonal/unitary matrix from an *RQ* factorization determined by p?gerqf (unblocked algorithm).
p?pbtrsv	s,d,c,z	Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a banded matrix computed by p?pbtrf.
p?pttrsv	s,d,c,z	Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a tridiagonal matrix computed by p?pttrf.
p?potf2	s,d,c,z	Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix (local unblocked algorithm).
p?rscl	s,d,cs,zd	Multiplies a vector by the reciprocal of a real scalar.
p?sygs2/p?hegs2	s,d,c,z	Reduces a symmetric/Hermitian definite generalized eigenproblem to standard form, using the factorization results obtained from p?potrf (local unblocked algorithm).
p?sytd2/p?hetd2	s,d,c,z	Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (local unblocked algorithm).
p?trti2	s,d,c,z	Computes the inverse of a triangular matrix (local unblocked algorithm).

Routine Name	Data Types	Description
?lamsh	s,d	Sends multiple shifts through a small (single node) matrix to maximize the number of bulges that can be sent through.
?laref	s,d	Applies Householder reflectors to matrices on either their rows or columns.
?lasorte	s,d	Sorts eigenpairs by real and complex data types.
?lasrt2	s,d	Sorts numbers in increasing or decreasing order.
?stein2	s,d	Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix, using inverse iteration.
?dbtf2	s,d,c,z	Computes an *LU* factorization of a general band matrix with no pivoting (local unblocked algorithm).
?dbtrf	s,d,c,z	Computes an *LU* factorization of a general band matrix with no pivoting (local blocked algorithm).
?dttrf	s,d,c,z	Computes an *LU* factorization of a general tridiagonal matrix with no pivoting (local blocked algorithm).
?dttrsv	s,d,c,z	Solves a general tridiagonal system of linear equations using the *LU* factorization computed by ?dttrf.
?pttrsv	s,d,c,z	Solves a symmetric (Hermitian) positive-definite tridiagonal system of linear equations, using the LDL^H factorization computed by ?pttrf.
?steqr2	s,d	Computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit *QL* or *QR* method.

p?lacgv

Conjugates a complex vector.

Syntax

```
call pclacgv(n, x, ix, jx, descx, incx)
call pzlacgv(n, x, ix, jx, descx, incx)
```

Description

The routine conjugates a complex vector of length *n*, sub(*x*), where sub(*x*) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx* = *descx*(*m_*) and *X*(*ix*:*ix*+*n*-1, *jx*) if *incx* = 1.

Input Parameters

n	(global) INTEGER. The length of the distributed vector sub(*x*).
x	(local). COMPLEX for pclacgv COMPLEX*16 for pzlacgv.Pointer into the local memory to an array of DIMENSION (*lld_x*,*). On entry the vector to be conjugated *x*(*i*) = *X*(*ix*+(*jx*-1)*m_x*+(*i*-1)**incx*), 1 ≤ *i* ≤ *n*.
ix	(global) INTEGER.The row index in the global array *x* indicating the first row of sub(*x*).
jx	(global) INTEGER. The column index in the global array *x* indicating the first column of sub(*x*).
descx	(global and local) INTEGER. Array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *X*.
incx	(global) INTEGER.The global increment for the elements of *X*. Only two values of *incx* are supported in this version, namely 1 and *m_x*. *incx* must not be zero.

Output Parameters

x	(local). On exit, the conjugated vector.

p?max1

Finds the index of the element whose real part has maximum absolute value (similar to the Level 1 PBLAS p?amax*, but using the absolute value to the real part).*

Syntax

```
call pcmax1(n, amax, indx, x, ix, jx, descx, incx)
call pzmax1(n, amax, indx, x, ix, jx, descx, incx)
```

Description

This routine computes the global index of the maximum element in absolute value of a distributed vector sub(*x*). The global index is returned in *indx* and the value is returned in *amax*, where sub(*x*) denotes *X*(*ix*:*ix*+*n*-1, *jx*) if *incx* = 1, *X*(*ix*, *jx*:*jx*+*n*-1) if *incx* = *m_x*.

Input Parameters

n	(global) pointer to INTEGER. The number of components of the distributed vector sub(*x*). *n* ≥ 0.
x	(local) COMPLEX for pcmax1. COMPLEX*16 for pzmax1 Array containing the local pieces of a distributed matrix of dimension of at least ((*jx*-1)**m_x* + *ix* + (*n* - 1)*abs(*incx*)). This array contains the entries of the distributed vector sub (*x*).
ix	(global) INTEGER.The row index in the global array *X* indicating the first row of sub(*x*).
jx	(global) INTEGER. The column index in the global array *X* indicating the first column of sub(*x*)
descx	(global and local) INTEGER. Array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *X*.
incx	(global) INTEGER.The global increment for the elements of *X*. Only two values of *incx* are supported in this version, namely 1 and *m_x*. *incx* must not be zero.

Output Parameters

amax	(global output) pointer to REAL.The absolute value of the largest entry of the distributed vector sub(x) only in the scope of sub(x).
indx	(global output) pointer to INTEGER.The global index of the element of the distributed vector sub(x) whose real part has maximum absolute value.

?combamax1

Finds the element with maximum real part absolute value and its corresponding global index.

Syntax

```
call ccombamax1(v1, v2)
call zcombamax1(v1, v2)
```

Description

This routine finds the element having maximum real part absolute value as well as its corresponding global index.

Input Parameters

v1	(local) COMPLEX for ccombamax1 COMPLEX*16 for zcombamax1 Array, DIMENSION 2. The first maximum absolute value element and its global index. *v1*(1) = amax, *v1*(2) = indx.
v2	(local) COMPLEX for ccombamax1 COMPLEX*16 for zcombamax1 Array, DIMENSION 2. The second maximum absolute value element and its global index. *v2*(1) = amax, *v2*(2) = indx.

Output Parameters

v1	(local). The first maximum absolute value element and its global index. *v1*(1) = amax, *v1*(2) = indx.

p?sum1

Forms the 1-norm of a complex vector similar to Level 1 PBLAS p?asum*, but using the true absolute value.*

Syntax

call pscsum1(*n*, *asum*, *x*, *ix*, *jx*, *descx*, *incx*)

call pdzsum1(*n*, *asum*, *x*, *ix*, *jx*, *descx*, *incx*)

Description

This routine returns the sum of absolute values of a complex distributed vector sub(*x*) in *asum*, where sub(*x*) denotes *X*(*ix*:*ix*+*n*-1, *jx*:*jx*), if *incx* = 1, *X*(*ix*:*ix*, *jx*:*jx*+*n*-1), if *incx* = *m_x*.

Based on p?asum from the Level 1 PBLAS. The change is to use the 'genuine' absolute value.

Input Parameters

n	(global) pointer to INTEGER. The number of components of the distributed vector sub(*x*). *n* ≥ 0.
x	(local) COMPLEX for pscsum1 COMPLEX*16 for pdzsum1. Array containing the local pieces of a distributed matrix of dimension of at least ((*jx*-1)*m_x + *ix* + (*n* - 1)*abs(*incx*)). This array contains the entries of the distributed vector sub (*x*).
ix	(global) INTEGER.The row index in the global array *X* indicating the first row of sub(*x*).
jx	(global) INTEGER. The column index in the global array *X* indicating the first column of sub(*x*)

descx	(global and local) INTEGER. Array, DIMENSION 8. The array descriptor for the distributed matrix *X*.
incx	(global) INTEGER.The global increment for the elements of *X*. Only two values of *incx* are supported in this version, namely 1 and *m_x*.

Output Parameters

asum	(local) Pointer to REAL. The sum of absolute values of the distributed vector sub(*x*) only in its scope.

p?dbtrsv

Computes an LU factorization of a general triangular matrix with no pivoting. The routine is called by p?dbtrs.

Syntax

call psdbtrsv(*uplo*, *trans*, *n*, *bwl*, *bwu*, *nrhs*, *a*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

call pddbtrsv(*uplo*, *trans*, *n*, *bwl*, *bwu*, *nrhs*, *a*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

call pcdbtrsv(*uplo*, *trans*, *n*, *bwl*, *bwu*, *nrhs*, *a*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

call pzdbtrsv(*uplo*, *trans*, *n*, *bwl*, *bwu*, *nrhs*, *a*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

Description

This routines solves a banded triangular system of linear equations

$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs)$ or

$A(1:n, ja:ja+n-1)^T * X = B(ib:ib+n-1, 1:nrhs)$ (for real flavors); $A(1:n, ja:ja+n-1)^H * X = B(ib:ib+n-1, 1:nrhs)$ (for complex flavors),

where $A(1:n, ja:ja+n-1)$ is a banded triangular matrix factor produced by the Gaussian elimination code PD@(dom_pre)BTRF and is stored in $A(1:n, ja:ja+n-1)$ and af. The matrix stored in $A(1:n, ja:ja+n-1)$ is either upper or lower triangular according to $uplo$, and the choice of solving $A(1:n, ja:ja+n-1)$ or $A(1:n, ja:ja+n-1)^T$ is dictated by the user by the parameter $trans$.

Routine p?dbtrf must be called first.

Input Parameters

uplo	(global) CHARACTER. If *uplo*='U', the upper triangle of $A(1:n, ja:ja+n-1)$ is stored, if *uplo* = 'L', the lower triangle of $A(1:n, ja:ja+n-1)$ is stored.
trans	(global) CHARACTER. If *trans* = 'N', solve with $A(1:n, ja:ja+n-1)$, if *trans* = 'C', solve with conjugate transpose $A(1:n, ja:ja+n-1)$.
n	(global) INTEGER. The order of the distributed submatrix A;($n \geq 0$).
bwl	(global) INTEGER. Number of subdiagonals. $0 \leq bwl \leq n-1$.
bwu	(global) INTEGER. Number of subdiagonals. $0 \leq bwu \leq n-1$.
nrhs	(global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix B ($nrhs \geq 0$).
a	(local). REAL for psdbtrsv DOUBLE PRECISION for pddbtrsv COMPLEX for pcdbtrsv COMPLEX*16 for pzdbtrsv. Pointer into the local memory to an array with first DIMENSION $lld\_a \geq (bwl+bwu+1)$(stored in $desca$). On entry, this array contains the local pieces of the n-by-n unsymmetric banded distributed Cholesky factor L or $L^T*A(1:n, ja:ja+n-1)$. This local portion is stored in the packed

banded format used in LAPACK. Please see the Application Notes below and the ScaLAPACK manual for more detail on the format of distributed matrices.

ja (global) INTEGER. The index in the global array *a* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).

if 1*d* type (*dtype_a* = 501 or 502), *dlen*≥ 7;

if 2*d* type (*dtype_a* = 1), *dlen*≥ 9. The array descriptor for the distributed matrix *A*. Contains information of mapping of *A* to memory.

b (local)
REAL for psdbtrsv
DOUBLE PRECISION for pddbtrsv
COMPLEX for pcdbtrsv
COMPLEX*16 for pzdbtrsv.
Pointer into the local memory to an array of local lead DIMENSION *lld_b*≥*nb*. On entry, this array contains the local pieces of the right-hand sides *B*(*ib*:*ib*+*n*-1, 1:*nrhs*).

ib (global) INTEGER. The row index in the global array *b* that points to the first row of the matrix to be operated on (which may be either all of *b* or a submatrix of *B*).

descb (global and local) INTEGER array of DIMENSION (*dlen_*).

if 1*d* type (*dtype_b* =502), *dlen*≥7;

if 2*d* type (*dtype_b* =1), *dlen*≥9. The array descriptor for the distributed matrix *B*. Contains information of mapping *B* to memory.

laf (local)
INTEGER. Size of user-input Auxiliary Filling space *af*.

laf must be ≥*nb**(*bwl*+*bwu*)+6*max(*bwl*, *bwu*)*max(*bwl*, *bwu*). If *laf* is not large enough, an error code is returned and the minimum acceptable size will be returned in *af*(1).

work (local).
REAL for psdbtrsv
DOUBLE PRECISION for pddbtrsv

COMPLEX for pcdbtrsv
COMPLEX*16 for pzdbtrsv.
Temporary workspace. This space may be overwritten in between calls to routines.
work must be the size given in *lwork*.

lwork (local or global) INTEGER.
Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

lwork≥ max(*bwl*, *bwu*)*nrhs*.

Output Parameters

a (local).
This local portion is stored in the packed banded format used in LAPACK. Please see the ScaLAPACK manual for more detail on the format of distributed matrices.

b On exit, this contains the local piece of the solutions distributed matrix *X*.

af (local).
REAL for psdbtrsv
DOUBLE PRECISION for pddbtrsv
COMPLEX for pcdbtrsv
COMPLEX*16 for pzdbtrsv.
Auxiliary Filling Space. Filling is created during the factorization routine p?dbtrf and this is stored in *af*. If a linear system is to be solved using p?dbtrf after the factorization routine, *af* must not be altered after the factorization.

work On exit, *work*(1) contains the minimal *lwork*.

info (local).
INTEGER. If *info* = 0, the execution is successful.
< 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?dttrsv

Computes an LU factorization of a general band matrix, using partial pivoting with row interchanges. The routine is called by p?dttrs.

Syntax

call psdttrsv(*uplo*, *trans*, *n*, *nrhs*, *dl*, *d*, *du*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

call pddttrsv(*uplo*, *trans*, *n*, *nrhs*, *dl*, *d*, *du*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

call pcdttrsv(*uplo*, *trans*, *n*, *nrhs*, *dl*, *d*, *du*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

call pzdttrsv(*uplo*, *trans*, *n*, *nrhs*, *dl*, *d*, *du*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

Description

This routine solves a tridiagonal triangular system of linear equations

A(1 :*n*, *ja*:*ja*+*n*-1) * *X* = *B*(*ib*:*ib*+*n*-1, 1 :*nrhs*) or

A(1 :*n*, *ja*:*ja*+*n*-1)^T * *X* = *B*(*ib*:*ib*+*n*-1, 1 :*nrhs*) for real flavors; *A*(1 :*n*, *ja*:*ja*+*n*-1)^H * *X* = *B*(*ib*:*ib*+*n*-1, 1 :*nrhs*) for complex flavors,

where *A*(1 :*n*, *ja*:*ja*+*n*-1) is a tridiagonal matrix factor produced by the Gaussian elimination code PS@(dom_pre)TTRF and is stored in *A*(1 :*n*, *ja*:*ja*+*n*-1) and *af*.

The matrix stored in *A*(1 :*n*, *ja*:*ja*+*n*-1) is either upper or lower triangular according to *uplo*, and the choice of solving *A*(1 :*n*, *ja*:*ja*+*n*-1) or *A*(1 :*n*, *ja*:*ja*+*n*-1)^T is dictated by the user by the parameter *trans*.

Routine p?dttrf must be called first.

Input Parameters

uplo (global) CHARACTER.
If *uplo*='U', the upper triangle of *A*(1:*n*, *ja*:*ja*+*n*-1) is stored,
if *uplo* = 'L', the lower triangle of *A*(1:*n*, *ja*:*ja*+*n*-1) is stored.

trans (global) CHARACTER.
If *trans* = 'N', solve with *A*(1:*n*, *ja*:*ja*+*n*−1),
if *trans* = 'C', solve with conjugate transpose *A*(1:*n*, *ja*:*ja*+*n*−1).

n (global) INTEGER. The order of the distributed submatrix *A*; (*n*≥ 0).

nrhs (global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix *B*(*ib*:*ib*+*n*−1, 1:*nrhs*). (*nrhs*≥ 0).

dl (local).
REAL for psdttrsv
DOUBLE PRECISION for pddttrsv
COMPLEX for pcdttrsv
COMPLEX*16 for pzdttrsv.
Pointer to local part of global vector storing the lower diagonal of the matrix.
Globally, *dl*(1) is not referenced, and *dl* must be aligned with *d*.

Must be of size ≥*desca*(*nb_*).

d (local).
REAL for psdttrsv
DOUBLE PRECISION for pddttrsv
COMPLEX for pcdttrsv
COMPLEX*16 for pzdttrsv.
Pointer to local part of global vector storing the main diagonal of the matrix.

du (local).
REAL for psdttrsv
DOUBLE PRECISION for pddttrsv
COMPLEX for pcdttrsv
COMPLEX*16 for pzdttrsv.
Pointer to local part of global vector storing the upper diagonal of the matrix.
Globally, *du*(*n*) is not referenced, and *du* must be aligned with *d*.

ja	(global) INTEGER. The index in the global array *a* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).
desca	(global and local). INTEGER array of DIMENSION (*dlen_*). if 1*d* type (*dtype_a* = 501 or 502), *dlen*≥ 7; if 2*d* type (*dtype_a* = 1), *dlen*≥ 9. The array descriptor for the distributed matrix *A*. Contains information of mapping of *A* to memory.
b	(local) REAL for psdttrsv DOUBLE PRECISION for pddttrsv COMPLEX for pcdttrsv COMPLEX*16 for pzdttrsv. Pointer into the local memory to an array of local lead DIMENSION *lld_b*≥*nb*. On entry, this array contains the local pieces of the right-hand sides *B*(*ib*:*ib*+*n*−1, 1 :*nrhs*).
ib	(global). INTEGER. The row index in the global array *b* that points to the first row of the matrix to be operated on (which may be either all of *b* or a submatrix of *B*).
descb	(global and local).INTEGER array of DIMENSION (*dlen_*). if 1*d* type (*dtype_b* = 502), *dlen*≥7; if 2*d* type (*dtype_b* = 1), *dlen*≥ 9. The array descriptor for the distributed matrix *B*. Contains information of mapping *B* to memory.
laf	(local). INTEGER. Size of user-input Auxiliary Filling space *af*. *laf* must be ≥ 2*(*nb*+2). If *laf* is not large enough, an error code is returned and the minimum acceptable size will be returned in *af*(1).
work	(local). REAL for psdttrsv DOUBLE PRECISION for pddttrsv COMPLEX for pcdttrsv

COMPLEX*16 for pzdttrsv.
Temporary workspace. This space may be overwritten in between calls to routines.
work must be the size given in *lwork*.

lwork (local or global).INTEGER.
Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

lwork≥ 10**npcol*+4**nrhs*.

Output Parameters

dl (local).
On exit, this array contains information containing the factors of the matrix.

d On exit, this array contains information containing the factors of the matrix. Must be of size ≥*desca* (*nb_*).

b On exit, this contains the local piece of the solutions distributed matrix X.

af (local).
REAL for psdttrsv
DOUBLE PRECISION for pddttrsv
COMPLEX for pcdttrsv
COMPLEX*16 for pzdttrsv.
Auxiliary Filling Space. Filling is created during the factorization routine p?dttrf and this is stored in *af*. If a linear system is to be solved using p?dttrs after the factorization routine, *af* must not be altered after the factorization.

work On exit, *work*(1) contains the minimal *lwork*.

info (local). INTEGER.
If *info*=0, the execution is successful.
if *info*< 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?gebd2

Reduces a general rectangular matrix to real bidiagonal form by an orthogonal/unitary transformation (unblocked algorithm).

Syntax

```
call psgebd2(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pdgebd2(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pcgebd2(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pzgebd2(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
```

Description

This routine reduces a real/complex general *m*-by-*n* distributed matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) to upper or lower bidiagonal form *B* by an orthogonal/unitary transformation: *Q*' * sub(*A*) * *P* = *B*.

If *m* ≥ *n*, *B* is the upper bidiagonal; if *m* < *n*, *B* is the lower bidiagonal.

Input Parameters

m — (global) INTEGER.
The number of rows of the distributed submatrix sub(*A*). (*m* ≥ 0).

n — (global) INTEGER.
The order of the distributed submatrix sub(*A*). (*n* ≥ 0).

a — (local).
REAL for psgebd2
DOUBLE PRECISION for pdgebd2
COMPLEX for pcgebd2
COMPLEX*16 for pzgebd2.
Pointer into the local memory to an array of DIMENSION(*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the general distributed matrix sub(*A*).

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.

work (local).
REAL for psgebd2
DOUBLE PRECISION for pdgebd2
COMPLEX for pcgebd2
COMPLEX*16 for pzgebd2.
This is a workspace array of DIMENSION (*lwork*).

lwork (local or global) INTEGER.
The dimension of the array *work*.

lwork is local input and must be at least *lwork* ≥ max(*mpa*0, *nqa*0), where *nb* = *mb_a* = *nb_a*, *iroffa* = mod(*ia*-1, *nb*), *iarow* = indxg2p (*ia*, *nb*, *myrow*, *rsrc_a*, *nprow*), *iacol* = indxg2p (*ja*, *nb*, *mycol*, *csrc_a*, *npcol*), *mpa*0 = numroc(*m*+*iroffa*, *nb*, *myrow*, *iarow*, *nprow*), *nqa*0 = numroc(*n*+*icoffa*, *nb*, *mycol*, *iacol*, *npcol*). indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a (local).

On exit, if *m* ≥ *n*, the diagonal and the first superdiagonal of sub(*A*) are overwritten with the upper bidiagonal matrix *B*; the elements below the diagonal, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors, and the elements above the first

superdiagonal, with the array *taup*, represent the orthogonal matrix *P* as a product of elementary reflectors. If *m* < *n*, the diagonal and the first subdiagonal are overwritten with the lower bidiagonal matrix *B*; the elements below the first subdiagonal, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors, and the elements above the diagonal, with the array *taup*, represent the orthogonal matrix *P* as a product of elementary reflectors. See Applications Notes below.

d

(local)
REAL for psgebd2
DOUBLE PRECISION for pdgebd2
COMPLEX for pcgebd2
COMPLEX*16 for pzgebd2.

Array, DIMENSION *LOCc*(*ja*+min(*m*,*n*)-1) if *m* ≥ *n*; *LOCr*(*ia*+min(*m*,*n*)-1) otherwise. The distributed diagonal elements of the bidiagonal matrix *B*: *d*(*i*) = *a*(*i*,*i*). *d* is tied to the distributed matrix *A*.

e

(local)
REAL for psgebd2
DOUBLE PRECISION for pdgebd2
COMPLEX for pcgebd2
COMPLEX*16 for pzgebd2.

Array, DIMENSION *LOCc*(*ja*+min(*m*,*n*)-1) if *m* ≥ *n*; *LOCr*(*ia*+min(*m*,*n*)-2) otherwise. The distributed diagonal elements of the bidiagonal matrix *B*:

if *m* ≥ *n*, *e*(*i*) = *a*(*i*, *i*+1) for *i* = 1, 2, ... , *n*-1;
if *m* < *n*, *e*(*i*) = *a*(*i*+1, *i*) for *i* = 1, 2, ..., *m*-1.
e is tied to the distributed matrix *A*.

tauq

(local).
REAL for psgebd2
DOUBLE PRECISION for pdgebd2
COMPLEX for pcgebd2
COMPLEX*16 for pzgebd2.

	Array, DIMENSION*LOCc*(*ja*+min(*m*,*n*)-1). The scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Q*. *tauq* is tied to the distributed matrix *A*.
taup	(local). REAL for psgebd2 DOUBLE PRECISION for pdgebd2 COMPLEX for pcgebd2 COMPLEX*16 for pzgebd2. Array, DIMENSION *LOCr*(*ia*+min(*m*,*n*)-1). The scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *P*. *taup* is tied to the distributed matrix *A*.
work	On exit, *work*(1) returns the minimal and optimal *lwork*.
info	(local) INTEGER. If *info* = 0, the execution is successful. if *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrices *Q* and *P* are represented as products of elementary reflectors:

If *m*≥*n*,

Q = *H*(1) *H*(2) . . . *H*(*n*) and *P* = *G*(1) *G*(2). . . *G*(*n*-1)

Each *H*(i) and *G*(i) has the form:

H(i) = *I* - *tauq* **v* **v*' and *G*(i) = *I* - *taup* **u***u*',

where *tauq* and *taup* are real/complex scalars, and *v* and *u* are real/complex vectors. *v*(1:*i*-1) = 0, *v*(*i*) = 1, and *v*(*i*+*i*:*m*) is stored on exit in

A(*ia*+*i*-*ia*+*m*-1, *a*+*i*-1);

u(1:*i*) = 0, *u*(*i*+1) = 1, and *u*(*i*+2:*n*) is stored on exit in *A*(*ia*+*i*-1, *ja*+*i*+1:*ja*+*n*-1);

tauq is stored in *TAUQ*(*ja*+*i*-1) and *taup* in *TAUP*(*ia*+*i*-1).

If $m < n$,

$v(1: i) = 0$, $v(i+1) = 1$, and $v(i+2:m)$ is stored on exit in $A(ia+i+1: ia+m-1, ja+i-1)$;

$u(1: i-1) = 0$, $u(i) = 1$, and $u(i+1 :n)$ is stored on exit in $A(ia+i-1, ja+i:ja+n-1)$;

$tauq$ is stored in $TAUQ(ja+i-1)$ and $taup$ in $TAUP(ia+i-1)$.

The contents of sub(A) on exit are illustrated by the following examples:

$m = 6$ and $n = 5 (m > n)$:

$$\begin{bmatrix} d & e & u1 & u1 & u1 \\ v1 & d & e & u2 & u2 \\ v1 & v2 & d & e & u3 \\ v1 & v2 & v3 & d & e \\ v1 & v2 & v3 & v4 & d \\ v1 & v2 & v3 & v4 & v5 \end{bmatrix}$$

$m = 5$ and $n = 6 (m < n)$:

$$\begin{bmatrix} d & u1 & u1 & u1 & u1 & u1 \\ e & d & u2 & u2 & u2 & u2 \\ v1 & e & d & u3 & u3 & u3 \\ v1 & v2 & e & d & u4 & u4 \\ v1 & v2 & v3 & e & d & u5 \end{bmatrix}$$

where d and e denote diagonal and off-diagonal elements of B, vi denotes an element of the vector defining $H(i)$, and ui an element of the vector defining $G(i)$.

p?gehd2

Reduces a general matrix to upper Hessenberg form by an orthogonal/unitary similarity transformation (unblocked algorithm).

Syntax

```
call psgehd2(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pdgehd2(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pcgehd2(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pzgehd2(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
```

Description

This routine reduces a real/complex general distributed matrix sub(*A*) to upper Hessenberg form *H* by an orthogonal/unitary similarity transformation: *Q*' * sub(*A*) * *Q* = *H*, where sub(*A*) = *A*(*ia*+*n*-1 :*ia*+*n*-1, *ja*+*n*-1 :*ja*+*n*-1).

Input Parameters

n — (global) INTEGER. The order of the distributed submatrix *A*. (*n*≥ 0).

ilo, *ihi* — (global) INTEGER. It is assumed that sub(*A*) is already upper triangular in rows *ia*:*ia*+*ilo*-2 and *ia*+*ihi*:*ia*+*n*-1 and columns *ja*:*ja*+*jlo*-2 and *ja*+*jhi*:*ja*+*n*-1. See Application Notes for further information.

If *n*≥ 0, 1 ≤ *ilo* ≤ *ihi* ≤ *n*; otherwise set *ilo* = 1, *ihi* = *n*.

a — (local).
REAL for psgehd2
DOUBLE PRECISION for pdgehd2
COMPLEX for pcgehd2
COMPLEX*16 for pzgehd2.
Pointer into the local memory to an array of DIMENSION(*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the *n*-by-*n* general distributed matrix sub(*A*) to be reduced.

ia, *ja* — (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca — (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

work — (local).
REAL for psgehd2
DOUBLE PRECISION for pdgehd2
COMPLEX for pcgehd2
COMPLEX*16 for pzgehd2.
This is a workspace array of DIMENSION (*lwork*).

lwork — (local or global). INTEGER.

The dimension of the array *work*.

lwork is local input and must be at least *lwork*≥*nb* + max(*npa*0, *nb*), where *nb* = *mb_a* = *nb_a*, *iroffa* = mod(*ia*-1, *nb*), *iarow* = indxg2p (*ia*, *nb*, *myrow*, *rsrc_a*, *nprow*),*npa*0 = numroc(*ihi*+*iroffa*, *nb*, *myrow*, *iarow*, *nprow*).
indxg2p and numroc are ScaLAPACK tool functions;*myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a (local). On exit, the upper triangle and the first subdiagonal of sub(*A*) are overwritten with the upper Hessenberg matrix *H*, and the elements below the first subdiagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors. See Application Notes below.

tau (local).
REAL for psgehd2
DOUBLE PRECISION for pdgehd2
COMPLEX for pcgehd2
COMPLEX*16 for pzgehd2.
Array, DIMENSION *LOCc*(*ja*+*n*-2) The scalar factors of the elementary reflectors (see Application Notes below).
Elements *ja*:*ja*+*ilo*-2 and *ja*+*ihi*:*ja*+*n*-2 of *tau* are set to zero. *tau* is tied to the distributed matrix *A*.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local).INTEGER.
If *info* = 0, the execution is successful.

if *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of (*ihi-ilo*) elementary reflectors

Q = *H*(*ilo*) *H*(*ilo*+1) . . . *H*(*ihi*-1).

Each *H*(*i*) has the form

H(*i*) = *I* - *tau* **v* **v*',

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1: *i*) = 0, *v*(*i*+1) = 1 and *v*(*ihi*+1: *n*) = 0; *v*(*i*+2: *ihi*) is stored on exit in *A*(*ia*+*ilo*+*i*:*ia*+*ihi*-1, *ia*+*ilo*+*i*-2), and *tau* in *tau*(*ja*+*ilo*+*i*-2).

The contents of *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) are illustrated by the following example, with *n* = 7, *ilo* = 2 and *ihi* = 6:

on entry

$$\begin{bmatrix} a & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & & & & & & a \end{bmatrix}$$

on exit

$$\begin{bmatrix} a & a & h & h & h & h & a \\ & a & h & h & h & h & a \\ & h & h & h & h & h & h \\ & v2 & h & h & h & h & h \\ & v2 & v3 & h & h & h & h \\ & v2 & v3 & v4 & h & h & h \\ & & & & & & a \end{bmatrix}$$

where *a* denotes an element of the original matrix sub(*A*), *h* denotes a modified element of the upper Hessenberg matrix *H*, and *vi* denotes an element of the vector defining *H*(*ja*+*ilo*+*i*-2).

p?gelq2

Computes an LQ factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```
call psgelq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgelq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgelq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgelq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

This routine computes an *LQ* factorization of a real/complex distributed *m*-by-*n* matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) = *L* **Q*.

Input Parameters

m — (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(*A*). (*m*≥0).

n — (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*).
(*n*≥0).

a — (local).
REAL for psgelq2
DOUBLE PRECISION for pdgelq2
COMPLEX for pcgelq2
COMPLEX*16 for pzgelq2.
Pointer into the local memory to an array of DIMENSION(*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the *m*-by-*n* distributed matrix sub(*A*) which is to be factored.

ia, *ja* — (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

work (local).
REAL for psgelq2
DOUBLE PRECISION for pdgelq2
COMPLEX for pcgelq2
COMPLEX*16 for pzgelq2.
This is a workspace array of DIMENSION (*lwork*).

lwork (local or global) INTEGER.
The dimension of the array *work*.

lwork is local input and must be at least *lwork*≥*nq*0 + max(1, *mp*0), where *iroff* = mod(*ia*-1, *mb_a*), *icoff* = mod(*ja*-1, *nb_a*), *iarow* = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*), *iacol* = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*), *mp*0 = numroc(*m*+*iroff*, *mb_a*, *myrow*, *iarow*, *nprow*), *nq*0 = numroc(*n*+*icoff*, *nb_a*, *mycol*, *iacol*, *npcol*),
indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = −1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a (local).
On exit, the elements on and below the diagonal of sub(*A*) contain the *m* by min(*m*,*n*) lower trapezoidal matrix *L* (*L* is lower triangular if *m* ≤ *n*); the elements above the diagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see Application Notes below).

tau (local).
REAL for psgelq2

	DOUBLE PRECISION for pdgelq2 COMPLEX for pcgelq2 COMPLEX*16 for pzgelq2. Array, DIMENSION *LOCr*(*ia*+min(*m*, *n*)-1). This array contains the scalar factors of the elementary reflectors. *tau* is tied to the distributed matrix *A*.
work	On exit, *work*(1) returns the minimal and optimal *lwork*.
info	(local).INTEGER. If *info* = 0, the execution is successful. if *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*ia*+*k*-1) *H*(*ia*+*k*-2). . . *H*(*ia*) for real flavors,*Q* = *H*(*ia*+*k*-1)' *H*(*ia*+*k*-2)'. . . *H*(*ia*)' for complex flavors,

where *k* = min(*m*,*n*).

Each *H*(*i*) has the form

H(*i*) = *I* - *tau* * *v* * *v*'

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1: *i*-1) = 0 and *v*(*i*) = 1; *v*(*i*+1: *n*) (for real flavors) or conjg(*v*(*i*+1: *n*)) (for complex flavors) is stored on exit in *A*(*ia*+*i*-1,*ja*+*i*:*ja*+*n*-1), and *tau* in *TAU*(*ia*+*i*-1).

p?geql2

Computes a QL factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```
call psgeql2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgeql2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgeql2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgeql2(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine computes a *QL* factorization of a real/complex distributed *m*-by-*n* matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1)= *Q* **L*.

Input Parameters

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(*A*). (*m*≥ 0).

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*). (*n*≥ 0).

a (local).
REAL for psgeql2
DOUBLE PRECISION for pdgeql2
COMPLEX for pcgeql2
COMPLEX*16 for pzgeql2.
Pointer into the local memory to an array of DIMENSION (*lld_a*,*LOCc* (*ja*+*n*-1)).
On entry, this array contains the local pieces of the *m*-by-*n* distributed matrix sub(*A*) which is to be factored.

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.

work (local).
REAL for psgeql2
DOUBLE PRECISION for pdgeql2
COMPLEX for pcgeql2
COMPLEX*16 for pzgeql2.
This is a workspace array of DIMENSION (*lwork*).

lwork (local or global) INTEGER.
The dimension of the array *work*.

lwork is local input and must be at least *lwork*≥*mp*0 + max(1, *nq*0), where *iroff* = mod(*ia*-1, *mb_a*), *icoff* = mod(*ja*-1, *nb_a*), *iarow* = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*), *iacol* = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*), *mp*0 = numroc(*m*+*iroff*, *mb_a*, *myrow*, *iarow*, *nprow*), *nq*0 = numroc(*n*+*icoff*, *nb_a*, *mycol*, *iacol*, *npcol*),
indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a (local).
On exit,
if *m* ≥ *n*, the lower triangle of the distributed submatrix *A*(*ia*+*m*-*n*:*ia*+*m*-1, *ja*:*ja*+*n*-1) contains the *n*-by-*n* lower triangular matrix *L*; if *m* ≤ *n*, the elements on and below the (*n*-*m*)-th superdiagonal contain the *m*-by-*n* lower trapezoidal matrix *L*; the remaining elements, with the array *tau*, represent the orthogonal/ unitary matrix *Q* as a product of elementary reflectors (see Application Notes below).

tau (local).
REAL for psgeql2
DOUBLE PRECISION for pdgeql2
COMPLEX for pcgeql2
COMPLEX*16 for pzgeql2.
Array, DIMENSION *LOCc*(*ja*+*n*-1). This array contains the scalar factors of the elementary reflectors. *tau* is tied to the distributed matrix *A*.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local). INTEGER.

If *info* = 0, the execution is successful. if *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*ja*+*k*-1) . . . *H*(*ja*+1) *H*(*ja*), where *k* = min(*m*,*n*).

Each *H*(*i*) has the form

H(*i*) = *I*- *tau* * *v* * *v*'

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(*m*-*k*+*i*+1: *m*) = 0 and *v*(*m*-*k*+*i*) = 1; *v*(1: *m*-*k*+*i*-1) is stored on exit in *A*(*ia*:*ia*+*m*-*k*+*i*-2, *ja*+*n*-*k*+*i*-1), and *tau* in *TAU*(*ja*+*n*-*k*+*i*-1).

p?geqr2

Computes a QR factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```
call psgeqr2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgeqr2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgeqr2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgeqr2(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

This routine computes a *QR* factorization of a real/complex distributed *m*-by-*n* matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1)= *Q***R*.

Input Parameters

m (global). INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(*A*). (*m*≥0).

n	(global).INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*). (*n*≥0).
a	(local). REAL for psgeqr2 DOUBLE PRECISION for pdgeqr2 COMPLEX for pcgeqr2 COMPLEX*16 for pzgeqr2. Pointer into the local memory to an array of DIMENSION (*lld_a*, *LOCc* (*ja*+*n*-1)). On entry, this array contains the local pieces of the *m*-by-*n* distributed matrix sub(*A*) which is to be factored.
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.
desca	(global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.
work	(local). REAL for psgeqr2 DOUBLE PRECISION for pdgeqr2 COMPLEX for pcgeqr2 COMPLEX*16 for pzgeqr2. This is a workspace array of DIMENSION (*lwork*).
lwork	(local or global). INTEGER. The dimension of the array *work*. *lwork* is local input and must be at least *lwork*≥*mp*0 + max(1, *nq*0), where *iroff* = mod(*ia*-1, *mb_a*), *icoff* = mod(*ja*-1, *nb_a*), *iarow* = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*), *iacol* = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*), *mp*0 = numroc(*m*+*iroff*, *mb_a*, *myrow*, *iarow*, *nprow*), *nq*0 = numroc(*n*+*icoff*, *nb_a*, *mycol*, *iacol*, *npcol*), indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a (local).
On exit, the elements on and above the diagonal of sub(*A*) contain the min(*m*,*n*) by *n* upper trapezoidal matrix *R* (*R* is upper triangular if *m*≥*n*); the elements below the diagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see Application Notes below).

tau (local).
REAL for psgeqr2
DOUBLE PRECISION for pdgeqr2
COMPLEX for pcgeqr2
COMPLEX*16 for pzgeqr2.
Array, DIMENSION *LOCc*(*ja*+min(*m*,*n*)-1). This array contains the scalar factors of the elementary reflectors. *tau* is tied to the distributed matrix *A*.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local). INTEGER.
If *info* = 0, the execution is successful. if *info* < 0:
If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*ja*) *H*(*ja*+1) . . . *H*(*ja*+*k*-1), where *k* = min(*m*,*n*).

Each *H*(*i*) has the form

H(*j*)= *I* - *tau* * *v* * *v*',

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1: *i*-1) = 0 and *v*(*i*) = 1; *v*(*i*+1: *m*) is stored on exit in *A*(*ia*+*i*:*ia*+*m*-1, *ja*+*i*-1), and *tau* in *TAU*(*ja*+*i*-1).

p?gerq2

Computes an RQ factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```
call psgerq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgerq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgerq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgerq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

This routine computes an *RQ* factorization of a real/complex distributed *m*-by-*n* matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) = *R***Q*.

Input Parameters

m (global). INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(*A*). (*m*≥0).

n (global).INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*). (*n*≥0).

a (local).
REAL for psgerq2
DOUBLE PRECISION for pdgerq2
COMPLEX for pcgerq2
COMPLEX*16 for pzgerq2.
Pointer into the local memory to an array of DIMENSION (*lld_a*,*LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the *m*-by-*n* distributed matrix sub(*A*) which is to be factored.

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.

work (local).
REAL for psgerq2
DOUBLE PRECISION for pdgerq2
COMPLEX for pcgerq2
COMPLEX*16 for pzgerq2.
This is a workspace array of DIMENSION (*lwork*).

lwork (local or global). INTEGER.
The dimension of the array *work*.

lwork is local input and must be at least *lwork*≥*nq*0 + max(1, *mp*0), where
iroff = mod(*ia*-1, *mb_a*), *icoff* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),
iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*), *mp*0 = numroc(*m*+*iroff*, *mb_a*, *myrow*, *iarow*, *nprow*),
*nq*0 = numroc(*n*+*icoff*, *nb_a*, *mycol*, *iacol*, *npcol*),
indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a (local).
On exit,

	if $m \leq n$, the upper triangle of $A(ia+m-n:ia+m-1, ja:ja+n-1)$ contains the m-by-m upper triangular matrix R; if $m \geq n$, the elements on and above the $(m-n)$-th subdiagonal contain the m-by-n upper trapezoidal matrix R; the remaining elements, with the array tau, represent the orthogonal/ unitary matrix Q as a product of elementary reflectors (see Application Notes below).

tau	(local).
	`REAL` for `psgerq2`
	`DOUBLE PRECISION` for `pdgerq2`
	`COMPLEX` for `pcgerq2`
	`COMPLEX*16` for `pzgerq2`.
	Array, `DIMENSION` $LOCr(ia+m-1)$. This array contains the scalar factors of the elementary reflectors. tau is tied to the distributed matrix A.

work	On exit, $work(1)$ returns the minimal and optimal $lwork$.
info	(local). `INTEGER`. If $info = 0$, the execution is successful. if $info < 0$: If the i-th argument is an array and the j-entry had an illegal value, then $info = -(i*100+j)$, if the i-th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$Q = H(ia)\ H(ia+1)\ .\ .\ .\ H(ia+k-1)$ for real flavors,

$Q = H(ia)'\ H(ia+1)'\ .\ .\ .\ H(ia+k-1)'$ for complex flavors,

where $k = \min(m, n)$.

Each $H(i)$ has the form

$H(i) = I - tau * v * v'$,

where tau is a real/complex scalar, and v is a real/complex vector with $v(n-k+i+1:n) = 0$ and $v(n-k+i) = 1$; $v(1:n-k+i-1)$ for real flavors or $conj(v(1:n-k+i-1))$ for complex flavors is stored on exit in $A(ia+m-k+i-1, ja:ja+n-k+i-2)$, and tau in $TAU(ia+m-k+i-1)$.

p?getf2

Computes an LU factorization of a general matrix, using partial pivoting with row interchanges (local blocked algorithm).

Syntax

```
call psgetf2(m, n, a, ia, ja, desca, ipiv, info)
call pdgetf2(m, n, a, ia, ja, desca, ipiv, info)
call pcgetf2(m, n, a, ia, ja, desca, ipiv, info)
call pzgetf2(m, n, a, ia, ja, desca, ipiv, info)
```

Description

This routine computes an *LU* factorization of a general *m*-by-*n* distributed matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) using partial pivoting with row interchanges.

The factorization has the form sub(*A*) = *P* * *L* * *U*, where *P* is a permutation matrix, *L* is lower triangular with unit diagonal elements (lower trapezoidal if *m*>*n*), and *U* is upper triangular (upper trapezoidal if *m* < *n*). This is the right-looking Parallel Level 2 BLAS version of the algorithm.

Input Parameters

m (global). INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(*A*). (*m*≥0).

n (global).INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*). (*nb_a* - mod(*ja*-1, *nb_a*)≥*n*≥0).

a (local).
REAL for psgetf2
DOUBLE PRECISION for pdgetf2
COMPLEX for pcgetf2
COMPLEX*16 for pzgetf2.
Pointer into the local memory to an array of DIMENSION(*lld_a*, *LOCc*(*ja*+*n*-1)).

On entry, this array contains the local pieces of the *m*-by-*n* distributed matrix sub(*A*).

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.

Output Parameters

ipiv (local). INTEGER.
Array, DIMENSION(*LOCr*(*m_a*) + *mb_a*). This array contains the pivoting information. *ipiv*(*i*) –> The global row that local row *i* was swapped with. This array is tied to the distributed matrix *A*.

info (local). INTEGER.
If *info* = 0: successful exit.
If *info* < 0:

- if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = –(*i**100+*j*),
- if the *i*-th argument is a scalar and had an illegal value, then *info* = – *i*.

If *info* > 0: If *info* = *k*, *u*(*ia*+*k*–1, *ja*+*k*–1) is exactly zero. The factorization has been completed, but the factor *u* is exactly singular, and division by zero will occur if it is used to solve a system of equations.

p?labrd

Reduces the first nb rows and columns of a general rectangular matrix A to real bidiagonal form by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A.

Syntax

```
call pslabrd(m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx, y, iy, jy, descy, work)
call pdlabrd(m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx, y, iy, jy, descy, work)
call pclabrd(m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx, y, iy, jy, descy, work)
call pzlabrd(m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx, y, iy, jy, descy, work)
```

Description

This routine reduces the first *nb* rows and columns of a real/complex general *m*-by-*n* distributed matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) to upper or lower bidiagonal form by an orthogonal/unitary transformation *Q*'* *A* * *P*, and returns the matrices *X* and *Y* necessary to apply the transformation to the unreduced part of sub(*A*).

If *m* ≥*n*, sub(*A*) is reduced to upper bidiagonal form; if *m* < *n*, sub(*A*) is reduced to lower bidiagonal form.

This is an auxiliary routine called by p?gebrd.

Input Parameters

m (global). INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(*A*). (*m* ≥ 0).

n (global).INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*). (*n* ≥ 0).

nb	(global) INTEGER. The number of leading rows and columns of sub(*A*) to be reduced.
a	(local). REAL for pslabrd DOUBLE PRECISION for pdlabrd COMPLEX for pclabrd COMPLEX*16 for pzlabrd. Pointer into the local memory to an array of DIMENSION(*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, this array contains the local pieces of the general distributed matrix sub(*A*).
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix sub(*A*), respectively.
desca	(global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.
ix, *jx*	(global) INTEGER. The row and column indices in the global array *x* indicating the first row and the first column of the submatrix sub(*X*), respectively.
descx	(global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix X.
iy, *jy*	(global) INTEGER. The row and column indices in the global array *y* indicating the first row and the first column of the submatrix sub(*Y*), respectively.
descy	(global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix Y.
work	(local). REAL for pslabrd DOUBLE PRECISION for pdlabrd COMPLEX for pclabrd COMPLEX*16 for pzlabrd Workspace array, DIMENSION(*lwork*) *lwork* ≥ *nb_a* + *nq*, with *nq* = numroc(*n* + mod(*ia*-1, *nb_y*), *nb_y*, *mycol*, *iacol*, *npcol*)

iacol = indxg2p (*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*)
indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

a (local)
On exit, the first *nb* rows and columns of the matrix are overwritten; the rest of the distributed matrix sub(*A*) is unchanged. if $m \geq n$, elements on and below the diagonal in the first *nb* columns, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors;and elements above the diagonal in the first *nb* rows, with the array *taup*, represent the orthogonal/unitary matrix *P* as a product of elementary reflectors.
If $m < n$, elements below the diagonal in the first *nb* columns, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors, and elements on and above the diagonal in the first *nb* rows, with the array *taup*, represent the orthogonal/unitary matrix *P* as a product of elementary reflectors. See Application Notes below.

d (local).
REAL for pslabrd
DOUBLE PRECISION for pdlabrd
COMPLEX for pclabrd
COMPLEX*16 for pzlabrd
Array, DIMENSION *LOCr*(*ia*+min(*m*,*n*)-1) if $m \geq n$; *LOCc*(*ja*+min(*m*,*n*)-1) otherwise. The distributed diagonal elements of the bidiagonal distributed matrix *B*: *d*(*i*) = *A*(*ia*+*i*-1, *ja*+*i*-1).
d is tied to the distributed matrix *A*.

e (local).
REAL for pslabrd
DOUBLE PRECISION for pdlabrd
COMPLEX for pclabrd

	COMPLEX*16 for pzlabrd Array, DIMENSION *LOCr*(*ia*+min(*m*,*n*)-1) if *m* ≥ *n*; *LOCc*(*ja*+min(*m*,*n*)-2) otherwise. The distributed off-diagonal elements of the bidiagonal distributed matrix *B*: if *m* ≥ *n*, *E*(*i*) = *A*(*ia*+*i*-1, *ja*+*i*) for *i* = 1, 2, ..., *n*-1; if *m* < *n*,*E*(*i*) = *A*(*ia*+*i*, *ja*+*i*-1) for *i* = 1, 2, ..., *m*-1. *e* is tied to the distributed matrix *A*.
tauq, *taup*	(local). REAL for pslabrd DOUBLE PRECISION for pdlabrd COMPLEX for pclabrd COMPLEX*16 for pzlabrd Array DIMENSION *LOCc*(*ja*+min(*m*, *n*)-1) for *tauq*, DIMENSION *LOCr*(*ia*+min(*m*, *n*)-1) for *taup*. The scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Q* for *tauq*, *P* for *taup*. *tauq* and *taup* are tied to the distributed matrix *A*. See Application Notes below.
x	(local) REAL for pslabrd DOUBLE PRECISION for pdlabrd COMPLEX for pclabrd COMPLEX*16 for pzlabrd Pointer into the local memory to an array of DIMENSION (*lld_x*, *nb*). On exit, the local pieces of the distributed *m*-by-*nb* matrix *X*(*ix*:*ix*+*m*-1, *jx*:*jx*+*nb*-1) required to update the unreduced part of sub(*A*).
y	(local). REAL for pslabrd DOUBLE PRECISION for pdlabrd COMPLEX for pclabrd COMPLEX*16 for pzlabrd

Pointer into the local memory to an array of DIMENSION (*lld_y*, *nb*). On exit, the local pieces of the distributed *n*-by-*nb* matrix *Y*(*iy*:*iy*+*n*-1, *jy*:*jy*+*nb*-1) required to update the unreduced part of sub(*A*).

Application Notes

The matrices *Q* and *P* are represented as products of elementary reflectors:

Q = *H*(1) *H*(2) . . . *H*(*nb*) and *P* = *G*(1) *G*(2). . . *G*(*nb*)

Each *H*(i) and *G*(i) has the form:

H(i) = *I* - *tauq* **v* **v*' and *G*(i) = *I* - *taup* **u* **u*',

where *tauq* and *taup* are real/complex scalars, and *v* and *u* are real/complex vectors.

If *m* ≥ *n*, *v*(1: *i*-1) = 0, *v*(*i*) = 1, and *v*(*i*:*m*) is stored on exit in

A(*ia*+*i*-1:*ia*+*m*-1, *ja*+*i*-1); *u*(1:*i*) = 0, *u*(*i*+1) = 1, and *u*(*i*+1:*n*) is stored on exit in *A*(*ia*+*i*-1, *ja*+*i*:*ja*+*n*-1); *tauq* is stored in *TAUQ*(*ja*+*i*-1) and *taup* in *TAUP*(*ia*+*i*-1).

If *m* < *n*, *v*(1: *i*) = 0, *v*(*i*+1) = 1, and *v*(*i*+1:*m*) is stored on exit in

A(*ia*+*i*+1:*ia*+*m*-1, *ja*+*i*-1); *u*(1:*i*-1) = 0, *u*(*i*) = 1, and *u*(*i*:*n*) is stored on exit in *A*(*ia*+*i*-1, *ja*+*i*:*ja*+*n*-1); *tauq* is stored in *TAUQ*(*ja*+*i*-1) and *taup* in *TAUP*(*ia*+*i*-1). The elements of the vectors *v* and *u* together form the *m*-by-*nb* matrix *V* and the *nb*-by-*n* matrix *U*' which are necessary, with *X* and *Y*, to apply the transformation to the unreduced part of the matrix, using a block update of the form: sub(*A*) := sub(*A*) - *V***Y*' - *X***U*'. The contents of sub(*A*) on exit are illustrated by the following examples with *nb* = 2:

$m = 6$ **and** $n = 5 (m > n)$:

$$\begin{bmatrix} 1 & 1 & u1 & u1 & u1 \\ v1 & 1 & 1 & u2 & u2 \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \end{bmatrix}$$

$m = 5$ **and** $n = 6 (m < n)$ **:**

$$\begin{bmatrix} 1 & u1 & u1 & u1 & u1 & u1 \\ 1 & 1 & u2 & u2 & u2 & u2 \\ v1 & 1 & a & a & a & a \\ v1 & v2 & a & a & a & a \\ v1 & v2 & a & a & a & a \end{bmatrix}$$

where *a* denotes an element of the original matrix which is unchanged, *vi* denotes an element of the vector defining *H(i)*, and *ui* an element of the vector defining *G(i)*.

p?lacon

Estimates the 1-norm of a square matrix, using the reverse communication for evaluating matrix-vector products.

Syntax

```
call pslacon(n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
call pdlacon(n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
call pclacon(n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
call pzlacon(n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
```

Description

This routine estimates the 1-norm of a square, real/unitary distributed matrix *A*. Reverse communication is used for evaluating matrix-vector products. *x* and *v* are aligned with the distributed matrix *A*, this information is implicitly contained within *iv*, *ix*, *descv*, and *descx*.

Input Parameters

n	(global).INTEGER. The length of the distributed vectors *v* and *x*. *n* ≥ 0.
v	(local). REAL for pslacon DOUBLE PRECISION for pdlacon COMPLEX for pclacon COMPLEX*16 for pzlacon. Pointer into the local memory to an array of DIMENSION *LOCr*(*n*+mod(*iv*-1, *mb_v*)). On the final return, *v* = *a*w*, where *est* = norm(*v*)/norm(*w*) (*w* is not returned).
iv, *jv*	(global) INTEGER. The row and column indices in the global array *v* indicating the first row and the first column of the submatrix *V*, respectively.
descv	(global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix V.
x	(local). REAL for pslacon DOUBLE PRECISION for pdlacon COMPLEX for pclacon COMPLEX*16 for pzlacon. Pointer into the local memory to an array of DIMENSION *LOCr*(*n*+mod(*ix*-1, *mb_x*)).
ix, *jx*	(global) INTEGER. The row and column indices in the global array *x* indicating the first row and the first column of the submatrix *X*, respectively.
descx	(global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix X.
isgn	(local). INTEGER. Array, DIMENSION *LOCr*(*n*+mod(*ix*-1, *mb_x*)). *isgn* is aligned with *x* and *v*.
kase	(local). INTEGER. On the initial call to p?lacon, *kase* should be 0.

Output Parameters

x	(local). On an intermediate return, X should be overwritten by *A***X*, if *kase*=1, *A*'**X*, if *kase*=2, p?lacon must be re-called with all the other parameters unchanged.
est	(global). REAL for single precision flavors DOUBLE PRECISION for double precision flavors
kase	(local) INTEGER. On an intermediate return, *kase* is 1 or 2, indicating whether *X* should be overwritten by *A***X*, or *A*'**X*. On the final return from p?lacon, *kase* is again 0.

p?laconsb

Looks for two consecutive small subdiagonal elements.

Syntax

```
call pslaconsb(a, desca, i, l, m, h44, h33, h43h34, buf, lwork)
call pdlaconsb(a, desca, i, l, m, h44, h33, h43h34, buf, lwork)
```

Description

This routine looks for two consecutive small subdiagonal elements by analyzing the effect of starting a double shift *QR* iteration given by *h44*, *h33*, and *h43h34* to see if this process makes a subdiagonal negligible.

Input Parameters

a	(global). REAL for pslaconsb DOUBLE PRECISION for pdlaconsb Array, DIMENSION (*desca* (*lld_*),*). On entry, the Hessenberg matrix whose tridiagonal part is being scanned. Unchanged on exit.
desca	(global and local) INTEGER. Array of DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

i	(global) INTEGER. The global location of the bottom of the unreduced submatrix of *A*. Unchanged on exit.
l	(global) INTEGER. The global location of the top of the unreduced submatrix of *A*. Unchanged on exit.
h44, *h33*, *h43h34*	(global). REAL for pslaconsb DOUBLE PRECISION for pdlaconsb These three values are for the double shift QR iteration.
lwork	(global) INTEGER. This must be at least 7*ceil(ceil((*i*-l)/*hbl*)/lcm(*nprow*, *npcol*)). Here lcm is least common multiple and *nprowxnpcol* is the logical grid size.

Output Parameters

m	(global). On exit, this yields the starting location of the *QR* double shift. This will satisfy: $l \le m \le i$-2.
buf	(local). REAL for pslaconsb DOUBLE PRECISION for pdlaconsb Array of size *lwork*.
lwork	(global). On exit, *lwork* is the size of the work buffer.

p?lacp2

Copies all or part of a distributed matrix to another distributed matrix.

Syntax

```
call pslacp2(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pdlacp2(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pclacp2(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pzlacp2(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
```

Description

This routine copies all or part of a distributed matrix *A* to another distributed matrix *B*. No communication is performed, p?lacp2 performs a local copy sub(*A*):= sub(*B*), where sub(*A*) denotes *A*(*ia*:*ia*+*m*-1, *a*:*ja*+*n*-1) and sub(*B*) denotes *B*(*ib*:*ib*+*m*-1, *jb*:*jb*+*n*-1).

p?lacp2 requires that only dimension of the matrix operands is distributed.

Input Parameters

uplo — (global) CHARACTER. Specifies the part of the distributed matrix sub(*A*) to be copied:
= 'U': Upper triangular part is copied; the strictly lower triangular part of sub(*A*) is not referenced;
= 'L': Lower triangular part is copied; the strictly upper triangular part of sub(*A*) is not referenced.
Otherwise: all of the matrix sub(*A*) is copied.

m — (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(*A*). (*m* ≥ 0).

n — (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*). (*n* ≥ 0).

a — (local).
REAL for pslacp2
DOUBLE PRECISION for pdlacp2
COMPLEX for pclacp2
COMPLEX*16 for pzlacp2.
Pointer into the local memory to an array of DIMENSION(*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the *m*-by-*n* distributed matrix sub(*A*).

ia, *ja* — (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of sub(*A*), respectively.

desca — (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.

ib, *jb* (global) INTEGER. The row and column indices in the global array *B* indicating the first row and the first column of sub(*B*), respectively.

descb (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix B.

Output Parameters

b (local).
REAL for pslacp2
DOUBLE PRECISION for pdlacp2
COMPLEX for pclacp2
COMPLEX*16 for pzlacp2.
Pointer into the local memory to an array of DIMENSION (*lld_b*, *LOCc*(*jb*+*n*-1)). This array contains on exit the local pieces of the distributed matrix sub(B) set as follows:
if *uplo* = 'U', *B*(*ib*+*i*-1, *jb*+*j*-1) = *A*(*ia*+*i*-1, *ja*+*j*-1), $1 \le i \le j$, $1 \le j \le n$;
if *uplo* = 'L', *B*(*ib*+*i*-1, *jb*+*j*-1) = *A*(*ia*+*i*-1, *ja*+*j*-1), $j \le i \le m$, $1 \le j \le n$;
otherwise, *B*(*ib*+*i*-1, *jb*+*j*-1) = *A*(*ia*+*i*-1, *ja*+*j*-1), $1 \le i \le m$, $1 \le j \le n$.

p?lacp3

Copies from a global parallel array into a local replicated array or vice versa.

Syntax

```
call pslacp3(m, i, a, desca, b, ldb, ii, jj, rev)
call pdlacp3(m, i, a, desca, b, ldb, ii, jj, rev)
```

Description

This is an auxiliary routine that copies from a global parallel array into a local replicated array or vise versa. Note that the entire submatrix that is copied gets placed on one node or more. The receiving node can be specified precisely, or all nodes can receive, or just one row or column of nodes.

Input Parameters

m	(global) INTEGER. *m* is the order of the square submatrix that is copied. *m* ≥ 0. Unchanged on exit.
i	(global) INTEGER. *A*(*i*, *i*) is the global location that the copying starts from. Unchanged on exit.
a	(global). REAL for pslacp3 DOUBLE PRECISION for pdlacp3 Array, DIMENSION (*desca*(*lld_*),*). On entry, the parallel matrix to be copied into or from.
desca	(global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.
b	(local). REAL for pslacp3 DOUBLE PRECISION for pdlacp3 Array, DIMENSION (*ldb*, *m*). If *rev* = 0, this is the global portion of the array *A*(*i*:*i*+*m*-1, *i*:*i*+*m*-1). If *rev* = 1, this is the unchanged on exit.
ldb	(local) INTEGER. The leading dimension of *B*.
ii	(global) INTEGER. By using *rev* 0 and 1, data can be sent out and returned again. If *rev* = 0, then *ii* is destination row index for the node(s) receiving the replicated *B*. If *ii* ≥ 0, *jj* ≥ 0, then node (*ii*, *jj*) receives the data. If *ii* = -1, *jj* ≥ 0, then all rows in column *jj* receive the data. If *ii* ≥ 0, *jj* = -1, then all cols in row *ii* receive the data. f *ii* = -1, *jj* = -1, then all nodes receive the data. If *rev* !=0, then *ii* is the source row index for the node(s) sending the replicated *B*.
jj	(global) INTEGER. Similar description as *ii* above.
rev	(global) INTEGER. Use *rev* = 0 to send global *A* into locally replicated *B* (on node (*ii*, *jj*)). Use *rev* != 0 to send locally replicated *B* from node (*ii*, *jj*) to its owner (which changes depending on its location in *A*) into the global *A*.

Output Parameters

a	(global). On exit, if *rev* = 1, the copied data. Unchanged on exit if *rev* = 0.
b	(local). If *rev* = 1, this is unchanged on exit.

p?lacpy

Copies all or part of one two-dimensional array to another.

Syntax

```
call pslacpy(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pdlacpy(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pclacpy(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pzlacpy(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
```

Description

This routine copies all or part of a distributed matrix *A* to another distributed matrix *B*. No communication is performed, p?lacpy performs a local copy sub(*A*):= sub(*B*), where sub(*A*) denotes *A*(*ia*:*ia*+*m*-1,*ja*:*ja*+*n*-1) and sub(*B*) denotes *B*(*ib*:*ib*+*m*-1,*jb*:*jb*+*n*-1).

Input Parameters

uplo	(global). CHARACTER. Specifies the part of the distributed matrix sub(*A*) to be copied: = 'U': Upper triangular part is copied; the strictly lower triangular part of sub(*A*) is not referenced; = 'L': Lower triangular part is copied; the strictly upper triangular part of sub(*A*) is not referenced. Otherwise: all of the matrix sub(*A*) is copied.
m	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(*A*). (*m* ≥0).
n	(global) INTEGER.

	The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*). (*n*≥0).
a	(local). REAL for pslacpy DOUBLE PRECISION for pdlacpy COMPLEX for pclacpy COMPLEX*16 for pzlacpy. Pointer into the local memory to an array of DIMENSION(*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, this array contains the local pieces of the distributed matrix sub(*A*).
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix sub(*A*), respectively.
desca	(global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.
ib, *jb*	(global) INTEGER. The row and column indices in the global array *B* indicating the first row and the first column of sub(*B*) respectively.
descb	(global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.

Output Parameters

b	(local). REAL for pslacpy DOUBLE PRECISION for pdlacpy COMPLEX for pclacpy COMPLEX*16 for pzlacpy. Pointer into the local memory to an array of DIMENSION (*lld_b*, *LOCc*(*jb*+*n*-1)). This array contains on exit the local pieces of the distributed matrix sub(B) set as follows: if *uplo* = 'U', *B*(*ib*+*i*-1, *jb*+*j*-1) = *A*(*ia*+*i*-1, *ja*+*j*-1), 1≤*i*≤*j*, 1≤*j*≤*n*;if *uplo* = 'L', *B*(*ib*+*i*-1, *jb*+*j*-1) = *A*(*ia*+*i*-1, *ja*+*j*-1),*j*≤*i*≤*m*, 1≤*j*≤*n*;

otherwise, $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$, $1 \le i \le m$, $1 \le j \le n$.

p?laevswp

Moves the eigenvectors from where they are computed to ScaLAPACK standard block cyclic array.

Syntax

```
call pslaevswp(n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork, lrwork)
call pdlaevswp(n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork, lrwork)
call pclaevswp(n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork, lrwork)
call pzlaevswp(n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork, lrwork)
```

Description

This routine moves the eigenvectors (potentially unsorted) from where they are computed, to a ScaLAPACK standard block cyclic array, sorted so that the corresponding eigenvalues are sorted.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

n (global). INTEGER.
The order of the matrix A. $n \ge 0$.

zin (local).
REAL for pslaevswp
DOUBLE PRECISION for pdlaevswp
COMPLEX for pclaevswp
COMPLEX*16 for pzlaevswp. Array, DIMENSION (*ldzi*, *nvs*(*iam*)). The eigenvectors on input. Each eigenvector resides entirely in one process. Each process holds a contiguous set of *nvs*(*iam*) eigenvectors. The first eigenvector which the process holds is: sum for *i*=[0, *iam*-1) of *nvs*(*i*).

ldzi	(local) INTEGER.The leading dimension of the *zin* array.
iz, *jz*	(global) INTEGER.The row and column indices in the global array *Z* indicating the first row and the first column of the submatrix *Z*, respectively.
descz	(global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix Z.
nvs	(global) INTEGER. Array, DIMENSION(*nprocs*+1) *nvs*(*i*) = number of processes number of eigenvectors held by processes [0, *i*-1) *nvs*(1) = number of eigen vectors held by[0, 1 -1) = 0 *nvs*(*nprocs*+1)= number of eigen vectors held by [0, *nprocs*)= total number of eigenvectors.
key	(global) INTEGER. Array, DIMENSION (*n*). Indicates the actual index (after sorting) for each of the eigenvectors.
rwork	(local). REAL for pslaevswp DOUBLE PRECISIONfor pdlaevswp COMPLEX for pclaevswp COMPLEX*16 for pzlaevswp. Array, DIMENSION (*lrwork*).
lrwork	(local) INTEGER. Dimension of *work*.

Output Parameters

z	(local). REAL for pslaevswp DOUBLE PRECISION for pdlaevswp COMPLEX for pclaevswp COMPLEX*16 for pzlaevswp. Array, global DIMENSION (*n*, *n*), local DIMENSION (*descz*(*dlen_*), *nq*). The eigenvectors on output. The eigenvectors are distributed in a block cyclic manner in both dimensions, with a block size of *nb*.

p?lahrd

Reduces the first nb columns of a general rectangular matrix A so that elements below the k-th subdiagonal are zero, by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A.

Syntax

```
call pslahrd(n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
call pdlahrd(n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
call pclahrd(n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
call pzlahrd(n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
```

Description

The routines reduces the first *nb* columns of a real general *n*-by-(*n*-*k*+1) distributed matrix *A*(*ia*:*ia*+*n*-1 , *ja*:*ja*+*n*-*k*) so that elements below the *k*-th subdiagonal are zero. The reduction is performed by an orthogonal/unitary similarity transformation *Q' * A * Q*. The routine returns the matrices *V* and *T* which determine *Q* as a block reflector *I - V*T*V'*, and also the matrix *Y = A * V * T*.

This is an auxiliary routine called by p?gehrd. In the following comments sub(*A*) denotes *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1).

Input Parameters

n — (global) INTEGER.
The order of the distributed submatrix sub(*A*). *n* ≥ 0.

k — (global) INTEGER.
The offset for the reduction. Elements below the *k*-th subdiagonal in the first *nb* columns are reduced to zero.

nb — (global) INTEGER.
The number of columns to be reduced.

a — (local).
REAL for pslahrd
DOUBLE PRECISION for pdlahrd

COMPLEX for pclahrd
COMPLEX*16 for pzlahrd.
Pointer into the local memory to an array of DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-*k*)). On entry, this array contains the local pieces of the *n*-by-(*n*-*k*+1) general distributed matrix *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-*k*).

ia, *ja* (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.

iy, *jy* (global) INTEGER. The row and column indices in the global array *Y* indicating the first row and the first column of the submatrix sub(*Y*), respectively.

descy (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix Y.

work (local).
REAL for pslahrd
DOUBLE PRECISION for pdlahrd
COMPLEX for pclahrd
COMPLEX*16 for pzlahrd.
Array, DIMENSION (*nb*).

Output Parameters

a (local).
On exit, the elements on and above the *k*-th subdiagonal in the first *nb* columns are overwritten with the corresponding elements of the reduced distributed matrix;the elements below the *k*-th subdiagonal, with the array *tau*, represent the matrix *Q* as a product of elementary reflectors. The other columns of *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-*k*) are unchanged. See Application Notes below.

tau (local)
REAL for pslahrd
DOUBLE PRECISION for pdlahrd
COMPLEX for pclahrd

COMPLEX*16 for pzlahrd.
Array, DIMENSION *LOCc*(*ja*+*n*-2). The scalar factors of the elementary reflectors (see Application Notes below). *tau* is tied to the distributed matrix *A*.

t (local)REAL for pslahrd
DOUBLE PRECISION for pdlahrd
COMPLEX for pclahrd
COMPLEX*16 for pzlahrd.
Array, DIMENSION (*nb_a*, *nb_a*) The upper triangular matrix *T*.

y (local).
REAL for pslahrd
DOUBLE PRECISION for pdlahrd
COMPLEX for pclahrd
COMPLEX*16 for pzlahrd.
Pointer into the local memory to an array of DIMENSION (*lld_y*, *nb_a*). On exit, this array contains the local pieces of the *n*-by-*nb* distributed matrix *Y*. *lld_y* ≥ *LOCr*(*ia*+*n*-1).

Application Notes

The matrix *Q* is represented as a product of *nb* elementary reflectors

Q = *H*(1) *H*(2) . . . *H*(*nb*).

Each *H*(*i*) has the form

H(*i*) = *i* - *tau* **v* * *v*',

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1: *i*+*k*-1)= 0, *v*(*i*+*k*)= 1; *v*(*i*+*k*+1:*n*) is stored on exit in *A*(*ia*+*i*+*k*:*ia*+*n*-1, *ja*+*i*-1), and *tau* in *TAU*(*ja*+*i*-1).

The elements of the vectors *v* together form the (*n*-*k*+1)-by-*nb* matrix *V* which is needed, with *T* and *Y*, to apply the transformation to the unreduced part of the matrix, using an update of the form: *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-*k*) := (*I*-*V***T***V*')*(*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-*k*)-*Y***V*'). The contents of *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-*k*) on exit are illustrated by the following example with *n* = 7, *k* = 3, and *nb* = 2:

$$\begin{bmatrix} a & h & a & a & a \\ a & h & a & a & a \\ a & h & a & a & a \\ h & h & a & a & a \\ v1 & h & a & a & a \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \end{bmatrix}$$

where *a* denotes an element of the original matrix *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-*k*), *h* denotes a modified element of the upper Hessenberg matrix *H*, and *vi* denotes an element of the vector defining *H*(*i*).

p?laiect

Exploits IEEE arithmetic to accelerate the computations of eigenvalues. (C interface function).

Syntax

```
void pslaiect(float *sigma, int *n, float *d, int *count);
void pdlaiectb(float *sigma, int *n, float *d, int *count);
void pdlaiectl(float *sigma, int *n, float *d, int *count);
```

Description

This routine computes the number of negative eigenvalues of ($A - \sigma I$). This implementation of the Sturm Sequence loop exploits IEEE arithmetic and has no conditionals in the innermost loop. The signbit for real routine `pslaiect` is assumed to be bit 32. Double precision routines `pdlaiectb` and `pdlaiectl` differ in the order of the double precision word storage and, consequently, in the signbit location. For `pdlaiectb`, the double precision word is stored in the big-endian word order and the signbit is assumed to be bit 32. For `pdlaiectl`, the double precision word is stored in the little-endian word order and the signbit is assumed to be bit 64.

Note that all arguments are call-by-reference so that this routine can be directly called from Fortran code.

This is a ScaLAPACK internal subroutine and arguments are not checked for unreasonable values.

Input Parameters

sigma — Real for pslaiect
DOUBLE PRECISION for pdlaiectb/pdlaiectl.
The shift. p?laiect finds the number of eigenvalues less than equal to *sigma*.

n — INTEGER. The order of the tridiagonal matrix *T*. $n \geq 1$.

d — Real for pslaiect
DOUBLE PRECISION for pdlaiectb/pdlaiectl.
Array of DIMENSION(2*n* –1).
On entry, this array contains the diagonals and the squares of the off-diagonal elements of the tridiagonal matrix *T*. These elements are assumed to be interleaved in memory for better cache performance. The diagonal entries of *T* are in the entries *d*(1), *d*(3),..., *d*(2*n*–1), while the squares of the off-diagonal entries are *d*(2), *d*(4), ..., *d*(2*n*–2). To avoid overflow, the matrix must be scaled so that its largest entry is no greater than $overflow^{(1/2)} * underflow^{(1/4)}$ in absolute value, and for greatest accuracy, it should not be much smaller than that.

Output Parameters

n — INTEGER. The count of the number of eigenvalues of *T* less than or equal to *sigma*.

p?lange

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a general rectangular matrix.

Syntax

```
val = pslange(norm, m, n, a, ia, ja, desca, work)
val = pdlange(norm, m, n, a, ia, ja, desca, work)
val = pclange(norm, m, n, a, ia, ja, desca, work)
val = pzlange(norm, m, n, a, ia, ja, desca, work)
```

Description

The function returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a distributed matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1).

p?lange returns the value

```
( max(abs(A(i,j))), norm = 'M' or 'm' with ia ≤ i ≤ ia+m-1,
( and ja ≤ j ≤ ja+n-1,
(
( norm1( sub(A)), norm = '1', 'O' or 'o'
(
( normI( sub(A)), norm = 'I' or 'i'
(
( normF( sub(A)), norm = 'F', 'f', 'E' or 'e',
```

where *norm1* denotes the 1-norm of a matrix (maximum column sum), *normI* denotes the infinity norm of a matrix (maximum row sum) and *normF* denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(*A*(*i*,*j*))) is not a matrix norm.

Input Parameters

norm (global) CHARACTER. Specifies the value to be returned by the routine as described above.

m (global). INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(A). When *m* = 0, p?lange is set to zero. *m* ≥ 0.

n (global). INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*).
When *n* = 0, p?lange is set to zero. *n* ≥ 0.

a (local).
Real for pslange
DOUBLE PRECISION for pdlange
COMPLEX for pclange
COMPLEX*16 for pzlange.
Pointer into the local memory to an array of DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1)) containing the local pieces of the distributed matrix sub(*A*).

ia, *ja* (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.

work (local).
Real for pslange
DOUBLE PRECISION for pdlange
COMPLEX for pclange
COMPLEX*16 for pzlange.
Array DIMENSION (*lwork*).
lwork ≥ 0 if *norm* = 'M' or 'm' (not referenced),
*nq*0 if *norm* = '1', 'O' or 'o',
*mp*0 if *norm* = 'I' or 'i',
0 if *norm* = 'F', 'f', 'E' or 'e' (not referenced),
where

iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),
iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),
*mp*0 = numroc(*m*+*iroffa*, *mb_a*, *myrow*, *iarow*, *nprow*),
*nq*0 = numroc(*n*+*icoffa*, *nb_a*, *mycol*, *iacol*, *npcol*),
indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

val	The value returned by the fuction.

p?lanhs

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of an upper Hessenberg matrix.

Syntax

val = pslanhs(*norm*, *n*, *a*, *ia*, *ja*, *desca*, *work*)

val = pdlanhs(*norm*, *n*, *a*, *ia*, *ja*, *desca*, *work*)

val = pclanhs(*norm*, *n*, *a*, *ia*, *ja*, *desca*, *work*)

val = pzlanhs(*norm*, *n*, *a*, *ia*, *ja*, *desca*, *work*)

Description

The function returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a distributed matrix sub(*A*) = *A*(*ia:ia+m-1*, *ja:ja+n-1*).

p?lanhs returns the value

(max(abs(*A*(*i*,*j*))), *norm* = 'M' or 'm' with *ia* ≤ *i* ≤ *ia*+*m*-1,

(and $ja \le j \le ja+n-1$,

(

(*norm1*(sub(*A*)), *norm* = '1', 'O' or 'o'

(

(*normI*(sub(*A*)), *norm* = 'I' or 'i'

(

(*normF*(sub(*A*)), *norm* = 'F', 'f', 'E' or 'e',

where *norm1* denotes the 1-norm of a matrix (maximum column sum), *normI* denotes the infinity norm of a matrix (maximum row sum) and *normF* denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(*A*(*i*,*j*))) is not a matrix norm.

Input Parameters

norm (global) CHARACTER. Specifies the value to be returned by the routine as described above.

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*).
When $n = 0$, p?lanhs is set to zero. $n \ge 0$.

a (local).
Real for pslanhs
DOUBLE PRECISION for pdlanhs
COMPLEX for pclanhs
COMPLEX*16 for pzlanhs.
Pointer into the local memory to an array of DIMENSION(*lld_a*, *LOCc*(*ja*+*n*-1)) containing the local pieces of the distributed matrix sub(*A*).

ia, *ja* (global) INTEGER.
The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.

work (local).

Real for pslanhs
DOUBLE PRECISION for pdlanhs
COMPLEX for pclanhs
COMPLEX*16 for pzlanh.
Array, DIMENSION (*lwork*).

lwork ≥ 0 if *norm* = 'M' or 'm' (not referenced),
*nq*0 if *norm* = '1', 'O' or 'o',
*mp*0 if *norm* = 'I' or 'i',
0 if *norm* = 'F', 'f', 'E' or 'e' (not referenced),
where
iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),
iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),
*mp*0 = numroc(*m*+*iroffa*, *mb_a*, *myrow*, *iarow*, *nprow*),
*nq*0 = numroc(*n*+*icoffa*, *nb_a*, *mycol*, *iacol*, *npcol*),
indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *imycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

val — The value returned by the fuction.

p?lansy, p?lanhe

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a real symmetric or a complex Hermitian matrix.

Syntax

```
val = pslansy(norm, uplo, n, a, ia, ja, desca, work)
val = pdlansy(norm, uplo, n, a, ia, ja, desca, work)
val = pclansy(norm, uplo, n, a, ia, ja, desca, work)
val = pzlansy(norm, uplo, n, a, ia, ja, desca, work)
val = pclanhe(norm, uplo, n, a, ia, ja, desca, work)
val = pzlanhe(norm, uplo, n, a, ia, ja, desca, work)
```

Description

The functions return the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a distributed matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1).

p?lansy, p?lanhe return the value

(max(abs(*A*(*i*,*j*))), *norm* = 'M' or 'm' with *ia* ≤ *i* ≤ *ia*+*m*-1,

(and *ja* ≤ *j* ≤ *ja*+*n*-1,

(

(*norm1*(sub(*A*)), *norm* = '1', 'O' or 'o'

(

(*normI*(sub(*A*)), *norm* = 'I' or 'i'

(

(*normF*(sub(*A*)), *norm* = 'F', 'f', 'E' or 'e',

where *norm1* denotes the 1-norm of a matrix (maximum column sum), *normI* denotes the infinity norm of a matrix (maximum row sum) and *normF* denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(*A*(*i*,*j*))) is not a matrix norm.

Input Parameters

norm	(global) CHARACTER. Specifies the value to be returned by the routine as described above.
uplo	(global) CHARACTER. Specifies whether the upper or lower triangular part of the symmetric matrix sub(*A*) is to be referenced. = 'U': Upper triangular part of sub(*A*) is referenced, = 'L': Lower triangular part of sub(*A*) is referenced.
n	(global) INTEGER. The number of columns to be operated on i.e the number of columns of the distributed submatrix sub(*A*). When *n* = 0, p?lansy is set to zero. *n* ≥ 0.
a	(local). Real for pslansy DOUBLE PRECISION for pdlansy COMPLEX for pclansy, pclanhe COMPLEX*16 for pzlansy, pzlanhe. Pointer into the local memory to an array of DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1)) containing the local pieces of the distributed matrix sub(*A*). If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular matrix whose norm is to be computed, and the strictly lower triangular part of this matrix is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular matrix whose norm is to be computed, and the strictly upper triangular part of sub(*A*) is not referenced.
ia, *ja*	(global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.
desca	(global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.
work	(local). Real for pslansy DOUBLE PRECISION for pdlansy COMPLEX for pclansy, pclanhe

COMPLEX*16 for pzlansy, pzlanhe.
Array DIMENSION (*lwork*).

lwork ≥ 0 if *norm* = 'M' or 'm' (not referenced),
2*nq0+np0+*ldw* if *norm* = '1', 'O' or 'o', 'I' or 'i',
where *ldw* is given by:

```
if( nprow.ne.npcol ) then
ldw = mb_a*ceil(ceil(np0/mb_a)/(lcm/nprow))
else
ldw = 0
end if
```

0 if *norm* = 'F', 'f', 'E' or 'e' (not referenced),
where *lcm* is the least common multiple of *nprow* and *npcol*, *lcm* = ilcm(*nprow*, *npcol*) and ceil denotes the ceiling operation (iceil).
iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),
iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),
*mp*0 = numroc(*m*+*iroffa*, *mb_a*, *myrow*, *iarow*, *nprow*),
*nq*0 = numroc(*n*+*icoffa*, *nb_a*, *mycol*, *iacol*, *npcol*),
indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

val The value returned by the fuction.

p?lantr

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a triangular matrix.

Syntax

```
val = pslantr(norm, uplo, diag, m, n, a, ia, ja, desca, work)
val = pdlantr(norm, uplo, diag, m, n, a, ia, ja, desca, work)
val = pclantr(norm, uplo, diag, m, n, a, ia, ja, desca, work)
val = pzlantr(norm, uplo, diag, m, n, a, ia, ja, desca, work)
```

Description

The function returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular distributed matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1).

p?lantr returns the value

(max(abs(*A*(*i*,*j*))), *norm* = 'M' or 'm' with *ia* ≤ *i* ≤ *ia*+*m*-1,

(and *ja* ≤ *j* ≤ *ja*+*n*-1,

(

(*norm1*(sub(*A*)), *norm* = '1', 'O' or 'o'

(

(*normI*(sub(*A*)), *norm* = 'I' or 'i'

(

(*normF*(sub(*A*)), *norm* = 'F', 'f', 'E' or 'e',

where *norm1* denotes the 1-norm of a matrix (maximum column sum), *normI* denotes the infinity norm of a matrix (maximum row sum) and *normF* denotes the Frobenius norm of a matrix (square root of sum of squares). Note that max(abs(*A*(*i*,*j*))) is not a matrix norm.

Input Parameters

norm (global) CHARACTER. Specifies the value to be returned by the routine as described above.

uplo (global) CHARACTER.
Specifies whether the upper or lower triangular part of the symmetric matrix sub(*A*) is to be referenced.
= 'U': Upper trapezoidal,
= 'L': Lower trapezoidal.
Note that sub(*A*) is triangular instead of trapezoidal if *m* = *n*.

diag (global). CHARACTER.
Specifies whether or not the distributed matrix sub(*A*) has unit diagonal.
= 'N': Non-unit diagonal.
= 'U': Unit diagonal.

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(A). When *m* = 0, p?lantr is set to zero. *m* ≥ 0.

n (global) INTEGER.
The number of columns to be operated on i.e the number of columns of the distributed submatrix sub(*A*). When *n* = 0, p?lantr is set to zero. *n* ≥ 0.

a (local).
Real for pslantr
DOUBLE PRECISION for pdlantr
COMPLEX for pclantr
COMPLEX*16 for pzlantr.
Pointer into the local memory to an array of DIMENSION(*lld_a*, *LOCc*(*ja*+*n*−1)) containing the local pieces of the distributed matrix sub(*A*).

ia, *ja* (global) INTEGER.
The row and column indices in the global array *a* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.

work (local).
Real for pslantr
DOUBLE PRECISION for pdlantr
COMPLEX for pclantr
COMPLEX*16 for pzlantr.
Array DIMENSION (*lwork*).

lwork ≥ 0 if *norm* = 'M' or 'm' (not referenced),
*nq*0 if *norm* = '1', 'O' or 'o',
*mp*0 if *norm* = 'I' or 'i',
0 if *norm* = 'F', 'f', 'E' or 'e' (not referenced),
where *lcm* is the least common multiple of *nprow* and *npcol*
lcm = ilcm(*nprow*, *npcol*) and ceil denotes the ceiling operation (iceil).
iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),
iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),
*mp*0 = numroc(*m*+*iroffa*, *mb_a*, *myrow*, *iarow*, *nprow*),
*nq*0 = numroc(*n*+*icoffa*, *nb_a*, *mycol*, *iacol*, *npcol*),
indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

val The value returned by the fuction.

p?lapiv

Applies a permutation matrix to a general distributed matrix, resulting in row or column pivoting.

Syntax

call pslapiv(*direc*, *rowcol*, *pivroc*, *m*, *n*, *a*, *ia*, *ja*, *desca*, *ipiv*, *ip*, *jp*, *descip*, *iwork*)

call pdlapiv(*direc*, *rowcol*, *pivroc*, *m*, *n*, *a*, *ia*, *ja*, *desca*, *ipiv*, *ip*, *jp*, *descip*, *iwork*)

call pclapiv(*direc*, *rowcol*, *pivroc*, *m*, *n*, *a*, *ia*, *ja*, *desca*, *ipiv*, *ip*, *jp*, *descip*, *iwork*)

call pzlapiv(*direc*, *rowcol*, *pivroc*, *m*, *n*, *a*, *ia*, *ja*, *desca*, *ipiv*, *ip*, *jp*, *descip*, *iwork*)

Description

This routine applies either *P* (permutation matrix indicated by *ipiv*) or inv(*P*) to a general *m*-by-*n* distributed matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1), resulting in row or column pivoting. The pivot vector may be distributed across a process row or a column. The pivot vector should be aligned with the distributed matrix *A*. This routine will transpose the pivot vector, if necessary.

For example, if the row pivots should be applied to the columns of sub(*A*), pass *rowcol*='C' and *pivroc*='C'.

Input Parameters

direc (global) CHARACTER*1.
Specifies in which order the permutation is applied:
= 'F' (Forward). Applies pivots forward from top of matrix.
Computes *P**sub(*A*).
= 'B' (Backward) Applies pivots backward from bottom of matrix.
Computes inv(*P*)*sub(*A*).

rowcol (global) CHARACTER*1.
Specifies if the rows or columns are to be permuted:
= 'R' Rows will be permuted,

= 'C' Columns will be permuted.

pivroc (global) CHARACTER*1.
Specifies whether *ipiv* is distributed over a process row or column:
= 'R' *ipiv* is distributed over a process row,
= 'C' *ipiv* is distributed over a process column.

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(*A*). When *m* = 0, p?lapiv is set to zero. *m* ≥ 0.

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*).
When *n* = 0, p?lapiv is set to zero. *n* ≥ 0.

a (local).
Real for pslapiv
DOUBLE PRECISION for pdlapiv
COMPLEX for pclapiv
COMPLEX*16 for pzlapiv.
Pointer into the local memory to an array of DIMENSION(*lld_a*, *LOCc*(*ja*+*n*-1)) containing the local pieces of the distributed matrix sub(*A*).

ia, *ja* (global) INTEGER.
The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.

ipiv (local). INTEGER.
Array, DIMENSION (*lipiv*) ;
when *rowcol*='R' or 'r':
lipiv≥*LOCr*(*ia*+*m*-1) + *mb_a* if *pivroc*='C' or 'c',
lipiv≥*LOCc*(*m* + *mod*(*jp*-1, *nb_p*)) if *pivroc*='R' or 'r', and,
when *rowcol*='C' or 'c':

lipiv≥*LOCr*(*n* + mod(*ip*-1, *mb_p*)) if *pivroc*='C' or 'c',

lipiv≥*LOCc*(*ja*+*n*-1) + *nb_a* if *pivroc*='R' or 'r'.

This array contains the pivoting information. *ipiv(i)* is the global row (column), local row (column) *i* was swapped with. When *rowcol*='R' or 'r' and *pivroc*='C' or 'c', or *rowcol*='C' or 'c' and *pivroc*='R' or 'r', the last piece of this array of size *mb_a* (resp. *nb_a*) is used as workspace. In those cases, this array is tied to the distributed matrix *A*.

ip, *jp* (global) INTEGER. The row and column indices in the global array *P* indicating the first row and the first column of the submatrix sub(*P*), respectively.

descip (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed vector *ipiv*.

iwork (local). INTEGER.
Array, DIMENSION (*ldw*), where *ldw* is equal to the workspace necessary for transposition, and the storage of the tranposed *ipiv* :

Let *lcm* be the least common multiple of *nprow* and *npcol*.

```
If( rowcol.eq.'r' .and.  pivroc. eq.'r') then
      If( nprow.eq. npcol) then
ldw = LOCr( n_p + mod(jp-1, nb_p) ) + nb_p
     else
ldw = LOCr( n_p + mod(jp-1, nb_p) )+
nb_p * ceil( ceil(LOCc(n_p)/nb_p) / (lcm/npcol)
 )
   end if
 else if( rowcol.eq.'c' .and.  pivroc.eq.'c')
then
       if( nprow.eq.
npcol ) then
ldw = LOCc( m_p + mod(ip-1, mb_p) ) + mb_p
 else
ldw = LOCc( m_p + mod(ip-1, mb_p) ) +
mb_p *ceil(ceil(LOCr(m_p)/mb_p) / (lcm/nprow) )
    end if
 else
iwork is not referenced.
 end if.
```

Output Parameters

a (local).
On exit, the local pieces of the permuted distributed submatrix.

p?laqge

Scales a general rectangular matrix, using row and column scaling factors computed by p?geequ .

Syntax

```
call pslaqge(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
call pdlaqge(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
call pclaqge(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
call pzlaqge(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
```

Description

This routine equilibrates a general *m*-by-*n* distributed matrix sub(*A*) = *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) using the row and scaling factors in the vectors *r* and *c* computed by p?geequ.

Input Parameters

m (global). INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(*A*). (*m* ≥ 0).

n (global).INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*). (*n* ≥ 0).

a (local).
REAL for pslaqge
DOUBLE PRECISION for pdlaqge
COMPLEX for pclaqge
COMPLEX*16 for pzlaqge.
Pointer into the local memory to an array of DIMENSION(*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the distributed matrix sub(*A*).

ia, *ja* (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca	(global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.
r	(local). REAL for pslaqge DOUBLE PRECISION for pdlaqge COMPLEX for pclaqge COMPLEX*16 for pzlaqge. Array, DIMENSION *LOCr*(*m_a*). The row scale factors for sub(*A*). *r* is aligned with the distributed matrix *A*, and replicated across every process column. *r* is tied to the distributed matrix *A*.
c	(local). REAL for pslaqge DOUBLE PRECISION for pdlaqge COMPLEX for pclaqge COMPLEX*16 for pzlaqge. Array, DIMENSION*LOCc*(*n_a*). The row scale factors for sub(*A*). *c* is aligned with the distributed matrix *A*, and replicated across every process column. *c* is tied to the distributed matrix *A*.
rowcnd	(local). REAL for pslaqge DOUBLE PRECISION for pdlaqge COMPLEX for pclaqge COMPLEX*16 for pzlaqge. The global ratio of the smallest *r(i)* to the largest *r(i)*, *ia* ≤ *i* ≤ *ia*+*m*-1.
colcnd	(local). REAL for pslaqge DOUBLE PRECISION for pdlaqge COMPLEX for pclaqge COMPLEX*16 for pzlaqge. The global ratio of the smallest *c(i)* to the largest *c(i)*, *ia* ≤ *i* ≤ *ia*+*n*-1.
amax	(global). REAL for pslaqge DOUBLE PRECISION for pdlaqge

COMPLEX for pclaqge
COMPLEX*16 for pzlaqge.
Absolute value of largest distributed submatrix entry.

Output Parameters

a (local).
On exit, the equilibrated distributed matrix. See *equed* for the form of the equilibrated distributed submatrix.

equed (global) CHARACTER.
Specifies the form of equilibration that was done.
= 'N': No equilibration
= 'R': Row equilibration, that is, sub(*A*) has been pre-multiplied by diag(r(ia:ia+m-1)),
= 'C': column equilibration, that is, sub(*A*) has been post-multiplied by diag(c(ja:ja+n-1)),
= 'B': Both row and column equilibration, that is, sub(*A*) has been replaced by diag(r(ia:ia+m-1))* sub(*A*) * diag(c(ja:ja+n-1)).

p?laqsy

Scales a symmetric/Hermitian matrix, using scaling factors computed by p?poequ *.*

Syntax

```
call pslaqsy(uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
call pdlaqsy(uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
call pclaqsy(uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
call pzlaqsy(uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
```

Description

This routine equilibrates a symmetric distributed matrix sub(*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) using the scaling factors in the vectors *sr* and *sc*. The scaling factors are computed by p?poequ.

Input Parameters

uplo (global) CHARACTER. Specifies the upper or lower triangular part of the symmetric distributed matrix sub(*A*)is to be referenced:
= 'U': Upper triangular part;
= 'L': Lower triangular part.

n (global) INTEGER.
The order of the distributed submatrix sub(*A*). $n \geq 0$.

a (local).
REAL for pslaqsy
DOUBLE PRECISION for pdlaqsy
COMPLEX for pclaqsy
COMPLEX*16 for pzlaqsy.
Pointer into the local memory to an array of DIMENSION (*lld_a*,*LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the distributed matrix sub(*A*). On entry, the local pieces of the distributed symmetric matrix sub(*A*).
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and the strictly lower triangular part of sub(*A*) is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and the strictly upper triangular part of sub(*A*) is not referenced.

ia, *ja* (global) INTEGER.
The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.

sr (local)
REAL for pslaqsy
DOUBLE PRECISION for pdlaqsy
COMPLEX for pclaqsy
COMPLEX*16 for pzlaqsy.

Array, DIMENSION *LOCr*(*m_a*). The scale factors for *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1). *sr* is aligned with the distributed matrix *A*, and replicated across every process column. *sr* is tied to the distributed matrix *A*.

sc (local)
REAL for pslaqsy
DOUBLE PRECISION for pdlaqsy
COMPLEX for pclaqsy
COMPLEX*16 for pzlaqsy.
Array, DIMENSION *LOCc*(*m_a*). The scale factors for *A* (*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1). *sr* is aligned with the distributed matrix *A*, and replicated across every process column. *sr* is tied to the distributed matrix *A*.

scond (global). REAL for pslaqsy
DOUBLE PRECISION for pdlaqsy
COMPLEX for pclaqsy
COMPLEX*16 for pzlaqsy.
Ratio of the smallest *sr(i)* (respectively *sc(j)*) to the largest *sr(i)* (respectively *sc(j)*), with $ia \le i \le ia+n-1$ and $ja \le j \le ja+n-1$.

amax (global).
REAL for pslaqsy
DOUBLE PRECISION for pdlaqsy
COMPLEX for pclaqsy
COMPLEX*16 for pzlaqsy.
Absolute value of largest distributed submatrix entry.

Output Parameters

a On exit,
if *equed* = 'Y', the equilibrated matrix:
diag(*sr*(*ia*:*ia*+*n*-1)) * sub(*A*) *
diag(*sc*(*ja*:*ja*+*n*-1)).

equed (global) CHARACTER*1.
Specifies whether or not equilibration was done.
= 'N': No equilibration.

= 'Y': Equilibration was done, that is, sub(*A*) has been replaced by:
diag(*sr*(*ia*:*ia*+*n*-1))* sub(*A*) *
diag(*sc*(*ja*:*ja*+*n*-1)).

p?lared1d

Redistributes an array assuming that the input array, bycol, is distributed across rows and that all process columns contain the same copy of bycol.

Syntax

call pslared1d(*n*, *ia*, *ja*, *desc*, *bycol*, *byall*, *work*, *lwork*)

call pdlared1d(*n*, *ia*, *ja*, *desc*, *bycol*, *byall*, *work*, *lwork*)

Description

This routine redistributes a 1D array. It assumes that the input array *bycol* is distributed across rows and that all process column contain the same copy of *bycol*. The output array *byall* is identical on all processes and contains the entire array.

Input Parameters

np = Number of local rows in *bycol*()

n	(global). INTEGER. The number of elements to be redistributed. $n \geq 0$.
ia, *ja*	(global) INTEGER. *ia*, *ja* must be equal to 1.
desc	(global and local) INTEGER array, DIMENSION 8. A 2D array descirptor, which describes *bycol*.
bycol	(local). REAL for pslared1d DOUBLE PRECISION for pdlared1d COMPLEX for pclared1d COMPLEX*16 for pzlared1d. Distributed block cyclic array global DIMENSION (*n*), local DIMENSION *np*. *bycol* is distributed across the process rows. All process columns are assumed to contain the same value.

work (local).
REAL for pslared1d
DOUBLE PRECISION for pdlared1d
COMPLEX for pclared1d
COMPLEX*16 for pzlared1d.
DIMENSION (*lwork*). Used to hold the buffers sent from one process to another.

lwork (local)
INTEGER. The size of the *work* array. *lwork* ≥ numroc(*n*, *desc*(*nb_*), 0, 0, *npcol*).

Output Parameters

byall (global). REAL for pslared1d
DOUBLE PRECISION for pdlared1d
COMPLEX for pclared1d
COMPLEX*16 for pzlared1d.
Global DIMENSION (*n*), local DIMENSION (*n*). *byall* is exactly duplicated on all processes. It contains the same values as *bycol*, but it is replicated across all processes rather than being distributed.

p?lared2d

Redistributes an array assuming that the input array byrow is distributed across columns and that all process rows contain the same copy of byrow.

Syntax

```
call pslared2d(n, ia, ja, desc, byrow, byall, work, lwork)
call pdlared2d(n, ia, ja, desc, byrow, byall, work, lwork)
```

Description

This routine redistributes a 1D array. It assumes that the input array *byrow* is distributed across columns and that all process rows contain the same copy of *byrow*. The output array *byall* will be identical on all processes and will contain the entire array.

Input Parameters

np = Number of local rows in *byrow*()

n	(global) INTEGER. The number of elements to be redistributed. *n* ≥ 0.
ia, *ja*	(global) INTEGER. *ia*, *ja* must be equal to 1.
desc	(global and local) INTEGER array, DIMENSION (*dlen_*). A 2D array descirptor, which describes *byrow*.
byrow	(local). REAL for pslared2d DOUBLE PRECISION for pdlared2d COMPLEX for pclared2d COMPLEX*16 for pzlared2d. Distributed block cyclic array global DIMENSION (*n*), local DIMENSION *np*. *bycol* is distributed across the process columns. All process rows are assumed to contain the same value.
work	(local). REAL for pslared2d DOUBLE PRECISION for pdlared2d COMPLEX for pclared2d COMPLEX*16 for pzlared2d. DIMENSION (*lwork*). Used to hold the buffers sent from one process to another.
lwork	(local).INTEGER. The size of the *work* array. *lwork* ≥ numroc(*n*, *desc*(*nb_*), 0, 0, *npcol*).

Output Parameters

byall	(global). REAL for pslared2d DOUBLE PRECISION for pdlared2d COMPLEX for pclared2d COMPLEX*16 for pzlared2d. Global DIMENSION(*n*), local DIMENSION (*n*). *byall* is exactly duplicated on all processes. It contains the same values as *bycol*, but it is replicated across all processes rather than being distributed.

p?larf

Applies an elementary reflector to a general rectangular matrix.

Syntax

```
call pslarf(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pdlarf(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pclarf(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pzlarf(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
```

Description

This routine applies a real/complex elementary reflector Q (or Q^T) to a real/complex m-by-n distributed matrix sub(C) = $C(ic:ic+m-1, jc:jc+n-1)$, from either the left or the right. Q is represented in the form

$Q = I - tau * v * v'$,

where tau is a real/complex scalar and v is a real/complex vector.

If $tau = 0$, then Q is taken to be the unit matrix.

Input Parameters

side (global). CHARACTER.
= 'L': form Q * sub(C),
= 'R': form sub(C)* Q, $Q = Q^T$.

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(A). ($m \geq 0$).

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(A). ($n \geq 0$).

v (local).
REAL for pslarf
DOUBLE PRECISION for pdlarf
COMPLEX for pclarf

	COMPLEX*16 for pzlarf. Pointer into the local memory to an array of DIMENSION (*lld_v*,*) containing the local pieces of the distributed vectors *V* representing the Householder transformation *Q*, *v*(*iv*:*iv*+*m*-1, *jv*) if *side* = 'L' and *incv* = 1, *v*(*iv*, *jv*:*jv*+*m*-1) if *side* = 'L' and *incv* = *m_v*, *v*(*iv*:*iv*+*n*-1, *jv*) if *side* = 'R' and *incv* = 1, *v*(*iv*, *jv*:*jv*+*n*-1) if *side* = 'R' and *incv* = *m_v*. The vector *v* is the representation of *Q*. *v* is not used if *tau* = 0.
iv, *jv*	(global) INTEGER. The row and column indices in the global array *V* indicating the first row and the first column of the submatrix sub(*V*), respectively.
descv	(global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix V.
incv	(global) INTEGER. The global increment for the elements of *v*. Only two values of *incv* are supported in this version, namely 1 and *m_v*. *incv* must not be zero.
tau	(local). REAL for pslarf DOUBLE PRECISION for pdlarf COMPLEX for pclarf COMPLEX*16 for pzlarf. Array, DIMENSION *LOCc*(*jv*) if *incv* = 1, and *LOCr*(*iv*) otherwise. This array contains the Householder scalars related to the Householder vectors. *tau* is tied to the distributed matrix *v*.
c	(local). REAL for pslarf DOUBLE PRECISION for pdlarf COMPLEX for pclarf COMPLEX*16 for pzlarf. Pointer into the local memory to an array of DIMENSION(*lld_c*, *LOCc*(*jc*+*n*-1)), containing the local pieces of sub(*C*).
ic, *jc*	(global) INTEGER.

The row and column indices in the global array *c* indicating the first row and the first column of the submatrix sub(*C*), respectively.

descc (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local).
REAL for pslarf
DOUBLE PRECISION for pdlarf
COMPLEX for pclarf
COMPLEX*16 for pzlarf.
Array, DIMENSION (*lwork*).
If *incv* = 1,
if *side* = 'L',
if *ivcol* = *iccol*,
lwork≥*nqc*0
else
lwork≥*mpc*0 + max(1, *nqc*0)

```
end if
  else if side = 'R' ,
lwork≥nqc0 + max( max( 1, mpc0), numroc(numroc( n+
icoffc,nb_v,0,0,npcol),nb_v,0,0,lcmq ) )
  end if
 else if  incv = m_v,
 if side = 'L',
lwork≥mpc0 + max( max( 1, nqc0 ), numroc(
numroc(m+iroffc,mb_v,0,0,nprow ),mb_v,0,0, lcmp ) )
 else if side = 'R',
 if  ivrow = icrow,
lwork≥mpc0
    else
lwork≥nqc0 + max( 1, mpc0 )
end if
  end if
 end if,
```

where *lcm* is the least common multiple of *nprow* and *npcol* and *lcm* = *ilcm*(*nprow*, *npcol*), *lcmp* = *lcm*/*nprow*, *lcmq* = *lcm*/*npcol*,
iroffc = mod(*ic*-1, *mb_c*), *icoffc* = *mod*(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),
iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),
*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, *myrow*, *icrow*, *nprow*),

nqc0 = numroc(*n+icoffc*, *nb_c*, *mycol*, *iccol*, *npcol*),
ilcm, indxg2p, and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

c (local).
On exit, sub(*C*) is overwritten by the *Q* * sub(*C*) if *side* = 'L',
or sub(*C*) * *Q* if *side* = 'R'.

p?larfb

Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.

Syntax

call pslarfb(*side*, *trans*, *direct*, *storev*, *m*, *n*, *k*, *v*, *iv*, *jv*, *descv*, *t*, *c*, *ic*, *jc*, *descc*, *work*)

call pdlarfb(*side*, *trans*, *direct*, *storev*, *m*, *n*, *k*, *v*, *iv*, *jv*, *descv*, *t*, *c*, *ic*, *jc*, *descc*, *work*)

call pclarfb(*side*, *trans*, *direct*, *storev*, *m*, *n*, *k*, *v*, *iv*, *jv*, *descv*, *t*, *c*, *ic*, *jc*, *descc*, *work*)

call pzlarfb(*side*, *trans*, *direct*, *storev*, *m*, *n*, *k*, *v*, *iv*, *jv*, *descv*, *t*, *c*, *ic*, *jc*, *descc*, *work*)

Description

This routine applies a real/complex block reflector Q or its transpose Q^T/conjugate transpose Q^H to a real/complex distributed m-by-n matrix sub(C) = $C(ic:ic+m-1, jc:jc+n-1)$ from the left or the right.

Input Parameters

side (global).CHARACTER.

	if *side* = 'L': apply Q or Q^T for real flavors (Q^H for complex flavors) from the Left; if *side* = 'R': apply Q or $Q^T$for real flavors (Q^H for complex flavors) from the Right.
trans	(global).CHARACTER. if *trans* = 'N': no transpose, apply Q; for real flavors, if *trans*='T': transpose, apply Q^T for complex flavors, if *trans* = 'C': conjugate transpose, apply Q^H;
direct	(global) CHARACTER. Indicates how Q is formed from a product of elementary reflectors. if *direct* = 'F': $Q = H(1)\ H(2)\ .\ .\ .\ H(k)$ (Forward) if *direct* = 'B': $Q = H(k)\ .\ .\ .\ H(2)\ H(1)$ (Backward)
storev	(global) CHARACTER. Indicates how the vectors that define the elementary reflectors are stored: if *storev* = 'C': Columnwise if *storev* = 'R': Rowwise.
m	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(C). ($m \geq 0$).
n	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(C). ($n \geq 0$).
k	(global) INTEGER. The order of the matrix T.
v	(local). REAL for pslarfb DOUBLE PRECISION for pdlarfb COMPLEX for pclarfb COMPLEX*16 for pzlarfb. Pointer into the local memory to an array of DIMENSION (*lld_v*, *LOCc*(*jv*+*k*-1)) if *storev* = 'C', (*lld_v*, *LOCc*(*jv*+*m*-1)) if *storev* = 'R' and *side* = 'L',

(*lld_v*, *LOCc*(*jv*+*n*-1)) if *storev* = 'R' and *side* = 'R'.
It contains the local pieces of the distributed vectors *V* representing the Householder transformation.

if *storev* = 'C' and *side* = 'L', *lld_v* ≥ max(1,*LOCr*(*iv*+*m*-1));

if *storev* = 'C' and *side* = 'R', *lld_v* ≥ max(1,*LOCr*(*iv*+*n*-1));

if *storev* = 'R', *lld_v* ≥ *LOCr*(*jv*+*k*-1).

iv, *jv* (global) INTEGER.
The row and column indices in the global array *V* indicating the first row and the first column of the submatrix sub(*V*), respectively.

descv (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *V*.

c (local).
REAL for pslarfb
DOUBLE PRECISION for pdlarfb
COMPLEX for pclarfb
COMPLEX*16 for pzlarfb.
Pointer into the local memory to an array of DIMENSION(*lld_c*, *LOCc*(*jc*+*n*-1)), containing the local pieces of sub(*C*).

ic, *jc* (global) INTEGER. The row and column indices in the global array *C* indicating the first row and the first column of the submatrix sub(*C*), respectively.

descc (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local).
REAL for pslarfb
DOUBLE PRECISION for pdlarfb
COMPLEX for pclarfb
COMPLEX*16 for pzlarfb.
Workspace array, DIMENSION (*lwork*).
If *storev* = 'C',

```
if side = 'L',

lwork ≥ ( nqc0 +  mpc0 ) * k

else if  side = 'R',

lwork ≥ ( nqc0 + max( npv0 + numroc( numroc( n
+ icoffc,

nb_v, 0, 0, npcol ), nb_v, 0, 0, lcmq ),

mpc0 ) ) * k

end if


else if storev = 'R' ,

if side = 'L' ,

lwork ≥ ( mpc0 + max( mqv0 + numroc( numroc(
m+iroffc,

mb_v, 0, 0, nprow ), mb_v, 0, 0, lcmp ),

nqc0 ) ) * k

     else if side = 'R',

lwork ≥ ( mpc0 + nqc0 ) * k

end if

end if,

where
```

lcmq = *lcm* / *npcol* with lcm = iclm(*nprow*, *npcol*),

iroffv = mod(*iv*-1, *mb_v*), *icoffv* = mod(*jv*-1, *nb_v*),

ivrow = indxg2p(*iv*, *mb_v*, *myrow*, *rsrc_v*, *nprow*),

ivcol = indxg2p(*jv*, *nb_v*, *mycol*, *csrc_v*, *npcol*),

MqV0 = numroc(*m*+*icoffv*, *nb_v*, *mycol*, *ivcol*, *npcol*),

NpV0 = numroc(*n*+*iroffv*, *mb_v*, *myrow*, *ivrow*, *nprow*),

iroffc = mod(*ic*-1, *mb_c*), *icoffc* = mod(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),

iccol = indxg2p(

jc, *nb_c*, *mycol*, *csrc_c*, *npcol*),

MpC0 = numroc(*m*+*iroffc*, *mb_c*, *myrow*, *icrow*, *nprow*),

NpC0 = numroc(*n*+*icoffc*, *mb_c*, *myrow*, *icrow*, *nprow*),

NqC0 = numroc(*n*+*icoffc*, *nb_c*, *mycol*, *iccol*, *npcol*),

ilcm, indxg2p, and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

t (local).
REAL for pslarfb
DOUBLE PRECISION for pdlarfb
COMPLEX for pclarfb
COMPLEX*16 for pzlarfb.

	Array, DIMENSION(*mb_v*, *mb_v*) if *storev* = 'R', and (*nb_v*, *nb_v*) if *storev* = 'C'. The triangular matrix *t* is the representation of the block reflector.
c	(local). On exit, sub(*C*) is overwritten by the *Q* * sub(*C*), or *Q*' *sub(*C*) or sub(*C*)**Q* or sub(*C*)**Q*'.

p?larfc

Applies the conjugate transpose of an elementary reflector to a general matrix.

Syntax

call pclarfc(*side*, *m*, *n*, *v*, *iv*, *jv*, *descv*, *incv*, *tau*, *c*, *ic*, *jc*, *descc*, *work*)

call pzlarfc(*side*, *m*, *n*, *v*, *iv*, *jv*, *descv*, *incv*, *tau*, *c*, *ic*, *jc*, *descc*, *work*)

Description

This routine applies a complex elementary reflector Q^H to a complex *m*-by-*n* distributed matrix sub(*C*) = *C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1), from either the left or the right. *Q* is represented in the form

Q = *i* - *tau* * *v* * *v*',

where *tau* is a complex scalar and *v* is a complex vector.

If *tau* = 0, then *Q* is taken to be the unit matrix.

Input Parameters

side	(global).CHARACTER. if *side* = 'L': form Q^H*sub(*C*) ; if *side* = 'R': form sub (*C*)*Q^H.
m	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(*C*). (*m* ≥ 0).
n	(global) INTEGER.

The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*C*). (*n* ≥ 0).

v (local).
COMPLEX for pclarfc
COMPLEX*16 for pzlarfc.
Pointer into the local memory to an array of DIMENSION (*lld_v*,*) containing the local pieces of the distributed vectors *v* representing the Householder transformation *Q*,
v(*iv*:*iv*+*m*-1, *jv*) if *side* = 'L' and *incv* = 1,
v(*iv*, *jv*:*jv*+*m*-1) if *side* = 'L' and *incv* = *m_v*,
v(*iv*:*iv*+*n*-1, *jv*) if *side* = 'R' and *incv* = 1,
v(*iv*, *jv*:*jv*+*n*-1) if *side* = 'R' and *incv* = *m_v*.
The vector *v* is the representation of *Q*. *v* is not used if *tau* = 0.

iv, *jv* (global) INTEGER.
The row and column indices in the global array *V* indicating the first row and the first column of the submatrix sub(*V*), respectively.

descv (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *V*.

incv (global) INTEGER.
The global increment for the elements of *v*. Only two values of *incv* are supported in this version, namely 1 and *m_v*. *incv* must not be zero.

tau (local)
COMPLEX for pclarfc
COMPLEX*16 for pzlarfc.
Array, DIMENSION *LOCc*(*jv*) if *incv* = 1, and *LOCr*(*iv*) otherwise. This array contains the Householder scalars related to the Householder vectors.
tau is tied to the distributed matrix *V*.

c (local).
COMPLEX for pclarfc
COMPLEX*16 for pzlarfc.

Pointer into the local memory to an array of DIMENSION (*lld_c*, *LOCc*(*jc*+*n*-1)), containing the local pieces of sub(*C*).

ic, *jc* (global) INTEGER.
The row and column indices in the global array *C* indicating the first row and the first column of the submatrix sub(*C*), respectively.

descc (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local).
COMPLEX for pclarfc
COMPLEX*16 for pzlarfc.
Workspace array, DIMENSION (*lwork*).

```
   If incv = 1,
        if side = 'L' ,
             if ivcol = iccol,
lwork ≥ nqc0
              else
lwork ≥ mpc0 + max( 1, nqc0 )
     end if
    else if side = 'R',
lwork ≥ nqc0 + max( max( 1, mpc0 ), numroc(
numroc(
n+icoffc,nb_v,0,0,npcol ), nb_v,0,0,lcmq ) )
end if
```

```
else if  incv = m_v,
  if  side = 'L',
lwork ≥ mpc0 + max( max( 1, nqc0 ), numroc(
numroc(
m+iroffc,mb_v,0,0,nprow ),mb_v,0,0,lcmp ) )
else if side = 'R' ,

if ivrow = icrow,
lwork ≥ mpc0
  else
lwork ≥ nqc0 + max( 1, mpc0 )
         end if
       end if
    end if,
```

where *lcm* is the least common multiple of *nprow* and *npcol* and *lcm* = ilcm(*nprow*, *npcol*), lcmp = *lcm* / *nprow*, *lcmq* = *lcm* / *npcol*,
iroffc = mod(*ic*-1, *mb_c*), *icoffc* = mod(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),
iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),
*mpc*0 = numroc(*m*+*iroffc*, *mb_c*, *myrow*, *icrow*, *nprow*),
*nqc*0 = numroc(*n*+*icoffc*, *nb_c*, *mycol*, *iccol*, *npcol*),
ilcm, indxg2p, and numroc are ScaLAPACK tool functions;*myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

c	(local). On exit, sub(*C*) is overwritten by the Q^H * sub(*C*) if *side* = 'L', or sub(*C*) * Q^H if *side* = 'R'.

p?larfg

Generates an elementary reflector (Householder matrix).

Syntax

call pslarfg(*n*, *alpha*, *iax*, *jax*, *x*, *ix*, *jx*, *descx*, *incx*, *tau*)

call pdlarfg(*n*, *alpha*, *iax*, *jax*, *x*, *ix*, *jx*, *descx*, *incx*, *tau*)

call pclarfg(*n*, *alpha*, *iax*, *jax*, *x*, *ix*, *jx*, *descx*, *incx*, *tau*)

call pzlarfg(*n*, *alpha*, *iax*, *jax*, *x*, *ix*, *jx*, *descx*, *incx*, *tau*)

Description

This routine generates a real/complex elementary reflector *H* of order *n*, such that

H * sub(*X*) = *H* * (*x*(*iax*, *jax*)) = (*alpha*), *H*' * *H* = *i*,

(*x*) (0)

where *alpha* is a scalar (a real scalar - for complex flavors), and sub(*X*) is an (*n*-1)-element real/complex distributed vector *X*(*ix*:*ix*+*n*-2, *jx*) if *incx* = 1 and *X*(*ix*, *jx*:*jx*+*n*-2) if *incx* = *descx*(*m_*). *H* is represented in the form

H = *I* - *tau* * (1) * (1 *v*') ,

(*v*)

where *tau* is a real/complex scalar and *v* is a real/complex (*n*-1)-element vector. Note that *H* is not Hermitian.

If the elements of sub(*X*) are all zero (and *X*(*iax*, *jax*) is real for complex flavors), then *tau* = 0 and *H* is taken to be the unit matrix.

Otherwise 1 ≤ real(*tau*) ≤ 2 and abs(*tau*-1) ≤ 1.

Input Parameters

n (global) INTEGER.
The global order of the elementary reflector. *n* ≥ 0.

iax, *jax* (global) INTEGER.
The global row and column indices in *x* of *X*(*iax*, *jax*).

x (local).
Real for pslarfg
DOUBLE PRECISION for pdlarfg
COMPLEX for pclarfg
COMPLEX*16 for pzlarfg.
Pointer into the local memory to an array of DIMENSION (*lld_x*, *). This array contains the local pieces of the distributed vector sub(*X*). Before entry, the incremented array sub(*X*) must contain vector *x*.

ix, *jx* (global) INTEGER.
The row and column indices in the global array *X* indicating the first row and the first column of sub(*X*), respectively.

descx (global and local) INTEGER.
Array of DIMENSION (*dlen_*). The array descriptor for the distributed matrix *X*.

incx (global) INTEGER.
The global increment for the elements of *x*. Only two values of *incx* are supported in this version, namely 1 and *m_x*. *incx* must not be zero.

Output Parameters

alpha (local)
REAL for pslafg
DOUBLE PRECISION for pdlafg
COMPLEX for pclafg
COMPLEX*16 for pzlafg.
On exit, *alpha* is computed in the process scope having the vector sub(*X*).

x (local).
On exit, it is overwritten with the vector *v*.

tau	(local). Real for pslarfg DOUBLE PRECISION for pdlarfg COMPLEX for pclarfg COMPLEX*16 for pzlarfg. Array, DIMENSION *LOCc*(*jx*) if *incx* = 1, and *LOCr*(*ix*) otherwise. This array contains the Householder scalars related to the Householder vectors. *tau* is tied to the distributed matrix *X*.

p?larft

*Forms the triangular vector T of a block reflector H=I-V*T*V^H.*

Syntax

```
call pslarft(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pdlarft(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pclarft(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pzlarft(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
```

Description

This routine forms the triangular factor *T* of a real/complex block reflector *H* of order *n*, which is defined as a product of *k* elementary reflectors.

If *direct* = 'F', *H* = *H*(1) *H*(2) . . . *H*(*k*) and *T* is upper triangular;

If *direct* = 'B', *H* = *H*(*k*) . . . *H*(2) *H*(1) and *T* is lower triangular.

If *storev* = 'C', the vector which defines the elementary reflector *H*(*i*) is stored in the *i*-th column of the distributed matrix *V*, and

H = *I* - *V* * *T* * *V*'

If *storev* = 'R', the vector which defines the elementary reflector *H*(*i*) is stored in the *i*-th row of the distributed matrix *V*, and

H = *I* - *V*' * *T* *V.

Input Parameters

direct (global) CHARACTER*1.
Specifies the order in which the elementary reflectors are multiplied to form the block reflector:
if *direct* = 'F': $H = H(1)\ H(2)\ .\ .\ .\ H(k)$ (forward)
if *direct* = 'B': $H = H(k)\ .\ .\ .\ H(2)\ H(1)$ (backward).

storev (global) CHARACTER*1.
Specifies how the vectors that define the elementary reflectors are stored (See Application Notes below):
if *storev* = 'C': columnwise;
if *storev* = 'R': rowwise.

n (global) INTEGER.
The order of the block reflector H. $n \geq 0$.

k (global) INTEGER.
The order of the triangular factor T (= the number of elementary reflectors).
$1 \leq k \leq mb\_v$ (= nb_v).

v REAL for pslarft
DOUBLE PRECISION for pdlarft
COMPLEX for pclarft
COMPLEX*16 for pzlarft.
Pointer into the local memory to an array of local DIMENSION (*LOCr*(*iv*+*n*-1), *LOCc*(*jv*+*k*-1)) if *storev* = 'C', and (*LOCr*(*iv*+*k*-1), *LOCc*(*jv*+*n*-1)) if *storev* = 'R'.
The distributed matrix V contains the Householder vectors. (See Application Notes below).

iv, *jv* (global) INTEGER.
The row and column indices in the global array V indicating the first row and the first column of the submatrix sub(V), respectively.

descv (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix V.

tau (local)
REAL for pslarft

DOUBLE PRECISION for pdlarft
COMPLEX for pclarft
COMPLEX*16 for pzlarft.
Array, DIMENSION *LOCr*(*iv*+*k*-1) if *incv* = *m_v*, and *LOCc*(*jv*+*k*-1) otherwise. This array contains the Householder scalars related to the Householder vectors. *tau* is tied to the distributed matrix *V*.

work (local).
REAL for pslarft
DOUBLE PRECISION for pdlarft
COMPLEX for pclarft
COMPLEX*16 for pzlarft.
Workspace array, DIMENSION (*k**(*k*-1)/2).

Output Parameters

v REAL for pslarft
DOUBLE PRECISION for pdlarft
COMPLEX for pclarft
COMPLEX*16 for pzlarft.

t (local)
REAL for pslarft
DOUBLE PRECISION for pdlarft
COMPLEX for pclarft
COMPLEX*16 for pzlarft.
Array, DIMENSION (*nb_v*,*nb_v*) if *storev* = 'Col', and (*mb_v*,*mb_v*) otherwise. It contains the *k*-by-*k* triangular factor of the block reflector associated with *v*. If *direct* = 'F', *t* is upper triangular;
if *direct* = 'B', *t* is lower triangular.

Application Notes

The shape of the matrix *V* and the storage of the vectors that define the *H*(*i*) is best illustrated by the following example with *n* = 5 and *k* = 3. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

direct = 'F' and *storev* = 'C':

$$V(iv : iv + n - 1,\ jv : jv + k - 1) = \begin{bmatrix} 1 & & \\ v1 & 1 & \\ v1 & v2 & 1 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \end{bmatrix}$$

direct = 'F' and *storev* =

$$V(iv : iv + k - 1,\ jv : jv + n - 1) = \begin{bmatrix} 1 & \end{bmatrix}$$

direct = 'B' and *storev* = 'C'

$$V(iv : iv + n - 1,\ jv : jv + k - 1) = \begin{bmatrix} v1 & v2v & v3 \\ v1 & v2 & v3 \\ 1 & v2 & v3 \\ & 1 & v3 \\ & & 1 \end{bmatrix}$$

direct = 'B' and *storev* =

$$V(iv : iv + k - 1,\ jv : jv + n - 1) = \begin{bmatrix} v1 \\ v2 \\ v3 \end{bmatrix}$$

p?larz

Applies an elementary reflector as returned by `p?tzrzf` *to a general matrix.*

Syntax

```
call pslarz(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pdlarz(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pclarz(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pzlarz(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
```

Description

This routine applies a real/complex elementary reflector Q (or Q^T) to a real/complex m-by-n distributed matrix sub(C) = $C(ic:ic+m-1, jc:jc+n-1)$, from either the left or the right. Q is represented in the form

$Q = I - tau * v * v'$,

where tau is a real/complex scalar and v is a real/complex vector.

If $tau = 0$, then Q is taken to be the unit matrix.

Q is a product of k elementary reflectors as returned by `p?tzrzf`.

Input Parameters

side	(global) CHARACTER. if *side* = 'L': form Q^* sub(C), if *side* = 'R': form sub (C)*Q, Q=Q^T (for real flavors).
m	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(C). ($m \geq 0$).
n	(global) INTEGER.

The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*C*). (*n* ≥ 0).

l (global). INTEGER.
The columns of the distributed submatrix sub(*A*) containing the meaningful part of the Householder reflectors. If *side* = 'L', *m* ≥ *l* ≥ 0,
if *side* = 'R', *n* ≥ *l* ≥ 0.

v (local).
REAL for pslarz
DOUBLE PRECISION for pdlarz
COMPLEX for pclarz
COMPLEX*16 for pzlarz.
Pointer into the local memory to an array of DIMENSION (*lld_v*,*) containing the local pieces of the distributed vectors *v* representing the Householder transformation *Q*,
v(*iv*:*iv*+*l*-1, *jv*) if *side* = 'L' and *incv* = 1,
v(*iv*, *jv*:*jv*+*l*-1) if *side* = 'L' and *incv* = *m_v*,
v(*iv*:*iv*+*l*-1, *jv*) if *side* = 'R' and *incv* = 1,
v(*iv*, *jv*:*jv*+*l*-1) if *side* = 'R' and *incv* = *m_v*.
The vector *v* in the representation of *Q*. *v* is not used if *tau* = 0.

iv, *jv* (global) INTEGER. The row and column indices in the global array *V* indicating the first row and the first column of the submatrix sub(*V*), respectively.

descv (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *V*.

incv (global) INTEGER.
The global increment for the elements of *v*. Only two values of *incv* are supported in this version, namely 1 and *m_v*. *incv* must not be zero.

tau (local)
REAL for pslarz
DOUBLE PRECISION for pdlarz
COMPLEX for pclarz
COMPLEX*16 for pzlarz.

	Array, DIMENSION *LOCc*(*jv*) if *incv* = 1, and *LOCr*(*iv*) otherwise. This array contains the Householder scalars related to the Householder vectors. *tau* is tied to the distributed matrix *V*.
c	(local). REAL for pslarz DOUBLE PRECISION for pdlarz COMPLEX for pclarz COMPLEX*16 for pzlarz. Pointer into the local memory to an array of DIMENSION (*lld_c*, *LOCc*(*jc*+*n*-1)), containing the local pieces of sub(*C*).
ic, *jc*	(global) INTEGER. The row and column indices in the global array *C* indicating the first row and the first column of the submatrix sub(*C*), respectively.
descc	(global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *C*.
work	(local). REAL for pslarz DOUBLE PRECISION for pdlarz COMPLEX for pclarz COMPLEX*16 for pzlarz.

Array, DIMENSION (*lwork*)

If *incv* = 1,

if *side* = 'L' ,

if *ivcol* = *iccol*,

lwork ≥ *NqC*0

else

lwork ≥ *MpC*0 + max(1, *NqC*0)

end if

else if *side* = 'R' ,

lwork ≥ *NqC*0 + max(max(1, *MpC*0),
numroc(numroc(*n*+*icoffc*,*nb_v*,0,0,*npcol*),*nb_v*,0,0,*lcmq*))

end if

else if *incv* = *m_v*,

if *side* = 'L' ,

lwork ≥ *MpC*0 + max(max(1, *NqC*0),
numroc(numroc(*m*+*iroffc*,*mb_v*,0,0,*nprow*),*mb_v*,0,0,*lcmp*))

else if *side* = 'R' ,

if *ivrow* = *icrow*,

lwork ≥ *MpC*0

else

lwork ≥ *NqC*0 + max(1, *MpC*0)

end if

end if

end if,

where *lcm* is the least common multiple of *nprow* and *npcol*
and
lcm = ilcm(*nprow*, *npcol*), *lcmp* = lcm / *nprow*,
lcmq = lcm / *npcol*,

iroffc = mod(*ic*-1, *mb_c*), *icoffc* = mod(*jc*-1, *nb_c*), *icrow* = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*), *iccol* = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*), *mpc*0 = numroc(*m*+*iroffc*, *mb_c*, *myrow*, *icrow*, *nprow*), *nqc*0 = numroc(*n*+*icoffc*, *nb_c*, *mycol*, *iccol*, *npcol*), ilcm, indxg2p, and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

c	(local). On exit, sub(*C*) is overwritten by the *Q* * sub(*C*) if *side* = 'L', or sub(*C*) * *Q* if *side* = 'R'.

p?larzb

Applies a block reflector or its transpose/conjugate-transpose as returned by p?tzrzf *to a general matrix.*

Syntax

call pslarzb(*side*, *trans*, *direct*, *storev*, *m*, *n*, *k*, *l*, *v*, *iv*, *jv*, *descv*, *t*, *c*, *ic*, *jc*, *descc*, *work*)

call pdlarzb(*side*, *trans*, *direct*, *storev*, *m*, *n*, *k*, *l*, *v*, *iv*, *jv*, *descv*, *t*, *c*, *ic*, *jc*, *descc*, *work*)

call pclarzb(*side*, *trans*, *direct*, *storev*, *m*, *n*, *k*, *l*, *v*, *iv*, *jv*, *descv*, *t*, *c*, *ic*, *jc*, *descc*, *work*)

call pzlarzb(*side*, *trans*, *direct*, *storev*, *m*, *n*, *k*, *l*, *v*, *iv*, *jv*, *descv*, *t*, *c*, *ic*, *jc*, *descc*, *work*)

Description

This routine applies a real/complex block reflector *Q* or its transpose Q^T (conjugate transpose Q^H for complex flavors) to a real/complex distributed *m*-by-*n* matrix sub(*C*) = *C*(*ic*:*ic*+*m*–1, *jc*:*jc*+*n*–1) from the left or the right.

Q is a product of *k* elementary reflectors as returned by p?tzrzf.

Currently, only *storev* = 'R' and *direct* = 'B' are supported.

Input Parameters

side (global) CHARACTER.
if *side* = 'L': apply Q or Q^T (Q^H for complex flavors) from the Left;
if *side* = 'R': apply Q or Q^T (Q^H for complex flavors) from the Right.

trans (global) CHARACTER.
if *trans* = 'N': No transpose, apply Q;
If *trans*='T': Transpose, apply Q^T (real flavors);
If *trans*='C': Conjugate transpose, apply Q^H (complex flavors).

direct (global) CHARACTER.
Indicates how H is formed from a product of elementary reflectors.
if *direct* = 'F': $H = H(1)\ H(2)\ .\ .\ .\ H(k)$ (forward, not supported)
if *direct* = 'B': $H = H(k)\ .\ .\ .\ H(2)\ H(1)$ (backward)

storev (global) CHARACTER.
Indicates how the vectors that define the elementary reflectors are stored:
if *storev* = 'C': columnwise (not supported).
if *storev* = 'R': rowwise.

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(C). ($m \geq 0$).

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(C). ($n \geq 0$).

k (global) INTEGER.
The order of the matrix T. (= the number of elementary reflectors whose product defines the block reflector).

l (global) INTEGER.

	The columns of the distributed submatrix sub(*A*) containing the meaningful part of the Householder reflectors. If *side* = 'L', *m* ≥ *l* ≥ 0, if *side* = 'R', *n* ≥ *l* ≥ 0.
v	(local). REAL for pslarzb DOUBLE PRECISION for pdlarzb COMPLEX for pclarzb COMPLEX*16 for pzlarzb. Pointer into the local memory to an array of DIMENSION(*lld_v*, *LOCc*(*jv*+*m*-1)) if *side* = 'L', (*lld_v*, *LOCc*(*jv*+*m*-1)) if *side* = 'R'. It contains the local pieces of the distributed vectors *V* representing the Householder transformation as returned by p?tzrzf. *lld_v* ≥ *LOCr*(*iv*+*k*-1).
iv, *jv*	(global) INTEGER. The row and column indices in the global array *V* indicating the first row and the first column of the submatrix sub(*V*), respectively.
descv	(global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *V*.
t	(local) REAL for pslarzb DOUBLE PRECISION for pdlarzb COMPLEX for pclarzb COMPLEX*16 for pzlarzb. Array, DIMENSION *mb_v* by *mb_v*. The lower triangular matrix *T* in the representation of the block reflector.
c	(local). REAL for pslarfb DOUBLE PRECISION for pdlarfb COMPLEX for pclarfb COMPLEX*16 for pzlarfb.

Pointer into the local memory to an array of DIMENSION(*lld_c*, *LOCc*(*jc*+*n*-1)).
On entry, the *m*-by-*n* distributed matrix sub(*C*).

ic, *jc* (global) INTEGER.
The row and column indices in the global array *c* indicating the first row and the first column of the submatrix sub(*C*), respectively.

descc (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local).
REAL for pslarzb
DOUBLE PRECISION for pdlarzb
COMPLEX for pclarzb
COMPLEX*16 for pzlarzb.
Array, DIMENSION (*lwork*).

If *storev* = 'C' ,

if *side* = 'L' ,

lwork ≥(*NqC*0 + *MpC*0)* *k*

else if *side* = 'R' ,

lwork ≥ (*NqC*0 + max(*NpV*0 +
numroc(numroc(*n*+*icoffc*, *nb_v*, 0, 0, *npcol*),
nb_v, 0, 0, *lcmq*), *mpc*0))* *k*

end if

else if *storev* = 'R' ,

if *side* = 'L' ,

lwork ≥ (*mpc*0 + max(*mqv*0 + numroc(numroc(
m+*iroffc*, *mb_v*, 0, 0, *nprow*), *mb_v*, 0, 0, *lcmp*),
*nqc*0))* *k*

else if *side* = 'R' ,

lwork ≥ (*MpC*0 + *NqC*0) * *k*

end if

end if,

where *lcmq* = *lcm*/*npcol* with *lcm* = iclm(*nprow*, *npcol*),
iroffv = mod(*iv*-1, *mb_v*), *icoffv* = mod(*jv*-1, *nb_v*),
ivrow = indxg2p(*iv*, *mb_v*, *myrow*, *rsrc_v*, *nprow*),
ivcol = indxg2p(*jv*, *nb_v*, *mycol*, *csrc_v*, *npcol*),
MqV0 = numroc(*m*+*icoffv*, *nb_v*, *mycol*, *ivcol*, *npcol*),
NpV0 = numroc(*n*+*iroffv*, *mb_v*, *myrow*, *ivrow*, *nprow*),
iroffc = mod(*ic*-1, *mb_c*), *icoffc*= *mod*(*jc*-1, *nb_c*),
icrow= indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),
iccol= indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),
MpC0 = numroc(*m*+*iroffc*, *mb_c*, *myrow*, *icrow*, *nprow*),
NpC0 = numroc(*n*+*icoffc*, *mb_c*, *myrow*, *icrow*, *nprow*),
NqC0 = numroc(*n*+*icoffc*, *nb_c*, *mycol*, *iccol*, *npcol*),
ilcm, indxg2p, and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

c (local).
On exit, sub(*C*) is overwritten by the *Q**sub(*C*), or *Q*'*sub(*C*), or sub(*C*)**Q*, or sub(*C*)* *Q*'.

p?larzc

Applies (multiplies by) the conjugate transpose of an elementary reflector as returned by `p?tzrzf` *to a general matrix.*

Syntax

```
call pclarzc(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
```

```
call pzlarzc(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
```

Description

This routine applies a complex elementary reflector Q^H to a complex m-by-n distributed matrix sub(C) = $C(ic:ic+m-1, jc:jc+n-1)$, from either the left or the right. Q is represented in the form

```
Q = i - tau * v * v',
```

where tau is a complex scalar and v is a complex vector.

If $tau = 0$, then Q is taken to be the unit matrix.

Q is a product of k elementary reflectors as returned by `p?tzrzf`.

Input Parameters

side	(global) `CHARACTER`. if *side* = `'L'`: form Q^H *sub(C); if *side* = `'R'`: form sub(C)*Q^H.
m	(global) `INTEGER`. The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(C). ($m \geq 0$).
n	(global) `INTEGER`. The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(C). ($n \geq 0$).
l	(global) `INTEGER`.

	The columns of the distributed submatrix sub(*A*) containing the meaningful part of the Householder reflectors. If *side* = 'L', *m* ≥ *l* ≥ 0, if *side* = 'R', *n* ≥ *l* ≥ 0.
v	(local). COMPLEX for pclarzc COMPLEX*16 for pzlarzc. Pointer into the local memory to an array of DIMENSION (*lld_v*,*) containing the local pieces of the distributed vectors *v* representing the Householder transformation *Q*, *v*(*iv*:*iv*+*l*-1, *jv*) if *side* = 'L' and *incv* = 1, *v*(*iv*, *jv*:*jv*+*l*-1) if *side* = 'L' and *incv* = *m_v*, *v*(*iv*:*iv*+*l*-1, *jv*) if *side* = 'R' and *incv* = 1, *v*(*iv*, *jv*:*jv*+*l*-1) if *side* = 'R' and *incv* = *m_v*. The vector v in the representation of *Q*. *v* is not used if *tau* = 0.
iv, *jv*	(global) INTEGER. The row and column indices in the global array *V* indicating the first row and the first column of the submatrix sub(*V*), respectively.
descv	(global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *V*.
incv	(global). INTEGER. The global increment for the elements of *v*. Only two values of *incv* are supported in this version, namely 1 and *m_v*. *incv* must not be zero.
tau	(local) COMPLEX for pclarzc COMPLEX*16 for pzlarzc. Array, DIMENSION*LOCc*(*jv*) if *incv* = 1, and *LOCr*(*iv*) otherwise. This array contains the Householder scalars related to the Householder vectors. *tau* is tied to the distributed matrix *V*.
c	(local). COMPLEX for pclarzc COMPLEX*16 for pzlarzc.

	Pointer into the local memory to an array of DIMENSION(*lld_c*, *LOCc*(*jc*+*n*-1)), containing the local pieces of sub(*C*).
ic, *jc*	(global) INTEGER. The row and column indices in the global array *C* indicating the first row and the first column of the submatrix sub(*C*), respectively.
descc	(global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local).

If *incv* = 1,

if *side* = 'L' ,

if *ivcol* = *iccol*,

lwork ≥ *NqC*0

else

lwork ≥ *MpC*0 + max(1, *NqC*0)

end if

else if *side* = 'R' ,

lwork ≥ *nqc*0 + max(max(1, *mpc*0), numroc(numroc(*n*+*icoffc*,*nb_v*,0,0,*npcol*),*nb_v*,0,0,*lcmq*))

end if

else if *incv* = *m_v*,

if *side* = 'L' ,

lwork ≥ *mpc*0 + max(max(1, *nqc*0), numroc(numroc(*m*+*iroffc*,*mb_v*,0,0,*nprow*),*mb_v*,0,0,*lcmp*))

else if *side* = 'R',

if *ivrow* = *icrow*,

lwork ≥ *mpc*0

else

lwork ≥ *nqc*0 + max(1, *mpc*0)

end if

end if

end if,

where *lcm* is the least common multiple of *nprow* and *npcol*; *lcm* = *ilcm*(*nprow*, *npcol*), *lcmp* = *lcm*/*nprow*, *lcmq*= *lcm*/*npcol*,

iroffc = mod(*ic*-1, *mb_c*), *icoffc*= mod(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),
iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),
MpC0 = numroc(*m*+*iroffc*, *mb_c*, *myrow*, *icrow*, *nprow*),
NqC0 = numroc(*n*+*icoffc*, *nb_c*, *mycol*, *iccol*, *npcol*),
ilcm, *indxg2p*, and *numroc* are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

c (local).
On exit, sub(*C*) is overwritten by the Q^H*sub(*C*) if *side* = 'L', or sub(*C*)*Q^H if *side* = 'R'.

p?larzt

*Forms the triangular factor T of a block reflector H=I-V*T*V^H as returned by* p?tzrzf.

Syntax

call pslarzt(*direct*, *storev*, *n*, *k*, *v*, *iv*, *jv*, *descv*, *tau*, *t*, *work*)

call pdlarzt(*direct*, *storev*, *n*, *k*, *v*, *iv*, *jv*, *descv*, *tau*, *t*, *work*)

call pclarzt(*direct*, *storev*, *n*, *k*, *v*, *iv*, *jv*, *descv*, *tau*, *t*, *work*)

call pzlarzt(*direct*, *storev*, *n*, *k*, *v*, *iv*, *jv*, *descv*, *tau*, *t*, *work*)

Description

This routine forms the triangular factor *T* of a real/complex block reflector *H* of order > *n*, which is defined as a product of *k* elementary reflectors as returned by p?tzrzf.

If *direct* = 'F', *H* = *H*(1) *H*(2) . . . *H*(*k*) and *T* is upper triangular;

If *direct* = 'B', *H* = *H*(*k*) . . . *H*(2) *H*(1) and *T* is lower triangular.

If *storev* = 'C', the vector which defines the elementary reflector *H*(*i*) is stored in the *i*-th column of the array *v*, and

H = *i* - *v* * *t* * *v*'

If *storev* = 'R', the vector which defines the elementary reflector *H*(*i*) is stored in the *i*-th row of the array *v*, and

H = *i* - *v*' * *t* * *v*

Currently, only *storev* = 'R' and *direct* = 'B' are supported.

Input Parameters

direct	(global) CHARACTER. Specifies the order in which the elementary reflectors are multiplied to form the block reflector: if *direct* = 'F': *H* = *H*(1) *H*(2) . . . *H*(*k*) (Forward, not supported) if *direct* = 'B': *H* = *H*(*k*) . . . *H*(2) *H*(1) (Backward).
storev	(global) CHARACTER. Specifies how the vectors which define the elementary reflectors are stored: if *storev* = 'C': columnwise (not supported); if *storev* = 'R': rowwise.
n	(global). INTEGER. The order of the block reflector *H*. *n* ≥ 0.
k	(global). INTEGER. The order of the triangular factor *T* (= the number of elementary reflectors). 1 ≤ *k* ≤ *mb_v* (= *nb_v*).
v	REAL for pslarzt DOUBLE PRECISION for pdlarzt COMPLEX for pclarzt COMPLEX*16 for pzlarzt. Pointer into the local memory to an array of local DIMENSION(*LOCr*(*iv*+*k*-1), *LOCc*(*jv*+*n*-1)). The distributed matrix *V* contains the Householder vectors. See Application Notes below.

iv, *jv* — (global) INTEGER. The row and column indices in the global array *V* indicating the first row and the first column of the submatrix sub(*V*), respectively.

descv — (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *V*.

tau — (local)
REAL for pslarzt
DOUBLE PRECISION for pdlarzt
COMPLEX for pclarzt
COMPLEX*16 for pzlarzt.
Array, DIMENSION *LOCr*(*iv*+*k*-1) if *incv* = *m_v*, and *LOCc*(*jv*+*k*-1) otherwise. This array contains the Householder scalars related to the Householder vectors. *tau* is tied to the distributed matrix *V*.

work — (local).
REAL for pslarzt
DOUBLE PRECISION for pdlarzt
COMPLEX for pclarzt
COMPLEX*16 for pzlarzt.
Workspace array, DIMENSION(*k**(*k*-1)/2).

Output Parameters

v — REAL for pslarzt
DOUBLE PRECISION for pdlarzt
COMPLEX for pclarzt
COMPLEX*16 for pzlarzt.

t — (local)
REAL for pslarzt
DOUBLE PRECISION for pdlarzt
COMPLEX for pclarzt
COMPLEX*16 for pzlarzt.
Array, DIMENSION (*mb_v*, *mb_v*). It contains the *k*-by-*k* triangular factor of the block reflector associated with *v*. *t* is lower triangular.

Application Notes

The shape of the matrix `V` and the storage of the vectors which define the `H(i)` is best illustrated by the following example with `n = 5` and `k = 3`. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

direct='F' and *storev*='C'

$$V = \begin{bmatrix} v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \end{bmatrix} \begin{matrix} . & . & . \\ . & . & . \\ 1 & . & . \\ & 1 & . \\ & & 1 \end{matrix}$$

direct='F' and *storev*='R':

$$\begin{bmatrix} \overbrace{v1 & v1 & v1 & v1 & v1}^{V} & . & . & . & . & 1 \\ v2 & v2 & v2 & v2 & v2 & . & . & . & 1 & \\ v3 & v3 & v3 & v3 & v3 & . & . & 1 & & \end{bmatrix}$$

direct='*B*' and *storev*='*C*':

$$
v = \begin{bmatrix} 1 & & & \\ . & 1 & & \\ . & . & 1 & \\ . & . & . & \\ . & . & . & \end{bmatrix}
\begin{bmatrix} v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \end{bmatrix}
$$

direct='*B*' and *storev*='*R*':

$$
\begin{bmatrix} & & & & & \overbrace{\qquad\qquad}^{v} & & & & \\ 1 & . & . & . & . & v1 & v1 & v1 & v1 & v1 \\ . & 1 & . & . & . & v2 & v2 & v2 & v2 & v2 \\ . & . & 1 & . & . & v3 & v3 & v3 & v3 & v3 \\ & & & & & & & & & \end{bmatrix}
$$

p?lascl

Multiplies a general rectangular matrix by a real scalar defined as C_{to}/C_{from}.

Syntax

```
call pslascl(type, cfrom, cto, m, n, a, ia, ja, desca, info)
call pdlascl(type, cfrom, cto, m, n, a, ia, ja, desca, info)
call pclascl(type, cfrom, cto, m, n, a, ia, ja, desca, info)
call pzlascl(type, cfrom, cto, m, n, a, ia, ja, desca, info)
```

Description

This routine multiplies the *m*-by-*n* real/complex distributed matrix sub(*A*) denoting *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) by the real/complex scalar *cto*/*cfrom*. This is done without over/underflow as long as the final result *cto* * *A*(*i*,*j*)/*cfrom* does not over/underflow. *type* specifies that sub(*A*) may be full, upper triangular, lower triangular or upper Hessenberg.

Input Parameters

type — (global) CHARACTER.
type indices of the storage type of the input distributed matrix.
if *type* = 'G': sub(*A*) is a full matrix,
if *type* = 'L': sub(*A*) is a lower triangular matrix,
if *type* = 'U': sub(*A*) is an upper triangular matrix,
if *type* = 'H': sub(*A*) is an upper Hessenberg matrix.

cfrom, *cto* — (global)
REAL for pslascl/pclascl
DOUBLE PRECISION for pdlascl/pzlascl.
The distributed matrix sub(*A*) is multiplied by *cto*/*cfrom* . *A*(*i*,*j*) is computed without over/underflow if the final result *cto***A*(*i*,*j*)/*cfrom* can be represented without over/underflow. *cfrom* must be nonzero.

m — (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(*A*). (*m* ≥ 0).

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*). (*n* ≥ 0).

a (local input/local output)
REAL for pslascl
DOUBLE PRECISION for pdlascl
COMPLEX for pclascl
COMPLEX*16 for pzlascl.
Pointer into the local memory to an array of DIMENSION(*lld_a*, *LOCc*(*ja*+*n*-1)).
This array contains the local pieces of the distributed matrix sub(*A*).

ia, *ja* (global) INTEGER.
The column and row indices in the global array *A* indicating the first row and column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER .
Array of DIMENSION (*dlen_*).The array descriptor for the distributed matrix *A*.

Output Parameters

a (local).
On exit, this array contains the local pieces of the distributed matrix multiplied by *cto/cfrom*.

info (local)
INTEGER.
if *info* = 0: the execution is successful.
if *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?laset

Initializes the offdiagonal elements of a matrix to alpha *and the diagonal elements to* beta.

Syntax

```
call pslaset(uplo, m, n, alpha, beta, a, ia, ja, desca)
call pdlaset(uplo, m, n, alpha, beta, a, ia, ja, desca)
call pclaset(uplo, m, n, alpha, beta, a, ia, ja, desca)
call pzlaset(uplo, m, n, alpha, beta, a, ia, ja, desca)
```

Description

This routine initializes an *m*-by-*n* distributed matrix sub(*A*) denoting *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) to *beta* on the diagonal and *alpha* on the offdiagonals.

Input Parameters

uplo — (global) CHARACTER.
Specifies the part of the distributed matrix sub(*A*) to be set:
if *uplo* = 'U': upper triangular part is set; the strictly lower triangular part of sub(*A*) is not changed;
if *uplo* = 'L': lower triangular part is set; the strictly upper triangular part of sub(*A*) is not changed.
Otherwise: all of the matrix sub(*A*) is set.

m — (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(*A*). (*m* ≥ 0).

n — (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*). (*n* ≥ 0).

alpha — (global).
REAL for pslaset
DOUBLE PRECISION for pdlaset
COMPLEX for pclaset
COMPLEX*16 for pzlaset.

The constant to which the offdiagonal elements are to be set.

beta (global).
REAL for pslaset
DOUBLE PRECISION for pdlaset
COMPLEX for pclaset
COMPLEX*16 for pzlaset.
The constant to which the diagonal elements are to be set.

Output Parameters

a (local).
REAL for pslaset
DOUBLE PRECISION for pdlaset
COMPLEX for pclaset
COMPLEX*16 for pzlaset.
Pointer into the local memory to an array of DIMENSION(*lld_a*, *LOCc*(*ja*+*n*-1)).
This array contains the local pieces of the distributed matrix sub(*A*) to be set. On exit, the leading *m*-by-*n* submatrix sub(*A*) is set as follows:

if *uplo* = 'U', *A*(*ia*+*i*-1, *ja*+*j*-1) = *alpha*, 1 ≤ *i* ≤ *j*-1, 1 ≤ *j* ≤ *n*,

if *uplo* = 'L', *A*(*ia*+*i*-1, *ja*+*j*-1) = *alpha*, *j*+1 ≤ *i* ≤ *m*, 1 ≤ *j* ≤ *n*,

otherwise, *A*(*ia*+*i*-1, *ja*+*j*-1) = *alpha*, 1 ≤ *i* ≤ *m*, 1 ≤ *j* ≤ *n*, *ia*+*i*.*ne*.*ja*+*j*, and, for all *uplo*, *A*(*ia*+*i*-1, *ja*+*i*-1) = *beta*, 1 ≤ *i* ≤ min(*m*,*n*).

ia, *ja* (global) INTEGER.
The column and row indices in the global array *A* indicating the first row and column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER .
Array of DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

p?lasmsub

Looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero.

Syntax

```
call pslasmsub(a, desca, i, l, k, smlnum, buf, lwork)
call pdlasmsub(a, desca, i, l, k, smlnum, buf, lwork)
```

Description

This routine looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero. This routine does a global maximum and must be called by all processes.

Input Parameters

a	(global) REAL for pslasmsub DOUBLE PRECISION for pdlasmsub Array, DIMENSION(*desca*(*lld_*),*). On entry, the Hessenberg matrix whose tridiagonal part is being scanned. Unchanged on exit.
desca	(global and local) INTEGER. Array of DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.
i	(global) INTEGER. The global location of the bottom of the unreduced submatrix of *A*. Unchanged on exit.
l	(global) INTEGER. The global location of the top of the unreduced submatrix of *A*. Unchanged on exit.
smlnum	(global) REAL for pslasmsub DOUBLE PRECISION for pdlasmsub On entry, a “small number” for the given matrix. Unchanged on exit.
lwork	(global) INTEGER.

On exit, *lwork* is the size of the work buffer.
This must be at least 2*ceil(ceil((*i*-*l*)/*hbl*)/ lcm(*nprow*,*npcol*)). Here lcm is least common multiple, and *nprow* x *npcol* is the logical grid size.

Output Parameters

k (global) INTEGER.
On exit, this yields the bottom portion of the unreduced submatrix. This will satisfy: *l* ≤ *m* ≤ *i*-1.

buf (local).
REAL for pslasmsub
DOUBLE PRECISION for pdlasmsub
Array of size *lwork*.

p?lassq

Updates a sum of squares represented in scaled form.

Syntax

call pslassq(*n*, *x*, *ix*, *jx*, *descx*, *incx*, *scale*, *sumsq*)

call pdlassq(*n*, *x*, *ix*, *jx*, *descx*, *incx*, *scale*, *sumsq*)

call pclassq(*n*, *x*, *ix*, *jx*, *descx*, *incx*, *scale*, *sumsq*)

call pzlassq(*n*, *x*, *ix*, *jx*, *descx*, *incx*, *scale*, *sumsq*)

Description

This routine returns the values *scl* and *smsq* such that
$scl^2 * smsq = x(1)^2 + ... + x(n)^2 + scale^2 * sumsq$,

where *x*(*i*) = sub(*X*) = *X*(*ix* + (*jx*-1)**descx*(*m_*) + (*i* - 1)**incx*) for pslassq/pdlassq and *x*(*i*) = sub(*X*) = abs(*X*(*ix* + (*jx*-1)**descx*(*m_*) + (*i* - 1)**incx*) for pclassq/pzlassq.

For real routines pslassq/pdlassq the value of *sumsq* is assumed to be non-negative and *scl* returns the value
scl = max(*scale*, abs(*x*(*i*))).

For complex routines pclassq/pzlassq the value of *sumsq* is assumed to be at least unity and the value of *ssq* will then satisfy

$1.0 \leq ssq \leq sumsq + 2n$

Value *scale* is assumed to be non-negative and *scl* returns the value

$$scl = \max_i \left(scale, \mathrm{abs}\left(\mathrm{real}\left(x\left(i\right)\right)\right), \mathrm{abs}\left(\mathrm{aimag}\left(x\left(i\right)\right)\right)\right)$$

For all routines p?lassq values *scale* and *sumsq* must be supplied in *scale* and *sumsq* respectively, and *scale* and *sumsq* are overwritten by *scl* and *ssq* respectively.

All routines p?lassq make only one pass through the vector sub(*x*).

Input Parameters

n — (global) INTEGER.
The length of the distributed vector sub(*x*).

x — REAL for pslassq
DOUBLE PRECISION for pdlassq
COMPLEX for pclassq
COMPLEX*16 for pzlassq.
The vector for which a scaled sum of squares is computed:
x(*ix* + (*jx*-1)**m_x* + (*i* - 1)**incx*), 1 ≤ *i* ≤ *n*.

ix — (global) INTEGER.
The row index in the global array *X* indicating the first row of sub(*X*).

jx — (global) INTEGER.
The column index in the global array *X* indicating the first column of sub(*X*).

descx — (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *X*.

incx — (global) INTEGER.
The global increment for the elements of *X*. Only two values of *incx* are supported in this version, namely 1 and *m_x*.
The argument *incx* must not equal zero.

scale — (local).

REAL for pslassq/pclassq
DOUBLE PRECISION for pdlassq/pzlassq.
On entry, the value *scale* in the equation above.

sumsq (local)
REAL for pslassq/pclassq
DOUBLE PRECISION for pdlassq/pzlassq.
On entry, the value *sumsq* in the equation above.

Output Parameters

scale (local).
On exit, *scale* is overwritten with *scl* , the scaling factor for the sum of squares.

sumsq (local).
On exit, *sumsq* is overwritten with the value *smsq*, the basic sum of squares from which *scl* has been factored out.

p?laswp

Performs a series of row interchanges on a general rectangular matrix.

Syntax

```
call pslaswp(direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
call pdlaswp(direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
call pclaswp(direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
call pzlaswp(direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
```

Description

This routine performs a series of row or column interchanges on the distributed matrix sub(*A*)=*A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1). One interchange is initiated for each of rows or columns *k1* through *k2* of sub(*A*). This routine assumes that the pivoting information has already been broadcast along the process row or column. Also note that this routine will only work for *k1*-*k2* being in the same *mb* (or *nb*) block. If you want to pivot a full matrix, use p?lapiv.

Input Parameters

direc (global) CHARACTER.
Specifies in which order the permutation is applied:
= 'F' - forward,
= 'B' - backward.

rowcol (global) CHARACTER.
Specifies if the rows or columns are permuted:
= 'R' - rows,
= 'C' - columns.

n (global) INTEGER.
If *rowcol*='R', the length of the rows of the distributed matrix *A*(*, *ja*:*ja*+*n*-1) to be permuted;
If *rowcol*='C', the length of the columns of the distributed matrix *A*(*ia*:*ia*+*n*-1 , *) to be permuted;

a (local)
REAL for pslaswp
DOUBLE PRECISION for pdlaswp
COMPLEX for pclaswp
COMPLEX*16 for pzlaswp.
Pointer into the local memory to an array of DIMENSION (*lld_a*, *). On entry, this array contains the local pieces of the distributed matrix to which the row/columns interchanges will be applied.

ia (global) INTEGER.
The row index in the global array *A* indicating the first row of sub(*A*).

ja (global) INTEGER.
The column index in the global array *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *A*.

k1 (global) INTEGER.
The first element of *ipiv* for which a row or column interchange will be done.

k2 (global) INTEGER.

The last element of *ipiv* for which a row or column interchange will be done.

ipiv (local)
INTEGER. Array, DIMENSION *LOCr*(*m_a*)+*mb_a* for row pivoting and *LOCr*(*n_a*)+*nb_a* for column pivoting. This array is tied to the matrix *A*, *ipiv*(*k*)=*l* implies rows (or columns) *k* and *l* are to be interchanged.

Output Parameters

A (local)
REAL for pslaswp
DOUBLE PRECISION for pdlaswp
COMPLEX for pclaswp
COMPLEX*16 for pzlaswp.
On exit, the permuted distributed matrix.

p?latra

Computes the trace of a general square distributed matrix.

Syntax

val = pslatra(*n*, *a*, *ia*, *ja*, *desca*)

val = pdlatra(*n*, *a*, *ia*, *ja*, *desca*)

val = pclatra(*n*, *a*, *ia*, *ja*, *desca*)

val = pzlatra(*n*, *a*, *ia*, *ja*, *desca*)

Description

This function computes the trace of an *n*-by-*n* distributed matrix sub(*A*) denoting *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1). The result is left on every process of the grid.

Input Parameters

n (global) INTEGER.
The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub(*A*). *n* ≥ 0.

a	(local).
	`Real` for `pslatra`
	`DOUBLE PRECISION` for `pdlatra`
	`COMPLEX` for `pclatra`
	`COMPLEX*16` for `pzlatra`.
	Pointer into the local memory to an array of `DIMENSION(lld_a, LOCc(ja+n-1))` containing the local pieces of the distributed matrix, the trace of which is to be computed.

ia, *ja*	(global) `INTEGER`. The row and column indices respectively in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.
desca	(global and local) `INTEGER` array of `DIMENSION` (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

val	The value returned by the fuction.

p?latrd

Reduces the first nb rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.

Syntax

```
call pslatrd(uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw, work)
call pdlatrd(uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw, work)
call pclatrd(uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw, work)
call pzlatrd(uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw, work)
```

Description

This routine reduces *nb* rows and columns of a real symmetric or complex Hermitian matrix sub(*A*)= *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) to symmetric/complex tridiagonal form by an orthogonal/unitary similarity transformation *Q*'* sub(*A*)* *Q*, and returns the matrices *V* and *W*, which are needed to apply the transformation to the unreduced part of sub(*A*).

If *uplo* = U, p?latrd reduces the last *nb* rows and columns of a matrix, of which the upper triangle is supplied;

if *uplo* = L, p?latrd reduces the first *nb* rows and columns of a matrix, of which the lower triangle is supplied.

This is an auxiliary routine called by p?sytrd/p?hetrd.

Input Parameters

uplo (global) CHARACTER.
Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix sub(*A*) is stored:
= 'U': Upper triangular
= L: Lower triangular.

n (global) INTEGER.
The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub(*A*). *n* ≥ 0.

nb (global) INTEGER.
The number of rows and columns to be reduced.

a REAL for pslatrd
DOUBLE PRECISION for pdlatrd
COMPLEX for pclatrd
COMPLEX*16 for pzlatrd.
Pointer into the local memory to an array of DIMENSION(*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the symmetric/Hermitian distributed matrix sub(*A*).
If *uplo* = U, the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.
If *uplo* = L, the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

ia (global) INTEGER.
The row index in the global array *A* indicating the first row of sub(*A*).

ja (global) INTEGER.

The column index in the global array *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

iw (global) INTEGER.
The row index in the global array *W* indicating the first row of sub(*W*).

jw (global) INTEGER.
The column index in the global array *W* indicating the first column of sub(*W*).

descw (global and local) INTEGER array of DIMENSION (*dlen_*). The array descriptor for the distributed matrix *W*.

work (local)
REAL for pslatrd
DOUBLE PRECISION for pdlatrd
COMPLEX for pclatrd
COMPLEX*16 for pzlatrd.
Workspace array of DIMENSION (*nb_a*).

Output Parameters

a (local)
On exit, if *uplo* = 'U', the last *nb* columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of sub(*A*); the elements above the diagonal with the array *tau* represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors;
if *uplo* = 'L', the first *nb* columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of sub(*A*); the elements below the diagonal with the array *tau* represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.

d (local)
REAL for pslatrd/pclatrd
DOUBLE PRECISION for pdlatrd/pzlatrd.
Array, DIMENSION *LOCc*(*ja*+*n*-1).

The diagonal elements of the tridiagonal matrix *T*: *d*(i) = *a*(i,i). *d* is tied to the distributed matrix *A*.

e (local)
REAL for pslatrd/pclatrd
DOUBLE PRECISION for pdlatrd/pzlatrd.
Array, DIMENSION *LOCc*(*ja*+*n*-1) if *uplo* = 'U', *LOCc*(*ja*+*n*-2) otherwise.
The off-diagonal elements of the tridiagonal matrix *T*:
e(*i*) = *a*(*i*, *i* + 1) if *uplo* = 'U',
e(*i*) = *a*(*i* + 1, *i*) if *uplo* = L.
e is tied to the distributed matrix *A*.

tau (local)
REAL for pslatrd
DOUBLE PRECISION for pdlatrd
COMPLEX for pclatrd
COMPLEX*16 for pzlatrd.
Array, DIMENSION *LOCc*(*ja*+*n*-1). This array contains the scalar factors *tau* of the elementary reflectors. *tau* is tied to the distributed matrix *A*.

w (local)
REAL for pslatrd
DOUBLE PRECISION for pdlatrd
COMPLEX for pclatrd
COMPLEX*16 for pzlatrd.
Pointer into the local memory to an array of DIMENSION (*lld_w*, *nb_w*). This array contains the local pieces of the *n*-by-*nb_w* matrix *w* required to update the unreduced part of sub(*A*).

Application Notes

If *uplo* = 'U', the matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*n*) *H*(*n*-*1*) . . . *H*(*n*-*nb*+1)

Each *H*(i) has the form

H(i) = *I* - *tau***v***v*' ,

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(*i*:*n*) = 0 and *v*(*i*-1) = 1; *v*(1:*i*-1) is stored on exit in *A*(*ia*:*ia*+*i*-1, *ja*+*i*), and *tau* in *tau*(*ja*+*i*-1).

If *uplo* = L, the matrix *Q* is represented as a product of elementary reflectors

Q = *H*(1) *H*(2) . . . *H*(*nb*)

Each *H*(i) has the form

H(i) = *I* - *tau***v***v*' ,

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1:*i*) = 0 and *v*(*i*+1) = 1; *v*(*i*+2: *n*) is stored on exit in *A*(*ia*+*i*+1: *ia*+*n*-1, *ja*+*i*-1), and *tau* in *tau*(*ja*+*i*-1).

The elements of the vectors *v* together form the *n*-by-*nb* matrix *V* which is needed, with *W*, to apply the transformation to the unreduced part of the matrix, using a symmetric/Hermitian rank-2*k* update of the form:

sub(*A*) := sub(*A*) - *vw*' - *wv*'.

The contents of *a* on exit are illustrated by the following examples with

n = 5 and *nb* = 2:

if *uplo*='U':

$$\begin{bmatrix} a & a & a & v_4 & v_5 \\ & a & a & v_4 & v_5 \\ & & a & 1 & v_5 \\ & & & d & 1 \\ & & & & d \end{bmatrix}$$

if *uplo*='L':

$$\begin{bmatrix} d & & & & \\ 1 & d & & & \\ v_1 & 1 & a & & \\ v_1 & v_2 & a & a & \\ v_1 & v_2 & a & a & a \end{bmatrix}$$

where *d* denotes a diagonal element of the reduced matrix, *a* denotes an element of the original matrix that is unchanged, and v_i denotes an element of the vector defining *H*(i).

p?latrs

Solves a triangular system of equations with the scale factor set to prevent overflow.

Syntax

```
call pslatrs(uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx, scale, cnorm, work)
call pdlatrs(uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx, scale, cnorm, work)
call pclatrs(uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx, scale, cnorm, work)
call pzlatrs(uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx, scale, cnorm, work)
```

Description

This routine solves a triangular system of equations $Ax = \sigma b$, $A^T x = \sigma b$ or $A^H x = \sigma b$, where σ is a scale factor set to prevent overflow. The description of the routine will be extended in the future releases.

Input Parameters

uplo — CHARACTER*1.
Specifies whether the matrix A is upper or lower triangular.
= 'U': Upper triangular
= 'L': Lower triangular

trans — CHARACTER*1.
Specifies the operation applied to Ax.
= 'N': Solve $Ax = s*b$ (no transpose)
= 'T': Solve $A^T x = s*b$ (transpose)
= 'C': Solve $A^H x = s*b$ (conjugate transpose),
where s - is a scale factor

diag — CHARACTER*1.
Specifies whether or not the matrix A is unit triangular.
= 'N': Non-unit triangular
= 'U': Unit triangular

normin	CHARACTER*1. Specifies whether *cnorm* has been set or not. = 'Y': *cnorm* contains the column norms on entry; = 'N': *cnorm* is not set on entry. On exit, the norms will be computed and stored in *cnorm*.
n	INTEGER. The order of the matrix *A*. *n* ≥ 0
a	REAL for pslatrs/pclatrs DOUBLE PRECISION for pdlatrs/pzlatrs Array, DIMENSION (*lda*, *n*). Contains the triangular matrix *A*. If *uplo* = U, the leading *n*-by-*n* upper triangular part of the array *a* contains the upper triangular matrix, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of the array *a* contains the lower triangular matrix, and the strictly upper triangular part of *a* is not referenced. If *diag* = 'U', the diagonal elements of *a* are also not referenced and are assumed to be 1.
ia, *ja*	(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.
desca	(global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.
x	REAL for pslatrs/pclatrs DOUBLE PRECISION for pdlatrs/pzlatrs Array, DIMENSION (*n*). On entry, the right hand side *b* of the triangular system.
ix	(global) INTEGER.The row index in the global array *x* indicating the first row of sub(*x*).
jx	(global) INTEGER. The column index in the global array *x* indicating the first column of sub(*x*).
descx	(global and local) INTEGER. Array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *X*.

cnorm — REAL for pslatrs/pclatrs
DOUBLE PRECISION for pdlatrs/pzlatrs.
Array, DIMENSION (*n*). If *normin* = 'Y', *cnorm* is an input argument and *cnorm* (*j*) contains the norm of the off-diagonal part of the *j*-th column of *A*. If *trans* = 'N', *cnorm* (*j*) must be greater than or equal to the infinity-norm, and if *trans* = 'T' or 'C', *cnorm*(*j*) must be greater than or equal to the 1-norm.

work — (local).
REAL for pslatrs
DOUBLE PRECISION for pdlatrs
COMPLEX for pclatrs
COMPLEX*16 for pzlatrs.
Temporary workspace.

Output Parameters

X — On exit, *x* is overwritten by the solution vector *x*.

scale — REAL for pslatrs/pclatrs
DOUBLE PRECISION for pdlatrs/pzlatrs.
Array, DIMENSION (*lda*, *n*). The scaling factor *s* for the triangular system as described above.
If *scale* = 0, the matrix *A* is singular or badly scaled, and the vector *x* is an exact or approximate solution to *Ax* = 0.

cnorm — If *normin* = 'N', *cnorm* is an output argument and *cnorm*(*j*) returns the 1-norm of the off-diagonal part of the *j*-th column of *A*.

p?latrz

Reduces an upper trapezoidal matrix to upper triangular form by means of orthogonal/unitary transformations.

Syntax

```
call pslatrz(m, n, l, a, ia, ja, desca, tau, work)
call pdlatrz(m, n, l, a, ia, ja, desca, tau, work)
call pclatrz(m, n, l, a, ia, ja, desca, tau, work)
call pzlatrz(m, n, l, a, ia, ja, desca, tau, work)
```

Description

This routine reduces the m-by-n ($m \le n$) real/complex upper trapezoidal matrix $sub(A) = [A(ia:ia+m-1, ja:ja+m-1)\ A(ia:ia+m-1, ja+n-l:ja+n-1)]$ to upper triangular form by means of orthogonal/unitary transformations.

The upper trapezoidal matrix sub(A) is factored as

$sub(A) = (R\ 0)*Z$,

where Z is an n-by-n orthogonal/unitary matrix and R is an m-by-m upper triangular matrix.

Input Parameters

m	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(A). $m \ge 0$.
n	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(A). $n \ge 0$.
l	(global) INTEGER. The number of columns of the distributed submatrix sub(A) containing the meaningful part of the Householder reflectors. $l > 0$.
a	(local)

REAL for pslatrz
DOUBLE PRECISION for pdlatrz
COMPLEX for pclatrz
COMPLEX*16 for pzlatrz.
Pointer into the local memory to an array of DIMENSION(*lld_a*, *LOCc*(*ja*+*n*-1)). On entry, the local pieces of the *m*-by-*n* distributed matrix sub(*A*), which is to be factored.

ia (global) INTEGER.
The row index in the global array *A* indicating the first row of sub(*A*).

ja (global) INTEGER.
The column index in the global array *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *A*.

work (local)
REAL for pslatrz
DOUBLE PRECISION for pdlatrz
COMPLEX for pclatrz
COMPLEX*16 for pzlatrz.
Workspace array, DIMENSION (*lwork*).

lwork ≥ *nq*0 + max(1, *mp*0), where
iroff = mod(*ia*-1, *mb_a*),
icoff = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),
iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),
*mp*0 = numroc(*m*+*iroff*, *mb_a*, *myrow*, *iarow*, *nprow*),
*nq*0 = numroc(*n*+*icoff*, *nb_a*, *mycol*, *iacol*, *npcol*),
numroc, indxg2p, and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

a	On exit, the leading *m*-by-*m* upper triangular part of sub(*A*) contains the upper triangular matrix *R*, and elements *n*-*l*+1 to *n* of the first *m* rows of sub(*A*), with the array *tau*, represent the orthogonal/unitary matrix *Z* as a product of *m* elementary reflectors.

tau	(local)
	`REAL` for `pslatrz`
	`DOUBLE PRECISION` for `pdlatrz`
	`COMPLEX` for `pclatrz`
	`COMPLEX*16` for `pzlatrz`.
	Array, `DIMENSION(`*LOCr*`(`*ja*+*m*-1`))`. This array contains the scalar factors of the elementary reflectors. *tau* is tied to the distributed matrix *A*.

Application Notes

The factorization is obtained by Householder's method. The *k*-th transformation matrix, *Z*(*k*), which is used (or, in case of complex routines, whose conjugate transpose is used) to introduce zeros into the (*m* - *k* + 1)-th row of sub(*A*), is given in the form

$$Z(k) = \begin{bmatrix} I & 0 \\ 0 & T(k) \end{bmatrix},$$

where

$$T(k) = I - tau * u(k) * u(k)', \qquad u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

tau is a scalar and *z*(*k*) is an (*n*–*m*)-element vector. *tau* and *z*(*k*) are chosen to annihilate the elements of the *k*-th row of sub(*A*). The scalar *tau* is returned in the *k*-th element of *tau* and the vector *u*(*k*) in the *k*-th row of sub(*A*), such that the elements of *z*(*k*) are in *a*(*k*, *m* + 1), ..., *a*(*k*, *n*). The elements of *R* are returned in the upper triangular part of sub(*A*).

Z is given by

Z = *Z*(1) *Z*(2) ... *Z*(*m*).

p?lauu2

Computes the product $U*U^H$ or L^H*L, where U and L are upper or lower triangular matrices (local unblocked algorithm).

Syntax

```
call pslauu2(uplo, n, a, ia, ja, desca)
call pdlauu2(uplo, n, a, ia, ja, desca)
call pclauu2(uplo, n, a, ia, ja, desca)
call pzlauu2(uplo, n, a, ia, ja, desca)
```

Description

This routine computes the product *U*U'* or *L'*L*, where the triangular factor *U* or *L* is stored in the upper or lower triangular part of the distributed matrix

sub(*A*)= *A*(*ia*:*ia*+*n*–1, *ja*:*ja*+*n*–1).

If *uplo* = 'U' or 'u', then the upper triangle of the result is stored, overwriting the factor *U* in sub(*A*).

If *uplo* = 'L' or 'l', then the lower triangle of the result is stored, overwriting the factor *L* in sub(*A*).

This is the unblocked form of the algorithm, calling BLAS Level 2 Routines. No communication is performed by this routine, the matrix to operate on should be strictly local to one process.

Input Parameters

uplo (global) CHARACTER*1.

Specifies whether the triangular factor stored in the *matrix* sub(*A*) is upper or lower triangular:
= U: upper triangular
= L: lower triangular.

n (global) INTEGER.
The number of rows and columns to be operated on, that is, the order of the triangular factor *U* or *L*. *n* ≥ 0.

a (local)
REAL for pslauu2
DOUBLE PRECISION for pdlauu2
COMPLEX for pclauu2
COMPLEX*16 for pzlauu2.
Pointer into the local memory to an array of DIMENSION(*lld_a*, *LOCc*(*ja*+*n*-1). On entry, the local pieces of the triangular factor *U* or *L*.

ia (global) INTEGER.
The row index in the global array *A* indicating the first row of sub(*A*).

ja (global) INTEGER.
The column index in the global array *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *A*.

Output Parameters

a (local)
On exit, if *uplo* = 'U', the upper triangle of the distributed matrix sub(*A*) is overwritten with the upper triangle of the product *U***U*'; if *uplo* = 'L', the lower triangle of sub(*A*) is overwritten with the lower triangle of the product *L*'**L*.

p?lauum

Computes the product $U*U^H$ or L^H*L, where U and L are upper or lower triangular matrices.

Syntax

```
call pslauum(uplo, n, a, ia, ja, desca)
call pdlauum(uplo, n, a, ia, ja, desca)
call pclauum(uplo, n, a, ia, ja, desca)
call pzlauum(uplo, n, a, ia, ja, desca)
```

Description

This routine computes the product $U*U'$ or $L'*L$, where the triangular factor U or L is stored in the upper or lower triangular part of the matrix sub(A)= $A(ia:ia+n-1, ja:ja+n-1)$.

If *uplo* = 'U' or 'u', then the upper triangle of the result is stored, overwriting the factor U in sub(A). If *uplo* = 'L' or 'l', then the lower triangle of the result is stored, overwriting the factor L in sub(A).

This is the blocked form of the algorithm, calling Level 3 PBLAS.

Input Parameters

uplo (global) CHARACTER*1.
Specifies whether the triangular factor stored in the matrix sub(A) is upper or lower triangular:
= 'U': upper triangular
= 'L': lower triangular.

n (global) INTEGER.
The number of rows and columns to be operated on, that is, the order of the triangular factor U or L. $n \geq 0$.

a (local)
REAL for pslauum
DOUBLE PRECISION for pdlauum
COMPLEX for pclauum
COMPLEX*16 for pzlauum.

	Pointer into the local memory to an array of DIMENSION(*lld_a*, *LOCc*(*ja*+*n*-1). On entry, the local pieces of the triangular factor *U* or *L*.
ia	(global) INTEGER. The row index in the global array *A* indicating the first row of sub(*A*).
ja	(global) INTEGER. The column index in the global array *A* indicating the first column of sub(*A*).
desca	(global and local) INTEGER array of DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

a	(local) On exit, if *uplo* = 'U', the upper triangle of the distributed matrix sub(*A*) is overwritten with the upper triangle of the product *U***U*' ; if *uplo* = 'L', the lower triangle of sub(*A*) is overwritten with the lower triangle of the product *L*'**L*.

p?lawil

Forms the Wilkinson transform.

Syntax

```
call pslawil(ii, jj, m, a, desca, h44, h33, h43h34, v)
call pdlawil(ii, jj, m, a, desca, h44, h33, h43h34, v)
```

Description

This routine gets the transform given by *h44*, *h33*, and *h43h34* into *v* starting at row *m*.

Input Parameters

ii	(global) INTEGER. Row owner of *h*(*m*+2, *m*+2).
jj	(global) INTEGER. Column owner of *h*(*m*+2, *m*+2).

m	(global) INTEGER. On entry, the location from where the transform starts (row *m*). Unchanged on exit.
a	(global) REAL for pslawil DOUBLE PRECISION for pdlawil Array, DIMENSION (*desca(lld_)*,*). On entry, the Hessenberg matrix. Unchanged on exit.
desca	(global and local) INTEGER Array of DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*. Unchanged on exit.
h43h34	(global) REAL for pslawil DOUBLE PRECISION for pdlawil These three values are for the double shift *QR* iteration. Unchanged on exit.

Output Parameters

v	(global) REAL for pslawil DOUBLE PRECISION for pdlawil Array of size 3 that contains the transform on output.

p?org2l/p?ung2l

Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by p?geqlf *(unblocked algorithm).*

Syntax

```
call psorg2l(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorg2l(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcung2l(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzung2l(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine p?org2l/p?ung2l generates an *m*-by-*n* real/complex distributed matrix *Q* denoting *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) with orthonormal columns, which is defined as the last *n* columns of a product of *k* elementary reflectors of order *m*:

Q = *H*(*k*) . . . *H*(2) *H*(1) as returned by p?geqlf.

Input Parameters

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix *Q*. *m* ≥ 0.

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix *Q*. *m* ≥ *n* ≥ 0.

k (global) INTEGER.
The number of elementary reflectors whose product defines the matrix *Q*. *n*≥ *k* ≥ 0.

a REAL for psorg2l
DOUBLE PRECISION for pdorg2l
COMPLEX for pcung2l
COMPLEX*16 for pzung2l.
Pointer into the local memory to an array, DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1).
On entry, the *j*-th column must contain the vector that defines the elementary reflector *H*(*j*), *ja*+*n*-*k* ≤ *j* ≤ *ja*+*n*-*k*, as returned by p?geqlf in the *k* columns of its distributed matrix argument *A*(*ia*:*,*ja*+*n*-*k*:*ja*+*n*-1).

ia (global) INTEGER.
The row index in the global array *A* indicating the first row of sub(*A*).

ja (global) INTEGER.
The column index in the global array *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)
REAL for psorg2l
DOUBLE PRECISION for pdorg2l
COMPLEX for pcung2l
COMPLEX*16 for pzung2l.
Array, DIMENSION *LOCc*(*ja*+*n*-1).
This array contains the scalar factor *tau*(*j*) of the elementary reflector *H*(*j*), as returned by p?geqlf.

work (local)
REAL for psorg2l
DOUBLE PRECISION for pdorg2l
COMPLEX for pcung2l
COMPLEX*16 for pzung2l.
Workspace array, DIMENSION (*lwork*).

lwork (local or global) INTEGER.
The dimension of the array *work*.

lwork is local input and must be at least *lwork* ≥ *mpa*0 + max(1, *nqa*0), where
iroffa = mod(*ia*-1, *mb_a*),
icoffa = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),
iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),
*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, *myrow*, *iarow*, *nprow*),
*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, *mycol*, *iacol*, *npcol*).
indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a	On exit, this array contains the local pieces of the *m*-by-*n* distributed matrix *Q*.
work	On exit, *work*(1) returns the minimal and optimal *lwork*.
info	(local) INTEGER. = 0: successful exit < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100 +*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?org2r/p?ung2r

Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by p?geqrf *(unblocked algorithm).*

Syntax

```
call psorg2r(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorg2r(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcung2r(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzung2r(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine p?org2r/p?ung2r generates an *m*-by-*n* real/complex matrix *Q* denoting *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) with orthonormal columns, which is defined as the first *n* columns of a product of *k* elementary reflectors of order *m*:

Q = *H*(1) *H*(2) . . . *H*(*k*)

as returned by p?geqrf.

Input Parameters

m	(global) INTEGER.

The number of rows to be operated on, that is, the number of rows of the distributed submatrix Q. $m \geq 0$.

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix Q. $m \geq n \geq 0$.

k (global) INTEGER.
The number of elementary reflectors whose product defines the matrix Q. $n \geq k \geq 0$.

a REAL for psorg2r
DOUBLE PRECISION for pdorg2r
COMPLEX for pcung2r
COMPLEX*16 for pzung2r.
Pointer into the local memory to an array, DIMENSION(*lld_a*, *LOCc*(*ja*+*n*-1).
On entry, the j-th column must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by p?geqrf in the k columns of its distributed matrix argument $A(ia:*,ja:ja+k-1)$.

ia (global) INTEGER.
The row index in the global array A indicating the first row of sub(A).

ja (global) INTEGER.
The column index in the global array A indicating the first column of sub(A).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix A.

tau (local)
REAL for psorg2r
DOUBLE PRECISION for pdorg2r
COMPLEX for pcung2r
COMPLEX*16 for pzung2r.
Array, DIMENSION *LOCc*(*ja*+*k*-1).

This array contains the scalar factor *tau*(*j*) of the elementary reflector *H*(*j*), as returned by p?geqrf. This array is tied to the distributed matrix *A*.

work (local)
REAL for psorg2r
DOUBLE PRECISION for pdorg2r
COMPLEX for pcung2r
COMPLEX*16 for pzung2r.
Workspace array, DIMENSION (*lwork*).

lwork (local or global) INTEGER.
The dimension of the array *work*.

lwork is local input and must be at least *lwork* ≥ *mpa*0 + max(1, *nqa*0),
where
iroffa = mod(*ia*-1, *mb_a* , *icoffa* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),
iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),
*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, *myrow*, *iarow*, *nprow*),
*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, *mycol*, *iacol*, *npcol*).
indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit, this array contains the local pieces of the *m*-by-*n* distributed matrix *Q*.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local) INTEGER.

= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,
then *info* = - (*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value,
then *info* = -*i*.

p?orgl2/p?ungl2

Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by p?gelqf *(unblocked algorithm).*

Syntax

```
call psorgl2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorgl2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcungl2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungl2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine p?orgl2/p?ungl2 generates a *m*-by-*n* real/complex matrix *Q* denoting *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) with orthonormal rows, which is defined as the first *m* rows of a product of *k* elementary reflectors of order *n*

Q = *H*(*k*) . . . *H*(2) *H*(1) (for real flavors),

Q = *H*(*k*)' . . . *H*(2)' *H*(1)' (for complex flavors)

as returned by p?gelqf.

Input Parameters

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix *Q*. *m* ≥ 0.

n (global) INTEGER.

	The number of columns to be operated on, that is, the number of columns of the distributed submatrix Q. $n \geq m \geq 0$.
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q. $m \geq k \geq 0$.
a	REAL for psorgl2 DOUBLE PRECISION for pdorgl2 COMPLEX for pcungl2 COMPLEX*16 for pzungl2. Pointer into the local memory to an array, DIMENSION (*lld_a*, *LOCc*(*ja*+*n*-1). On entry, the i-th row must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gelqf in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$.
ia	(global) INTEGER. The row index in the global array A indicating the first row of sub(A).
ja	(global) INTEGER. The column index in the global array A indicating the first column of sub(A).
desca	(global and local) INTEGER array of DIMENSION (*dlen_*). The array descriptor for the distributed matrix A.
tau	(local) REAL for psorgl2 DOUBLE PRECISION for pdorgl2 COMPLEX for pcungl2 COMPLEX*16 for pzungl2. Array, DIMENSION *LOCr*(*ja*+*k*-1). This array contains the scalar factors *tau*(*i*) of the elementary reflectors $H(i)$, as returned by p?gelqf. This array is tied to the distributed matrix A.
WORK	(local) REAL for psorgl2

DOUBLE PRECISION for pdorgl2
COMPLEX for pcungl2
COMPLEX*16 for pzungl2.
Workspace array, DIMENSION (*lwork*).

lwork (local or global) INTEGER.
The dimension of the array *work*.

lwork is local input and must be at least *lwork* ≥ *nqa*0 + max(1, *mpa*0), where
iroffa = mod(*ia*-1, *mb_a*),
icoffa = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),
iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),
*mpa*0 = numroc(*m*+*iroffa*, *mb_a*, *myrow*, *iarow*, *nprow*),
*nqa*0 = numroc(*n*+*icoffa*, *nb_a*, *mycol*, *iacol*, *npcol*).
indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a On exit, this array contains the local pieces of the *m*-by-*n* distributed matrix *Q*.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local) INTEGER.
= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,
then *info* = - (*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value,
then *info* = -*i*.

p?orgr2/p?ungr2

Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by p?gerqf *(unblocked algorithm).*

Syntax

```
call psorgr2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorgr2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcungr2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungr2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The routine p?orgr2/p?ungr2 generates an *m*-by-*n* real/complex matrix *Q* denoting *A*(*ia*:*ia*+*m*-1, *ja*:*ja*+*n*-1) with orthonormal rows, which is defined as the last *m* rows of a product of *k* elementary reflectors of order *n*

Q = *H*(1) *H*(2) . . . *H*(*k*) (for real flavors)

Q = *H*(1)' *H*(2)' . . . *H*(*k*)' (for complex flavors)

as returned by p?gerqf.

Input Parameters

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix *Q*. *m* ≥ 0.

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix *Q*. *n* ≥ *m* ≥ 0.

k (global) INTEGER.
The number of elementary reflectors whose product defines the matrix *Q*. *m* ≥ *k* ≥ 0.

a REAL for psorgr2
DOUBLE PRECISION for pdorgr2

COMPLEX for pcungr2
COMPLEX*16 for pzungr2.
Pointer into the local memory to an array, DIMENSION(*lld_a*, *LOCc*(*ja*+*n*-1).
On entry, the *i*-th row must contain the vector that defines the elementary reflector *H*(*i*), *ia*+*m*-*k* ≤ *i* ≤ *ia*+*m*-1, as returned by p?gerqf in the *k* rows of its distributed matrix argument *A*(*ia*+*m*-*k*:*ia*+*m*-1, *ja*:*).

ia (global) INTEGER.
The row index in the global array *A* indicating the first row of sub(*A*).

ja (global) INTEGER.
The column index in the global array *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *A*.

tau (local)
REAL for psorgr2
DOUBLE PRECISION for pdorgr2
COMPLEX for pcungr2
COMPLEX*16 for pzungr2.
Array, DIMENSION *LOCr*(*ja*+*m*-1). This array contains the scalar factors *tau*(*i*) of the elementary reflectors *H*(*i*), as returned by p?gerqf. This array is tied to the distributed matrix *A*.

work (local)
REAL for psorgr2
DOUBLE PRECISION for pdorgr2
COMPLEX for pcungr2
COMPLEX*16 for pzungr2.
Workspace array, DIMENSION (*lwork*).

lwork (local or global) INTEGER.
The dimension of the array *work*.

lwork is local input and must be at least *lwork* ≥ *nqa*0 + max(1, *mpa*0), where *iroffa* = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprow*),
iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),
mpa0 = numroc(*m*+*iroffa*, *mb_a*, *myrow*, *iarow*, *nprow*),
nqa0 = numroc(*n*+*icoffa*, *nb_a*, *mycol*, *iacol*, *npcol*).
indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a — On exit, this array contains the local pieces of the *m*-by-*n* distributed matrix *Q*.

work — On exit, *work*(1) returns the minimal and optimal *lwork*.

info — (local) INTEGER.
= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,
then *info* = - (*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value,
then *info* = -*i*.

p?orm2l/p?unm2l

Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by `p?geqlf` *(unblocked algorithm).*

Syntax

```
call psorm2l(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)
call pdorm2l(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)
call pcunm2l(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)
call pzunm2l(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)
```

Description

The routine `p?orm2l/p?unm2l` overwrites the general real/complex m-by-n distributed matrix sub $(C)=C(ic:ic+m-1,jc:jc+n-1)$ with

	side = 'L'	*side* = 'R'
trans = 'N'	Q*sub (C)	sub (C)*Q
trans = 'T' (for real flavors)	Q^T * sub(C)	sub(C)*Q^T
trans = 'C' (for complex flavors)	Q^H * sub(C)	sub(C)*Q^H

where Q is a real orthogonal or complex unitary matrix defined as the product of k elementary reflectors

$Q = H(k) \ldots H(2)\ H(1)$

as returned by `p?geqlf` . Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

side (global) CHARACTER.

= 'L': apply Q or Q^T for real flavors (Q^H for complex flavors) from the left,
= 'R': apply Q or Q^T for real flavors (Q^H for complex flavors) from the right.

trans (global) CHARACTER.
= 'N': apply Q (no transpose)
= 'T': apply Q^T (transpose, for real flavors)
= 'C': apply Q^H (conjugate transpose, for complex flavors)

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(C). $m \geq 0$.

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(C). $n \geq 0$.

k (global) INTEGER.
The number of elementary reflectors whose product defines the matrix Q.

If $side$ = 'L', $m \geq k \geq 0$;

if $side$ = 'R', $n \geq k \geq 0$.

a (local)
REAL for psorm2l
DOUBLE PRECISION for pdorm2l
COMPLEX for pcunm2l
COMPLEX*16 for pzunm2l.
Pointer into the local memory to an array,
DIMENSION(lld_a, $LOCc(ja+k-1)$).
On entry, the j-th row must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by p?geqlf in the k columns of its distributed matrix argument $A(ia:*,ja:ja+k-1)$. The argument $A(ia:*,ja:ja+k-1)$ is modified by the routine but restored on exit.

If $side$ = 'L', lld_a ≥ max(1, $LOCr(ia+m-1)$),

if $side$ = 'R', lld_a ≥ max(1, $LOCr(ia+n-1)$).

ia (global) INTEGER.
The row index in the global array *A* indicating the first row of sub(*A*).

ja (global) INTEGER.
The column index in the global array *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *A*.

tau (local)
REAL for psorm2l
DOUBLE PRECISION for pdorm2l
COMPLEX for pcunm2l
COMPLEX*16 for pzunm2l.
Array, DIMENSION*LOCc*(*ja*+*n*−1). This array contains the scalar factor *tau*(*j*) of the elementary reflector *H*(*j*), as returned by p?geqlf. This array is tied to the distributed matrix *A*.

c (local)
REAL for psorm2l
DOUBLE PRECISION for pdorm2l
COMPLEX for pcunm2l
COMPLEX*16 for pzunm2l.
Pointer into the local memory to an array, DIMENSION(*lld_c*, *LOCc*(*jc*+*n*−1)).On entry, the local pieces of the distributed matrix sub (*C*).

ic (global) INTEGER.
The row index in the global array *C* indicating the first row of sub(*C*).

jc (global) INTEGER.
The column index in the global array *C* indicating the first column of sub(*C*).

descc (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *C*.

work (local)
REAL for psorm2l
DOUBLE PRECISION for pdorm2l

COMPLEX for pcunm2l
COMPLEX*16 for pzunm2l.
Workspace array, DIMENSION (*lwork*).
On exit, *work*(1) returns the minimal and optimal *lwork*.

lwork (local or global) INTEGER.
The dimension of the array *work*.
lwork is local input and must be at least

if *side* = 'L', *lwork* ≥ *mpc0* + max(1, *nqc0*),

if *side* = 'R', *lwork* ≥ *nqc0* + max(max(1, *mpc0*), numroc(numroc(*n*+*icoffc*, *nb_a*, 0, 0, *npcol*), *nb_a*, 0, 0, *lcmq*)),
where
lcmq = *lcm*/*npcol*,
lcm = iclm(*nprow*, *npcol*),
iroffc = mod(*ic*-1, *mb_c*),
icoffc = mod(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),
iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),
Mqc0 = numroc(*m*+*icoffc*, *nb_c*, *mycol*, *icrow*, *nprow*),
Npc0 = numroc(*n*+*iroffc*, *mb_c*, *myrow*, *iccol*, *npcol*),
ilcm, indxg2p, and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c On exit, sub(*C*) is overwritten by *Q**sub(*C*), or *Q*'*sub(*C*), or sub(*C*)**Q*', or sub(*C*)**Q*.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local) INTEGER.

= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,
then *info* = - (*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value,
then *info* = -*i*.

NOTE. The distributed submatrices *A*(*ia*:*, *ja*:*) and *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) must verify some alignment properties, namely the following expressions should be true:

If *side* = 'L', (*mb_a*.eq.*mb_c* .AND. *iroffa*.eq.*iroffc* .AND. *iarow*.eq.*icrow*)

If *side* = 'R', (*mb_a*.eq.*nb_c* .AND. *iroffa*.eq.*iroffc*).

p?orm2r/p?unm2r

Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by p?geqrf *(unblocked algorithm).*

Syntax

call psorm2r(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pdorm2r(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pcunm2r(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pzunm2r(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Description

The routine p?orm2r/p?unm2r overwrites the general real/complex *m*-by-*n* distributed matrix sub (*C*)=*C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1) with

side = 'L'	*side* = 'R'

trans = 'N'	Q*sub (C)	sub (C)*Q
trans = 'T' (for real flavors)	Q^T * sub(C)	sub(C)*Q^T
trans = 'C' (for complex flavors)	Q^H * sub(C)	sub(C)*Q^H

where Q is a real orthogonal or complex unitary matrix defined as the product of k elementary reflectors

$Q = H(k) . . . H(2) H(1)$

as returned by p?geqrf . Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

side (global) CHARACTER.
= 'L': apply Q or Q^T for real flavors (Q^H for complex flavors) from the left,
= 'R': apply Q or Q^T for real flavors (Q^H for complex flavors) from the right.

trans (global) CHARACTER.
= 'N': apply Q (no transpose)
= 'T': apply Q^T (transpose, for real flavors)
= 'C': apply Q^H (conjugate transpose, for complex flavors)

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(C). $m \geq 0$.

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(C). $n \geq 0$.

k (global) INTEGER.
The number of elementary reflectors whose product defines the matrix Q.
If *side* = 'L', $m \geq k \geq 0$;
if *side* = 'R', $n \geq k \geq 0$.

a (local)

REAL for psorm2r
DOUBLE PRECISION for pdorm2r
COMPLEX for pcunm2r
COMPLEX*16 for pzunm2r.
Pointer into the local memory to an array,
DIMENSION(*lld_a*, *LOCc*(*ja*+*k*-1).
On entry, the *j*-th column must contain the vector that defines the elementary reflector *H*(*j*), *ja* ≤ *j* ≤ *ja*+*k*-1, as returned by p?geqrf in the *k* columns of its distributed matrix argument *A*(*ia*:*,ja*:*ja*+*k*-1). The argument *A*(*ia*:*,ja*:*ja*+*k*-1) is modified by the routine but restored on exit.

If *side* = 'L', lld_a ≥ max(1, *LOCr*(*ia*+*m*-1)),

if *side* = 'R', lld_a ≥ max(1, *LOCr*(*ia*+*n*-1)).

ia (global) INTEGER.
The row index in the global array *A* indicating the first row of sub(*A*).

ja (global) INTEGER.
The column index in the global array *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *A*.

tau (local)
REAL for psorm2r
DOUBLE PRECISION for pdorm2r
COMPLEX for pcunm2r
COMPLEX*16 for pzunm2r.
Array, DIMENSION *LOCc*(*ja*+*k*-1). This array contains the scalar factors *tau*(*j*) of the elementary reflector *H*(*j*), as returned by p?geqrf. This array is tied to the distributed matrix *A*.

c (local)
REAL for psorm2r
DOUBLE PRECISION for pdorm2r
COMPLEX for pcunm2r
COMPLEX*16 for pzunm2r.

Pointer into the local memory to an array, DIMENSION(*lld_c*, *LOCc*(*jc*+*n*-1)).
On entry, the local pieces of the distributed matrix sub (*C*).

ic (global) INTEGER.
The row index in the global array *C* indicating the first row of sub(*C*).

jc (global) INTEGER.
The column index in the global array *C* indicating the first column of sub(*C*).

descc (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *C*.

work (local)
REAL for psorm2r
DOUBLE PRECISION for pdorm2r
COMPLEX for pcunm2r
COMPLEX*16 for pzunm2r.
Workspace array, DIMENSION (*lwork*).

lwork (local or global) INTEGER.
The dimension of the array *work*.
lwork is local input and must be at least

if *side* = 'L', *lwork* ≥ *mpc*0 + max(1, *nqc*0),

if *side* = 'R', *lwork* ≥ *nqc*0 + max(max(1, *mpc*0), numroc(numroc(*n*+*icoffc*, *nb_a*, 0, 0, *npcol*), *nb_a*, 0, 0, *lcmq*)),
where
lcmq = *lcm*/*npcol* ,
lcm = *iclm*(*nprow*, *npcol*),
iroffc = mod(*ic*-*1*, *mb_c*),
icoffc = mod(*jc*-*1*, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),
iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),
Mqc0 = numroc(*m*+*icoffc*, *nb_c*, *mycol*, *icrow*, *nprow*),
Npc0 = numroc(*n*+*iroffc*, *mb_c*, *myrow*, *iccol*, *npcol*),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — On exit, sub(*C*) is overwritten by *Q**sub(*C*), or *Q*'*sub(*C*), or sub(*C*)**Q*', or sub(*C*)**Q*.

work — On exit, *work*(1) returns the minimal and optimal *lwork*.

info — (local) INTEGER.
= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,
then *info* = - (*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value,
then *info* = -*i*.

NOTE. The distributed submatrices *A*(*ia*:*, *ja*:*) and *C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1) must verify some alignment properties, namely the following expressions should be true:

If *side* = 'L', (*mb_a*.eq.*mb_c* .AND. *iroffa*.eq.*iroffc* .AND. *iarow*.eq.*icrow*)

If *side* = 'R', (*mb_a*.eq.*nb_c* .AND. *iroffa*.eq.*iroffc*).

p?orml2/p?unml2

Multiplies a general matrix by the orthogonal/unitary matrix from an LQ factorization determined by p?gelqf *(unblocked algorithm).*

Syntax

call psorml2(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pdorml2(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pcunml2(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pzunml2(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Description

The routine p?orml2/p?unml2 overwrites the general real/complex m-by-n distributed matrix sub $(C)=C(ic:ic+m-1, jc:jc+n-1)$ with

	side = 'L'	*side* = 'R'
trans = 'N'	Q*sub (C)	sub (C)*Q
trans = 'T' (for real flavors)	Q^T * sub(C)	sub(C)*Q^T
trans = 'C' (for complex flavors)	Q^H * sub(C)	sub(C)*Q^H

where Q is a real orthogonal or complex unitary distributed matrix defined as the product of k elementary reflectors

$Q = H(k) . . . H(2) H(1)$ (for real flavors)

$Q = H(k)' . . . H(2)' H(1)'$ (for complex flavors)

as returned by p?gelqf . Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

side (global) CHARACTER.
= 'L': apply Q or Q^T for real flavors (Q^H for complex flavors) from the left,
= 'R': apply Q or Q^T for real flavors (Q^H for complex flavors) from the right.

trans (global) CHARACTER.
= 'N': apply Q (no transpose)
= 'T': apply Q^T (transpose, for real flavors)
= 'C': apply Q^H (conjugate transpose, for complex flavors)

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(C). $m \geq 0$.

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(C). $n \geq 0$.

k (global) INTEGER.
The number of elementary reflectors whose product defines the matrix Q.
If *side* = 'L', $m \geq k \geq 0$;
if *side* = 'R', $n \geq k \geq 0$.

a (local)
REAL for psorml2
DOUBLE PRECISION for pdorml2
COMPLEX for pcunml2
COMPLEX*16 for pzunml2.
Pointer into the local memory to an array, DIMENSION (*lld_a*, *LOCc*(*ja*+*m*-1) if *side*='L',
(*lld_a*, *LOCc*(*ja*+*n*-1) if *side*='R',
where *lld_a* ≥ max (1, *LOCr*(*ia*+*k*-1)).
On entry, the i-th row must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gelqf in the k rows of its distributed matrix

argument *A*(*ia*:*ia*+*k*-1, *ja*:*). The argument *A*(*ia*:*ia*+*k*-1, *ja*:*) is modified by the routine but restored on exit.

ia (global) INTEGER.
The row index in the global array *A* indicating the first row of sub(*A*).

ja (global) INTEGER.
The column index in the global array *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *A*.

tau (local)
REAL for psorml2
DOUBLE PRECISION for pdorml2
COMPLEX for pcunml2
COMPLEX*16 for pzunml2.
Array, DIMENSION *LOCc*(*ia*+*k*-1). This array contains the scalar factors *tau*(*i*) of the elementary reflector *H*(*i*), as returned by p?gelqf. This array is tied to the distributed matrix *A*.

c (local)
REAL for psorml2
DOUBLE PRECISION for pdorml2
COMPLEX for pcunml2
COMPLEX*16 for pzunml2.
Pointer into the local memory to an array, DIMENSION(*lld_c*, *LOCc*(*jc*+*n*-1)). On entry, the local pieces of the distributed matrix sub (*C*).

ic (global) INTEGER.
The row index in the global array *C* indicating the first row of sub(*C*).

jc (global) INTEGER.
The column index in the global array *C* indicating the first column of sub(*C*).

descc (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *C*.

work (local)
REAL for psorml2
DOUBLE PRECISION for pdorml2
COMPLEX for pcunml2
COMPLEX*16 for pzunml2.
Workspace array, DIMENSION (*lwork*).

lwork (local or global) INTEGER.
The dimension of the array *work*.
lwork is local input and must be at least

if *side* = 'L', *lwork* ≥ *mqc*0 + max(max(1, *npc*0), numroc(numroc(*m*+*icoffc*, *mb_a*, 0, 0, *nprow*), *mb_a*, 0, 0, *lcmp*)),

if *side* = 'R', *lwork* ≥ *npc*0 + max(1, *mqc*0),

where
lcmp = *lcm* / *nprow*,
lcm = *iclm*(*nprow*, *npcol*),
iroffc = mod(*ic-1*, *mb_c*),
icoffc = mod(*jc-1*, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),
iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),
Mpc0 = numroc(*m*+*icoffc*, *mb_c*, *mycol*, *icrow*, *nprow*),
Nqc0 = numroc(*n*+*iroffc*, *nb_c*, *myrow*, *iccol*, *npcol*),
ilcm, indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c On exit, sub(*C*) is overwritten by *Q**sub(*C*) ,or *Q*'*sub(*C*), or sub(*C*)**Q*', or sub(*C*)**Q*.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info	(local) INTEGER. = 0: successful exit < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

NOTE. The distributed submatrices *A*(*ia*:*, *ja*:*) and *C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1) must verify some alignment properties, namely the following expressions should be true:

If *side* = 'L', (*nb_a*.eq.*mb_c* .AND. *icoffa*.eq.*iroffc*)

If *side* = 'R', (*nb_a*.eq.*nb_c* .AND. *icoffa*.eq.*icoffc* .AND. *iacol*.eq.*iccol*).

p?ormr2/p?unmr2

Multiplies a general matrix by the orthogonal/unitary matrix from an RQ factorization determined by p?gerqf *(unblocked algorithm).*

Syntax

call psormr2(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pdormr2(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pcunmr2(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pzunmr2(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Description

The routine p?ormr2/p?unmr2 overwrites the general real/complex *m*-by-*n* distributed matrix sub (*C*)=*C*(*ic*:*ic*+*m*-1, *jc*:*jc*+*n*-1) with

side = 'L'	*side* = 'R'

trans = 'N'	Q*sub (C)	sub (C)*Q
trans = 'T' (for real flavors)	Q^T * sub(C)	sub(C)*Q^T
trans = 'C' (for complex flavors)	Q^H * sub(C)	sub(C)*Q^H

where Q is a real orthogonal or complex unitary distributed matrix defined as the product of k elementary reflectors

$Q = H(1)\ H(2). \ . \ . \ H(k)$ (for real flavors)

$Q = H(1)'\ H(2)'\ . \ . \ . \ H(k)'$ (for complex flavors)

as returned by p?gerqf . Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

side (global) CHARACTER.
= 'L': apply Q or Q^T for real flavors (Q^H for complex flavors) from the left,
= 'R': apply Q or Q^T for real flavors (Q^H for complex flavors) from the right.

trans (global) CHARACTER.
= 'N': apply Q (no transpose)
= 'T': apply Q^T (transpose, for real flavors)
= 'C': apply Q^H (conjugate transpose, for complex flavors)

m (global) INTEGER.
The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(C). $m \geq 0$.

n (global) INTEGER.
The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(C). $n \geq 0$.

k (global) INTEGER.
The number of elementary reflectors whose product defines the matrix Q.
If *side* = 'L', $m \geq k \geq 0$;
if *side* = 'R', $n \geq k \geq 0$.

a (local)
REAL for psormr2
DOUBLE PRECISION for pdormr2
COMPLEX for pcunmr2
COMPLEX*16 for pzunmr2.
Pointer into the local memory to an array, DIMENSION (*lld_a*, *LOCc*(*ja*+*m*-1) if *side*='L',
(*lld_a*, *LOCc*(*ja*+*n*-1) if *side*='R',
where *lld_a* ≥ max (1, *LOCr*(*ia*+*k*-1)).
On entry, the *i*-th row must contain the vector that defines the elementary reflector *H*(*i*), *ia* ≤ *i* ≤ *ia*+*k*-1, as returned by p?gerqf in the *k* rows of its distributed matrix argument *A*(*ia*:*ia*+*k*-1, *ja*:*).
The argument *A*(*ia*:*ia*+*k*-1, *ja*:*) is modified by the routine but restored on exit.

ia (global) INTEGER.
The row index in the global array *A* indicating the first row of sub(*A*).

ja (global) INTEGER.
The column index in the global array *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *A*.

tau (local)
REAL for psormr2
DOUBLE PRECISION for pdormr2
COMPLEX for pcunmr2
COMPLEX*16 for pzunmr2.
Array, DIMENSION *LOCc*(*ia*+*k*-1). This array contains the scalar factors *tau*(*i*) of the elementary reflector *H*(*i*), as returned by p?gerqf. This array is tied to the distributed matrix *A*.

c (local)
REAL for psormr2
DOUBLE PRECISION for pdormr2
COMPLEX for pcunmr2

COMPLEX*16 for pzunmr2.
Pointer into the local memory to an array, DIMENSION(*lld_c*, *LOCc*(*jc*+*n*-1)). On entry, the local pieces of the distributed matrix sub (*C*).

ic (global) INTEGER.
The row index in the global array *C* indicating the first row of sub(*C*).

jc (global) INTEGER.
The column index in the global array *C* indicating the first column of sub(*C*).

descc (global and local) INTEGER array of DIMENSION (*dlen_*).
The array descriptor for the distributed matrix *C*.

work (local)
REAL for psormr2
DOUBLE PRECISION for pdormr2
COMPLEX for pcunmr2
COMPLEX*16 for pzunmr2.
Workspace array, DIMENSION (*lwork*).

lwork (local or global) INTEGER.
The dimension of the array *work*.
lwork is local input and must be at least

if *side* = 'L', *lwork* ≥ *mpc*0 + max(max(1, *nqc*0), numroc(numroc(*m*+*iroffc*, *mb_a*, 0, 0, *nprow*), *mb_a*, 0, 0, *lcmp*)),

if *side* = 'R', *lwork* ≥ *nqc*0 + max(1, *mpc*0),

where *lcmp* = *lcm*/*nprow*,
lcm = iclm(*nprow*, *npcol*),
iroffc = mod(*ic*-1, *mb_c*),
icoffc = mod(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),
iccol = indxg2p(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),
Mpc0 = numroc(*m*+*iroffc*, *mb_c*, *myrow*, *icrow*, *nprow*),
Nqc0 = numroc(*n*+*icoffc*, *nb_c*, *mycol*, *iccol*, *npcol*),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c — On exit, sub(*C*) is overwritten by *Q**sub(*C*), or *Q*'*sub(*C*), or sub(*C*)**Q*', or sub(*C*)**Q*.

work — On exit, *work*(1) returns the minimal and optimal *lwork*.

info — (local) INTEGER.
= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,
then *info* = - (*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value,
then *info* = -*i*.

NOTE. The distributed submatrices *A*(*ia*:*, *ja*:*) and *C*(*ic*:*ic*+*m*-1,*jc*:*jc*+*n*-1) must verify some alignment properties, namely the following expressions should be true:

If *side* = 'L', (*nb_a*.eq.*mb_c* .AND. *icoffa*.eq.*iroffc*)

If *side* = 'R', (*nb_a*.eq.*nb_c* .AND. *icoffa*.eq.*icoffc* .AND. *iacol*.eq.*iccol*).

p?pbtrsv

Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a banded matrix computed by p?pbtrf*.*

Syntax

call pspbtrsv(*uplo*, *trans*, *n*, *bw*, *nrhs*, *a*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

call pdpbtrsv(*uplo*, *trans*, *n*, *bw*, *nrhs*, *a*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

call pcpbtrsv(*uplo*, *trans*, *n*, *bw*, *nrhs*, *a*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

call pzpbtrsv(*uplo*, *trans*, *n*, *bw*, *nrhs*, *a*, *ja*, *desca*, *b*, *ib*, *descb*, *af*, *laf*, *work*, *lwork*, *info*)

Description

The routine p?pbtrsv solves a banded triangular system of linear equations

$A(1{:}n,\ ja{:}ja+n-1)*X = B(jb{:}jb+n-1,\ 1{:}nrhs)$

or

$A(1{:}n,\ ja{:}ja+n-1)^T *X = B(jb{:}jb+n-1,\ 1{:}nrhs)$ for real flavors,

$A(1{:}n,\ ja{:}ja+n-1)^H *X = B(jb{:}jb+n-1,\ 1{:}nrhs)$ for complex flavors,

where $A(1{:}n,\ ja{:}ja+n-1)$ is a banded triangular matrix factor produced by the Cholesky factorization code p?pbtrf and is stored in $A(1{:}n,\ ja{:}ja+n-1)$ and *af*. The matrix stored in $A(1{:}n,\ ja{:}ja+n-1)$ is either upper or lower triangular according to *uplo*, and the choice of solving $A(1{:}n,\ ja{:}ja+n-1)$ or $A(1{:}n,\ ja{:}ja+n-1)^T$ for real flavors and $A(1{:}n,\ ja{:}ja+n-1)^H$ for complex flavors respectively is dictated by the user by the parameter *trans*.

Routine p?pbtrf must be called first.

Input Parameters

uplo (global) CHARACTER. Must be 'U' or 'L'.
If *uplo* = 'U', upper triangle of $A(1{:}n,\ ja{:}ja+n-1)$ is stored;

If *uplo* = 'L', lower triangle of *A*(1:*n*, *ja*:*ja*+*n*-1) is stored.

trans (global) CHARACTER. Must be 'N' or 'T' or 'C'.
If *trans* = 'N', solve with *A*(1:*n*, *ja*:*ja*+*n*-1);
If *trans* = 'T' or 'C' for real flavors, solve with $A(1:n, ja:ja+n-1)^T$.
If *trans* = 'C' for complex flavors, solve with conjugate transpose $(A(1:n, ja:ja+n-1)^H$.

n (global) INTEGER.
The number of rows and columns to be operated on, that is, the order of the distributed submatrix *A*(1:*n*, *ja*:*ja*+*n*-1). *n* ≥ 0.

bw (global) INTEGER.
The number of subdiagonals in 'L' or 'U', 0 ≤ *bw* ≤ *n*-1.

nrhs (global) INTEGER.
The number of right hand sides; the number of columns of the distributed submatrix *B*(*jb*:*jb*+*n*-1, 1:*nrhs*); *nrhs* ≥ 0.

a (local)
REAL for pspbtrsv
DOUBLE PRECISION for pdpbtrsv
COMPLEX for pcpbtrsv
COMPLEX*16 for pzpbtrsv.
Pointer into the local memory to an array with the first DIMENSION *lld_a* ≥(*bw*+1), stored in *desca*.
On entry, this array contains the local pieces of the *n*-by-*n* symmetric banded distributed Cholesky factor L or L^T*A(1:*n*, *ja*:*ja*+*n*-1).
This local portion is stored in the packed banded format used in LAPACK. Please see the Application Notes below and the ScaLAPACK manual for more detail on the format of distributed matrices.

ja (global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

If 1D type (*dtype_a* = 501), then *dlen* ≥ 7;

If 2D type (*dtype_a* = 1), then *dlen* ≥ 9.
Contains information on mapping of *A* to memory. Please, see ScaLAPACK manual for full description and options.

b (local)
REAL for pspbtrsv
DOUBLE PRECISION for pdpbtrsv
COMPLEX for pcpbtrsv
COMPLEX*16 for pzpbtrsv.
Pointer into the local memory to an array of local lead

DIMENSION *lld_b* ≥ *nb*.
On entry, this array contains the local pieces of the right hand sides *B*(*jb*:*jb*+*n*-1, 1:*nrhs*).

ib (global) INTEGER. The row index in the global array *B* that points to the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *B*.

If 1D type (*dtype_b* = 502), then *dlen*≥ 7;

If 2D type (*dtype_b* = 1), then *dlen*≥ 9.
Contains information on mapping of *B* to memory. Please, see ScaLAPACK manual for full description and options.

laf (local)
INTEGER. The size of user-input auxiliary Fillin space *af*.

Must be *laf* ≥ (*nb*+2**bw*)**bw* . If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work (local)
REAL for pspbtrsv
DOUBLE PRECISION for pdpbtrsv
COMPLEX for pcpbtrsv
COMPLEX*16 for pzpbtrsv.

	The array *work* is a temporary workspace array of DIMENSION *lwork*. This space may be overwritten in between calls to routines.
lwork	(local or global) INTEGER. The size of the user-input workspace *work*, must be at least *lwork* ≥ *bw*nrhs*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

Output Parameters

af	(local) REAL for pspbtrsv DOUBLE PRECISION for pdpbtrsv COMPLEX for pcpbtrsv COMPLEX*16 for pzpbtrsv. The array *af* is of DIMENSION *laf*. It contains auxiliary Fillin space. Fillin is created during the factorization routine p?pbtrf and this is stored in *af*. If a linear system is to be solved using p?pbtrs after the factorization routine, *af* must not be altered after the factorization.
b	On exit, this array contains the local piece of the solutions distributed matrix *X*.
work(1)	On exit, *work*(1) contains the minimum value of *lwork*.
info	(local) INTEGER. = 0: successful exit < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

If the factorization routine and the solve routine are to be called separately to solve various sets of right-hand sides using the same coefficient matrix, the auxiliary space *af* must not be altered between calls to the factorization routine and the solve routine.

The best algorithm for solving banded and tridiagonal linear systems depends on a variety of parameters, especially the bandwidth. Currently, only algorithms designed for the case `N/P >> bw` are implemented. These algorithms go by many names, including Divide and Conquer, Partitioning, domain decomposition-type, etc.

The Divide and Conquer algorithm assumes the matrix is narrowly banded compared with the number of equations. In this situation, it is best to distribute the input matrix `A` one-dimensionally, with columns atomic and rows divided amongst the processes. The basic algorithm divides the banded matrix up into `P` pieces with one stored on each processor, and then proceeds in 2 phases for the factorization or 3 for the solution of a linear system.

1. **Local Phase**: The individual pieces are factored independently and in parallel. These factors are applied to the matrix creating fill-in, which is stored in a non-inspectable way in auxiliary space `af`. Mathematically, this is equivalent to reordering the matrix `A` as PAP^T and then factoring the principal leading submatrix of size equal to the sum of the sizes of the matrices factored on each processor. The factors of these submatrices overwrite the corresponding parts of `A` in memory.
2. **Reduced System Phase**: A small `(bw*(P-1))` system is formed representing interaction of the larger blocks and is stored (as are its factors) in the space `af`. A parallel Block Cyclic Reduction algorithm is used. For a linear system, a parallel front solve followed by an analogous backsolve, both using the structure of the factored matrix, are performed.
3. **Backsubsitution Phase**: For a linear system, a local backsubstitution is performed on each processor in parallel.

p?pttrsv

Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a tridiagonal matrix computed by p?pttrf *.*

Syntax

call pspttrsv(*uplo, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info*)

call pdpttrsv(*uplo, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info*)

call pcpttrsv(*uplo, trans, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info*)

call pzpttrsv(*uplo, trans, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info*)

Description

This routine solves a tridiagonal triangular system of linear equations

$A(1{:}n,\ ja{:}ja+n-1)*X = B(jb{:}jb+n-1,\ 1{:}nrhs)$

or

$A(1{:}n,\ ja{:}ja+n-1)^{T}*X = B(jb{:}jb+n-1,\ 1{:}nrhs)$ for real flavors,

$A(1{:}n,\ ja{:}ja+n-1)^{H}*X = B(jb{:}jb+n-1,\ 1{:}nrhs)$ for complex flavors,

where $A(1{:}n,\ ja{:}ja+n-1)$ is a tridiagonal triangular matrix factor produced by the Cholesky factorization code p?pttrf and is stored in $A(1{:}n,\ ja{:}ja+n-1)$ and *af*. The matrix stored in $A(1{:}n,\ ja{:}ja+n-1)$ is either upper or lower triangular according to *uplo*, and the choice of solving $A(1{:}n,\ ja{:}ja+n-1)$ or $A(1{:}n,\ ja{:}ja+n-1)^{T}$ for real flavors and $A(1{:}n,\ ja{:}ja+n-1)^{H}$ for complex flavors respectively is dictated by the user by the parameter *trans*.

Routine p?pttrf must be called first.

Input Parameters

uplo (global) CHARACTER. Must be 'U' or 'L'.
If *uplo* = 'U', upper triangle of $A(1{:}n,\ ja{:}ja+n-1)$ is stored;

If *uplo* = 'L', lower triangle of *A*(1:*n*, *ja*:*ja*+*n*-1) is stored.

trans (global) CHARACTER. Must be 'N' or 'C'.
If *trans* = 'N', solve with *A*(1:*n*, *ja*:*ja*+*n*-1);
If *trans* = 'C' (for complex flavors), solve with conjugate transpose $(A(1{:}n, ja{:}ja{+}n{-}1))^H$.

n (global) INTEGER.
The number of rows and columns to be operated on, that is, the order of the distributed submatrix *A*(1:*n*, *ja*:*ja*+*n*-1). *n* ≥ 0.

nrhs (global) INTEGER.
The number of right hand sides; the number of columns of the distributed submatrix *B*(*jb*:*jb*+*n*-1, 1:*nrhs*); *nrhs* ≥ 0.

d (local)
REAL for pspttrsv
DOUBLE PRECISION for pdpttrsv
COMPLEX for pcpttrsv
COMPLEX*16 for pzpttrsv.
Pointer to the local part of the global vector storing the main diagonal of the matrix; must be of size ≥ *desca*(*nb_*).

e (local)
REAL for pspttrsv
DOUBLE PRECISION for pdpttrsv
COMPLEX for pcpttrsv
COMPLEX*16 for pzpttrsv.
Pointer to the local part of the global vector storing the upper diagonal of the matrix; must be of size ≥ *desca*(*nb_*). Globally, *du*(*n*) is not referenced, and *du* must be aligned with *d*.

ja (global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

If 1D type (*dtype_a* = 501 or 502), then *dlen* ≥ 7;

If 2D type (*dtype_a* = 1), then *dlen* ≥ 9.

Contains information on mapping of *A* to memory. Please, see ScaLAPACK manual for full description and options.

b (local)
REAL for pspttrsv
DOUBLE PRECISION for pdpttrsv
COMPLEX for pcpttrsv
COMPLEX*16 for pzpttrsv.
Pointer into the local memory to an array of local lead

DIMENSION *lld_b* ≥ *nb*.

On entry, this array contains the local pieces of the right hand sides *B*(*jb*:*jb*+*n*−1, 1:*nrhs*).

ib (global) INTEGER. The row index in the global array *B* that points to the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *B*.

If 1D type (*dtype_b* = 502), then *dlen* ≥ 7;

If 2D type (*dtype_b* = 1), then *dlen* ≥ 9.

Contains information on mapping of *B* to memory. Please, see ScaLAPACK manual for full description and options.

laf (local)
INTEGER. The size of user-input auxiliary Fillin space *af*.

Must be *laf* ≥ (*nb*+2**bw*)**bw*.

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work (local)
REAL for pspttrsv
DOUBLE PRECISION for pdpttrsv
COMPLEX for pcpttrsv
COMPLEX*16 for pzpttrsv.
The array *work* is a temporary workspace array of DIMENSION *lwork*. This space may be overwritten in between calls to routines.

lwork (local or global) INTEGER. The size of the user-input workspace *work*, must be at least *lwork* ≥(10+2*min(100, *nrhs*))*npcol+4*nrhs*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

Output Parameters

d, *e* (local).
REAL for pspttrsv
DOUBLE PRECISION for pdpttrsv
COMPLEX for pcpttrsv
COMPLEX*16 for pzpttrsv.
On exit, these arrays contain information on the factors of the matrix.

af (local)
REAL for pspttrsv
DOUBLE PRECISION for pdpttrsv
COMPLEX for pcpttrsv
COMPLEX*16 for pzpttrsv.
The array *af* is of DIMENSION *laf*. It contains auxiliary Fillin space. Fillin is created during the factorization routine p?pbtrf and this is stored in *af*. If a linear system is to be solved using p?pttrs after the factorization routine, *af* must not be altered after the factorization.

b On exit, this array contains the local piece of the solutions distributed matrix *X*.

work(1) On exit, *work*(1) contains the minimum value of *lwork*.

info (local) INTEGER.
= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,
then *info* = - (*i*100+*j*),
if the *i*-th argument is a scalar and had an illegal value,
then *info* = -*i*.

p?potf2

Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix (local unblocked algorithm).

Syntax

```
call pspotf2(uplo, n, a, ia, ja, desca, info)
call pdpotf2(uplo, n, a, ia, ja, desca, info)
call pcpotf2(uplo, n, a, ia, ja, desca, info)
call pzpotf2(uplo, n, a, ia, ja, desca, info)
```

Description

This routine computes the Cholesky factorization of a real symmetric or complex Hermitian positive definite distributed matrix sub `(A)=A(ia:ia+n-1, ja:ja+n-1)`.

The factorization has the form

sub `(A) = U' U`, if `uplo` = `'U'`, or

sub `(A) = LL'`, if `uplo` = `'L'`,

where `U` is an upper triangular matrix and `L` is lower triangular.

Input Parameters

uplo	(global) CHARACTER. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix `A` is stored. = `'U'`: upper triangle of sub (`A`) is stored; = `'L'`: lower triangle of sub (`A`) is stored.
n	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub (`A`). `n` ≥ 0.
a	(local) REAL for pspotf2 DOUBLE PRECISION or pdpotf2 COMPLEX for pcpotf2 COMPLEX*16 for pzpotf2.

Pointer into the local memory to an array of DIMENSION(*lld_a*, *LOCc*(*ja*+*n*-1)) containing the local pieces of the *n*-by-*n* symmetric distributed matrix sub(*A*) to be factored.
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular matrix and the strictly lower triangular part of this matrix is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular matrix and the strictly upper triangular part of sub(*A*) is not referenced.

ia, *ja* (global) INTEGER.
The row and column indices in the global array *A* indicating the first row and the first column of the sub(*A*), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

a (local)
On exit,
if *uplo* = 'U', the upper triangular part of the distributed matrix contains the Cholesky factor U;
if *uplo* = 'L', the lower triangular part of the distributed matrix contains the Cholesky factor L.

info (local) INTEGER.
= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,
then *info* = - (*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value,
then *info* = -*i*.
> 0: if *info* = *k*, the leading minor of order *k* is not positive definite, and the factorization could not be completed.

p?rscl

Multiplies a vector by the reciprocal of a real scalar.

Syntax

```
call psrscl(n, sa, sx, ix, jx, descx, incx)
call pdrscl(n, sa, sx, ix, jx, descx, incx)
call pcsrscl(n, sa, sx, ix, jx, descx, incx)
call pzdrscl(n, sa, sx, ix, jx, descx, incx)
```

Description

This routine multiplies an n-element real/complex vector sub(x) by the real scalar 1/a. This is done without overflow or underflow as long as the final result sub(x)/a does not overflow or underflow.

sub(x) denotes $x(ix:ix+n-1, jx:jx)$, if $incx = 1$,

and $x(ix:ix, jx:jx+n-1)$, if $incx = m\_x$.

Input Parameters

n	(global) INTEGER. The number of components of the distributed vector sub(x). $n \geq 0$.
sa	REAL for psrscl/pcsrscl DOUBLE PRECISION for pdrscl/pzdrscl. The scalar a that is used to divide each component of the vector x. This argument must be ≥ 0, or the subroutine will divide by zero.
sx	REAL forpsrscl DOUBLE PRECISION for pdrscl COMPLEX for pcsrscl COMPLEX*16 for pzdrscl. Array containing the local pieces of a distributed matrix of DIMENSION of at least ((*jx*-1)*m_x + *ix* + (*n*-1)*abs(*incx*)). This array contains the entries of the distributed vector sub(x).

ix	(global) INTEGER.The row index of the submatrix of the distributed matrix *X* to operate on.
jx	(global) INTEGER. The column index of the submatrix of the distributed matrix *X* to operate on.
descx	(global and local). INTEGER. Array of DIMENSION 8. The array descriptor for the distributed matrix *X*.
incx	(global) INTEGER. The increment for the elements of *X*. This version supports only two values of *incx*, namely 1 and *m_x*.

Output Parameters

sx	On exit, the result *x*/*a*.

p?sygs2/p?hegs2

Reduces a symmetric/Hermitian definite generalized eigenproblem to standard form, using the factorization results obtained from p?potrf *(local unblocked algorithm).*

Syntax

```
call pssygs2(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
call pdsygs2(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
call pchegs2(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
call pzhegs2(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
```

Description

The routine p?sygs2/p?hegs2 reduces a real symmetric-definite or a complex Hermitian-definite generalized eigenproblem to standard form.

sub(*A*) denotes *A*(*ia*:*ia*+*n*–1, *ja*:*ja*+*n*–1) and sub(*B*) denotes *B*(*ib*:*ib*+*n*–1, *jb*:*jb*+*n*–1).

If *ibtype* = 1, the problem is

sub(*A*)**x* = λ*sub(*B*)**x*,

and sub(*A*) is overwritten by

inv(*U*^{T})*sub(*A*)*inv(*U*) or inv(*L*)*sub(*A*)*inv(*L*^{T}) for real flavors and

inv(*U*^{H})*sub(*A*)*inv(*U*) or inv(*L*)*sub(*A*)*inv(*L*^{H}) for complex flavors.

If *ibtype* = 2 or 3, the problem is

sub(*A*)*sub(*B*)*x* = λ*x* or sub(*B*)*sub(*A*)*x* =λ*x*,

and sub(*A*) is overwritten

by *U**sub(*A*)**U*^{T} or *L****T**sub(*A*)**L* for real flavors and

by *U**sub(*A*)**U*^{H} or *L****H**sub(*A*)**L* for complex flavors.

sub(*B*) must have been previously factorized as *U*^{T}**U* or *L***L*^{T} (for real flavors) or as *U*^{H}**U* or *L***L*^{H} (for complex flavors) by p?potrf.

Input Parameters

ibtype	(global) INTEGER. = 1: compute inv(*U*^{T})*sub(*A*)*inv(*U*) or inv(*L*)*sub(*A*)*inv(*L*^{T}) for real subroutines, and inv(*U*^{H})*sub(*A*)*inv(*U*) or inv(*L*)*sub(*A*)*inv(*L*^{H}) for complex subroutines; = 2 or 3: compute *U**sub(*A*)**U*^{T} or *L*^{T}*sub(*A*)**L* for real subroutines, and by *U**sub(*A*)**U*^{H} or *L*^{H}*sub(*A*)**L* for complex subroutines.
uplo	(global) CHARACTER Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix sub(*A*) is stored, and how sub(*B*) is factorized. = 'U': Upper triangular of sub(*A*) is stored and sub(*B*) is factorized as *U*^{T}**U* (for real subroutines) or as *U*^{H}**U* (for complex subroutines). = 'L': Lower triangular of sub(*A*) is stored and sub(*B*) is factorized as *L***L*^{T} (for real subroutines) or as *L***L*^{H} (for complex subroutines)
n	(global) INTEGER. The order of the matrices sub(*A*) and sub(*B*). *n* ≥ 0.
a	(local)

REAL for pssygs2
DOUBLE PRECISION for pdsygs2
COMPLEX for pchegs2
COMPLEX*16 for pzhegs2.
Pointer into the local memory to an array,
DIMENSION(*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the *n*-by-*n* symmetric/Hermitian distributed matrix sub(*A*).
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and the strictly lower triangular part of sub(*A*) is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and the strictly upper triangular part of sub(*A*) is not referenced.

ia, *ja* (global) INTEGER.
The row and column indices in the global array *A* indicating the first row and the first column of the sub(*A*), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

B (local)
REAL for pssygs2
DOUBLE PRECISION for pdsygs2
COMPLEX for pchegs2
COMPLEX*16 for pzhegs2.
Pointer into the local memory to an array,
DIMENSION(*lld_b*, *LOCc*(*jb*+*n*-1)).
On entry, this array contains the local pieces of the triangular factor from the Cholesky factorization of sub(*B*) as returned by p?potrf.

ib, *jb* (global) INTEGER.
The row and column indices in the global array *B* indicating the first row and the first column of the sub(*B*), respectively.

descb (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *B*.

Output Parameters

a	(local) On exit, if *info* = 0, the transformed matrix is stored in the same format as sub(*A*).
info	INTEGER. = 0: successful exit. < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?sytd2/p?hetd2

Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (local unblocked algorithm).

Syntax

```
call pssytd2(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pdsytd2(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pchetd2(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pzhetd2(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
```

Description

The routine p?sytd2/p?hetd2 reduces a real symmetric/complex Hermitian matrix sub(*A*) to symmetric/Hermitian tridiagonal form *T* by an orthogonal/unitary similarity transformation:

Q' sub(*A*)*Q* = *T*, where sub(*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1).

Input Parameters

uplo	(global) CHARACTER. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix sub(*A*) is stored: = 'U': upper triangular

= 'L': lower triangular

n (global) INTEGER.
The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub(*A*). *n* ≥ 0.

a (local)
REAL for pssytd2
DOUBLE PRECISION for pdsytd2
COMPLEX for pchetd2
COMPLEX*16 for pzhetd2.
Pointer into the local memory to an array, DIMENSION(*lld_a*, *LOCc*(*ja*+*n*−1)).
On entry, this array contains the local pieces of the *n*-by-*n* symmetric/Hermitian distributed matrix sub(*A*).
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and the strictly lower triangular part of sub(*A*) is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and the strictly upper triangular part of sub(*A*) is not referenced.

ia, *ja* (global) INTEGER.
The row and column indices in the global array *A* indicating the first row and the first column of the sub(*A*), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

work (local)
REAL for pssytd2
DOUBLE PRECISION for pdsytd2
COMPLEX for pchetd2
COMPLEX*16 for pzhetd2.
The array *work* is a temporary workspace array of DIMENSION *lwork*.

Output Parameters

a On exit, if *uplo* = 'U', the diagonal and first superdiagonal of sub(*A*) are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements above the first

	superdiagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors; if *uplo* = 'L', the diagonal and first subdiagonal of *A* are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements below the first subdiagonal, with the array *tau,* represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors. See the Application Notes below.
d	(local) REAL for pssytd2/pchetd2 DOUBLE PRECISION for pdsytd2/pzhetd2. Array, DIMENSION(*LOCc*(*ja*+*n*-1)). The diagonal elements of the tridiagonal matrix *T*: *d*(i) = *a*(i,i); *d* is tied to the distributed matrix *A*.
e	(local) REAL for pssytd2/pchetd2 DOUBLE PRECISION for pdsytd2/pzhetd2. Array, DIMENSION(*LOCc*(*ja*+*n*-1)), if *uplo* = 'U', *LOCc*(*ja*+*n*-2) otherwise. The off-diagonal elements of the tridiagonal matrix *T*: *e*(i) = *a*(i,i+1) if *uplo* = 'U', *e*(i) = *a*(i+1,i) if *uplo* = 'L'. *e* is tied to the distributed matrix *A*.
tau	(local) REAL for pssytd2 DOUBLE PRECISION for pdsytd2 COMPLEX for pchetd2 COMPLEX*16 for pzhetd2. Array, DIMENSION(*LOCc*(*ja*+*n*-1)). The scalar factors of the elementary reflectors. *tau* is tied to the distributed matrix *A*.
work(1)	On exit, *work*(1) returns the minimal and optimal value of *lwork*.
lwork	(local or global) INTEGER. The dimension of the workspace array *work*. *lwork* is local input and must be at least *lwork* ≥ 3*n*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

info (local) INTEGER.
= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,
then *info* = - (*i**100),
if the *i*-th argument is a scalar and had an illegal value,
then *info* = -*i*.

Application Notes

If *uplo* = 'U', the matrix *Q* is represented as a product of elementary reflectors

Q = *H*(*n*-*1*). . . *H*(2) *H*(1)

Each *H*(i) has the form

H(i) = *I* - *tau***v***v*',

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(*i*+1:*n*) = 0 and *v*(*i*) = 1; *v*(1:*i*-1) is stored on exit in *A*(*ia*:*ia*+*i*-2, *ja*+*i*), and *tau* in *TAU*(*ja*+*i*-1).

If *uplo* = 'L', the matrix *Q* is represented as a product of elementary reflectors

Q = *H*(1) *H*(2) . . . *H*(*n*-1).

Each *H*(i) has the form

H(i) = *I* - *tau***v***v*' ,

where *tau* is a real/complex scalar, and *v* is a real/complex vector with *v*(1:*i*) = 0 and *v*(*i*+1) = 1; *v*(*i*+2:*n*) is stored on exit in *A*(*ia*+*i*+1:*ia*+*n*-1, *ja*+*i*-1), and *tau* in *TAU*(*ja*+*i*-1).

The contents of sub (*A*) on exit are illustrated by the following examples with *n* = 5:

if *uplo*='U':

$$\begin{bmatrix} d & e & v_2 & v_3 & v_4 \\ & d & e & v_3 & v_4 \\ & & d & e & v_4 \\ & & & d & e \\ & & & & d \end{bmatrix}$$

if *uplo*='L':

$$\begin{bmatrix} d & & & & \\ e & d & & & \\ v1 & e & d & & \\ v1 & v_2 & e & d & \\ v1 & v_2 & v_3 & e & d \end{bmatrix}$$

where d and e denotes diagonal and off-diagonal elements of T, and v_i denotes an element of the vector defining H(i).

NOTE. The distributed submatrix sub(*A*) must verify some alignment properties, namely the following expression should be true:

(*mb_a*.eq.*nb_a* .AND. *iroffa*.eq.*icoffa*) with *iroffa* = mod(*ia* - 1, *mb_a*) and *icoffa* = mod(*ja* -1, *nb_a*).

p?trti2

Computes the inverse of a triangular matrix (local unblocked algorithm).

Syntax

```
call pstrti2(uplo, diag, n, a, ia, ja, desca, info)
call pdtrti2(uplo, diag, n, a, ia, ja, desca, info)
call pctrti2(uplo, diag, n, a, ia, ja, desca, info)
call pztrti2(uplo, diag, n, a, ia, ja, desca, info)
```

Description

This routine computes the inverse of a real/complex upper or lower triangular block matrix sub (*A*) = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1).

This matrix should be contained in one and only one process memory space (local operation).

Input Parameters

uplo (global) CHARACTER*1.
Specifies whether the matrix sub (*A*) is upper or lower triangular.
= 'U': sub (*A*) is upper triangular
= 'L': sub (*A*) is lower triangular.

diag (global) CHARACTER*1.
Specifies whether or not the matrix *A* is unit triangular.
= 'N': sub (*A*) is non-unit triangular
= 'U': sub (*A*) is unit triangular.

n (global) INTEGER.
The number of rows and columns to be operated on, i.e., the order of the distributed submatrix sub(*A*). $n \geq 0$.

a (local)
REAL for pstrti2
DOUBLE PRECISION for pdtrti2
COMPLEX for pctrti2
COMPLEX*16 for pztrti2.
Pointer into the local memory to an array, DIMENSION(*lld_a*, *LOCc*(*ja*+*n*-1)).
On entry, this array contains the local pieces of the triangular matrix sub(*A*).
If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of the matrix sub(*A*) contains the upper triangular part of the matrix, and the strictly lower triangular part of sub(*A*) is not referenced.
If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of the matrix sub(*A*) contains the lower triangular part of the matrix, and the strictly upper triangular part of sub(*A*) is not referenced. If *diag* = 'U', the diagonal elements of sub(*A*) are not referenced either and are assumed to be 1.

ia, *ja* (global) INTEGER.
The row and column indices in the global array *A* indicating the first row and the first column of the sub(*A*), respectively.

desca (global and local) INTEGER array, DIMENSION (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

a	On exit, the (triangular) inverse of the original matrix, in the same storage format.
info	INTEGER. = 0: successful exit < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

?lamsh

Sends multiple shifts through a small (single node) matrix to maximize the number of bulges that can be sent through.

Syntax

```
call slamsh(s, lds, nbulge, jblk, h, ldh, n, ulp)
call dlamsh(s, lds, nbulge, jblk, h, ldh, n, ulp)
```

Description

This routine sends multiple shifts through a small (single node) matrix to see how small consecutive subdiagonal elements are modified by subsequent shifts in an effort to maximize the number of bulges that can be sent through. The subroutine should only be called when there are multiple shifts/bulges (*nbulge* > 1) and the first shift is starting in the middle of an unreduced Hessenberg matrix because of two or more small consecutive subdiagonal elements.

Input Parameters

s	(local) INTEGER. REAL for slamsh DOUBLE PRECISION for dlamsh Array, DIMENSION (*lds*,*).

On entry, the matrix of shifts. Only the 2x2 diagonal of *s* is referenced. It is assumed that *s* has *jblk* double shifts (size 2).

lds (local) INTEGER.
On entry, the leading dimension of *S*; unchanged on exit.
$1 < nbulge \leq jblk \leq lds/2$.

nbulge (local) INTEGER.
On entry, the number of bulges to send through *h* (>1). *nbulge* should be less than the maximum determined (*jblk*). $1 < nbulge \leq jblk \leq lds/2$.

jblk (local) INTEGER.
On entry, the leading dimension of *S*; unchanged on exit.

h (local) INTEGER.
REAL for slamsh
DOUBLE PRECISION for dlamsh
Array, DIMENSION (*lds*, *n*).
On entry, the local matrix to apply the shifts on.
h should be aligned so that the starting row is 2.

ldh (local)
INTEGER.
On entry, the leading dimension of *H*; unchanged on exit.

n (local) INTEGER.
On entry, the size of *H*. If all the bulges are expected to go through, *n* should be at least $4nbulge+2$. Otherwise, *nbulge* may be reduced by this routine.

ulp (local)
REAL for slamsh
DOUBLE PRECISION for dlamsh
On entry, machine precision. Unchanged on exit.

Output Parameters

s On exit, the data is rearranged in the best order for applying.

nbulge On exit, the maximum number of bulges that can be sent through.

h On exit, the data is destroyed.

?laref

Applies Householder reflectors to matrices on either their rows or columns.

Syntax

```
call slaref(type, a, lda, wantz, z, ldz, block, irow1, icol1, istart, istop,
itmp1, itmp2, liloz, lihiz, vecs, v2, v3, t1, t2, t3)
```

```
call dlaref(type, a, lda, wantz, z, ldz, block, irow1, icol1, istart, istop,
itmp1, itmp2, liloz, lihiz, vecs, v2, v3, t1, t2, t3)
```

Description

This routine applies one or several Householder reflectors of size 3 to one or two matrices (if column is specified) on either their rows or columns.

Input Parameters

type (global) CHRACTER*1.
If *type* = 'R', apply reflectors to the rows of the matrix (apply from left). Otherwise, apply reflectors to the columns of the matrix. Unchanged on exit.

a (global) REAL for slaref
DOUBLE PRECISION for dlaref
Array, DIMENSION (*lda*, *).
On entry, the matrix to receive the reflections.

lda (local) INTEGER.
On entry, the leading dimension of *A*; unchanged on exit.

wantz (global) LOGICAL.
If *wantz* = .TRUE., apply any column reflections to *Z* as well.
If *wantz* = .FALSE., do no additional work on *Z*.

z (global) REAL for slaref
DOUBLE PRECISION for dlaref
Array, DIMENSION (*ldz*, *).
On entry, the second matrix to receive column reflections.

ldz (local) INTEGER.
On entry, the leading dimension of *Z*; unchanged on exit.

block (global). LOGICAL.
= .TRUE. : apply several reflectors at once and read their data from the *vecs* array;
= .FALSE. : apply the single reflector given by *v2*, *v3*, *t1*, *t2*, and *t3*.

irow1 (local) INTEGER.
On entry, the local row element of the matrix A.

icol1 (local) INTEGER.
On entry, the local column element of the matrix A.

istart (global) INTEGER.
Specifies the "number" of the first reflector.
istart is used as an index into *vecs* if *block* is set. *istart* is ignored if *block* is .FALSE. .

istop (global) INTEGER.
Specifies the "number" of the last reflector.
istop is used as an index into *vecs* if *block* is set. *istop* is ignored if *block* is .FALSE. .

itmp1 (local) INTEGER.
Starting range into *A*. For rows, this is the local first column. For columns, this is the local first row.

itmp2 (local) INTEGER.
Ending range into *A*. For rows, this is the local last column. For columns, this is the local last row.

liloz, *lihiz* (local). INTEGER.
Serve the same purpose as *itmp1*, *itmp2* but for *Z* when *wantz* is set.

vecs (global)
REAL for slaref
DOUBLE PRECISION for dlaref.
Array of size 3 **n* (matrix size). This array holds the size 3 reflectors one after another and is only accessed when *block* is .TRUE.

v2,*v3*,*t1*,*t2*,*t3* (global). INTEGER.
REAL for slaref
DOUBLE PRECISION for dlaref.

	These parameters hold information on a single size 3 Householder reflector and are read when *block* is .FALSE., and overwritten when *block* is .TRUE..

Output Parameters

a	On exit, the updated matrix.
z	Changed only if *wantz* is set. If *wantz* is .FALSE. , *z* is not referenced.
irow1	Undefined.
icol1	Undefined.
v2,*v3*,*t1*,*t2*,*t3*	These parameters are read when *block* is .FALSE., and overwritten when *block* is .TRUE..

?lasorte

Sorts eigenpairs by real and complex data types.

Syntax

```
call slasorte(s, lds, j, out, info)
call dlasorte(s, lds, j, out, info)
```

Description

This routine sorts eigenpairs so that real eigenpairs are together and complex eigenpairs are together. This helps to employ 2x2 shifts easily since every 2nd subdiagonal is guaranteed to be zero. This routine does no parallel work and makes no calls.

Input Parameters

s	(local) INTEGER. REAL for slasorte DOUBLE PRECISION for dlasorte Array, DIMENSION (*lds*). On entry, a matrix already in Schur form.
lds	(local) INTEGER. On entry, the leading dimension of the array *s*; unchanged on exit.

j	(local) INTEGER. On entry, the order of the matrix *S*; unchanged on exit.
out	(local) INTEGER. REAL for slasorte DOUBLE PRECISION for dlasorte Array, DIMENSION (2**j*). The work buffer required by the routine.
info	(local) INTEGER. Set, if the input matrix had an odd number of real eigenvalues and things could not be paired or if the input matrix *S* was not originally in Schur form. 0 indicates successful completion.

Output Parameters

s	On exit, the diagonal blocks of *S* have been rewritten to pair the eigenvalues. The resulting matrix is no longer similar to the input.
out	Work buffer.

?lasrt2

Sorts numbers in increasing or decreasing order.

Syntax

```
call slasrt2(id, n, d, key, info)
call dlasrt2(id, n, d, key, info)
```

Description

This routine is modified LAPACK routine ?lasrt, which sorts the numbers in *d* in increasing order (if *id* = 'I') or in decreasing order (if *id* = 'D'). It uses Quick Sort, reverting to Insertion Sort on arrays of size ≤ 20. Dimension of STACK limits *n* to about 2^{32}.

Input Parameters

id	CHARACTER*1. = 'I': sort *d* in increasing order;

	= 'D': sort *d* in decreasing order.
n	INTEGER. The length of the array *d*.
d	REAL for slasrt2 DOUBLE PRECISION for dlasrt2. Array, DIMENSION (*n*). On entry, the array to be sorted.
key	INTEGER. Array, DIMENSION (*n*). On entry, *key* contains a key to each of the entries in *d*(). Typically, *key*(*i*) = *i* for all *i* .

Output Parameters

D	On exit, *d* has been sorted into increasing order (*d*(1) ≤ ... ≤ *d*(*n*)) or into decreasing order (*d*(1) ≥ ... ≥ *d*(*n*)), depending on *id*.
info	INTEGER. = 0: successful exit < 0: if *info* = -i, the *i*-th argument had an illegal value.
key	On exit, *key* is permuted in exactly the same manner as *d*() was permuted from input to output. Therefore, if *key*(*i*) = *i* for all *i* upon input, then *d*_out(*i*) = *d*_in(*key*(*i*)).

?stein2

Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix, using inverse iteration.

Syntax

call sstein2(*n*, *d*, *e*, *m*, *w*, *iblock*, *isplit*, *orfac*, *z*, *ldz*, *work*, *iwork*, *ifail*, *info*)

call dstein2(*n*, *d*, *e*, *m*, *w*, *iblock*, *isplit*, *orfac*, *z*, *ldz*, *work*, *iwork*, *ifail*, *info*)

Description

This routine is a modified LAPACK routine ?stein. It computes the eigenvectors of a real symmetric tridiagonal matrix *T* corresponding to specified eigenvalues, using inverse iteration.

The maximum number of iterations allowed for each eigenvector is specified by an internal parameter *maxits* (currently set to 5).

Input Parameters

n	INTEGER. The order of the matrix *T* (*n* ≥ 0).
m	INTEGER. The number of eigenvectors to be found (0 ≤ *m* ≤ *n*).
d, *e*, *w*	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: *d*(*), DIMENSION (*n*). The *n* diagonal elements of the tridiagonal matrix *T*. *e*(*), DIMENSION (*n*). The (*n*-1) subdiagonal elements of the tridiagonal matrix *T*, in elements 1 to *n*-1. *e*(*n*) need not be set. *w*(*), DIMENSION (*n*). The first *m* elements of *w* contain the eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block. (The output array *w* from ?stebz with ORDER = 'B' is expected here). The dimension of *w* must be at least max(1, *n*).
iblock	INTEGER. Array, DIMENSION (*n*). The submatrix indices associated with the corresponding eigenvalues in *w* ; *iblock*(*i*) = 1, if eigenvalue *w*(*i*) belongs to the first submatrix from the top, *iblock*(*i*) = 2, if eigenvalue *w*(*i*) belongs to the second submatrix, etc. (The output array *iblock* from ?stebz is expected here).
isplit	INTEGER. Array, DIMENSION (*n*).

	The splitting points, at which *T* breaks up into submatrices. The first submatrix consists of rows/columns 1 to *isplit*(1), the second submatrix consists of rows/columns *isplit*(1)+1 through *isplit*(2), etc. (The output array *isplit* from ?stebz is expected here).
orfac	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. *orfac* specifies which eigenvectors should be orthogonalized. Eigenvectors that correspond to eigenvalues which are within *orfac**|| *T* || of each other are to be orthogonalized.
ldz	INTEGER. The leading dimension of the output array *z*; *ldz* ≥ max(1, *n*).
work	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Workspace array, DIMENSION (5*n*).
iwork	INTEGER. Workspace array, DIMENSION (*n*).

Output Parameters

z	REAL for sstein2 DOUBLE PRECISION for dstein2 Array, DIMENSION (*ldz*, *m*). The computed eigenvectors. The eigenvector associated with the eigenvalue *w*(*i*) is stored in the *i*-th column of *z*. Any vector that fails to converge is set to its current iterate after *maxits* iterations.
ifail	INTEGER. Array, DIMENSION (*m*). On normal exit, all elements of *ifail* are zero. If one or more eigenvectors fail to converge after *maxits* iterations, then their indices are stored in the array *ifail*.
info	INTEGER. *info* = 0, the exit is successful. *info* < 0: if *info* = –*i*, the *i*-th had an illegal value. *info* > 0: if *info* = *i*, then *i* eigenvectors failed to converge in *maxits* iterations. Their indices are stored in the array *ifail*.

?dbtf2

Computes an LU factorization of a general band matrix with no pivoting (local unblocked algorithm).

Syntax

```
call sdbtf2(m, n, kl, ku, ab, ldab, info)
call ddbtf2(m, n, kl, ku, ab, ldab, info)
call cdbtf2(m, n, kl, ku, ab, ldab, info)
call zdbtf2(m, n, kl, ku, ab, ldab, info)
```

Description

This routine computes an *LU* factorization of a general real/complex *m*-by-*n* band matrix *A* without using partial pivoting with row interchanges.

This is the unblocked version of the algorithm, calling BLAS Routines and Functions.

Input Parameters

m — INTEGER. The number of rows of the matrix *A*(*m* ≥ 0).

n — INTEGER. The number of columns in *A*(*n* ≥ 0).

kl — INTEGER. The number of sub-diagonals within the band of *A*(*kl* ≥ 0).

ku — INTEGER. The number of super-diagonals within the band of *A*(*ku* ≥ 0).

ab — REAL for sdbtf2
DOUBLE PRECISION for ddbtf2
COMPLEX for cdbtf2
COMPLEX*16 for zdbtf2.
Array, DIMENSION (*ldab*, *n*).
The matrix *A* in band storage, in rows *kl*+1 to 2*kl*+*ku*+1; rows 1 to *kl* of the array need not be set. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:
ab(*kl*+*ku*+1+*i*-*j*,*j*) = *A*(*i*,*j*) for max(1,*j*-*ku*) ≤ *i* ≤ min(*m*,*j*+*kl*).

ldab	INTEGER. The leading dimension of the array *ab*. (*ldab* ≥ 2*kl* + *ku* +1)

Output Parameters

ab	On exit, details of the factorization: *U* is stored as an upper triangular band matrix with *kl*+*ku* superdiagonals in rows 1 to *kl*+*ku*+1, and the multipliers used during the factorization are stored in rows *kl*+*ku*+2 to 2**kl*+*ku*+1. See the Application Notes below for further details.
info	INTEGER. = 0: successful exit < 0: if *info* = – *i*, the *i*-th argument had an illegal value, > 0: if *info* = + *i*, *u*(*i*,*i*) is 0. The factorization has been completed, but the factor *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

Application Notes

The band storage scheme is illustrated by the following example, when *m* = *n* = 6, *kl* = 2, *ku* = 1:

on entry

$$\begin{bmatrix} * & a12 & a23 & a34 & a45 & a56 \\ a11 & a22 & a33 & a44 & a55 & a66 \\ a21 & a32 & a43 & a54 & a65 & * \\ a31 & a42 & a53 & a64 & * & * \end{bmatrix}$$

on exit

$$\begin{bmatrix} * & u12 & u23 \\ u11 & u22 & u33 \\ m21 & m32 & m43 \\ m31 & m42 & m53 \end{bmatrix}$$

The routine does not use array elements marked *; elements marked + need not be set on entry, but the routine requires them to store elements of *U*, because of fill-in resulting from the row interchanges.

?dbtrf

Computes an LU factorization of a general band matrix with no pivoting (local blocked algorithm).

Syntax

```
call sdbtrf(m, n, kl, ku, ab, ldab, info)
call ddbtrf(m, n, kl, ku, ab, ldab, info)
call cdbtrf(m, n, kl, ku, ab, ldab, info)
call zdbtrf(m, n, kl, ku, ab, ldab, info)
```

Description

This routine computes an LU factorization of a real m-by-n band matrix A without using partial pivoting or row interchanges.

This is the blocked version of the algorithm, calling BLAS Routines and Functions.

Input Parameters

m	INTEGER. The number of rows of the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
kl	INTEGER. The number of sub-diagonals within the band of A ($kl \geq 0$).
ku	INTEGER. The number of super-diagonals within the band of A ($ku \geq 0$).
ab	REAL for sdbtrf DOUBLE PRECISION for ddbtrf COMPLEX for cdbtrf COMPLEX*16 for zdbtrf. Array, DIMENSION (*ldab*, *n*). The matrix A in band storage, in rows $kl+1$ to $2kl+ku+1$; rows 1 to kl of the array need not be set. The j-th column of A is stored in the j-th column of the array *ab* as follows: $ab(kl+ku+1+i-j,j) = A(i,j)$ for $\max(1,j-ku) \leq i \leq \min(m,j+kl)$.

ldab	INTEGER. The leading dimension of the array *ab*. (*ldab* ≥ 2*kl* + *ku* +1)

Output Parameters

ab	On exit, details of the factorization: *U* is stored as an upper triangular band matrix with *kl*+*ku* superdiagonals in rows 1 to *kl*+*ku*+1, and the multipliers used during the factorization are stored in rows *kl*+*ku*+2 to 2**kl*+*ku*+1. See the Application Notes below for further details.
info	INTEGER. = 0: successful exit < 0: if *info* = – *i*, the *i*-th argument had an illegal value, > 0: if *info* = + *i*, *u*(*i*,*i*) is 0. The factorization has been completed, but the factor *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

Application Notes

The band storage scheme is illustrated by the following example, when *m* = *n* = 6, *kl* = 2, *ku* = 1:

on entry

$$\begin{bmatrix} * & a12 & a23 & a34 & a45 & a56 \\ a11 & a22 & a33 & a44 & a55 & a66 \\ a21 & a32 & a43 & a54 & a65 & * \\ a31 & a42 & a53 & a64 & * & * \end{bmatrix}$$

on exit

$$\begin{bmatrix} * & u12 & u23 \\ u11 & u22 & u33 \\ m21 & m32 & m43 \\ m31 & m42 & m53 \end{bmatrix}$$

The routine does not use array elements marked *.

?dttrf

Computes an LU factorization of a general tridiagonal matrix with no pivoting (local blocked algorithm).

Syntax

```
call sdttrf(n, dl, d, du, info)
call ddttrf(n, dl, d, du, info)
call cdttrf(n, dl, d, du, info)
call zdttrf(n, dl, d, du, info)
```

Description

This routine computes an *LU* factorization of a real or complex tridiagonal matrix *A* using elimination without partial pivoting.

The factorization has the form *A = L*U*, where *L* is a product of unit lower bidiagonal matrices and *U* is upper triangular with nonzeros only in the main diagonal and first superdiagonal.

Input Parameters

n	INTEGER. The order of the matrix *A*(*n* ≥ 0).
dl, *d*, *du*	REAL for sdttrf DOUBLE PRECISION for ddttrf COMPLEX for cdttrf COMPLEX*16 for zdttrf. Arrays containing elements of *A*. The array *dl* of DIMENSION(*n* – 1) contains the sub-diagonal elements of *A*. The array *d* of DIMENSION *n* contains the diagonal elements of *A*. The array *du* of DIMENSION(*n* – 1) contains the super-diagonal elements of *A*.

Output Parameters

dl	Overwritten by the (*n*–1) multipliers that define the matrix *L* from the *LU* factorization of *A*.

d	Overwritten by the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*.
du	Overwritten by the (*n*-1) elements of the first super-diagonal of *U*.
info	INTEGER. = 0: successful exit < 0: if *info* = -*i*, the *i*-th argument had an illegal value, > 0: if *info* = *i*, *u*(*i*,*i*) is exactly 0. The factorization has been completed, but the factor *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

?dttrsv

Solves a general tridiagonal system of linear equations using the LU factorization computed by ?dttrf.

Syntax

```
call sdttrsv(uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
call ddttrsv(uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
call cdttrsv(uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
call zdttrsv(uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
```

Description

This routine solves one of the following systems of linear equations:

$L*X = B$, $L^T*X = B$, or $L^H*X = B$,

$U*X = B$, $U^T*X = B$, or $U^H*X = B$

with factors of the tridiagonal matrix *A* from the *LU* factorization computed by ?dttrf.

Input Parameters

uplo	CHARACTER*1. Specifies whether to solve with *L* or *U*.
trans	CHARACTER. Must be 'N' or 'T' or 'C'.

	Indicates the form of the equations: If *trans* = 'N', then *A*X* = *B* is solved for *X* (no transpose). If *trans* = 'T', then $A^T*X = B$ is solved for *X* (transpose). If *trans* = 'C', then $A^H*X = B$ is solved for *X* (conjugate transpose).
n	INTEGER. The order of the matrix *A*(*n* ≥ 0).
nrhs	INTEGER. The number of right-hand sides, that is, the number of columns in the matrix *B*(*nrhs* ≥ 0).
dl,*d*,*du*,*b*	REAL for sdttrsv DOUBLE PRECISION for ddttrsv COMPLEX for cdttrsv COMPLEX*16 for zdttrsv. Arrays of DIMENSIONs: *dl*(*n* -1), *d*(*n*), *du*(*n* -1), *b*(*ldb*,*nrhs*). The array *dl* contains the (*n* – 1) multipliers that define the matrix *L* from the *LU* factorization of *A*. The array *d* contains *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*. The array *du* contains the (*n* – 1) elements of the first super-diagonal of *U*. On entry, the array *b* contains the right-hand side matrix *B*.
ldb	INTEGER. The leading dimension of the array *b*; *ldb* ≥ max(1, *n*).

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

?pttrsv

*Solves a symmetric (Hermitian) positive-definite tridiagonal system of linear equations, using the L*D*L^H factorization computed by* ?pttrf.

Syntax

```
call spttrsv(trans, n, nrhs, d, e, b, ldb, info)
call dpttrsv(trans, n, nrhs, d, e, b, ldb, info)
call cpttrsv(uplo, trans, n, nrhs, d, e, b, ldb, info)
call zpttrsv(uplo, trans, n, nrhs, d, e, b, ldb, info)
```

Description

This routine solves one of the triangular systems:

$L^T*X = B$, or $L*X = B$ for real flavors,

or

$L*X = B$, or $L^H*X = B$,

$U*X = B$, or $U^H*X = B$ for complex flavors,

where L (or U for complex flavors) is the Cholesky factor of a Hermitian positive-definite tridiagonal matrix A such that

$A = L*D*L^H$ (computed by spttrf/dpttrf)

or

$A = U^H*D*U$ or $A = L*D*L^H$ (computed by cpttrf/zpttrf).

Input Parameters

uplo — CHARACTER*1. Must be 'U' or 'L'.
Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix A is stored and the form of the factorization:
If *uplo* = 'U', e is the superdiagonal of U, and $A = U'*D*U$;
if *uplo* = 'L', e is the subdiagonal of L, and $A = L*D*L'$.
The two forms are equivalent, if A is real.

trans CHARACTER.
Specifies the form of the system of equations:
for real flavors:
if *trans* = 'N': $L*X = B$ (no transpose)
if *trans* = 'T': $L^T*X = B$ (transpose)
for complex flavors:
if *trans* = 'N': $U*X = B$ or $L*X = B$ (no transpose)
if *trans* = 'C': $U^H*X = B$ or $L^H*X = B$ (conjugate transpose).

n INTEGER. The order of the tridiagonal matrix *A*. $n \geq 0$.

nrhs INTEGER. The number of right hand sides, that is, the number of columns of the matrix *B*. $nrhs \geq 0$.

d REAL array, DIMENSION (*n*). The *n* diagonal elements of the diagonal matrix *D* from the factorization computed by ?pttrf.

e COMPLEX array, DIMENSION(*n*-1). The (*n*-1) off-diagonal elements of the unit bidiagonal factor *U* or *L* from the factorization computed by ?pttrf. See *uplo*.

b COMPLEX array, DIMENSION (*ldb*, *nrhs*).
On entry, the right hand side matrix *B*.

ldb INTEGER.
The leading dimension of the array *b*.
$ldb \geq \max(1, n)$.

Output Parameters

b On exit, the solution matrix *X*.

info INTEGER.
= 0: successful exit
< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

?steqr2

Computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method.

Syntax

```
call ssteqr2(compz, n, d, e, z, ldz, nr, work, info)
call dsteqr2(compz, n, d, e, z, ldz, nr, work, info)
```

Description

This routine is a modified version of LAPACK routine ?steqr. The routine ?steqr2 computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method. ?steqr2 is modified from ?steqr to allow each ScaLAPACK process running ?steqr2 to perform updates on a distributed matrix Q. Proper usage of ?steqr2 can be gleaned from examination of ScaLAPACK routine p?syev.

Input Parameters

compz	CHARACTER*1. Must be 'N' or 'I'. If *compz* = 'N', the routine computes eigenvalues only. If *compz* = 'I', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix *T*. *z* must be initialized to the identity matrix by p?laset or ?laset prior to entering this subroutine.
n	INTEGER. The order of the matrix *T* (*n* ≥ 0).
d, *e*, *work*	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: *d* contains the diagonal elements of *T*. The dimension of *d* must be at least max(1, *n*). *e* contains the (*n*-1) subdiagonal elements of *T*. The dimension of *e* must be at least max(1, *n*-1). *work* is a workspace array. The dimension of *work* is max(1, 2**n*-2). If *compz* = 'N', then *work* is not referenced.
z	(local) REAL for ssteqr2 DOUBLE PRECISION for dsteqr2

Array, global DIMENSION (*n*, *n*), local DIMENSION (*ldz*, *nr*). If *compz* = 'V', then *z* contains the orthogonal matrix used in the reduction to tridiagonal form.

ldz — INTEGER. The leading dimension of the array *z*. Constraints:

ldz ≥ 1,

ldz ≥ max(1, *n*), if eigenvectors are desired.

nr — INTEGER. *nr* = max(1, *numroc*(*n*, *nb*, *myprow*, 0, *nprocs*)).
If *compz* = 'N', then *nr* is not referenced.

Output Parameters

d — REAL array, DIMENSION (*n*), for ssteqr2.
DOUBLE PRECISION array, DIMENSION (*n*), for dsteqr2.
On exit, the eigenvalues in ascending order, if *info* = 0. See also *info*.

e — REAL array, DIMENSION (*n*-1), for ssteqr2.
DOUBLE PRECISION array, DIMENSION (*n*-1), for dsteqr2.
On exit, *e* has been destroyed.

z — (local)
REAL for ssteqr2
DOUBLE PRECISION for dsteqr2
Array, global DIMENSION (*n*, *n*), local DIMENSION (*ldz*, *nr*).
On exit, if *info* = 0, then,
if *compz* = 'V', *z* contains the orthonormal eigenvectors of the original symmetric matrix, and if *compz* = 'I', *z* contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If *compz* = 'N', then *z* is not referenced.

info — INTEGER.
info = 0, the exit is successful.
info < 0: if *info* = -*i*, the *i*-th had an illegal value.
info > 0: the algorithm has failed to find all the eigenvalues in a total of 30*n* iterations;
if *info* = *i*, then *i* elements of *e* have not converged to zero; on exit, *d* and *e* contain the elements of a symmetric tridiagonal matrix, which is orthogonally similar to the original matrix.

Utility Functions and Routines

This section describes ScaLAPACK utility functions and routines. Summary information about these routines is given in the following table:

Table 7-2 ScaLAPACK Utility Functions and Routines

Routine Name	Data Types	Description
p?labad	s,d	Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.
p?lachkieee	s,d	Performs a simple check for the features of the IEEE standard. (C interface function).
p?lamch	s,d	Determines machine parameters for floating-point arithmetic.
p?lasnbt	s,d	Computes the position of the sign bit of a floating-point number. (C interface function).
pxerbla		Error handling routine called by ScaLAPACK routines.

p?labad

Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.

Syntax

```
call pslabad(ictxt, small, large)
call pdlabad(ictxt, small, large)
```

Description

This routine takes as input the values computed by p?lamch for underflow and overflow, and returns the square root of each of these values if the log of *large* is sufficiently large. This subroutine is intended to identify machines with a large exponent range, such as the Crays, and redefine the underflow and overflow limits to be the square roots of the values computed by p?lamch. This subroutine is needed because p?lamch does not compensate for poor arithmetic in the upper half of the exponent range, as is found on a Cray.

In addition, this routine performs a global minimization and maximization on these values, to support heterogeneous computing networks.

Input Parameters

ictxt (global) INTEGER.
The BLACS context handle in which the computation takes place.

small (local).
REAL PRECISION for pslabad.
DOUBLE PRECISION for pdlabad.
On entry, the underflow threshold as computed by p?lamch.

large (local).
REAL PRECISION for pslabad.
DOUBLE PRECISION for pdlabad.
On entry, the overflow threshold as computed by p?lamch.

Output Parameters

small (local).
On exit, if *log10(large)* is sufficiently large, the square root of *small*, otherwise unchanged.

large (local).
On exit, if *log10(large)* is sufficiently large, the square root of *large*, otherwise unchanged.

p?lachkieee

Performs a simple check for the features of the IEEE standard. (C interface function).

Syntax

```
void pslachkieee(int *isieee, float *rmax, float *rmin);
void pdlachkieee(int *isieee, float *rmax, float *rmin);
```

Description

This routine performs a simple check to make sure that the features of the IEEE standard are implemented. In some implementations, p?lachkieee may not return.

Note that all arguments are call-by-reference so that this routine can be directly called from Fortran code.

This is a ScaLAPACK internal subroutine and arguments are not checked for unreasonable values.

Input Parameters

rmax	(local).
	`REAL` for `pslachkieee`
	`DOUBLE PRECISION` for `pdlachkieee`
	The overflow threshold `(= ?lamch ('O'))`.

rmin	(local).
	`REAL` for `pslachkieee`
	`DOUBLE PRECISION` for `pdlachkieee`
	The underflow threshold `(= ?lamch ('U'))`.

Output Parameters

isieee	(local). `INTEGER`. On exit, *isieee* `= 1` implies that all the features of the IEEE standard that we rely on are implemented. On exit, *isieee* `= 0` implies that some the features of the IEEE standard that we rely on are missing.

p?lamch

Determines machine parameters for floating-point arithmetic.

Syntax

val = pslamch(*ictxt*, *cmach*)

val = pdlamch(*ictxt*, *cmach*)

Description

This function determines single precision machine parameters.

Input Parameters

ictxt (global). INTEGER.The BLACS context handle in which the computation takes place.

cmach (global) CHARACTER*1.
Specifies the value to be returned by p?lamch:
= 'E' or 'e', p?lamch := eps
= 'S' or 's' , p?lamch := sfmin
= 'B' or 'b', p?lamch := base
= 'P' or 'p', p?lamch := eps*base
= 'N' or 'n', p?lamch := t
= 'R' or 'r', p?lamch := rnd
= 'M' or 'm', p?lamch := emin
= 'U' or 'u', p?lamch := rmin
= 'L' or 'l', p?lamch := emax
= 'O' or 'o', p?lamch := rmax,
where
eps = relative machine precision
sfmin = safe minimum, such that 1/sfmin does not overflow
base = base of the machine
prec = eps*base
t = number of (base) digits in the mantissa
rnd = 1.0 when rounding occurs in addition, 0.0 otherwise
emin = minimum exponent before (gradual) underflow
rmin = underflow threshold - $base^{(emin-1)}$
emax = largest exponent before overflow
rmax = overflow threshold - $(base^{emax})*(1-eps)$

Output Parameters

val Value returned by the fuction.

p?lasnbt

Computes the position of the sign bit of a floating-point number. (C interface function).

Syntax

```
void pslasnbt(int *ieflag);
void pdlasnbt(int *ieflag);
```

Description

This routine finds the position of the signbit of a single/double precision floating point number. This routine assumes IEEE arithmetic, and hence, tests only the 32-nd bit (for single precision) or 32-nd and 64-th bits (for double precision) as a possibility for the signbit. `sizeof(int)` is assumed equal to 4 bytes.

If a compile time flag (`NO_IEEE`) indicates that the machine does not have IEEE arithmetic, *ieflag* = `0` is returned.

Output Parameters

ieflag	`INTEGER`. This flag indicates the position of the signbit of any single/double precision floating point number. *ieflag* = `0`, if the compile time flag `NO_IEEE` indicates that the machine does not have IEEE arithmetic, or if `sizeof(int)` is different from 4 bytes. *ieflag* = `1` indicates that the signbit is the 32-nd bit for a single precision routine. In the case of a double precision routine: *ieflag* = `1` indicates that the signbit is the 32-nd bit (Big Endian). *ieflag* = `2` indicates that the signbit is the 64-th bit (Little Endian).

pxerbla

Error handling routine called by ScaLAPACK routines.

Syntax

```
call pxerbla(ictxt, srname, info)
```

Description

This routine is an error handler for the ScaLAPACK routines. It is called by a ScaLAPACK routine if an input parameter has an invalid value. A message is printed. Program execution is not terminated. For the ScaLAPACK driver and computational routines, a `RETURN` statement is issued following the call to `pxerbla`.

Control returns to the higher-level calling routine, and it is left to the user to determine how the program should proceed. However, in the specialized low-level ScaLAPACK routines (auxiliary routines that are Level 2 equivalents of computational routines), the call to `pxerbla()` is immediately followed by a call to `BLACS_ABORT()` to terminate program execution since recovery from an error at this level in the computation is not possible.

It is always good practice to check for a nonzero value of *info* on return from a ScaLAPACK routine. Installers may consider modifying this routine in order to call system-specific exception-handling facilities.

Input Parameters

ictxt (global) `INTEGER`
The BLACS context handle, indicating the global context of the operation. The context itself is global.

srname (global) `CHARACTER*6`
The name of the routine which called `pxerbla`.

info (global) `INTEGER`.
The position of the invalid parameter in the parameter list of the calling routine.

8 Sparse Solver Routines

Intel® Math Kernel Library (Intel® MKL) provides user-callable sparse solver software to solve real or complex, symmetric, structurally symmetric or non-symmetric, positive definite, indefinite or Hermitian sparse linear system of equations.

The terms and concepts required to understand the use of the Intel MKL direct sparse solver subroutines are discussed in the Linear Solvers Basics appendix. If you are familiar with linear sparse solvers and sparse matrix storage schemes, you can omit reading these sections and go directly to the interface descriptions. The direct sparse solver PARDISO* is described in the section that follows. After that, two alternative interfaces (direct sparse solver and iterative sparse solver) that consist of several Intel MKL routines implementing the step-by-step solution process are described.

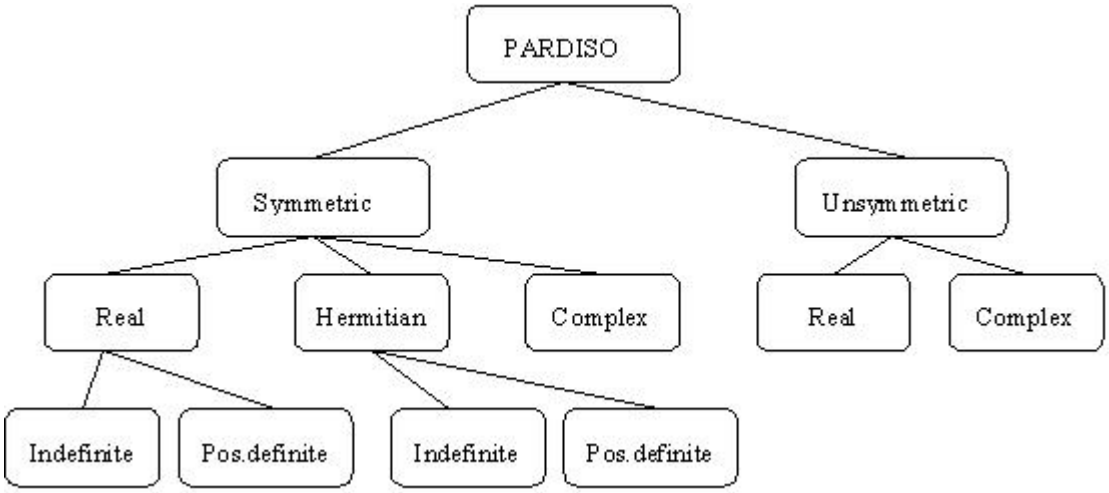
PARDISO - Parallel Direct Sparse Solver Interface

This section describes the interface to the shared-memory multiprocessing parallel direct sparse solver known as PARDISO. The interface is Fortran, but can be called from C programs by observing Fortran parameter passing and naming conventions used by the supported compilers and operating systems. A discussion of the algorithms used in PARDISO and more information on the solver can be found at http://www.computational.unibas.ch/cs/scicomp.

The PARDISO package is a high-performance, robust, memory efficient and easy to use software for solving large sparse symmetric and unsymmetric linear systems of equations on shared memory multiprocessors. The solver uses a combination of left- and right-looking Level-3 BLAS supernode techniques [Schenk00-2]. In order to improve sequential and parallel sparse numerical factorization performance, the algorithms are based on a Level-3 BLAS update and pipelining parallelism is exploited with a combination of left- and right-looking supernode techniques [Schenk00, Schenk01, Schenk02, Schenk03]. The parallel pivoting methods allow complete supernode pivoting in order to compromise numerical stability and scalability during the factorization process. For sufficiently large problem sizes, numerical experiments demonstrate that the scalability of the parallel algorithm is nearly independent of the shared-memory multiprocessing architecture and a speedup of up to seven using eight processors has been observed.

PARDISO supports, as illustrated in Figure 8-1, a wide range of sparse matrix types and computes the solution of real or complex, symmetric, structurally symmetric or unsymmetric, positive definite, indefinite or Hermitian sparse linear system of equations on shared-memory multiprocessing architectures.

Figure 8-1 Sparse Matrices That Can be Solved With PARDISO



You can find example code that uses PARDISO interface routine to solve systems of linear equations in PARDISO Code Examples section in the Appendix C .

pardiso

Calculates the solution of a set of sparse linear equations with multiple right-hand sides.

Syntax

Fortran:

```
call pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja, perm, nrhs, iparm,
msglvl, b, x, error)
```

C:

```
pardiso (pt, &maxfct, &mnum, &mtype, &phase, &n, a, ia, ja, perm, &nrhs,
iparm, &msglvl, b, x, &error);
```

(An underscore may or may not be required after "pardiso" depending on the OS and compiler conventions for that OS).

Interface:

```
SUBROUTINE pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja, perm, nrhs,
iparm, msglvl, b, x, error)

INTEGER *4 pt (64)

INTEGER *4 maxfct, mnum, mtype, phase, n, nrhs, error, ia(*), ja(*), perm(*),
iparm(*)

REAL *8 a(*), b(n,nrhs), x(n,nrhs)
```

Note that the above interface is given for the 32-bit architectures. For 64-bit architectures, the argument *pt(64)* must be defined as INTEGER*8 type.

Description

PARDISO calculates the solution of a set of sparse linear equations

$AX = B$

with multiple right-hand sides, using a parallel LU, LDL or LL^T factorization, where A is an n-by-n matrix, and X and B are n-by-$nrhs$ matrices. PARDISO performs the following analysis steps depending on the structure of the input matrix A.

Symmetric Matrices: The solver first computes a symmetric fill-in reducing permutation P based on either the minimum degree algorithm [Liu85] or the nested dissection algorithm from the METIS package [Karypis98] (included with Intel MKL), followed by the parallel left-right looking numerical Cholesky factorization [Schenk00-2] of $PAP^T = LL^T$ for symmetric positive-definite matrices, or $PAP^T = LDL^T$ for symmetric indefinite matrices. The solver uses diagonal pivoting, or 1x1 and 2x2 Bunch and Kaufman pivoting for symmetric indefinite matrices, and an approximation of X is found by forward and backward substitution and iterative refinements.

The coefficient matrix is perturbed whenever numerically acceptable 1x1 and 2x2 pivots cannot be found within the diagonal supernode block. One or two passes of iterative refinements may be required to correct the effect of the perturbations. This restricting notion of pivoting with iterative refinements is effective for highly indefinite symmetric systems. Furthermore the accuracy of this method is for a large set of matrices from different applications areas as accurate as a direct factorization method that uses complete sparse pivoting techniques [Schenk04]. Another possiblity to improve the pivoting accuracy is to use symmetric weighted matchings algorithms. These methods identify large entries in the coefficient matrix A that, if permuted close to the diagonal, permit the factorization process to identify more acceptable

pivots and proceed with fewer pivot perturbations. The methods are based on maximum weighted matchings and improve the quality of the factor in a complementary way to the alternative idea of using more complete pivoting techniques.

The inertia is also computed for real symmetric indefinite matrices.

Structurally Symmetric Matrices: The solver first computes a symmetric fill-in reducing permutation P followed by the parallel numerical factorization of $PAP^T = QLU^T$. The solver uses partial pivoting in the supernodes and an approximation of X is found by forward and backward substitution and iterative refinements.

Unsymmetric Matrices: The solver first computes a non-symmetric permutation P_{MPS} and scaling matrices D_r and D_c with the aim to place large entries on the diagonal which enhances greatly the reliability of the numerical factorization process [Duff99]. In the next step the solver computes a fill-in reducing permutation P based on the matrix $P_{MPS}A + (P_{MPS}A)_T$ followed by the parallel numerical factorization

$QLUR = PP_{MPS}D_rAD_cP$

with supernode pivoting matrices Q and R. When the factorization algorithm reaches a point where it cannot factorize the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy similar to [Li99]. The magnitude of the potential pivot is tested against a constant threshold of $\texttt{alpha} = \texttt{eps}*||A2||_{\texttt{inf}}$, where eps is the machine precision, $A2 = P*P_{MPS}*D_r*A*D_c*P$, and $||A2||_{\texttt{inf}}$ is the infinity norm of the scaled and permuted matrix A. Therefore any tiny pivots encountered during elimination are set to the sign $(l_{ii})*\texttt{eps}*||A2||_{\texttt{inf}}$ - this trades off some numerical stability for the ability to keep pivots from getting too small. Although many failures could render the factorization well-defined but essentially useless, in practice it is observed that the diagonal elements are rarely modified for a large class of matrices. The result of this pivoting approach is that the factorization is, in general, not exact and iterative refinement may be needed.

Direct-Iterative Preconditioning for Unsymmetric Linear Systems. The solver also allows a combination of direct and iterative methods [Sonn89] in order to accelerate the linear solution process for transient simulation. A majority of applications of sparse solvers require solutions of systems with gradually changing values of the nonzero coefficient matrix, but the same identical sparsity pattern. In these applications, the analysis phase of the solvers has to be performed only once and the numerical factorizations are the important time-consuming steps during the simulation. PARDISO uses a numerical factorization $A = LU$ for the first system and applies these exact factors L and U for the next steps in a preconditioned Krylow-Subspace iteration. If the iteration does not converge, the solver will automatically switch back to the numerical factorization. This method can be applied for unsymmetric matrices in PARDISO and the user can select the method using only one input parameter. For further details see the parameter description ($iparm(4)$, $iparm(20)$).

The sparse data storage in PARDISO follows the scheme described in Sparse Matrix Storage Format section with *ja* standing for *columns*, *ia* for *rowIndex*, and *a* for *values*. The algorithms in PARDISO require column indices ja to be increasingly ordered per row and the presence of the diagonal element per row for any symmetric or structurally symmetric matrix. The unsymmetric matrices need no diagonal elements in the PARDISO solver.

There are four tasks that PARDISO is capable of performing, namely analysis and symbolic factorization, numerical factorization, forward and backward substitution including iterative refinement and finally the termination to release all internal solver memory. When an input data structure is not accessed in a call, a NULL pointer or any valid address can be passed as a place holder for that argument.

Input Parameters

pt

INTEGER*4 for 32 -bit operating systems;
INTEGER*8 for 64 -bit operating systems
Array, DIMENSION (64)
On entry, this is the solver internal data address pointer. Theses addresses are passed to the solver and all related internal memory management is organized through this pointer

NOTE. *pt* is an integer array with 64 entries. It is very important that the pointer is initialized with zero at the first call of PARDISO. After that first call you should never modify the pointer, as a serious memory leak can occur. The integer length should be 4-byte on 32-bit operating systems and 8-byte on 64-bit operating systems.

maxfct

INTEGER
Maximal number of factors with identical nonzero sparsity structure that the user would like to keep at the same time in memory. It is possible to store several different factorizations with the same nonzero structure at the same time in the internal data management of the solver. In most of the applications this value is equal to 1.

PARDISO can process several matrices with identical matrix sparsity pattern and is able to store the factors of these matrices at the same time. Matrices with different sparsity structure can be kept in memory with different memory address pointers *pt*.

mnum INTEGER

Actual matrix for the solution phase. With this scalar you can define the matrix that you would like to factorize. The value must be: $1 \le mnum \le maxfct$.

In most of the applications this value is equal to 1.

mtype INTEGER

This scalar value defines the matrix type. The PARDISO solver supports the following matrices:

= 1	real and structurally symmetric matrix
= 2	real and symmetric positive definite matrix
= -2	real and symmetric indefinite matrix
= 3	complex and structurally symmetric matrix
= *4*	complex and Hermitian positive definite matrix
= *-4*	complex and Hermitian indefinite matrix
= *6*	complex and symmetric matrix
= *11*	real and unsymmetric matrix
= *13*	complex and unsymmetric matrix

This parameter influences the pivoting method.

phase INTEGER

Controls the execution of the solver. It is a two-digit integer *ij* ($10i + j$, $1 \le i \le 3$, $i < j \le 3$ for normal execution modes). The *i* digit indicates the starting phase of execution, and *j* indicates the ending phase. PARDISO has the following phases of execution:

- Phase 1: Fill-reduction analysis and symbolic factorization
- Phase 2: Numerical factorization
- Phase 3: Forward and Backward solve including iterative refinements
- Termination and Memory Release Phase (*phase*≤ 0)

If a previous call to the routine has computed information from previous phases, execution may start at any phase. The *phase* parameter can have the following values:

phase	**Solver Execution Steps**
11	Analysis
12	Analysis, numerical factorization
13	Analysis, numerical factorization, solve, iterative refinement
22	Numerical factorization
23	Numerical factorization, solve, iterative refinement
33	Solve, iterative refinement
0	Release internal memory for *L* and *U* matrix number *mnum*
-1	Release all internal memory for all matrices

n INTEGER

Number of equations. This is the number of equations in the sparse linear systems of equations *A*X = B*. Constraint: *n* > 0.

a REAL/COMPLEX

Array. Contains the nonzero values of the coefficient matrix *A* corresponding to the indices in *ja*. The size of *a* is the same as that of *ja* and the coefficient matrix can be either real or complex. The matrix must be stored in compressed sparse row format with increasing values of *ja* for each row. Refer to *values* array description in Sparse Matrix Storage Format for more details.

NOTE. The nonzeros of each row of the matrix *A* must be stored in increasing order. For symmetric or structural symmetric matrices it is also important that the diagonal elements are also available and stored in the matrix. If the matrix is symmetric, then the array *a* is only accessed in the factorization phase, in the triangular solution and iterative refinement phase. Unsymmetric matrices are accessed in all phases of the solution process.

ia INTEGER

Array, dimension (*n*+1). For *i*≤*n*, *ia*(*i*) points to the first column index of row *i* in the array *ja* in compressed sparse row format. That is, *ia*(*i*) gives the index of the element in array *a* that contains the first non-zero element from row *i* of *A*. The last element *ia*(*n*+1) is taken to be equal to the number of non-zeros in *A*, plus one. Refer to *rowIndex* array description in Sparse Matrix Storage Format for more details.The array *ia* is also accessed in all phases of the solution process. Note that the row and columns numbers start from 1.

ja INTEGER

Array. *ja*(*) contains column indices of the sparse matrix *A* stored in compressed sparse row format. The indices in each row must be sorted in increasing order.The array *ja* is also accessed in all phases of the solution process. For symmetric and structurally symmetric matrices it is assumed that zero diagonal elements are also stored in the list of nonzeros in *a* and *ja*. For symmetric matrices, the solver needs only the upper triangular part of the system as is shown for *columns* array in Sparse Matrix Storage Format.

perm INTEGER

Array, dimension (*n*). Holds the permutation vector of size *n*. The array *perm* is defined as follows. Let *A* be the original matrix and $B = P*A*P^T$ be the permuted matrix. Row

(column) *i* of *A* is the *perm*(i) row (column) of *B*. The numbering of the array must start by 1 and it must describe a permutation.
On entry, you can apply your own fill-in reducing ordering to the solver. The permutation vector *perm* is only accessed if *iparm*(5) = 1.

nrhs INTEGER
Number of right-hand sides that need to be solved for.

iparm INTEGER
Array, dimension (64). This array is used to pass various parameters to PARDISO and to return some useful information after the execution of the solver. If *iparm*(1) = 0, then PARDISO fills *iparm*(1), and *iparm*(4) through *iparm*(64) with default values and uses them. Note that there is no default values for *iparm*(3) and this value must always be supplied by the user, whether *iparm*(1) is 0 or 1.
Individual components of the *iparm* array are described below (some of them are described in the Output Parameters section).
iparm(1) - use default values.
If *iparm*(1) = 0, then *iparm*(2) and *iparm*(4) through *iparm*(64) are filled with default values, otherwise the user has to supply all values in *iparm* from *iparm*(2) to *iparm*(64).
iparm(2) - fill-in reducing ordering.
iparm(2) controls the fill-in reducing ordering for the input matrix. If *iparm*(2) is 0, then the minimum degree algorithm is applied [Li99], if *iparm*(2) is 2, the solver uses the nested dissection algorithm from the METIS package [Karypis98]. The default value of *iparm*(2) is 2.
iparm(3) - number of processors.
iparm(3) must contain the number of processors that are available for the parallel execution. The number must be equal to the OpenMP environment variable OMP_NUM_THREADS.

CAUTION. If the user has not explicitly set OMP_NUM_THREADS, then this value can be set by the operating system to the maximal numbers of processors on the system. It is therefore always recommended to control the parallel execution of the solver by explicitly setting OMP_NUM_THREADS. If less processors are available than specified, the execution may slow down instead of speeding up.

There is no default value for *iparm*(3).
iparm(4) - preconditioned CGS.
This parameter controls preconditioned CGS [Sonn89] for unsymmetric or structural symmetric matrices and Conjugate-Gradients for symmetric matrices. *iparm*(4) has the form

iparm(4)= 10*L+K

The *K* and *L* values have the meanings as follow.

Value of *K*	Description
0	The factorization is always computed as required by *phase*.
1	CGS iteration replaces the computation of *LU*. The preconditioner is *LU* that was computed at a previous step (the first step or last step with a failure) in a sequence of solutions needed for identical sparsity patterns.
2	CG iteration for symmetric matrices replaces the computation of *LU*. The preconditioner is *LU* that was computed at a previous step (the first step or last step with a failure) in a sequence of solutions needed for identical sparsity patterns.

Value *L*:
The value *L* controls the stopping criterion of the Krylow-Subspace iteration:

$eps_{CGS} = 10^{-L}$ is used in the stopping criterion

$||dx_i|| / ||dx_1|| < eps_{CGS}$

with $||dx_i|| = ||inv(L*U)*r_i||$ and r_i is the residuum at iteration i of the preconditioned Krylow-Subspace iteration.

Strategy: A maximum number of 150 iterations is fixed by expecting that the iteration will converge before consuming half the factorization time. Intermediate convergence rates and residuum excursions are checked and can terminate the iteration process. If *phase* =23, then the factorization for a given *A* is automatically recomputed in these cases where the Krylow-Subspace iteration failed, and the corresponding direct solution is returned. Otherwise the solution from the preconditioned Krylow-Subspace iteration is returned. Using *phase* =33 results in an error message (*error* =4) if the stopping criteria for the Krylow-Subspace iteration can not be reached. More information on the failure can be obtained from *iparm*(20).

The default is *iparm*(4)=0, and other values are only recommended for an advanced user. *iparm*(4) must be greater or equal to zero.

Examples:

iparm(4)	Description
31	*LU*-preconditioned CGS iteration with a stopping criterion of 1.0E-3 for unsymmetric matrices
61	*LU*-preconditioned CGS iteration with a stopping criterion of 1.0E-6 for unsymmetric matrices
62	*LU*-preconditioned CGS iteration with a stopping criterion of 1.0E-6 for symmetric matrices

iparm(5) - user permutation.

This parameter controls whether the user supplied fill-in reducing permutation is used instead of the integrated multiple-minimum degree or nested dissection algorithms.

This option may be useful for testing reordering algorithms or adapting the code to special applications problems (for instance, to move zero diagonal elements to the end $P*A*P^T$). For definition of the permutation, see description of the *perm* parameter.
The default value of *iparm*(5) is 0.
iparm(6) - write solution on x.
If *iparm*(6) is 0 (which is the default), then the array *x* contains the solution and the value of *b* is not changed. If *iparm*(6) is 1, then the solver will store the solution on the right hand side *b*.
Note that the array *x* is always used. The default value of *iparm*(6) is 0.
iparm(8)
On entry to the solve and iterative refinement step, *iparm*(8) should be set to the maximum number of iterative refinement steps that the solver will perform. The solver will not perform more than the absolute value of *iparm*(8) steps of iterative refinement and will stop the process if a satisfactory level of accuracy of the solution in terms of backward error has been achieved.
Note that if *iparm*(8) < 0, the accumulation of the residuum is using enhanced precision real and complex data types.
Perturbed pivots result in iterative refinement (independent of *iparm*(8)=0) and the iteration number executed is reported on *iparm*(7).
The solver will automatically perform two steps of iterative refinements when perturbed pivots have been obtained during the numerical factorization and *iparm*(8) was equal to zero.
The number of performed iterative refinement steps is reported on *iparm*(8).
The default value for *iparm*(8) is 0.
iparm(9)
This parameter is reserved for future use. Its value must be set to 0.
iparm(10) - pivoting perturbation.

This parameter instructs PARDISO how to handle small pivots or zero pivots for unsymmetric matrices (*mtype* =11 or *mtype* =13) and symmetric matrices (*mtype* =-2, *mtype* =-4, or *mtype* =6). For these matrices the solver uses a complete supernode pivoting approach. When the factorization algorithm reaches a point where it cannot factorize the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy similar to [Li99], [Schenk04].

The magnitude of the potential pivot is tested against a constant threshold of

$$alpha = eps * ||A2||_{inf},$$

where $eps = 10^{(-iparm(10))}$, $A2 = P * P_{MPS} * D_r * A * D_c * P$, and $||A2||_{inf}$ is the infinity norm of the scaled and permuted matrix *A*. Any tiny pivots encountered during elimination are set to the sign $(l_{ii}) * eps * ||A2||_{inf}$ - this trades off some numerical stability for the ability to keep pivots from getting too small. Small pivots are therefore perturbed with $eps = 10^{(-iparm(10))}$.

The default value of *iparm*(10) is 13 and therefore eps = 1.0E-13 for unsymmetric matrices (*mtype* =11 or *mtype* =13).

The default value of *iparm*(10) is 8, and therefore eps = 1.0E-8 for symmetric indefinite matrices (*mtype* =-2, *mtype* =-4, or *mtype* =6).

iparm(11)- scaling vectors.

PARDISO uses a maximum weight matching algorithm to permute large elements on the diagonal and to scale the matrix so that the diagonal elements are equal to 1 and the absolute value of the off-diagonal entries are less or equal to 1. This scaling method is only applied to unsymmetric matrices (*mtype* =11 or *mtype* =13). The scaling can also be used for symmetric indefinite matrices (*mtype* =-2, *mtype* =-4, or *mtype* =6) in case that symmetric weighted matchings is applied (*iparm*(13)= 1).

It is recommended to use *iparm*(11) = 1 (scaling) and *iparm*(13) = 1 (matchings) for highly indefinite symmetric matrices, for example from interior point optimizations or

saddle point problems. It is also very important to note that the user must provided in the analysis phase (*phase*=11) the numerical values of the matrix *A* in case of scalings and symmetric weighted matchings.
The default value of *iparm*(11) is 1 for unsymmetric matrices (*mtype* =11 or *mtype* =13). The default value of *iparm*(11) is 0 for symmetric indefinite matrices (*mtype* =-2, *mtype* =-4, or *mtype* =6).
iparm(12)
This parameter is reserved for future use. Its value must be set to 0.
iparm(13) - improved accuracy using (non-)symmetric weighted matchings.
PARDISO can use a maximum weighted matching algorithm to permute large elements close the diagonal. This strategy adds an additional level of reliability to our factorization methods and can be seen as a complement to the alternative idea of using more complete pivoting techniques during the numerical factorization.
It is recommended to use *iparm*(11)=1 (scalings) and *iparm*(13)=1 (matchings) for highly indefinite symmetric matrices, for example from interior point optimizations or saddle point problems. It is also very important to note that the user must provided in the analysis phase (*phase* =11) the numerical values of the matrix *A* in case of scalings and symmetric weighted matchings.
The default value of *iparm*(13) is 1 for unsymmetric matrices (*mtype* =11 or *mtype* =13). The default value of *iparm*(13) is 0 for symmetric matrices (*mtype* =-2, *mtype* =-4, or *mtype* =6).
iparm(18)
The solver will report the numbers of nonzeros on the factors if *iparm*(18)< 0 on entry.
The default value of *iparm*(18) is -1.
iparm(19)- MFlops of factorization.
If *iparm*(19)< 0 on entry, the solver will report MFlop (1.0E6) that are necessary to factor the matrix *A*. This will increase the reordering time.
The default value of *iparm*(19) is 0.

iparm(21) - pivoting for symmetric indefinite matrices. *iparm*(21) controls the pivoting method for sparse symmetric indefinite matrices. If *iparm*(21) is 0, then 1x1 diagonal pivoting is used. If *iparm*(21) is 1, then 1x1 and 2x2 Bunch and Kaufman pivoting will be used within the factorization process. It is also recommended to use *iparm*(11)=1 (scalings) and *iparm*(13)=1 (matchings) for highly indefinite symmetric matrices, for example from interior point optimizations or saddle point problems. Bunch and Kaufman pivoting is available for matrices: *mtype* =-2, *mtype* =-4, or *mtype* =6.
The default value of *iparm*(21) is 0.

msglvl — INTEGER
Message level information. If *msglvl* = 0 then PARDISO generates no output, if *msglvl* = 1 the solver prints statistical information to the screen.

b — REAL/COMPLEX
Array, dimension (*n*,*nrhs*). On entry, contains the right hand side vector/matrix *B*. Note that *b* is only accessed in the solution phase.

Output Parameters

pt — This parameter contains internal address pointers.

iparm — On output, some *iparm* values will report useful information, for example, numbers of nonzeros in the factors, and so on.
iparm(7) - number of performed iterative refinement steps. The number of iterative refinement steps that are actually performed during the solve step.
iparm(14) - number of perturbed pivots.
After factorization, *iparm*(14) contains the number of perturbed pivots during the elimination process for *mtype* =11, *mtype* =13, *mtype* =-2, *mtype* =-4, or *mtype* =-6.
iparm(15) - peak memory symbolic factorization.
The parameter *iparm*(15) provides the user with the total peak memory in KBytes that the solver needed during the analysis and symbolic factorization phase. This value is only computed in phase 1.

iparm(16) - permanent memory symbolic factorization.
The parameter *iparm*(16) provides the user with the permanent memory in KBytes that the solver needed from the analysis and symbolic factorization phase in the factorization and solve phases. This value is only computed in phase 1.
iparm(17) - memory numerical factorization and solution.
The parameter *iparm*(17) provides the user with the total double precision memory consumption (KBytes) of the solver for the factorization and solve phases. This value is only computed in phase 2.
Note that the total peak memory solver consumption is max(*iparm*(15), *iparm*(16)+*iparm*(17))
iparm(18) - number of nonzeros in factors.
The solver will report the numbers of nonzeros on the factors if *iparm*(18) < 0 on entry.
iparm(19) - MFlops of factorization.
Number of operations in MFlop (1.0E6 operations) that are necessary to factor the matrix *A* are returned to the user if *iparm*(19) < 0 on entry.
iparm(20) - CG/CGS diagnostics.
The value is used to give CG/CGS diagnostics (for example, the number of iterations and cause of failure):
If *iparm*(20)> 0, CGS succeeded, and the number of iterations executed are reported in *iparm*(20).
If *iparm*(20)< 0, iterations executed, but CG/CGS failed. The error report details in *iparm*(20) are of the form: *iparm*(20)= - it_cgs*10 - cgs_error.
If *phase* is 23, then the factors *L*, *U* are recomputed for the matrix *A* and the error flag *error* is zero in case of a successful factorization. If *phase* is 33, then *error* = -4 signals the failure.
Description of cgs_error is given in the table below:

cgs_error	Description
1	- fluctuations of the residuum are too large
2	- $\|\|dx_{max\_it\_cgs/2}\|\|$ too large (slow convergence)

cgs_error	Description
3	- stopping criterion not reached at max_it_cgs
4	- perturbed pivots caused iterative refinement
5	- factorization is too fast for this matrix. It is better to use the factorization method with *iparm*(4)=0

iparm(22)- inertia: number of positive eigenvalues.
The parameter *iparm*(22) reports the number of positive eigenvalues for symmetric indefinite matrices.
iparm(23)- inertia: number of negative eigenvalues.
The parameter *iparm*(23) reports the number of negative eigenvalues for symmetric indefinite matrices.
iparm(24) to *iparm*(64)
These parameters are reserved for future use. Their values must be set to 0.

b On output, the array is replaced with the solution if *iparm*(6) = 1.

x REAL/COMPLEX
Array, dimension (*n,nrhs*). On output, contains solution if *iparm*(6)=0. Note that *x* is only accessed in the solution phase.

error INTEGER
The error indicator according to the below table:

error	Information
0	no error
-1	input inconsistent
-2	not enough memory
-3	reordering problem
-4	zero pivot, numerical factorization or iterative refinement problem
-5	unclassified (internal) error
-6	preordering failed (matrix types 11, 13 only)

error	**Information**
-7	diagonal matrix problem

Direct Sparse Solver (DSS) Interface Routines

The Intel MKL supports an alternative to PARDISO interface for the direct sparse solver referred to here as DSS interface. The DSS interface implements a group of user-callable routines that are used in the step-by-step solving process and exploits the general scheme described in Linear Solvers Basics for solving sparse systems of linear equations. This interface also includes one routine for gathering statistics related to the solving process and an auxiliary routine for passing character strings from Fortran routines to C routines.

The solving process is conceptually divided into six phases, as shown in Table 8-1 which lists the names of the routines, grouped by phase, and describes their general use.

Table 8-1 DSS Interface Routines

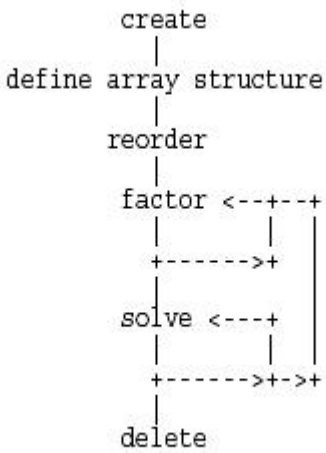
Routine	Description
dss_create	Initializes the solver and creates the basic data structures necessary for the solver. This routine must be called before any other DSS routine.
dss_define_structure	Used to inform the solver of the locations of the non-zero elements of the array.
dss_reorder	Based on the non-zero structure of the matrix, this routine computes a permutation vector to reduce fill-in during the factoring process.
dss_factor_real, dss_factor_complex	Computes the LU, LDT^t or LL^T factorization of a real or complex matrix.
dss_solve_real, dss_solve_complex	Computes the solution vector for a system of equations based on the factorization computed by the previous phase.
dss_delete	Deletes all of the data structures created during the solutions process.
dss_statistics	Returns statistics about various phases of the solving process. Used to gather statistics in the following areas: time taken to do reordering, time taken to

Routine	Description
	do factorization, problem solving duration, determinant of a matrix, inertia of a matrix, and number of floating point operations taken during factorization. Can be invoked at any phase of the solving process after the "reorder" phase, but before the "delete" phase. Note that appropriate argument(s) must be supplied to this routine to correspond to phase at which it is invoked.
`mkl_cvt_to_null_terminated_str`	Used to pass character strings from Fortran routines to C routines.

To find a single solution vector for a single system of equations with a single right hand side, the Intel MKL DSS interface routines are invoked in the order in which they are listed in Table 8-1 , with the exception of `dss_statistics`, which is invoked as described in the table.

However, in certain applications it is necessary to produce solution vectors for multiple right-hand sides for a given factorization and/or factor several matrices with the same non-zero structure. Consequently, it is necessary to be able to invoke the Intel MKL sparse routines in an order other than listed in the table. The following diagram in Figure 8-2 indicates the typical order(s) in which the DSS interface routines can be invoked.

Figure 8-2 Typical order for invoking DSS interface routines



You can find example code that uses DSS interface routines to solve systems of linear equations in Direct Sparse Solver Examples section in the appendix.

DSS Interface Description

As noted in Memory Allocation and Handles section, each DSS routine either reads or writes an opaque data object called a handle. Because the declaration of a handle varies from language to language, it is declared as being of type `MKL_DSS_HANDLE` in this documentation. You can refer to Memory Allocation and Handles to determine the correct method for declaring a handle argument.

All other types in this documentation refer to the standard Fortran types, `INTEGER`, `REAL`, `COMPLEX`, `DOUBLE PRECISION`, and `DOUBLE COMPLEX`.

C and C++ programmers should refer to Calling Direct Sparse Solver Routines From C/C++ for information on mapping Fortran types to C/C++ types.

Routine Options

All of the DSS routines have an integer argument (below referred to as *opt*) for passing various options to the routines. The permissible values for *opt* should be specified using only the symbol constants defined in the language-specific header files (see Implementation Details). All of the routines accept options for setting the message and termination level as described in Table 8-2 . Additionally, all routines accept the option `MKL_DSS_DEFAULTS`, which establishes the documented default options for each DSS routine.

Table 8-2 Symbolic Names for the Message and Termination Level Options

Message Level	Termination Level
MKL_DSS_MSG_LVL_SUCCESS	MKL_DSS_TERM_LVL_SUCCESS
MKL_DSS_MSG_LVL_INFO	MKL_DSS_TERM_LVL_INFO
MKL_DSS_MSG_LVL_WARNING	MKL_DSS_TERM_LVL_WARNING
MKL_DSS_MSG_LVL_ERROR	MKL_DSS_TERM_LVL_ERROR
MKL_DSS_MSG_LVL_FATAL	MKL_DSS_TERM_LVL_FATAL

The settings for message and termination level can be set on any call to a DSS routine. However, once set to a particular level, they remain at that level until they are changed in another call to a DSS routine.

Users can specify multiple options to a DSS routine by adding the options together. For example, to set the message level to debug and the termination level to error for all DSS routines, use the call:

```
CALL dss_create( handle, MKL_DSS_MSG_LVL_INFO + MKL_DSS_TERM_LVL_ERROR)
```

User Data Arrays

Many of the DSS routines take arrays of user data as input. For example, user arrays are passed to the routine `dss_define_structure` to describe the location of the non-zero entries in the matrix. In order to minimize storage requirements and improve overall run-time efficiency, the Intel MKL DSS routines do not make copies of the user input arrays.

WARNING. Users cannot modify the contents of these arrays after they are passed to one of the solver routines.

dss_create

Initializes the solver.

Syntax

`dss_create(handle, opt)`

Input Parameters

opt	`INTEGER` Options passing argument. The default value is `MKL_DSS_MSG_LVL_WARNING + MKL_DSS_TERM_LVL_ERROR`.

Output Parameters

handle	Data object of `MKL_DSS_HANDLE` type (see Interface Description).

Description

The routine `dss_create` is called to initialize the solver. After the call to `dss_create`, all subsequent invocations of Intel MKL DSS routines should use the value of handle returned by `dss_create`.

WARNING. Do not write the value of handle directly.

Return Values

`MKL_DSS_SUCCESS`

MKL_DSS_INVALID_OPTION

MKL_DSS_OUT_OF_MEMORY

dss_define_structure

Communicates to the solver locations of non-zero elements in the matrix.

Syntax

dss_define_structure(*handle*, *opt*, *rowIndex*, *nRows*, *nCols*, *columns*, *nNonZeros*);

Input Parameters

opt	INTEGER. Option passing argument. The default option for the matrix structure is MKL_DSS_SYMMETRIC.
rowIndex	INTEGER. Array of size min(*nRows*, *nCols*)+1. Defines the location of non-zero entries in the matrix.
nRows	INTEGER. Number of rows in the matrix.
nCols	INTEGER. Number of columns in the matrix.
columns	INTEGER. Array of size *nNonZeros*. Defines the location of non-zero entries in the matrix.
nNonZeros	INTEGER. Number of non-zero elements in the matrix.

Output Parameters

handle	Data object of MKL_DSS_HANDLE type (see Interface Description).

Description

The routine dss_define_structure communicates to the solver the locations of the *nNonZeros* number of non-zero elements in a matrix of size *nRows* by *nCols*.

Note that currently Intel MKL DSS software only operates on square matrices, so *nRows* must be equal to *nCols*.

To communicate the locations of non-zeros in the matrix, do the following:

1. Define the general non-zero structure of the matrix by specifying one of the following values for the options argument *opt*:

- MKL_DSS_SYMMETRIC_STRUCTURE
- MKL_DSS_SYMMETRIC
- MKL_DSS_NON_SYMMETRIC

2. Provide the actual locations of the non-zeros by means of the arrays *rowIndex* and *columns* (see Sparse Matrix Storage Format).

Return Values

MKL_DSS_SUCCESS

MKL_DSS_STATE_ERR

MKL_DSS_INVALID_OPTION

MKL_DSS_COL_ERR

MKL_DSS_NOT_SQUARE

MKL_DSS_TOO_FEW_VALUES

MKL_DSS_TOO_MANY_VALUES

dss_reorder

Computes permutation vector that minimizes the fill-in during the factorization phase.

Syntax

dss_reorder(*handle*, *opt*, *perm*)

Input Parameters

opt	INTEGER. Option passing argument. The default option for the permutation type is MKL_DSS_AUTO_ORDER.
perm	INTEGER. Array of length *nRows*. Contains a user-defined permutation vector (accessed only if *opt* contains MKL_DSS_MY_ORDER).

Output Parameters

handle	Data object of MKL_DSS_HANDLE type (see Interface Description).

Description

If *opt* contains the options MKL_DSS_AUTO_ORDER, then dss_reorder computes a permutation vector that minimizes the fill-in during the factorization phase. For this option, the *perm* array is never accessed.

If *opt* contains the option MKL_DSS_MY_ORDER, then the array *perm* is considered to be a permutation vector supplied by the user. In this case, the array *perm* is of length *nRows*, where *nRows* is the number of rows in the matrix as defined by the previous call to dss_define_structure.

Return Values

MKL_DSS_SUCCESS

MKL_DSS_STATE_ERR

MKL_DSS_INVALID_OPTION

MKL_DSS_OUT_OF_MEMORY

dss_factor_real, dss_factor_complex

Compute the factorization of the matrix with previously specified location.

Syntax

dss_factor_real(*handle*, *opt*, *rValues*)

dss_factor_complex(*handle*, *opt*, *cValues*)

Input Parameters

handle	Data object of MKL_DSS_HANDLE type (see Interface Description).
opt	INTEGER Option passing argument. The default option for the matrix type is MKL_DSS_POSITIVE_DEFINITE.
rValues	DOUBLE PRECISION. Array of size *nNonZeros*. Contains real non-zero elements of the matrix.

cValues	DOUBLE COMPLEX. Array of size *nNonZeros*. Contains complex non-zero elements of the matrix.

Description

These routines compute the factorization of the matrix whose non-zero locations were previously specified by a call to dss_define_structure and whose non-zero values are given in the array *rValues* or *cValues*. The arrays *rValues* and *cValues* are assumed to be of length *nNonZeros* as defined in a previous call to dss_define_structure.

The *opt* argument should contain one of the following options:

- MKL_DSS_POSITIVE_DEFINITE,
- MKL_DSS_INDEFINITE,
- MKL_DSS_HERMITIAN_POSITIVE_DEFINITE,
- MKL_DSS_HERMITIAN_INDEFINITE ,

depending on whether the non-zero values in *rValues* and *cValues* describe a positive definite, indefinite, or Hermitian matrix.

Return Values

MKL_DSS_SUCCESS

MKL_DSS_STATE_ERR

MKL_DSS_INVALID_OPTION

MKL_DSS_OPTION_CONFLICT

MKL_DSS_OUT_OF_MEMORY

MKL_DSS_ZERO_PIVOT

dss_solve_real, dss_solve_complex

Compute the corresponding solutions vector and place it in the output array.

Syntax

dss_solve_real(*handle*, *opt*, *rRhsValues*, *nRhs*, *rSolValues*)

dss_solve_complex(*handle*, *opt*, *cRhsValues*, *nRhs*, *cSolValues*)

Input Parameters

handle	Data object of MKL_DSS_HANDLE type (see Interface Description).
opt	INTEGER. Option passing argument.
nRhs	INTEGER. Number of the right-hand sides in the linear equation.
rRhsValues	DOUBLE PRECISION. Array of size *nRows* by *nRhs*. Contains real right-hand side vectors.
cRhsValues	DOUBLE COMPLEX. Array of size *nRows* by *nRhs*. Contains complex right-hand side vectors.

Output Parameters

rSolValues	DOUBLE PRECISION. Array of size *nCols* by *nRhs*. Contains real solution vectors.
cSolValues	DOUBLE COMPLEX. Array of size *nCols* by *nRhs*. Contains complex solution vectors.

Description

For each right hand side column vector defined in *?RhsValues* (where *?* is one of *r* or *c*), these routines compute the corresponding solutions vector and place it in the array *?SolValues*.

The lengths of the right-hand side and solution vectors, *nCols* and *nRows* respectively, are assumed to have been defined in a previous call to dss_define_structure.

Return Values

MKL_DSS_SUCCESS

MKL_DSS_STATE_ERR

MKL_DSS_INVALID_OPTION

MKL_DSS_OUT_OF_MEMORY

dss_delete

Deletes all of data structures created during the solutions process.

Syntax

`dss_delete(handle, opt)`

Input Parameters

opt	`INTEGER`. Options passing argument. The default value is `MKL_DSS_MSG_LVL_WARNING + MKL_DSS_TERM_LVL_ERROR`.

Output Parameters

handle	Data object of `MKL_DSS_HANDLE` type (see Interface Description).

Description

The routine `dss_delete` is called to delete all of the data structures created during the solutions process.

Return Values

`MKL_DSS_SUCCESS`

`MKL_DSS_INVALID_OPTION`

`MKL_DSS_OUT_OF_MEMORY`

dss_statistics

Returns statistics about various phases of the solving process.

Syntax

`dss_statistics(handle, opt, statArr, retValues)`

Input Parameters

handle Data object of MKL_DSS_HANDLE type (see Interface Description).

opt INTEGER Options passing argument.

statArr STRING Input string that defines the type of the returned statistics. Can include one or more of the following string constants (case of the input string has no effect):

ReorderTime	Amount of time taken to do the reordering.
FactorTime	Amount of time taken to do the factorization.
SolveTime	Amount of time taken to solve the problem after factorization.
Determinant	Determinant of the matrix A. For real matrices, determinant is returned as det_pow, det_base in two consecutive return array locations, where: 1.0 ≤abs(*det_base*) < 10.0 and determinant = *det_base**10$^{(det_pow)}$ For complex matrices, determinant is returned as *det_pow*, *det_re*, *det_im* in three consecutive return array locations, where: 1.0 ≤abs(*det_re*) + abs(*det_im*) < 10.0 and determinant = *det_re*, *det_im**10$^{(det_pow)}$
Inertia	Inertia of a real symmetric matrix is defined to be a triplet of nonnegative integers (*p*,*n*,*z*) where *p* is a number of positive eigenvalues, *n* is number of negative eigenvalues, and *z* is number of zero eigenvalues. Inertia will be returned as three consecutive return array locations as *p*,*n*,*z*.

	Computing `Inertia` is only recommended for stable matrices. Unstable matrices can lead to incorrect results. `Inertia` of a *k* by *k* real symmetric positive definite matrix is always `(`*k*`,0,0)`. Therefore `Inertia` is returned only in cases of real symmetric indefinite matrices. For all other matrix types, an error message is returned.
`Flops`	Number of floating point operations performed during factorization.

NOTE. To avoid problems in passing strings from Fortran to C, Fortran users must call the `mkl_cvt_to_null_terminated_str` routine before calling `dss_statistics`. Refer to the description of mkl_cvt_to_null_terminated_str for details.

Output Parameters

`retValues`	`DOUBLE PRECISION` Value of the statistics returned.

Description

The `dss_statistics` routine returns statistics about various phases of the solving process. Use this routine to gather statistics in the following areas:

- time taken to do reordering,
- time taken to do factorization,
- problem solving duration,
- determinant of a matrix,
- inertia of a matrix,
- number of floating point operations taken during factorization.

Statistics are returned corresponding to the specified input string. The value of the statistics is returned in double precision in a return array allocated by user.

For multiple statistics, string constants separated by commas can be used as input. Return values are put into the return array in the same order as specified in the input string.

Statistics should only be requested at appropriate stages of the solving process. For example, inquiring about `FactorTime` before a matrix is factored will lead to errors.

The following table shows the point at which each statistic can be called:

Table 8-3 Statistics Calling Sequences

Type of Statistics	When to Call
`ReorderTime`	After `dss_reorder` is completed successfully.
`FactorTime`	After `dss_factor_real` or `dss_factor_complex` is completed successfully.
`SolveTime`	After `dss_solve_real` or `dss_solve_complex` is completed successfully.
`Determinant`	After `dss_factor_real` or `dss_factor_complex` is completed successfully.
`Inertia`	After `dss_factor_real` is completed successfully and matrix is real, symmetric, and indefinite.
`Flops`	After `dss_factor_real` or `dss_factor_complex` is completed successfully.

Finding "time used to reorder" and "inertia" of a matrix.

The example below illustrates the use of the `dss_statistics` routine.

To find these values, call `dss_statistics`(*`handle`*, *`opt`*, *`statArr`*, *`retValues`*), where *`statArr`* is "`ReorderTime,Inertia`"

In this example, *`retValues`* will have the following values:

Return Values

`MKL_DSS_SUCCESS`

`MKL_DSS_STATISTICS_INVALID_MATRIX`

`MKL_DSS_STATISTICS_INVALID_STATE`

`MKL_DSS_STATISTICS_INVALID_STRING`

mkl_cvt_to_null_terminated_str

Passes character strings from Fortran routines to C routines.

Syntax

`mkl_cvt_to_null_terminated_str (destStr, destLen, srcStr)`

Input Parameters

destLen	INTEGER. Length of the output array *destStr*.
srcStr	STRING. Input string.

Output Parameters

destStr	INTEGER. One-dimensional array of integer.

Description

The routine `mkl_cvt_to_null_terminated_str` is used to pass character strings from Fortran routines to C routines. The strings are converted into integer arrays before being passed to C. Using this routine avoids the problems that can occur on some platforms when passing strings from Fortran to C. The use of this routine is highly recommended.

Implementation Details

Several aspects of the Intel MKL DSS interface are platform-specific and language-specific. In order to promote portability across platforms and ease of use across different languages, users are encouraged to include one of the Intel MKL DSS language-specific header files. Currently, there are three language specific header files:

- mkl_dss.f77 for F77 programs
- mkl_dss.f90 for F90 programs
- mkl_dss.h for C programs

These language-specific header files define symbolic constants for error returns, function options, certain defined data types, and function prototypes.

NOTE. It is strongly recommended that you refer to the constants for options, error returns, and message severities only by the symbolic names that are defined in the header files. Use of the Intel MKL DSS software without including one of the above header files is not supported.

Memory Allocation and Handles

In order to make the Intel MKL DSS routines as easy to use as possible, the routines do not require the user to allocate any temporary working storage. Any storage required by the solver (that is not a user input) is allocated by the solver itself. In order to allow multiple users to access the solver simultaneously, the solver keeps track of the storage allocated for a particular application by using an opaque data object called a **handle**.

Each of the Intel MKL DSS routines either creates, uses or deletes a handle. Consequently, user programs must be able to allocate storage for a handle. The exact syntax for allocating storage for a handle varies from language to language. To help standardize the handle declarations, the language-specific header files declare constants and defined data types that should be used when declaring a handle object in user code.

Fortran 90 programmers should declare a handle as:

```
INCLUDE "mkl_dss.f90"

TYPE(MKL_DSS_HANDLE) handle
```

C and C++ programmers should declare a handle as:

```
#include "mkl_dss.h"

_MKL_DSS_HANDLE_t handle;
```

Fortran 77 programmers using compilers that support eight byte integers, should declare a handle as:

```
INCLUDE "mkl_dss.f77"

INTEGER*8 handle
```

Otherwise they should replace `INTEGER*8` with `DOUBLE PRECISION`.

In addition to the necessary definition for the correct declaration of a handle, the include file also defines the following:

- function prototypes for languages that support prototypes
- symbolic constants that are used for the error returns
- user options for the solver routines

- message severity

Iterative Sparse Solvers based on Reverse Communication Interface (RCI ISS)

The Intel MKL supports an additional to PARDISO interface, namely, the iterative sparse solvers (ISS) based on reverse communication interface (RCI) referred to here as RCI ISS interface. The RCI ISS interface implements a group of user-callable routines that are used in the step-by-step solving process of a symmetric positive definite system (RCI Conjugate Gradient Solver, or RCI CG), and of a non-symmetric indefinite (non-degenerate) system (RCI Flexible Generalized Minimal RESidual Solver, or RCI FGMRES) of linear algebraic equations and exploits the general RCI scheme described in [Dong95]. The terms and concepts required to understand the use of the Intel MKL RCI ISS subroutines are discussed in the Linear Solvers Basics. RCI means that user himself must perform certain operations for the solver (for example, matrix-vector multiplications). When the solver needs the results of such operations, the user must pass them to the solver. This gives the great universality to the solver as it is independent of the specific implementation of the operations like the matrix-vector multiplication. However, this approach requires some additional work from the user. To simplify this task, the user can use the built-in sparse matrix-vector multiplications and triangular solvers routines (see Sparse BLAS Level 2 and Level 3.

NOTE. The RCI CG solver is implemented in two versions: for system of equations with single right hand side, and for system of equations with multiple right hand sides.

The CG method may fail to compute the solution or compute the wrong solution if the matrix of the system is not symmetric and positive definite.

The FGMRES method may fail if the matrix is degenerate.

The solving process is conceptually divided into four steps, as shown in the Table 8-4 , that lists the names of the routines, and describes their general use.

Table 8-4 RCI ISS Interface Routines

Routine	Description
dcg_init, dcgmrhs_init, dfgmres_init	Initializes the solver.
dcg_check, dcgmrhs_check, dfgmres_check	Checks the consistency and correctness of the user defined data.

Routine	Description
dcg, dcgmrhs, dfgmres	Computes the approximate solution vector.
dcg_get, dcgmrhs_get, dfgmres_get	Retrieves the number of the current iteration.

The Intel MKL RCI ISS interface routines are normally invoked in the order in which they are listed in Table8-4 , with the exception of dcg_get, dcgmrhs_get, and dfgmres_get routines that can be invoked at any place in the code. However, in this case some precautions should be taken to avoid the wrong results. Advanced users can change that order if they need it. For others it is strongly recommended to follow the above order of calls.

The following diagram in Figure 8-3 indicates the typical order in which the RCI ISS interface routines can be invoked.

Figure 8-3 Typical Order for Invoking RCI ISS interface Routines

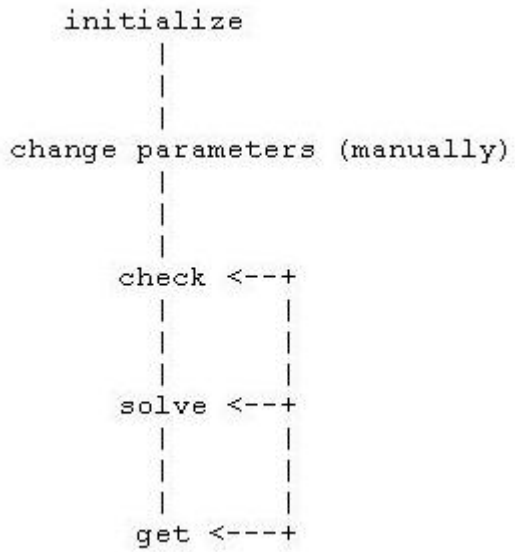


Figure 8-4 and Figure 8-5 show the general schemes of using the RCI CG and RCI FGMRES routines respectively.

Figure 8-4 General Scheme of Using RCI CG Routines

```
...
generate matrix A
generate preconditioner C (optional)
   call dcg_init(n, x, b, RCI_request, ipar, dpar, tmp)
   change parameters in ipar, dpar if necessary
   call dcg_check(n, x, b, RCI_request, ipar, dpar, tmp)
1  call dcg(n, x, b, RCI_request, ipar, dpar, tmp)
   if (RCI_request.eq.1) then
    multiply the matrix A by tmp(1:n,1) and put the result in tmp(1:n,2)
   It is possible to use MKL Sparse BLAS Level 2 subroutines for this operation
c  proceed with CG iterations
     goto 1
   endif
   if (RCI_request.eq.2)then
    do the stopping test
    if (test not passed) then
c  proceed with CG iterations
     go to 1
    else
c  stop CG iterations
     goto 2
    endif
   endif
   if (RCI_request.eq.3) then (optional)
   apply the preconditioner C inverse to tmp(1:n,3) and put the result in tmp(1:n,4)
```

```
c  proceed with CG iterations
    goto 1
   end
2  call dcg_get(n, x, b, RCI_request, ipar, dpar, tmp, itercount)
  current iteration number is in itercount
  the computed approximation is in the array x
```

Figure 8-5 General Scheme of Using RCI FGMRES Routines

```
...
generate matrix A
generate preconditioner C (optional)
   call dfgmres_init(n, x, b, RCI_request, ipar, dpar, tmp)
    change parameters in ipar, dpar if necessary
    call dfgmres_check(n, x, b, RCI_request, ipar, dpar, tmp)
1   call dfgmres(n, x, b, RCI_request, ipar, dpar, tmp)
    if (RCI_request.eq.1) then
     multiply the matrix A by tmp(ipar(22)) and put the result in tmp(ipar(23))
     It is possible to use MKL Sparse BLAS Level 2 subroutines for this operation
c  proceed with FGMRES iterations
    goto 1
   endif
   if (RCI_request.eq.2) then
    do the stopping test
    if (test not passed) then
c  proceed with FGMRES iterations
     go to 1
    else
c  stop FGMRES iterations
     goto 2
```

```
    endif
  endif
  if (RCI_request.eq.3) then (optional)
  apply the preconditioner C inverse to tmp(ipar(22)) and put the result in tmp(ipar(23))
c  proceed with FGMRES iterations
    goto 1
  endif
  if (RCI_request.eq.4) then
  check the norm of the next orthogonal vector, it is contained in dpar(7)
    if (the norm is not zero up to rounding/computational errors) then
c  proceed with FGMRES iterations
     goto 1
    else
c  stop FGMRES iterations
     goto 2
    endif
  endif
```

2 call `dfgmres_get(n, x, b, RCI_request, ipar, dpar, tmp, itercount)`

current iteration number is in `itercount`

the computed approximation is in the array `x`

Note that for the FGMRES method the array `x` initially contains the current initial approximation to the solution that can be updated only by calling the `dfgmres_get` routine that updates the solution in accordance with the computations performed by the `dfgmres` routine.

The pseudo codes in these figures demonstrate two main differences in use of RCI CG and RCI FGMRES interfaces. The first difference relates to the `RCI_request=3` (different locations in the `tmp` array which is 2-dimensional for CG and 1-dimensional for FGMRES), the second difference relates to `RCI_request=4` (RCI CG interface never produces `RCI_request=4`).

You can find example codes that use RCI ISS interface routines to solve systems of linear equations in the Iterative Sparse Solver Code Example section in the Appendix C.

CG Interface Description

All types in this documentation refer to the standard Fortran types, `INTEGER`, and `DOUBLE PRECISION`.

C and C++ programmers should refer to the section Calling Sparse Solver Routines From C/C++ for information on mapping Fortran types to C/C++ types.

Each routine for the RCI CG solver is implemented in two versions: for system of equations with single right hand side (SRHS), and for system of equations with multiple right hand sides (MRHS). The routines for the system with MRHS contain the suffix `mrhs` in their names.

NOTE. The routines for the system with MRHS can be used with the names without suffix `mrhs`. To do this, the user must switch on the compiler's preprocessor and include the files mkl_solver.h for C/C++, or mkl_solver.f77 for FORTRAN.

Routines Options

All of the RCI CG routines have parameters for passing various options to the routines. The values for these parameters should be specified very carefully (see CG Common Parameters), and they can be changed during computations according to the user's needs.

NOTE. Users must provide correct and consistent parameters to the subroutines to avoid fails or wrong results.

User Data Arrays

Many of the RCI CG routines take arrays of user data as input. For example, user arrays are passed to the routine `dcg` to compute the solution of a system of linear algebraic equations. The Intel MKL RCI CG routines do not make copies of the user input arrays to minimize storage requirements and improve overall run-time efficiency.

CG Common Parameters

NOTE. The default and initial values listed below are assigned to the parameters by the calling the `dcg_init`/`dcgmrhs_init` routine.

n - INTEGER, this parameter sets the size of the problem in the dcg_init/dcgmrhs_init routine. All other routines uses *ipar*(1) parameter instead.

x - DOUBLE PRECISION array of size *n* for SRHS, or matrix of size *n* by *nrhs* for MRHS. This parameter contains the current approximation to the solution. Before the first call to the dcg/dcgmrhs routine, it contains the initial approximation to the solution.

nrhs - INTEGER, this parameter sets the number of right-hand sides for MRHS routines.

b - DOUBLE PRECISION array containing the single right-hand side vector, or matrix (*nrhs*, *n*) containing the right-hand side vectors.

RCI_request - INTEGER, this parameter is used to inform about the result of work of the RCI CG routines. The negative values of the parameter indicate that the routine is completed with errors or warnings. The 0 value indicates the successful completion of the task. The positive values mean that the user must perform certain actions, specifically:

RCI_request= 1 - multiply the matrix by *tmp*(1:*n*,1), put the result in *tmp*(1:*n*,2), and return the control to the dcg/dcgmrhs routine;

RCI_request= 2 - perform the stopping test(s). If they fail, return the control to the dcg/dcgmrhs routine. Otherwise, the solution is found and stored in the *x*;

RCI_request= 3 - for SRHS: apply the preconditioner to *tmp*(1:*n*,3), put the result in *tmp*(1:*n*,4), and return the control to the dcg routine;
- for MRHS: apply the preconditioner to *tmp*(:,3+*ipar*(3)), put the result in *tmp*(:,3), and return the control to the dcgmrhs routine.

Note that the dcg_get/dcgmrhs_get routine does not change the parameter *RCI_request*. This allows user to use this routine inside the Reverse Communication computations.

ipar - INTEGER array, of size 128 for SRHS, and of size (128+2*nrhs) for MRHS ; this parameter is used to specify the integer set of data for the RCI CG computations:

ipar(1) - specifies the size of the problem. The dcg_init/dcgmrhs_init routine assignes *ipar*(1)=*n*. All other routines uses this parameter instead of *n*. There is no default value for this parameter.

ipar(2) - specifies the type of output for error and warning messages that are generated by the RCI CG routines. The default value 6 means that all messages are displayed on the screen. Otherwise the error and warning messages are written to the newly created files dcg_errors.txt and dcg_check_warnings.txt respectively. Note that if *ipar*(6) and *ipar*(7) parameters are set to 0, error and warning messages are not generated at all.

ipar(3) - for SRHS: contains the current stage of the RCI CG computations, the initial value is 1;
- for MRHS: contains the right-hand side for which the calculations are currently performed.

NOTE. It is highly non-recommended to alter this variable during computations.

ipar(4) - contains the current iteration number, the initial value is 0.

ipar(5) - specifies the maximum number of iterations, the default value is *min{150,n}* .

ipar(6) - if the value is not equal to 0, the routines output error messages in accordance with the parameter *ipar*(2). Otherwise, the routines

do not output error messages at all, but they return a negative value of the parameter *RCI_request*. The default value is 1.

ipar(7) - if the value is not equal to 0, the routines output warning messages in accordance with the parameter *ipar*(2). Otherwise, the routines do not output warning messages at all, but they return a negative value of the parameter *RCI_request*. The default value is 1.

ipar(8) - if the value is not equal to 0, the dcg/dcgmrhs routine performs the stopping test for the maximum number of iterations, namely, *ipar*(4)≤*ipar*(5). Otherwise, the method is stopped and corresponding value is assigned to the *RCI_request*. If the value is 0, the routine does not perform this stopping test. The default value is 1.

ipar(9) - if the value is not equal to 0, the dcg/dcgmrhs routine performs the residual stopping test, namely, *dpar*(5)≤*dpar*(4)=*dpar*(1)**dpar*(3)+*dpar*(2). Otherwise, the method is stopped and corresponding value is assigned to the *RCI_request*. If the value is 0, the routine does not perform this stopping test. The default value is 0.

ipar(10) - if the value is not equal to 0, the dcg/dcgmrhs routine requests for the user defined stopping test by setting *RCI_request*=2. If the value is 0, the routine does not perform the user defined stopping test. The default value is 1.

NOTE. At least one of the parameters *ipar*(8)-*ipar*(10) must be set to 1.

ipar(11), - if the value is equal to 0, the dcg/dcgmrhs routine runs the non-preconditioned version of the corresponding Conjugate Gradient method. Otherwise, the routine runs the preconditioned version of the Conjugate Gradient method, and asks the user to perform the preconditioning step by setting the parameter *RCI_request*=3. The default value is 0.

ipar(11:128), *ipar*(11:128+2*nrhs) are reserved and not used in the current RCI CG SRHS and MRHS routines respectively.

NOTE. Advanced users can define the array in the code using RCI CG SRHS as follows: INTEGER *ipar*(11). However, to guarantee the compatibility with the future releases of the Intel MKL it is highly recommended to declare the array *ipar* of length 128.

dpar - DOUBLE PRECISION array, for SRHS of size 128, for MRHS of size (128+2*nrhs); this parameter is used to specify the double precision set of data for the RCI CG computations, specifically:

dpar(1) - specifies the relative tolerance, the default value is 1.0D-6;

dpar(2) - specifies the absolute tolerance, the default value is 0.0D-0;

dpar(3) - specifies the square norm of initial residual (if it is computed in the dcg/dcgmrhs routine), the initial value is 0;

dpar(4) - service variable, it is equal to *dpar*(1)*dpar(3)+*dpar*(2) (if it is computed in the dcg/dcgmrhs routine), the initial value is 0;

dpar(5) - specifies the square norm of current residual, the initial value is 0.0;

dpar(6) - specifies the square norm of residual from the previous iteration step (if available), the initial value is 0.0;

dpar(7) - contains the "alpha" parameter of the CG method, the initial value is 0.0;

dpar(8) - contains the "beta" parameter of the CG method, it is equal to *dpar*(5)/*dpar*(6), the initial value is 0.0;

dpar(9:128), *dpar*(9:128+2**nrhs*) are reserved and not used in the current RCI CG SRHS and MRHS routines respectively.

NOTE. Advanced users can define this array in the code using RCI CG SRHS as follows: DOUBLE PRECISION *dpar*(8). However, to guarantee the compatibility with the future releases of the Intel MKL it is highly recommended to declare the array *dpar* of length 128.

tmp - DOUBLE PRECISION array, for SRHS of size (*n*,4), for MRHS of size (*n*,3+*nrhs*); this parameter is used to supply the double precision temporary space for the RCI CG computations, specifically:

tmp(:,1) - specifies the current search direction. The initial value is 0.0;

tmp(:,2) - contains the matrix multiplied by the current search direction. The initial value is 0.0;

tmp(:,3) - contains the current residual. The initial value is 0.0;

tmp(:,4) - contains the inverse of the preconditioner applied to the current residual. There is no initial value for this parameter.

NOTE. Advanced users can define this array in the code using RCI CG SRHS as `DOUBLE PRECISION tmp(n,3)` if they run only non-preconditioned CG iterations.

FGMRES Interface Description

All types in this documentation refer to the standard Fortran types, `INTEGER`, and `DOUBLE PRECISION`.

C and C++ programmers should refer to the section Calling Sparse Solver Routines From C/C++ for information on mapping Fortran types to C/C++ types.

Routines Options

All of the RCI FGMRES routines have parameters for passing various options to the routines. The values for these parameters should be specified very carefully (see FGMRES Common Parameters), and they can be changed during computations according to the user's needs.

NOTE. Users must provide correct and consistent parameters to the subroutines to avoid fails or wrong results.

User Data Arrays

Many of the RCI FGMRES routines take arrays of user data as input. For example, user arrays are passed to the routine>`dfgmres` to compute the solution of a system of linear algebraic equations. In order to minimize storage requirements and improve overall run-time efficiency, the Intel MKL RCI FGMRES routines do not make copies of the user input arrays.

FGMRES Common Parameters

NOTE. The default and initial values listed below are assigned to the parameters by the calling the`dfgmres_init` routine.

n - INTEGER, this parameter sets the size of the problem in the dfgmres_init routine. All other routines uses *ipar*(1) parameter instead.

x - DOUBLE PRECISION array, this parameter contains the current approximation to the solution vector. Before the first call to the dfgmres routine, it contains the initial approximation to the solution vector.

b - DOUBLE PRECISION array, this parameter contains the right-hand side vector. Depending on user requests, it may contain the approximate solution later.

RCI_request - INTEGER, this parameter is used to inform about the result of work of the RCI FGMRES routines. The negative values of the parameter indicate that the routine is completed with errors or warnings. The 0 value indicates the successful completion of the task. The positive values mean that the user must perform certain actions, specifically:

- *RCI_request*= 1 - multiply the matrix by *tmp*(*ipar*(22)), put the result in *tmp*(*ipar*(23)), and return the control to the dfgmres routine;
- *RCI_request*= 2 - perform the stopping test(s). If they fail, return the control to the dfgres routine. Otherwise, the solution can be updated by a subsequent call to dfgmres_get routine;
- *RCI_request*= 3 - apply the preconditioner to *tmp*(*ipar*(22)), put the result in *tmp*(*ipar*(23)), and return the control to the dfgmres routine.
- *RCI_request*= 4 - check if the norm of the current orthogonal vector is not zero up to rounding/computational errors. Return the control to the dfgmres routine if it is not zero, otherwise complete the solution process by calling dfgmres_get routine.

ipar(128) - INTEGER array, this parameter is used to specify the integer set of data for the RCI FGMRES computations:

ipar(1) - specifies the size of the problem. The `dfgmres_init` routine assignes *ipar*(1)=*n*. All other routines uses this parameter instead of *n*. There is no default value for this parameter.

ipar(2) - specifies the type of output for error and warning messages that are generated by the RCI FGMRES routines. The default value `6` means that all messages are displayed on the screen. Otherwise the error and warning messages are written to the newly created file MKL_RCI_FGMRES_Log.txt. Note that if *ipar*(6) and *ipar*(7) parameters are set to 0, error and warning messages are not generated at all.

ipar(3) - contains the current stage of the RCI FGMRES computations, the initial value is 1.

NOTE. It is highly non-recommended to alter this variable during computations.

ipar(4) - contains the current iteration number, the initial value is 0.

ipar(5) - specifies the maximum number of iterations, the default value is *min* {150,*n*}.

ipar(6) - if the value is not equal to 0, the routines output error messages in accordance with the parameter *ipar*(2). Otherwise, the routines do not output error messages at all, but they return a negative value of the parameter *RCI_request*. The default value is 1.

ipar(7) - if the value is not equal to 0, the routines output warning messages in accordance with the parameter *ipar*(2). Otherwise, the routines do not output warning messages at

all, but they return a negative value of the parameter *RCI_request*. The default value is 1.

ipar(8) - if the value is not equal to 0, the dfmres routine performs the stopping test for the maximum number of iterations, namely, *ipar*(4)≤*ipar*(5). Otherwise, the method is stopped and corresponding value is assigned to the *RCI_request*. If the value is 0, the dfgmres routine does not perform this stopping test. The default value is 1.

ipar(9) - if the value is not equal to 0, the dfgmres routine performs the residual stopping test, namely, *dpar*(5)≤*dpar*(4) = *dpar*(1)·*dpar*(3)+*dpar*(2). If the criterion is fulfilled, the method is stopped and corresponding value is assigned to the *RCI_request*. If the value is 0, the dfgmres routine does not perform this stopping test. The default value is 0.

ipar(10) - if the value is not equal to 0, the dfgmres routine requests for the user defined stopping test by setting *RCI_request*=2. If the value is 0, the dfgmres routine does not perform the user defined stopping test. The default value is 1.

NOTE. At least one of the parameters *ipar*(8)-*ipar*(10) must be set to 1.

ipar(11) - if the value is equal to 0, the dfgmres routine runs the non-preconditioned version of the FGMRES method. Otherwise, the routine runs the preconditioned version of the FGMRES

method, and asks the user to perform the preconditioning step by setting the parameter *RCI_request*=3. The default value is 0.

ipar(12) - if the value is not equal to 0, the dfgmres routine performs the automatic test for zero norm of the currently generated vector, namely, *dpar*(7)≤*dpar*(8), where *dpar*(8) contains the tolerance value. Otherwise, the routine asks the user to perform this check by setting the parameter *RCI_request*=4. The default value is 0.

ipar(13) - if the value is equal to 0, the dfgmres_get routine updates the solution to the vector *x* according to the computations done by the dfgmres routine. If the value is positive, the routine writes the solution to the right hand side vector *b*. If the value is negative, the routine returns only the number of the current iteration, and does not update the solution. The default value is 0

NOTE. Advanced users may use the dfgmres_get routine at any place in the code. In this case special attention should be paid to the parameter *ipar*(13). The RCI FGMRES iterations can be continued after the call to dfgmres_get routine only if the parameter *ipar*(13) is not equal to zero. If *ipar*(13) is positive, then the updated solution will overwrite the right hand side in the vector *b*. If the user wants to run the restarted version of FGMRES with the same right hand side, it should be saved in a different memory location before the first call to dfgmres_get routine with positive *ipar*(13).

ipar(14) - contains the internal iteration counter that counts the number of iterations before the restart takes place. The initial value is 0.

NOTE. It is highly non-recommended to alter this variable during computations.

ipar(15) - specifies the length of the non-restarted FGMRES iterations. To run the restarted version of the FGMRES method, the user should assign to *ipar*(15) the number of iterations before the restart takes place. The default value is *min* {150, *n*}, that is, by default the non-restarted version of FGMRES method is used.

ipar(16) - service variable, specifies the location of the rotated Hessenberg matrix from which the matrix stored in the packed format (see "Matrix Arguments" in the Appendix B for details) is started in the tmp array.

ipar(17) - service variable, specifies the location of the rotation cosines from which the vector of cosines is started in the *tmp* array.

ipar(18) - service variable, specifies the location of the rotation sines from which the vector of sines is started in the *tmp* array.

ipar(19) - service variable, specifies the location of the rotated residual vector from which the vector is started in the *tmp* array.

ipar(20) - service variable, specifies the location of the least squares solution vector from which the vector is started in the *tmp* array.

ipar(21) - service variable, specifies the location of the set of preconditioned vectors from which the set is started in the *tmp* array. The memory

locations in the tmp array starting from *ipar*(21) are used only for preconditioned FGMRES method.

ipar(22) - specifies the memory location from which the first vector (source) used in operations requested via *RCI_request* is started in the *tmp* array.

ipar(23) - specifies the memory location from which the second vector (source) used in operations requested via *RCI_request* is started in the *tmp* array.

ipar(24:128) are reserved and not used in the current RCI FGMRES routines.

NOTE. Advanced users can define the array in the code as follows: INTEGER *ipar*(23). However, to guarantee the compatibility with the future releases of the Intel MKL it is highly recommended to declare the array *ipar* of length 128.

dpar(128) - DOUBLE PRECISION array, this parameter is used to specify the double precision set of data for the RCI CG computations, specifically:

dpar(1) - specifies the relative tolerance, the default value is 1.0D-6;

dpar(2) - specifies the absolute tolerance, the default value is 0.0D-0;

dpar(3) - pecifies the Euclidean norm of the initial residual (if it is computed in the dfgmres routine), the initial value is 0.0;

dpar(4) - service variable, it is equal to *dpar*(1)**dpar*(3)+*dpar*(2) (if it is computed in the dfgmres routine), the initial value is 0.0;

dpar(5) - specifies the Euclidean norm of the current residual, the initial value is 0.0;

dpar(6) - specifies the Euclidean norm of residual from the previous iteration step (if available), the initial value is 0.0;

dpar(7) - contains the norm of the generated vector, the initial value is 0.0;

NOTE. For reference only: in terms of [Saad03] this parameter is the coefficient $h_{k+1,k}$ of the Hessenberg matrix.

dpar(8) - contains the tolerance for the "zero" norm of the currently generated vector, the default value is 1.0D-12.

dpar(9:128) are reserved and not used in the current RCI FGMRES routines.

NOTE. Advanced users can define this array in the code as follows: `DOUBLE PRECISION` *dpar*(8). However, to guarantee the compatibility with the future releases of the Intel MKL it is highly recommended to declare the array *dpar* of length 128.

tmp -`DOUBLE PRECISION` array of size ((2**ipar*(15)+1)**n* + *ipar*(15)*(*ipar*(15)+9)/2 + 1)), this parameter is used to supply the double precision temporary space for the RCI FGMRES computations, specifically:

tmp(1:ipar(16)*−1)* - contains the sequence of generated by FGMRES method vectors. The initial value is 0.0 for the first part of this memory of length *n*;

tmp(ipar(16):ipar(17)–1) contains the rotated Hessenberg matrix generated by FGMRES method stored in the packed format. There is no initial value for this part of *tmp* array;

tmp(ipar(17):ipar(18)–1) - contains the rotation cosines vector generated by FGMRES method. There is no initial value for this part of *tmp* array;

tmp(ipar(18):ipar(19)–1) contains the rotation sines vector generated by FGMRES method. There is no initial value for this part of *tmp* array;

tmp(ipar(19):ipar(20)–1) contains the rotated residual vector generated by FGMRES method. There is no initial value for this part of *tmp* array;

tmp(ipar(20):ipar(21)–1) contains the solution vector to the least squares problem generated by FGMRES method. There is no initial value for this part of *tmp* array;

tmp(ipar(21):) - - contains the set of preconditioned vectors generated for FGMRES method by the user. This part of *tmp* array is not used if non-preconditioned version of FGMRES method is called. There is no initial value for this part of *tmp* array.

NOTE. Advanced users can define this array in the code as `DOUBLE PRECISION tmp((2*ipar(15)+1)*n + ipar(15)*(ipar(15)+9)/2 + 1))` if they run only non-preconditioned FGMRES iterations.

dcg_init

Initializes the solver.

Syntax

`dcg_init(n, x, b, RCI_request, ipar, dpar, tmp)`

Input Parameters

n	`INTEGER`. Contains the size of the problem, and size of arrays *x* and *b*.
x	`DOUBLE PRECISION` array of size *n*. Contains the initial approximation to the solution vector. Normally it is equal to 0 or to *b*.
b	`DOUBLE PRECISION` array of size *n*. Contains the right-hand side vector.

Output Parameters

RCI_request	`INTEGER`. Informs about the task completion.
ipar	`INTEGER` array of size `128`. Refer to the CG Common Parameters.
dpar	`DOUBLE PRECISION` array of size `128`. Refer to the CG Common Parameters.
tmp	`DOUBLE PRECISION` array of size `(n,4)`. Refer to the CG Common Parameters.

Description

The routine `dcg_init` is called to initialize the solver. After initialization all subsequent invocations of Intel MKL RCI CG routines can use the values of all parameters that are returned by `dcg_init`. Advanced users can skip this step and set the values to these parameters directly in the corresponding routines.

WARNING. Users can modify the contents of these arrays after they are passed to the solver routine only if they are sure that the values are correct and consistent. Basic check for correctness and consistency can be done by calling the `dcg_check` routine, but it does not guarantee that the method will work correctly.

Return Values

RCI_request= 0	The routine completed task normally.
RCI_request= -10000	The routine failed to complete the task.

dcg_check

Checks the consistency and correctness of the user defined data.

Syntax

dcg_check(*n*, *x*, *b*, *RCI_request*, *ipar*, *dpar*, *tmp*)

Input Parameters

n	INTEGER. Contains the size of the problem, and size of arrays *x* and *b*.
x	DOUBLE PRECISION array of size *n*. Contains the initial approximation to the solution vector. Normally it is equal to 0 or to *b*.
b	DOUBLE PRECISION array of size *n*. Contains the right-hand side vector.

Output Parameters

RCI_request	INTEGER. Informs about the task completion.
ipar	INTEGERarray of size 128. Refer to the CG Common Parameters.
dpar	DOUBLE PRECISION array of size 128. Refer to the CG Common Parameters.
tmp	DOUBLE PRECISION array of size (*n*,4). Refer to the CG Common Parameters.

Description

The routine dcg_check checks the consistency and correctness of the parameters to be passed to the solver routine dcg. However this operation does not guarantee that the method will be able to produce the correct result. It only reduces the chance to make a mistake in the parameters of the method. Advanced users can skip it if they are sure that the correct data is specified in the solver parameters.

WARNING. Users can modify the contents of these arrays after they are passed to the solver routine only if they are sure that the values are correct and consistent. Basic check for correctness and consistency can be done by calling the dcg_check routine, but it does not guarantee that the method will work correctly.

Note that the lengths of all vectors are assumed to have been defined in a previous call to dcg_init subroutine.

Return Values

RCI_request= 0	The routine completed task normally.
RCI_request= -1100	The routine is interrupted, errors occur.
RCI_request= -1001	The routine returns some warning messages.
RCI_request= -1010	The routine changed some parameters to make them consistent or correct.
RCI_request= -1011	The routine returns some warning messages and changed some parameters.

dcg

Computes the approximate solution vector.

Syntax

dcg(*n*, *x*, *b*, *RCI_request*, *ipar*, *dpar*, *tmp*)

Input Parameters

n	INTEGER. Contains the size of the problem, and size of arrays *x* and *b*.

x	DOUBLE PRECISION array of size *n*. Contains the initial approximation to the solution vector.
b	DOUBLE PRECISION array of size *n*. Contains the right-hand side vector.
tmp	DOUBLE PRECISION array of size (*n*,4). Refer to the CG Common Parameters.

Output Parameters

RCI_request	INTEGER. Informs about the task completion status.
x	DOUBLE PRECISION array of size *n*. Contains the updated approximation to the solution vector.
ipar	INTEGER array of size *128*. Refer to the CG Common Parameters.
dpar	DOUBLE PRECISION array of size *128*. Refer to the CG Common Parameters.
tmp	DOUBLE PRECISION array of size (*n*,4). Refer to the CG Common Parameters.

Description

The routine dcg computes the approximate solution vector using the CG method [Young71]. The value that was in the vector *x* before the first call, the routine dcg uses as an initial approximation to the solution. The parameter *RCI_request* inform the user about task completion status and ask for results of certain operations that are required to the solver.

Note that the lengths of all vectors are assumed to have been defined in a previous call to the dcg_init routine.

Return Values

RCI_request=0	The routine completed task normally, the solution is found and stored in the vector *x*. This occurs only if the stopping tests are fully automatic. For the user defined stopping tests, see the comments to the *RCI_request*= 2.

RCI_request=-1	The routine is interrupted because the maximal number of iterations is reached, but the relative stopping criterion is not satisfied (this occurs only if both tests are requested by the user).
RCI_request=-2	The routine is interrupted because the attempt to divide by zero occurs. This happens if the matrix is (almost) non-positive definite.
RCI_request=- 10	The routine is interrupted because the residual norm is invalid. (Probably, the data in *dpar*(6) were altered outside of the routine, or the dcg_check routine was not called).
RCI_request=-11	The routine is interrupted because it enters the infinite cycle. (Probably, the data in *ipar*(8), *ipar*(9), *ipar*(10) were altered outside of the routine, or the dcg_check routine was not called).
RCI_request= 1	Asks user to multiply the matrix by *tmp*(1:*n*,1), put the result in the *tmp*(1:*n*,2), and return the control back to the routine dcg.
RCI_request= 2	Asks user to perform the stopping test(s). If they fail, the user should return the control back to the dcg routine. Otherwise, the solution is found and stored in the vector *x*.
RCI_request= 3	Asks user to apply the preconditioner to *tmp*(:,3), put the result in the *tmp*(:,4), and return the control back to the routine dcg.

dcg_get

Retrieves the number of the current iteration.

Syntax

dcg_get(*n*, *x*, *b*, *RCI_request*, *ipar*, *dpar*, *tmp*, *itercount*)

Input Parameters

n	INTEGER. Contains the size of the problem, and size of arrays *x* and *b*.

x	DOUBLE PRECISION array of size *n*. Contains the initial approximation vector to the solution.
b	DOUBLE PRECISION array of size *n*. Contains the right-hand side vector.
RCI_request	INTEGER. This parameter is not used.
ipar	INTEGER array of size 128. Refer to the CG Common Parameters.
dpar	DOUBLE PRECISION array of size 128. Refer to the CG Common Parameters.
tmp	DOUBLE PRECISION array of size (*n*,4). Refer to the CG Common Parameters.

Output Parameters

itercount	INTEGER argument. Contains the value of the current iteration number.

Description

The routine dcg_get is called to retrieve the current iteration number of the solutions process.

Return Values

The routine dcg_get does not return any value.

dcgmrhs_init

Initializes the RCI CG solver with MHRS.

Syntax

dcgmrhs_init(*n*, *x*, *nrhs*, *b*, *method*, *RCI_request*, *ipar*, *dpar*, *tmp*)

Input Parameters

n	INTEGER. Contains the size of the problem, and size of arrays *x* and *b*.
x	DOUBLE PRECISION matrix of size *n* by *nrhs*. Contains the initial approximation to the solution vectors. Normally it is equal to 0 or to *b*.

nrhs	INTEGER. This parameter sets the number of right-hand sides.
b	DOUBLE PRECISION matrix of size (*nrhs*,*n*). Contains the right-hand side vectors.
method	INTEGER. Specifies the method of solution: 1 - CG with multiple right hand sides; default value. 2 - Block-CG (not supported now)

Output Parameters

RCI_request	INTEGER. Informs about the task completion.
ipar	INTEGER array of size *(128+2*nrhs)*. Refer to the CG Common Parameters.
dpar	DOUBLE PRECISION array of size *(128+2*nrhs)*. Refer to the CG Common Parameters.
tmp	DOUBLE PRECISION array of size (*n*, 3+*nrhs*). Refer to the CG Common Parameters.

Description

The routine dcgmrhs_init is called to initialize the solver. After initialization all subsequent invocations of Intel MKL RCI CG with multiple right hand sides (MRHS) routines can use the values of all parameters that are returned by dcgmrhs_init. Advanced users can skip this step and set the values to these parameters directly in the corresponding routines.

WARNING. Users can modify the contents of these arrays after they are passed to the solver routine only if they are sure that the values are correct and consistent. Basic check for correctness and consistency can be done by calling the dcgmrhs_check routine, but it does not guarantee that the method will work correctly.

NOTE. To use this routine with the name dcg_init, the user must switch on the compiler's preprocessor and include the files mkl_solver.h for C/C++, or mkl_solver.f77 for FORTRAN.

Return Values

RCI_request= 0	The routine completed task normally.

RCI_request= -10000	The routine failed to complete the task.

dcgmrhs_check

Checks the consistency and correctness of the user defined data.

Syntax

```
dcgmrhs_check(n, x, nrhs, b, RCI_request, ipar, dpar, tmp)
```

Input Parameters

n	INTEGER. Contains the size of the problem, and size of arrays *x* and *b*.
x	DOUBLE PRECISION matrix of size *n* by *nrhs*. Contains the initial approximation to the solution vectors. Normally it is equal to 0 or to *b*.
nrhs	INTEGER. This parameter sets the number of right-hand sides.
b	DOUBLE PRECISION matrix of size (*nrhs*,*n*). Contains the right-hand side vectors.

Output Parameters

RCI_request	INTEGER. Informs about the task completion.
ipar	INTEGER array of size *(128+2*nrhs)*. Refer to the CG Common Parameters.
dpar	DOUBLE PRECISION array of size *(128+2*nrhs)*. Refer to the CG Common Parameters.
tmp	DOUBLE PRECISION array of size (*n*, 3+*nrhs*). Refer to the CG Common Parameters.

Description

The routine dcgmrhs_check checks the consistency and correctness of the parameters to be passed to the solver routine dcgmrhs. However this operation does not guarantee that the method will be able to produce the correct result. It only reduces the chance to make a mistake in the parameters of the method. Advanced users can skip it if they are sure that the correct data is specified in the solver parameters.

WARNING. Users can modify the contents of these arrays after they are passed to the solver routine only if they are sure that the values are correct and consistent. Basic check for correctness and consistency can be done by calling the `dcgmrhs_check` routine, but it does not guarantee that the method will work correctly.

Note that the lengths of all vectors are assumed to have been defined in a previous call to `dcgmrhs_init` subroutine.

NOTE. To use this routine with the name `dcg_check`, the user must switch on the compiler's preprocessor and include the files mkl_solver.h for C/C++, or mkl_solver.f77 for FORTRAN.

Return Values

RCI_request= 0	The routine completed task normally.
RCI_request= -1100	The routine is interrupted, errors occur.
RCI_request= -1001	The routine returns some warning messages.
RCI_request= -1010	The routine changed some parameters to make them consistent or correct.
RCI_request= -1011	The routine returns some warning messages and changed some parameters.

dcgmrhs

Computes the approximate solution vectors.

Syntax

`dcgmrhs(n, x, nrhs, b, RCI_request, ipar, dpar, tmp)`

Input Parameters

n	`INTEGER`. Contains the size of the problem, and size of arrays *x* and *b*.
x	`DOUBLE PRECISION` matrix of size *n* by *nrhs*. Contains the initial approximation to the solution vectors.

nrhs	INTEGER. This parameter sets the number of right-hand sides.
b	DOUBLE PRECISION matrix of size (*nrhs*,*n*). Contains the right-hand side vectors.
tmp	DOUBLE PRECISION array of size (*n*, 3+*nrhs*). Refer to the CG Common Parameters.

Output Parameters

RCI_request	INTEGER. Informs about the task completion status.
x	DOUBLE PRECISION matrix of size *n* by *nrhs*. Contains the updated approximation to the solution vectors.
ipar	INTEGER array of size *(128+2*nrhs)*. Refer to the CG Common Parameters.
dpar	DOUBLE PRECISION array of size *(128+2*nrhs)*. Refer to the CG Common Parameters.
tmp	DOUBLE PRECISION array of size (*n*, 3+*nrhs*). Refer to the CG Common Parameters.

Description

The routine dcgmrhs computes the approximate solution vectors using the CG with multiple right hand sides (MRHS) method [Young71]. The value that was in the *x* before the first call, the routine dcgmrhs uses as an initial approximation to the solution. The parameter *RCI_request* inform the user about task completion status and ask for results of certain operations that are required to the solver.

Note that the lengths of all vectors are assumed to have been defined in a previous call to the dcgmrhs_init routine.

NOTE. To use this routine with the name dcg, the user must switch on the compiler's preprocessor and include the files mkl_solver.h for C/C++, or mkl_solver.f77 for FORTRAN.

Return Values

RCI_request=0	The routine completed task normally, the solution is found and stored in the matrixr *x*. This occurs only if the stopping tests are fully automatic. For the user defined stopping tests, see the comments to the *RCI_request*= 2.
RCI_request=-1	The routine is interrupted because the maximal number of iterations is reached, but the relative stopping criterion is not satisfied (this occurs only if both tests are requested by the user).
RCI_request=-2	The routine is interrupted because the attempt to divide by zero occurs. This happens if the matrix is (almost) non-positive definite.
RCI_request=- 10	The routine is interrupted because the residual norm is invalid. (Probably, the data in *dpar*(6) were altered outside of the routine, or the dcgmrhs_check routine was not called).
RCI_request=-11	The routine is interrupted because it enters the infinite cycle. (Probably, the data in *ipar*(8), *ipar*(9), *ipar*(10) were altered outside of the routine, or the dcgmrhs_check routine was not called).
RCI_request= 1	Asks user to multiply the matrix by *tmp*(1:*n*,1), put the result in the *tmp*(1:*n*,2), and return the control back to the routine dcgmrhs.
RCI_request= 2	Asks user to perform the stopping test(s). If they fail, the user should return the control back to the dcgmrhs routine. Otherwise, the solution is found and stored in the matrix *x*.
RCI_request= 3	Asks user to apply the preconditioner to *tmp*(:,3), put the result in the *tmp*(:,4), and return the control back to the routine dcgmrhs.

dcgmrhs_get

Retrieves the number of the current iteration.

Syntax

`dcgmrhs_get(n, x, b, RCI_request, ipar, dpar, tmp, itercount)`

Input Parameters

n	INTEGER. Contains the size of the problem, and size of arrays *x* and *b*.
x	DOUBLE PRECISION matrix of size *n* by *nrhs*. Contains the initial approximation to the solution vectors.
nrhs	INTEGER. This parameter sets the number of right-hand sides.
b	DOUBLE PRECISION matrix of size *(nrhs,n)*. Contains the right-hand side .
RCI_request	INTEGER. This parameter is not used.
ipar	INTEGER array of size *(128+2*nrhs)*. Refer to the CG Common Parameters.
dpar	DOUBLE PRECISION array of size *(128+2*nrhs)*. Refer to the CG Common Parameters.
tmp	DOUBLE PRECISION array of size *(n, 3+nrhs)*. Refer to the CG Common Parameters.

Output Parameters

itercount	INTEGER argument. Contains the value of the current iteration number.

Description

The routine dcgmrhs_get is called to retrieve the current iteration number of the solutions process.

NOTE. To use this routine with the name dcg_get, the user must switch on the compiler's preprocessor and include the files mkl_solver.h for C/C++, or mkl_solver.f77 for FORTRAN.

Return Values

The routine dcgmrhs_get does not return any value.

dfgmres_init

Initializes the solver.

Syntax

dfgmres_init(*n*, *x*, *b*, *RCI_request*, *ipar*, *dpar*, *tmp*)

Input Parameters

n	INTEGER. Contains the size of the problem, and size of arrays *x* and *b*.
x	DOUBLE PRECISION array of size *n*. Contains the initial approximation to the solution vector. Normally it is equal to 0 or to *b*.
b	DOUBLE PRECISION array of size *n*. Contains the right-hand side vector.

Output Parameters

RCI_request	INTEGER. Informs about the task completion.
ipar	INTEGER array of size 128. Refer to the FGMRES Common Parameters.
dpar	DOUBLE PRECISION array of size 128. Refer to the FGMRES Common Parameters.
tmp	DOUBLE PRECISION array of size ((2**ipar*(15)+1)**n*+*ipar*(15)**ipar*(15)+9)/2 + 1). Refer to the FGMRES Common Parameters.

Description

The routine dfgmres_init is called to initialize the solver. After initialization all subsequent invocations of Intel MKL RCI FGMRES routines can use the values of all parameters that are returned by dfgmres_init. Advanced users can skip this step and set the values to these parameters directly in the corresponding routines.

WARNING. Users can modify the contents of these arrays after they are passed to the solver routine only if they are sure that the values are correct and consistent. Basic check for correctness and consistency can be done by calling the `dfgmres_check` routine, but it does not guarantee that the method will work correctly.

Return Values

RCI_request= 0	The routine completed task normally.
RCI_request= -10000	The routine failed to complete the task.

dfgmres_check

Checks the consistency and correctness of the user defined data.

Syntax

`dfgmres_check(n, x, b, RCI_request, ipar, dpar, tmp)`

Input Parameters

n	INTEGER. Contains the size of the problem, and size of arrays *x* and *b*.
x	DOUBLE PRECISION array of size *n*. Contains the initial approximation to the solution vector. Normally it is equal to 0 or to *b*.
b	DOUBLE PRECISION array of size *n*. Contains the right-hand side vector.

Output Parameters

RCI_request	INTEGER. Informs about the task completion.
ipar	INTEGER array of size *128*. Refer to the FGMRES Common Parameters.
dpar	DOUBLE PRECISION array of size *128*. Refer to the FGMRES Common Parameters.

tmp	DOUBLE PRECISION array of size ((2*ipar(15)+1)*n+ipar(15)*ipar(15)+9)/2 + 1). Refer to the FGMRES Common Parameters.

Description

The routine dfgmres_check checks the consistency and correctness of the parameters to be passed to the solver routine dfgmres. However this operation does not guarantee that the method will be able to produce the correct result. It only reduces the chance to make a mistake in the parameters of the method. Advanced users can skip it if they are sure that the correct data is specified in the solver parameters.

WARNING. Users can modify the contents of these arrays after they are passed to the solver routine only if they are sure that the values are correct and consistent. Basic check for correctness and consistency can be done by calling the dfgmres_check routine, but it does not guarantee that the method will work correctly.

Note that the lengths of all vectors are assumed to have been defined in a previous call to dfgmres_init subroutine.

Return Values

RCI_request= 0	The routine completed task normally.
RCI_request= -1100	The routine is interrupted, errors occur.
RCI_request= -1001	The routine returns some warning messages.
RCI_request= -1010	The routine changed some parameters to make them consistent or correct.
RCI_request= -1011	The routine returns some warning messages and changed some parameters.

dfgmres

Makes the FGMRES iterations.

Syntax

dfgmres(*n*, *x*, *b*, *RCI_request*, *ipar*, *dpar*, *tmp*)

Input Parameters

n	INTEGER. Contains the size of the problem, and size of arrays *x* and *b*.
x	DOUBLE PRECISION array of size *n*. Contains the initial approximation to the solution vector.
b	DOUBLE PRECISION array of size *n*. Contains the right-hand side vector.
tmp	DOUBLE PRECISION array of size ((2*ipar(15)+1)*n+ipar(15)*ipar(15)+9)/2 + 1). Refer to the FGMRES Common Parameters.

Output Parameters

RCI_request	INTEGER. Informs about the task completion status.
ipar	INTEGER array of size *128*. Refer to the FGMRES Common Parameters.
dpar	DOUBLE PRECISION array of size *128*. Refer to the FGMRES Common Parameters.
tmp	DOUBLE PRECISION array of size ((2*ipar(15)+1)*n+ipar(15)*ipar(15)+9)/2 + 1). Refer to the FGMRES Common Parameters.

Description

The routine dfgmres performs the FGMRES iterations [Saad03]. The value that was in the vector *x* before the first call, the routine dfgmres uses as an initial approximation to the solution. To update the current approximation to the solution, the user should call dfgmres_get routine. The RCI FGMRES iterations can be continued after the call to the dfgmres_get routine only if the value of the parameter *ipar*(13) is not equal to 0 (default value). Note that the updated solution will overwrite the right hand side in the vector *b* if the parameter *ipar*(13) is positive, and it will be impossible to run the restarted version of FGMRES method. If the user wants to keep the right hand side, it should be saved in a different memory location before the first call to dfgmres_get routine with positive *ipar*(13).

The parameter *RCI_request* inform the user about task completion status and ask for results of certain operations that are required to the solver.

Note that the lengths of all vectors are assumed to have been defined in a previous call to the dfgmres_init routine.

Return Values

RCI_request= 0	The routine completed task normally, the solution is found. This occurs only if the stopping tests are fully automatic. For the user defined stopping tests, see the comments to the *RCI_request*= 2 or 4.
RCI_request= -1	The routine is interrupted because the maximal number of iterations is reached, but the relative stopping criterion is not satisfied (this occurs only if both tests are requested by the user).
RCI_request= -10	The routine is interrupted because the attempt to divide by zero occurs. Normally it happens if the matrix is (almost) degenerate. However it may happen if the parameter *dpar* is altered by mistale, or if the method is not stopped when the solution is found.
RCI_request= -11	The routine is interrupted because it enters the infinite cycle. (Probably, the data in *ipar*(8), *ipar*(9), *ipar*(10) were altered outside of the routine, or the dfgmres_check routine was not called).
RCI_request= -12	The routine is interrupted because the some errors are found in the method parameters. Normally this happens if the parameters *ipar* and *dpar* are altered by mistake outside the routine.
RCI_request= 1	Asks user to multiply the matrix by *tmp*(*ipar*(22)), put the result in the *tmp*(*ipar*(23)), and return the control back to the routine dfgmres.
RCI_request= 2	Asks user to perform the stopping test(s). If they fail, the user should return the control back to the dfgmres routine. Otherwise, the FGMRES solution is found, and the user should call the dfgmres_get routine and update the computed solution to the the vector *x*.
RCI_request= 3	Asks user to apply the inverse preconditioner to *ipar*(22), put the result in the *ipar*(23), and return the control back to the routine dfgmres.

RCI_request= 4	Asks user to check the norm of the currently generated vector. If it is not zero up to computational/rounding errors, the user should return the control back to the dfgmres routine. Otherwise, the FGMRES solution is found, and the user should call the dfgmres_get routine and update the computed solution to the the vector *x*.

dfgmres_get

Retrieves the number of the current iteration and updates the solution.

Syntax

dfgmres_get(*n*, *x*, *b*, *RCI_request*, *ipar*, *dpar*, *tmp*, *itercount*)

Input Parameters

n	INTEGER. Contains the size of the problem, and size of arrays *x* and *b*.
ipar	INTEGER array of size *128*. Refer to the FGMRES Common Parameters.
dpar	DOUBLE PRECISION array of size *128*. Refer to the FGMRES Common Parameters.
tmp	DOUBLE PRECISION array of size ((2**ipar*(15)+1)**n*+*ipar*(15)**ipar*(15)+9)/2 + 1). Refer to the FGMRES Common Parameters.

Output Parameters

x	DOUBLE PRECISION array of size *n*. If *ipar*(13)= 0, it contains the updated approximation to the solution according to the computations done in dfgmres routine. Otherwise, it is not changed.
b	DOUBLE PRECISION array of size *n*. If *ipar*(13)> 0, it contains the updated approximation to the solution according to the computations done in dfgmres routine. Otherwise, it is not changed.

RCI_request	INTEGER. Informs about the task completion.
itercount	INTEGER argument. Contains the value of the current iteration number.

Description

The routine dfgmres_get is called to retrieve the current iteration number of the solutions process and to update the solution according to the computations performed by the dfgmres routine. To retrieve the current iteration number only, set the parameter *ipar*(13) = -1 beforehand. This is normally recommended to do to proceed further with the computations. If the intermediate solution is needed, the method parameters should be set properly, see for details FGMRES Common Parameters and Iterative Sparse Solver Code Examples section in the Appendix C.

Return Values

RCI_request = 0	The routine completed task normally.
RCI_request = -12	The routine is interrupted because the some errors are found in the method parameters. Normally this happens if some of the parameters *ipar* and *dpar* are altered by mistake outside the routine.
RCI_request = -10000	The routine failed to complete the task.

Implementation Details

Several aspects of the Intel MKL RCI ISS interface are platform-specific and language-specific. In order to promote portability across platforms and ease of use across different languages, users are encouraged to include one of the Intel MKL RCI ISS language-specific header files. Currently, there is one language specific header file for C programs.

These language-specific header file defines function prototypes and they are the following:

```
void dcg_init(int *n, double *x, double *b, int *rci_request, int *ipar,
double dpar, double *tmp);
```

```
void dcg_check(int *n, double *x, double *b, int *rci_request, int *ipar,
double dpar, double *tmp);
```

```
void dcg(int *n, double *x, double *b, int *rci_request, int *ipar, double
dpar, double *tmp);
```

```
void dcg_get(int *n, double *x, double *b, int *rci_request, int *ipar,
double dpar, double *tmp, int *itercount);
```

```
void dcgmrhs_init(int *n, double *x, int *nRhs, double *b, int *method, int
*rci_request, int *ipar, double dpar, double *tmp);
void dcgmrhs_check(int *n, double *x, int *nRhs, double *b, int *rci_request,
int *ipar, double dpar, double *tmp);
void dcgmrhs(int *n, double *x, int *nRhs, double *b, int *rci_request, int
*ipar, double dpar, double *tmp);
void dcgmrhs_get(int *n, double *x, int *nRhs, double *b, int *rci_request,
int *ipar, double dpar, double *tmp, int *itercount);
void dfgmres_init(int *n, double *x, double *b, int *rci_request, int *ipar,
double dpar, double *tmp);
void dfgmres_check(int *n, double *x, double *b, int *rci_request, int *ipar,
double dpar, double *tmp);
void dfgmres(int *n, double *x, double *b, int *rci_request, int *ipar,
double dpar, double *tmp);
void dfgmres_get(int *n, double *x, double *b, int *rci_request, int *ipar,
double dpar, double *tmp, int *itercount);
```

NOTE. Use of the Intel MKL RCI ISS software without including the language specific header file is not supported.

Preconditioners or Accelerators based on Incomplete LU Factorization Technique

Usually, preconditioners or accelerators are used to accelerate an iterative solution process. In some cases, their use can reduce dramatically the number of iterations and thus lead to better solver performance. Although the terms ‘preconditioner’ and ‘accelerator’ are synonyms, hereafter only term ‘preconditioner’ will be used.

Currently, Intel MKL provides one preconditioner for PARDISO CSR matrix format (see Sparse Matrix Storage Format section). Full MKL compressed sparse row (CSR) format is not supported. The preconditioner is ILU0 preconditioner. It is based on a well-known factorization of the original matrix into a product of two triangular matrices: low triangular and upper triangular matrices. Usually, such a decomposition leads to some fill-in in the resulting matrix structure as compared to the original matrix. The distinctive feature of the ILU0 preconditioner is that it preserves the structure of the original matrix in the result.

The used algorithm is described in [Saad03]. The ILU0 preconditioner can be applied to any non-degenerate matrix. It can be used alone or together with the MKL RCI FGMRES solver (see Sparse Solver Routines). It is not recommended to use the preconditioner with MKL RCI CG solver because in general, it gives non-symmetric resulting matrix even if the original matrix is a symmetric one. Usually, an inverse of the preconditioner is required when it is used. For this purpose, for the ILU0 preconditioner a user should apply the Intel MKL triangular solver routine `mkl_dcsrtrsv` twice: for the low triangular part of the preconditioner, and then for its upper triangular part.

NOTE. Although ILU0 preconditioner can be applied to any non-degenerate matrix, in some cases the algorithm does not guarantee that it will terminate successfully and produce the required result. The result of ILU0 routine can be checked in practice only.

Preconditioner may increase the number of iterations for an arbitrary case of the system and initial guess and even ruin the convergence. It is user's responsibility to carefully use a suitable preconditioner.

General Scheme of Using ILU0 and RCI FGMRES Routines

The following pseudo code shows the general scheme of using the ILU0 preconditioner in the RCI FGMRES context.

```
...
generate matrix A
generate preconditioner C (optional)
  call dfgmres_init(n, x, b, RCI_request, ipar, dpar, tmp)
   change parameters in ipar, dpar if necessary
   call dcsrilu0(n, a, ia, ja, bilu0, ipar, dpar, ierr)
   call dfgmres_check(n, x, b, RCI_request, ipar, dpar, tmp)
1  call dfgmres(n, x, b, RCI_request, ipar, dpar, tmp)
   if (RCI_request.eq.1) then
    multiply the matrix A by tmp(ipar(22)) and put the result in tmp(ipar(23))
c  proceed with FGMRES iterations
    goto 1
   endif
```

```
      if (RCI_request.eq.2) then
       do the stopping test
       if (test not passed) then
c proceed with FGMRES iterations
        go to 1
       else
c  stop FGMRES iterations.
        goto 2
       endif
      endif
      if (RCI_request.eq.3) then
c  Below, trvec is an intermediate vector of length at least n
c  Here is the recommended use of the result produced by the ILU0 routine.
c  via standard Intel MKL Sparse Blas solver routine mkl_dcsrtrsv.
      call mkl_dcsrtrsv('L','N','U',n,bilu0,ia,ja,tmp(ipar(22)),trvec)
      call mkl_dcsrtrsv('U','N','N',n,bilu0,ia,ja,trvec,tmp(ipar(23)))
c proceed with FGMRES iterations
       goto 1
      endif
      if (RCI_request.eq.4) then
      check the norm of the next orthogonal vector, it is contained in dpar(7)
       if (the norm is not zero up to rounding/computational errors) then
c  proceed with FGMRES iterations
        goto 1
       else
c  stop FGMRES iterations
        goto 2
```

```
    endif
  endif
```

2 call `dfgmres_get(n, x, b, RCI_request, ipar, dpar, tmp, itercount)`

current iteration number is in *itercount*

the computed approximation is in the array *x*

You can find example codes that use RCI ISS interface routines to solve systems of linear equations in the Iterative Sparse Solver Code Example section in the Appendix C.

ILU0 Preconditioner Interface Description

The terms and concepts required to understand the use of the Intel MKL preconritioner routine are discussed in the Linear Solvers Basics. In this manual, we use FORTRAN style notations.

All types in this documentation refer to the standard Fortran types, `INTEGER`, and `DOUBLE PRECISION`.

C and C++ programmers should refer to the section Calling Sparse Solver Routines From C/C++ for information on mapping Fortran types to C/C++ types.

User Data Arrays

The ILU0 routine takes arrays of user data as input, for example, user arrays representing the original matrix. In order to minimize storage requirements and improve overall run-time efficiency, the Intel MKL ILU0 routine does not make copies of the user input arrays.

ILU0 Parameters

The preconditioner has parameters for passing various options to the routine. The values for these parameters should be specified very carefully, and they can be changed during computations according to the user's needs.

Some parameters are common with the FGMRES Common Parameters. Their default and initial values are specified by the routine `dfgmres_init` only. However, user can redefine these parameters by his own values. These parameters are listed below.

ipar`(2)` - specifies the type of output for error messages that are generated by the ILU0 routine. The default value 6 means that all messages are displayed on the screen. Otherwise the error messages are written to the newly created file MKL_PREC_log.txt. Note that if the parameter *ipar*`(6)` is set to 0, error messages are not generated at all.

ipar(6) - if its value is not equal to 0, the routine returns error messages in accordance with the parameter *ipar*(2). Otherwise, the routine does not generate error messages at all, but returns a negative value of the parameter *ierr*. The default value is 1.

NOTE. Users must provide correct and consistent parameters to the routine to avoid fails or wrong results.

dcsrilu0

ILU0 preconditioner based on incomplete LU factorization of a sparse matrix in the CSR format (PARDISO variation).

Syntax

Fortran:

```
call dcsrilu0(n, a, ia, ja, bilu0, ipar, dpar, ierr)
```

C:

```
dcsrilu0(&n, a, ia, ja, bilu0, ipar, dpar, &ierr);
```

Input Parameters

n	INTEGER. Size (number of rows or columns) of the original square *n*-by-*n*-matrix *A*.
a	DOUBLE PRECISION. Array containing the set of elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to values array description in the Sparse Matrix Storage Format section for more details.
ia	INTEGER. Array of size (*n*+1) containing begin indices of rows of matrix *A* such that *ia*(I) is the index in the array *A* of the first non-zero element from the row I. The value of the last element *ia*(*n*+1) is equal to the number of non-zeros in the matrix *A* plus one. Refer to the *rowIndex* array description in the Sparse Matrix Storage Format section for more details.

ja INTEGER. Array containing the column indices for each non-zero element of the matrix *A*. Its size is equal to the size of the array *a*. Refer to the *columns* array description in the Sparse Matrix Storage Format section for more details.

NOTE. Column indices should be put in increasing order for each row of matrix.

ipar INTEGER array of size *128*. This parameter is used to specify the integer set of data for both the ILU0 and RCI FGMRES computations. Refer to the *ipar* array description in the FGMRES Common Parameters for more details on FGMRES parameter entries. The entries that are specific to ILU0 are listed below.

ipar(31) - specifies how the routine operates if zero diagonal element occurs during calculation. If this parameter is set to 0 (default value set by the routine dfgmres_init) then that the calculations are stopped and the routine returns non-zero error value. Otherwise the value of the diagonal element is set to the specified value and the calculations are continued.

NOTE. Advanced users can define this array in the code as follows: INTEGER *ipar*(31). However, to guarantee the compatibility with the future releases of the Intel MKL it is highly recommended to declare the array *ipar* of length 128.

dpar DOUBLE PRECISION array of size *128*. This parameter is used to specify the double precision set of data for both the ILU0 and RCI FGMRES computations. Refer to the *dpar* array description in the FGMRES Common Parameters for more details on FGMRES parameter entries. The entries that are specific to ILU0 are listed below.

dpar(31) - specifies the small value that is compared with the diagonal elements during calculations; if the value of the diagonal element is smaller, then it is set to *dpar*(32), or the calculations are stopped, in accordance with *ipar*(31); the default value is 1.0D-16

NOTE. This parameter can be set to the negative value, because its absolute value is actually used in calculations.

If this parameter is set to 0, the comparison with the diagonal element is not performed.

dpar(32) - specifies the value that is assigned to the diagonal element if its value is less than *dpar*(31) (see above); the default value is 1.0D-10

NOTE. Advanced users can define this array in the code as follows: DOUBLE PRECOSION*dpar*(32). However, to guarantee the compatibility with the future releases of the Intel MKL it is highly recommended to declare the array *dpar* of length 128.

Output Parameters

bilu0	DOUBLE PRECISION. Array *B* stored in the PARDISO CSR format containing non-zero elements of the resulting preconditioning matrix. Its size is equal to the number of non-zero elements in the matrix *A*. Refer to the *values* array description in the Sparse Matrix Storage Format section for more details.
ierr	INTEGER. Error flag, informs about the routine completion status.

NOTE. To present the resulting preconditioning matrix in the CSR format the arrays *ia* (row indices) and *ja* (column indices) of the input matrix must be used.

Description

The routine dcsrilu0 computes a preconditioner *B* [Saad03] of a given sparse matrix *A* stored in the CSR format accepted in PARDISO:

*A~B=L*U* , where *L* is a low triangular matrix with unit diagonal, *U* is an upper triangular matrix with non-unit diagonal, and the portrait of the original matrix *A* is used to store the incomplete factors *L* and *U*.

Return Values

ierr=0	The routine completed task normally.
ierr=-101	The routine is interrupted, the error occurs: at least one diagonal element is omitted from the matrix in CSR format (see Sparse Matrix Storage Format).
ierr=-102	The routine is interrupted because the matrix contains zero diagonal element, routine can not perform operations.
ierr=-103	The routine is interrupted as the matrix contains too small diagonal element, and an overflow may occur because of the division by its value required to complete the task, or a bad approximation to ILU0 with use of this element will be computed.

ierr=-104	The routine is interrupted because the memory is insufficient for the internal work array.
ierr=-105	The routine is interrupted because the input matrix size *n* isless than or equal to 0.
ierr=-106	The routine is interrupted because the the column indices *ja* are placed in not increasing order.

Interfaces

Fortran 77 and Fortran 95:

```
SUBROUTINE dcsrilu0 (n, a, ia, ja, bilu0, ipar, dpar, ierr)
INTEGER n, ierr, ipar(128)
INTEGER ia(*), ja(*)
  DOUBLE PRECISION a(*), bilu0(*), dpar(128)
```

C:

```
void dcsrilu0 (int *n, double *a, int *ia, int *ja, double  *bilu0, int
*ipar, double *dpar, int *ierr);
```

Calling Sparse Solver Routines From C/C++

The calling interface for all of the Intel MKL sparse solver routines is designed to be used easily from Fortran 77 or Fortran 90. However, any of these routines can be invoked directly from C or C++ if users are familiar with the inter-language calling conventions of their platforms. These conventions include, but are not limited to, the argument passing mechanisms for the language, the data type mappings from Fortran to C/C++ and how Fortran external names are decorated on the platform.

In order to promote portability and to avoid having most users deal with these issues, the C header files provide a set of macros and type definitions that are intended to hide the inter-language calling conventions and provide an interface to the Intel MKL sparse solver routines that appears natural for C/C++.

For example, consider a hypothetical library routine, `foo`, that takes real vector of length *n*, and returns an integer status. Fortran users would access such a function as:

```
INTEGER n, status, foo

REAL x(*)

status = foo(x, n)
```

As noted above, for C users to invoke `foo`, they would need to know what C data types correspond to Fortran types `INTEGER` and `REAL`; what argument passing mechanism the Fortran compiler uses; and what, if any, name decoration the is performed by the Fortran compiler when generating the external symbol `foo`.

However, by using the C specific header file, for example mkl_solver.h, the invocation of `foo`, within a C program would look like:

```
#include "mkl_solver.h"

_INTEGER_t i, status;

_REAL_t x[];

status = foo( x, i );
```

Note that in the above example, the header file mkl_solver.h provides definitions for the types `_INTEGER_t` and `_REAL_t` that correspond to the Fortran types `INTEGER` and `REAL`.

In order to ease the use of Intel MKL sparse solver routines from C and C++, the general approach of providing C definitions of Fortran types is used throughout the library. Specifically, if an argument or result from a sparse solver is documented as having the Fortran language specific type XXX, then the C and C++ header files provide an appropriate C language type definitions for the name _XXX_t.

Caveat for C Users

One of the key differences between C/C++ and Fortran is the argument passing mechanisms for the languages: Fortran programs use pass-by-reference semantics and C/C++ programs use pass-by-value semantics. In the example in the previous section, the header file, mkl_solver.h, attempts to hide this difference, by defining a macro, `foo` that takes the address of the appropriate arguments. For example, on Tru64 UNIX, mkl_solver.h would define the macro as:

```
#define foo(a,b) foo_((a), &(b))
```

An important point to note when using the macro form of `foo` is how it deals with constants. If we write `foo( x, 10 )`, this is translated into `foo_( x, &10 )`. In a strictly ANSI compliant C compiler, it is not permissible to take the address of a constant, so a strictly conforming program would look like:

```
INTEGER_t iTen = 10;

_REAL_t * x;

status = foo( x, iTen );
```

However, some C compilers in a non-ANSI standard mode allow taking the address of a constant for ease of use with Fortran programs. Thus, the form shown as `foo( x, 10 )` is acceptable for these compilers.

9 Vector Mathematical Functions

This chapter describes Vector Mathematical Functions Library (VML), which is designed to compute mathematical functions on vector arguments. VML is an integral part of the Intel® MKL Kernel Library and the VML terminology is used here for simplicity in discussing this group of functions.

VML includes a set of highly optimized implementations of certain computationally expensive core mathematical functions (power, trigonometric, exponential, hyperbolic etc.) that operate on vectors of real and complex numbers.

Application programs that might significantly improve performance with VML include nonlinear programming software, integrals computation, and many others.

VML functions are divided into the following groups according to the operations they perform:

- VML Mathematical Functions compute values of mathematical functions (such as sine, cosine, exponential, logarithm and so on) on vectors with unit increment indexing.
- VML Pack/Unpack Functions convert to and from vectors with positive increment indexing, vector indexing and mask indexing (see Appendix B for details on vector indexing methods).
- VML Service Functions allow the user to set /get the accuracy mode, and set/get the error code.

VML mathematical functions take an input vector as argument, compute values of the respective function element-wise, and return the results in an output vector.

Data Types and Accuracy Modes

Mathematical and pack/unpack vector functions in VML have been implemented for vector arguments of single and double precision real data. Both Fortran- and C-interfaces to all functions, including VML service functions, are provided in the library. The differences in naming and calling the functions for Fortran- and C-interfaces are detailed in the Function Naming Conventions section below.

Each vector function from VML (for each data format) can work in two modes: High Accuracy (HA) and Low Accuracy (LA). For many functions, using the LA version will improve performance at the cost of accuracy.

For some cases, the advantage of relaxing the accuracy improves performance very little so the same function is employed for both versions. Error behavior depends not only on whether the HA or LA version is chosen, but also depends on the processor on which the software runs.

In addition, special value behavior may differ between the HA and LA versions of the functions. Any information on accuracy behavior can be found in the Intel MKL Release Notes.

Switching between the two modes (HA and LA) is accomplished by using `vmlSetMode(mode)` (see Table 9-12). The function `vmlGetMode()` will return the currently used mode. The High Accuracy mode is used by default.

Function Naming Conventions

Full names of all VML functions include only lowercase letters for Fortran-interface, whereas for C-interface names the lowercase letters are mixed with uppercase.

VML mathematical and pack/unpack function full names have the following structure:

v<*p*><name><*mod*>

The initial letter *v* is a prefix indicating that a function belongs to VML.

The <*p*> field is a precision prefix that indicates the data type:

s	`REAL` for Fortran-interface, or `float` for C-interface
d	`DOUBLE PRECISION` for Fortran-interface, or `double` for C-interface.
c	`COMPLEX` for Fortran-interface, or `MKL_Complex8` for C-interface.
z	`DOUBLE COMPLEX` for Fortran-interface, or `MKL_Complex16` for C-interface.

The <name> field indicates the function short name, with some of its letters in uppercase for C-interface (see for example Table 9-2 or Table 9-11).

The <*mod*> field (written in uppercase for C-interface) is present in pack/unpack functions only; it indicates the indexing method used:

i	indexing with positive increment
v	indexing with index vector
m	indexing with mask vector.

VML service function full names have the following structure:

vml<name>

where *vml* is a prefix indicating that a function belongs to VML, and <name> is the function short name, which includes some uppercase letters for C-ifglonterface (see Table 9-11). To call VML functions from an application program, use conventional function calls. For example, the VML exponential function for single precision real data can be called as

`call vsexp (n, a, y )` for Fortran-interface, or

`vsExp ( n, a, y);` for C-interface.

Functions Interface

The interface to VML functions includes function full names and the arguments list. The Fortran- and C-interface descriptions for different groups of VML functions are given below. Note that some functions (`Div`, `Pow`, and `Atan2`) have two input vectors *a* and *b* as their arguments, while `SinCos` function has two output vectors *y* and *z*.

VML Mathematical Functions

Fortran:

```
call v<p><name>( n, a, y )
call v<p><name>( n, a, b, y )
call v<p><name>( n, a, y, z )
```

C:

```
v<p><name>( n, a, y );
v<p><name>( n, a, b, y );
v<p><name>( n, a, y, z );
```

Pack Functions

Fortran:

```
call v<p>packi( n, a, inca, y )
call v<p>packv( n, a, ia, y )
call v<p>packm( n, a, ma, y )
```

C:

```
v<p>PackI( n, a, inca, y );
v<p>PackV( n, a, ia, y );
v<p>PackM( n, a, ma, y );
```

Unpack Functions

Fortran:

```
call v<p>unpacki( n, a, y, incy )
```

```
call v<p>unpackv( n, a, y, iy )
call v<p>unpackm( n, a, y, my )
```

C:

```
v<p>UnpackI( n, a, y, incy );
v<p>UnpackV( n, a, y, iy );
v<p>UnpackM( n, a, y, my );
```

Service Functions

Fortran:

```
oldmode = vmlsetmode( mode )
mode = vmlgetmode( )
olderr = vmlseterrstatus ( err )
err = vmlgeterrstatus( )
olderr = vmlclearerrstatus( )
oldcallback = vmlseterrorcallback( callback )
callback = vmlgeterrorcallback( )
oldcallback = vmlclearerrorcallback( )
```

C:

```
oldmode = vmlSetMode( mode );
mode = vmlGetMode( void );
olderr = vmlSetErrStatus ( err );
err = vmlGetErrStatus( void );
olderr = vmlClearErrStatus( void );
oldcallback = vmlSetErrorCallBack( callback );
callback = vmlGetErrorCallBack( void );
oldcallback = vmlClearErrorCallBack( void );
```

Input Parameters

n	number of elements to be calculated
a	first input vector
b	second input vector

inca	vector increment for the input vector *a*
ia	index vector for the input vector *a*
ma	mask vector for the input vector *a*
incy	vector increment for the output vector *y*
iy	index vector for the output vector *y*
my	mask vector for the output vector *y*
err	error code
mode	VML mode
callback	address of the callback function

Output Parameters

y	first output vector
z	second output vector
err	error code
mode	VML mode
olderr	former error code
oldmode	former VML mode
callback	address of the callback function
oldcallback	address of the former callback function

The data types of the parameters used in each function are specified in the respective function description section. All VML mathematical functions can perform in-place operations, which means that the same vector can be used as both input and output parameter. This holds true for functions with two input vectors as well, in which case one of them may be overwritten with the output vector. For functions with two output vectors, one of them may coincide with the input vector. But partially overlapping input and output vectors could lead to unpredictable results.

Vector Indexing Methods

Current VML mathematical functions work only with unit increment. Arrays with other increments, or more complicated indexing, can be accommodated by gathering the elements into a contiguous vector and then scattering them after the computation is complete.

Three following indexing methods are used to gather/scatter the vector elements in VML:

- positive increment
- index vector
- mask vector.

The indexing method used in a particular function is indicated by the indexing modifier (see the description of the *<mod>* field in Function Naming Conventions). For more information on indexing methods see Vector Arguments in VML in Appendix B.

Error Diagnostics

The VML library has its own error handler. The only difference for C- and Fortran- interfaces is that the Intel MKL error reporting routine `XERBLA` can be called after the Fortran- interface VML function encounters an error, and this routine gets information on *VML_STATUS_BADSIZE* and *VML_STATUS_BADMEM* input errors (see Table 9-14).

The VML error handler has the following properties:

1. The Error Status (*vmlErrStatus*) global variable is set after each VML function call. The possible values of this variable are shown in the Table 9-14 .
2. Depending on the VML mode, the error handler function invokes:
   - *errno* variable setting. The possible values are shown in the Table 9-1 .
   - writing error text information to the *stderr* stream
   - raising the appropriate exception on error, if necessary
   - calling the additional error handler callback function.

Table 9-1 Set Values of the *errno* Variable

Value of *errno*	Description
0	No errors are detected.
EINVAL	The array dimension is not positive.
EACCES	NULL pointer is passed.
EDOM	At least one of array values is out of a range of definition.
ERANGE	At least one of array values caused a singularity, overflow or underflow.

VML Mathematical Functions

This section describes VML functions which compute values of mathematical functions on real and complex vector arguments with unit increment.

Each function group is introduced by its short name, a brief description of its purpose, and the calling sequence for each type of data both for Fortran- and C-interfaces, as well as a description of the input/output arguments.

For all VML mathematical functions, the input range of parameters is equal to the mathematical range of definition in the set of defined values for the respective data type. Several VML functions, specifically `Div`, `Exp`, `Sinh`, `Cosh`, and `Pow`, can result in an overflow. For these functions, the respective input threshold values that mark off the precision overflow are specified in the function description section. Note that in these specifications, *`FLT_MAX`* denotes the maximum number representable in single precision real data type, while *`DBL_MAX`* denotes the maximum number representable in double precision real data type.

Table 9-2 lists available mathematical functions and data types associated with them.

Table 9-2 VML Mathematical Functions

Type of Distribution	Data Types	Description
Power and Root Functions		
`Inv`	*s*, *d*	Inversion of the vector elements
`Div`	*s*, *d*, *c*, *z*	Divide elements of one vector by elements of second vector
`Sqrt`	*s*, *d*	Square root of vector elements
`InvSqrt`	*s*, *d*	Inverse square root of vector elements
`Cbrt`	*s*, *d*	Cube root of vector elements
`InvCbrt`	*s*, *d*	Inverse cube root of vector elements
`Pow`	*s*, *d*, *c*, *z*	Each vector element raised to the specified power
`Powx`	*s*, *d*, *c*, *z*	Each vector element raised to the constant power
`Hypot`	*s*, *d*	Square root of sum of squares
Exponential and Logarithmic Functions		
`Exp`	*s*, *d*, *c*, *z*	Exponential of vector elements
`Ln`	*s*, *d*, *c*, *z*	Natural logarithm of vector elements
`Log10`	*s*, *d*, *c*, *z*	Denary logarithm of vector elements
Trigonometric Functions		
`Cos`	*s*, *d*, *c*, *z*	Cosine of vector elements
`Sin`	*s*, *d*, *c*, *z*	Sine of vector elements
`SinCos`	*s*, *d*	Sine and cosine of vector elements
`Tan`	*s*, *d*, *c*, *z*	Tangent of vector elements

Type of Distribution	Data Types	Description
Acos	*s*, *d*, *c*, *z*	Inverse cosine of vector elements
Asin	*s*, *d*, *c*, *z*	Inverse sine of vector elements
Atan	*s*, *d*, *c*, *z*	Inverse tangent of vector elements
Atan2	*s*, *d*	Four-quadrant inverse tangent of elements of two vectors
Hyperbolic Functions		
Cosh	*s*, *d*, *c*, *z*	Hyperbolic cosine of vector elements
Sinh	*s*, *d*, *c*, *z*	Hyperbolic sine of vector elements
Tanh	*s*, *d*, *c*, *z*	Hyperbolic tangent of vector elements
Acosh	*s*, *d*, *c*, *z*	Inverse hyperbolic cosine (nonnegative) of vector elements
Asinh	*s*, *d*, *c*, *z*	Inverse hyperbolic sine of vector elements
Atanh	*s*, *d*, *c*, *z*	Computes inverse hyperbolic tangent of vector elements.
Special Functions		
Erf	*s*, *d*	Error function value of vector elements
Erfc	*s*, *d*	Complementary error function value of vector elements
ErfInv	*s*, *d*	Inverse error function value of vector elements
Rounding Functions		
Floor	*s*, *d*	Rounding towards minus infinity
Ceil	*s*, *d*	Rounding towards plus infinity
Trunc	*s*, *d*	Rounding towards zero infinity
Round	*s*, *d*	Rounding to nearest integer
NearbyInt	*s*, *d*	Rounding according to current mode
Rint	*s*, *d*	Rounding according to current mode and raising inexact result exception
Modf	*s*, *d*	Integer and fraction parts

Inv

Performs element by element inversion of the vector.

Syntax

Fortran:

```
call vsinv( n, a, y )
call vdinv( n, a, y )
```

C:

```
call vsInv( n, a, y );
call vdInv( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.
a	**FORTRAN:** REAL, INTENT(IN) for vsinv DOUBLE PRECISION, INTENT(IN) for vdinv **C:** const float* for vsInv const double* for vdInv	**FORTRAN:** Array, specifies the input vector *a*. **C:** Pointer to an array that contains the input vector *a*.

Output Parameters

Name	Type	Description
y	**FORTRAN:** REAL for vsinv DOUBLE PRECISION for vdinv **C:** float* for vsInv double* for vdInv	**FORTRAN:** Array, specifies the output vector *y*. **C:** Pointer to an array that contains the output vector *y*.

Div

Performs element by element division of vector `a` *by vector* `b`

Syntax

Fortran:

```
call vsdiv( n, a, b, y )
call vddiv( n, a, b, y )
```

C:

```
vsDiv( n, a, b, y );
vdDiv( n, a, b, y );
```

Input Parameters

Name	Type	Description
`n`	**FORTRAN:** `INTEGER, INTENT(IN)`. **C:** `int`.	Specifies the number of elements to be calculated.

`a, b`	**FORTRAN:** `REAL, INTENT(IN)` for `vsdiv`
	`DOUBLE PRECISION, INTENT(IN)` for `vddiv`
	C: `const float*` for `vsDiv`
	`const double*` for `vdDiv`
		FORTRAN: Arrays, specify the input vectors `a` and `b`.
		C: Pointers to arrays that contain the input vectors `a` and `b`.

Table 9-3 Precision Overflow Thresholds for `Div` Function

Data Type	Threshold Limitations on Input Parameters
single precision	`abs(a[i]) < abs(b[i]) * FLT_MAX`
double precision	`abs(a[i]) < abs(b[i]) * DBL_MAX`

Output Parameters

Name	Type	Description

y	**FORTRAN:** `REAL` for `vsdiv`
	`DOUBLE PRECISION` for `vddiv`
	C: `float*` for `vsDiv`
	`double*` for `vdDiv`
		FORTRAN: Array, specifies the output vector *y*.
		C: Pointer to an array that contains the output vector *y*.

Sqrt

Computes a square root of vector elements.

Syntax

Fortran:

```
call vssqrt( n, a, y )
call vdsqrt( n, a, y )
call vcsqrt( n, a, y )
call vzsqrt( n, a, y )
```

C:

```
call vsSqrt( n, a, y );
call vdSqrt( n, a, y );
call vcSqrt( n, a, y );
call vzSqrt( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** `INTEGER, INTENT(IN)`. **C:** `int`.	Specifies the number of elements to be calculated.

Name	Type	Description

a	**FORTRAN:** `REAL, INTENT(IN)` for `vssqrt`
	`DOUBLE PRECISION, INTENT(IN)` for `vdsqrt`
	`COMPLEX, INTENT(IN)` for `vcsqrt`
	`DOUBLE COMPLEX, INTENT(IN)` for `vzsqrt`
	C: `const float*` for `vsSqrt`
	`const double*` for `vdSqrt`
	`const MKL_Complex8*` for `vcSqrt`
	`const MKL_Complex16*` for `vzSqrt`
		FORTRAN: Array, specifies the input vector *a*.
		C: Pointer to an array that contains the input vector *a*.

Output Parameters

Name	Type	Description

y	**FORTRAN:** `REAL` for `vssqrt`
	`DOUBLE PRECISION` for `vdsqrt`
	`COMPLEX` for `vcsqrt`
	`DOUBLE COMPLEX` for `vzsqrt`
	C: `float*` for `vsSqrt`
	`double*` for `vdSqrt`
	`MKL_Complex8*` for `vcSqrt`
	`MKL_Complex16*` for `vzSqrt`
		FORTRAN: Array, specifies the output vector *y*.
		C: Pointer to an array that contains the output vector *y*.

InvSqrt

Computes an inverse square root of vector elements.

Syntax

Fortran:

```
call vsinvsqrt( n, a, y )
call vdinvsqrt( n, a, y )
```

C:

```
call vsInvSqrt( n, a, y );
call vdInvSqrt( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.
a	**FORTRAN:** REAL, INTENT(IN) for vsinvsqrt DOUBLE PRECISION, INTENT(IN) for vdinvsqrt **C:** const float* for vsInvSqrt const double* for vdInvSqrt	**FORTRAN:** Array, specifies the input vector *a*. **C:** Pointer to an array that contains the input vector *a*.

Output Parameters

Name	Type	Description
y	**FORTRAN:** REAL for vsinvsqrt DOUBLE PRECISION for vdinvsqrt	**FORTRAN:** Array, specifies the output vector *y*.

Name	Type	Description
	C: `float*` for `vsInvSqrt` `double*` for `vdInvSqrt`	**C:** Pointer to an array that contains the output vector *y*.

Cbrt

Computes a cube root of vector elements.

Syntax

Fortran:

```
call vscbrt( n, a, y )
call vdcbrt( n, a, y )
```

C:

```
call vsCbrt( n, a, y );
call vdCbrt( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** `INTEGER`, `INTENT(IN)`. **C:** `int`.	Specifies the number of elements to be calculated.

a	**FORTRAN:** `REAL`, `INTENT(IN)` for `vscbrt`
	`DOUBLE PRECISION`, `INTENT(IN)` for `vdcbrt`
	C: `const float*` for `vsCbrt`
	`const double*` for `vdCbrt`
		FORTRAN: Array, specifies the input vector *a*.
		C: Pointer to an array that contains the input vector *a*.

Output Parameters

Name	Type	Description
y	**FORTRAN:** REAL for vscbrt DOUBLE PRECISION for vdcbrt **C:** float* for vsCbrt double* for vdCbrt	**FORTRAN:** Array, specifies the output vector *y*. **C:** Pointer to an array that contains the output vector *y*.

InvCbrt

Computes an inverse cube root of vector elements.

Syntax

Fortran:

```
call vsinvcbrt( n, a, y )
call vdinvcbrt( n, a, y )
```

C:

```
call vsInvCbrt( n, a, y );
call vdInvCbrt( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). int.	Specifies the number of elements to be calculated.
a	**FORTRAN:** REAL, INTENT(IN) for vsinvcbrt DOUBLE PRECISION, INTENT(IN) for vdinvcbrt **C:** const float* for vsInvCbrt	**FORTRAN:** Array, specifies the input vector *a*. **C:** Pointer to an array that contains the input vector *a*.

Name	Type	Description
	`const double*` for `vdInvCbrt`	

Output Parameters

Name	Type	Description

y	**FORTRAN:** `REAL` for `vsinvcbrt`
	`DOUBLE PRECISION` for `vdinvcbrt`
	C: `float*` for `vsInvCbrt`
	`double*` for `vdInvCbrt`
		FORTRAN: Array, specifies the output vector *y*.
		C: Pointer to an array that contains the output vector *y*.

Pow

Computes `a` to the power `b` for elements of two vectors.

Syntax

Fortran:

```
call vspow( n, a, b, y )
call vdpow( n, a, b, y )
call vcpow( n, a, b, y )
call vzpow( n, a, b, y )
```

C:

```
vsPow( n, a, b, y );
vdPow( n, a, b, y );
vcPow( n, a, b, y );
vzPow( n, a, b, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.
a, *b*	**FORTRAN:** REAL, INTENT(IN) for vspow DOUBLE PRECISION, INTENT(IN) for vdpow COMPLEX, INTENT(IN) for vcpow DOUBLE COMPLEX, INTENT(IN) for vzpow **C:** const float* for vsPow const double* for vdPow **C:** const MKL_Complex8* for vcPow const MKL_Complex16* for vzPow	**FORTRAN:** Arrays, specify the input vectors *a* and *b*. **C:** Pointers to arrays that contain the input vectors *a* and *b*.

Table 9-4 Precision Overflow Thresholds for Pow Real Function

Data Type	Threshold Limitations on Input Parameters
single precision	$abs(a[i]) < (FLT\_MAX)^{1/b[i]}$
double precision	$abs(a[i]) < (DBL\_MAX)^{1/b[i]}$

NOTE. Overflow can occur also in Pow complex function, but the exact formula is beyond the scope of this document.

Output Parameters

Name	Type	Description
y	**FORTRAN:** REAL for vspow DOUBLE PRECISION for vdpow COMPLEX for vcpow DOUBLE COMPLEX for vzpow **C:** float* for vsPow double* for vdPow MKL_Complex8* for vcPow MKL_Complex16* for vzPow	**FORTRAN:** Array, specifies the output vector *y*. **C:** Pointer to an array that contains the output vector *y*.

Description

The real function *Pow* has certain limitations on the input range of *a* and *b* parameters. Specifically, if *a[i]* is positive, then *b[i]* may be arbitrary. For negative *a[i]*, the value of *b[i]* must be integer (either positive or negative).

The complex function Pow has no such input range limitations.

Powx

Raises each element of a vector to the constant power.

Syntax

Fortran:

```
call vspowx( n, a, b, y )
call vdpowx( n, a, b, y )
call vcpowx( n, a, b, y )
call vzpowx( n, a, b, y )
```

C:

```
vsPowx( n, a, b, y );
vdPowx( n, a, b, y );
vcPowx( n, a, b, y );
vzPowx( n, a, b, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Number of elements to be calculated.
a	**FORTRAN:** REAL, INTENT(IN) for vspowx DOUBLE PRECISION, INTENT(IN) for vdpowx COMPLEX, INTENT(IN) for vcpowx DOUBLE COMPLEX, INTENT(IN) for vzpowx **C:** const float* for vsPowx const double* for vdPowx const MKL_Complex8* for vcPowx const MKL_Complex16* for vzPowx	**FORTRAN:** Array *a* that specifies the input vector **C:** Pointer to an array that contains the input vector *a*.
b	**FORTRAN:** REAL, INTENT(IN) for vspowx DOUBLE PRECISION, INTENT(IN) for vdpowx COMPLEX, INTENT(IN) for vcpowx DOUBLE COMPLEX, INTENT(IN) for vzpowx	**FORTRAN:** Scalar value *b* that is the constant power. **C:** Constant value for power *b*.

Name	Type	Description

	C: `const float` for `vsPowx`
	`const double` for `vdPowx`
	`const MKL_Complex8*` for `vcPowx`
	`const MKL_Complex16*` for `vzPowx`	

Table 9-5 Precision Overflow Thresholds for `Powx` Real Function

Data Type	Threshold Limitations on Input Parameters
single precision	$\text{abs}(a[i]) < (\text{FLT\_MAX})^{1/b}$
double precision	$\text{abs}(a[i]) < (\text{DBL\_MAX})^{1/b}$

NOTE. Overflow can occur also in `Powx` complex function, but the exact formula is beyond the scope of this document.

Output Parameters

Name	Type	Description

y	**FORTRAN:** `REAL` for `vspowx`
	`DOUBLE PRECISION` for `vdpowx`
	`COMPLEX` for `vcpowx`
	`DOUBLE COMPLEX` for `vzpowx`
	C: `float*` for `vsPowx`
	`double*` for `vdPowx`
	`MKL_Complex8*` for `vcPowx`
	`MKL_Complex16*` for `vzPowx`
		FORTRAN: Array, specifies the output vector *y*.
		C: Pointer to an array that contains the output vector *y*.

Description

The real function Powx has certain limitations on the input range of *a* and *b* parameters. Specifically, if *a[i]* is positive, then *b* may be arbitrary. For negative *a[i]*, the value of *b* must be integer (either positive or negative).

The complex function Powx has no such input range limitations.

Hypot

Computes a square root of sum of two squared elements.

Syntax

Fortran:

```
call vshypot( n, a, b, y )
call vdhypot( n, a, b, y )
```

C:

```
vsHypot( n, a, b, y );
vdHypot( n, a, b, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Number of elements to be calculated.
a,*b*	**FORTRAN:** REAL, INTENT(IN) for vshypot DOUBLE PRECISION, INTENT(IN) for vdhypot **C:** const float* for vsHypot const double* for vdHypot	**FORTRAN:** Arrays that specify the input vectors *a* and *b* **C:** Pointers to arrays that contain the input vectors *a* and *b*.

Table 9-6 Precision Overflow Thresholds for `Hypot` Function

Data Type	Threshold Limitations on Input Parameters
single precision	`abs(a[i]) < sqrt(FLT_MAX)`
	`abs(b[i]) < sqrt(FLT_MAX)`
double precision	`abs(a[i]) < sqrt(DBL_MAX)`
	`abs(b[i]) < sqrt(DBL_MAX)`

Output Parameters

Name	Type	Description

y	**FORTRAN:** `REAL` for `vshypot`
	`DOUBLE PRECISION` for `vdhypot`
	C: `float*` for `vsHypot`
	`double*` for `vdHypot`
		FORTRAN: Array, specifies the output vector *y*.
		C: Pointer to an array that contains the output vector *y*.

Exp

Computes an exponential of vector elements.

Syntax

Fortran:

```
call vsexp( n, a, y )
call vdexp( n, a, y )
call vcexp( n, a, y )
call vzexp( n, a, y )
```

C:

```
call vsExp( n, a, y );
call vdExp( n, a, y );
call vcExp( n, a, y );
call vzExp( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.
a	**FORTRAN:** REAL, INTENT(IN) for vsexp DOUBLE PRECISION, INTENT(IN) for vdexp COMPLEX, INTENT(IN) for vcexp DOUBLE COMPLEX, INTENT(IN) for vzexp **C:** const float* for vsExp const double* for vdExp const MKL_Complex8* for vcExp const MKL_Complex16* for vzExp	**FORTRAN:** Array, specifies the input vector *a*. **C:** Pointer to an array that contains the input vector *a*.

Table 9-7 Precision Overflow Thresholds for Exp Real Function

Data Type	Threshold Limitations on Input Parameters
single precision	*a*[i] < Ln(FLT_MAX)
double precision	*a*[i] < Ln(DBL_MAX)

NOTE. Overflow can occur also in Exp complex function, but the exact formula is beyond the scope of this document.

Output Parameters

Name	Type	Description
y	**FORTRAN:** REAL for vsexp DOUBLE PRECISION for vdexp COMPLEX for vcexp DOUBLE COMPLEX for vzexp **C:** float* for vsExp double* for vdExp MKL_Complex8* for vcExp MKL_Complex16* for vzExp	**FORTRAN:** Array, specifies the output vector *y*. **C:** Pointer to an array that contains the output vector *y*.

Ln

Computes natural logarithm of vector elements.

Syntax

Fortran:

```
call vsln( n, a, y )
call vdln( n, a, y )
call vcln( n, a, y )
call vzln( n, a, y )
```

C:

```
vsLn( n, a, y );
vdLn( n, a, y );
vcLn( n, a, y );
vzLn( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.
a	**FORTRAN:** REAL, INTENT(IN) for vsln DOUBLE PRECISION, INTENT(IN) for vdln **FORTRAN:** COMPLEX, INTENT(IN) for vcln DOUBLE COMPLEX, INTENT(IN) for vzln **C:** const float* for vsLn const double* for vdLn const MKL_Complex8* for vcLn const MKL_Complex16* for vzLn	**FORTRAN:** Array, specifies the input vector *a*. **C:** Pointer to an array that contains the input vector *a*.

Output Parameters

Name	Type	Description
y	**FORTRAN:** REAL for vsln DOUBLE PRECISION for vdln COMPLEX for vcln DOUBLE COMPLEX for vzln **C:** float* for vsLn double* for vdLn MKL_Complex8* for vcLn MKL_Complex16* for vzLn	**FORTRAN:** Array, specifies the output vector *y*. **C:** Pointer to an array that contains the output vector *y*.

Log10

Computes denary logarithm of vector elements.

Syntax

Fortran:

```
call vslog10( n, a, y )
call vdlog10( n, a, y )
call vclog10( n, a, y )
call vzlog10( n, a, y )
```

C:

```
call vsLog10( n, a, y );
call vdLog10( n, a, y );
call vcLog10( n, a, y );
call vzLog10( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.
a	**FORTRAN:** REAL, INTENT(IN) for vslog10 DOUBLE PRECISION, INTENT(IN) for vdlog10 COMPLEX, INTENT(IN) for vclog10 DOUBLE COMPLEX, INTENT(IN) for vzlog10 **C:** const float* for vsLog10 const double* for vdLog10	**FORTRAN:** Array, specifies the input vector *a*. **C:** Pointer to an array that contains the input vector *a*.

Name	Type	Description
	`const MKL_Complex8*` for `vcLog10` `const MKL_Complex16*` for `vzLog10`	

Output Parameters

Name	Type	Description

y	**FORTRAN:** `REAL` for `vslog10`
	`DOUBLE PRECISION` for `vdlog10`
	`COMPLEX` for `vclog10`
	`DOUBLE COMPLEX` for `vzlog10`
	C: `float*` for `vsLog10`
	`double*` for `vdLog10`
	`MKL_Complex8*` for `vcLog10`
	`MKL_Complex16*` for `vzLog10`
		FORTRAN: Array, specifies the output vector *y*.
		C: Pointer to an array that contains the output vector *y*.

Cos

Computes cosine of vector elements.

Syntax

Fortran:

```
call vscos( n, a, y )
call vdcos( n, a, y )
call vccos( n, a, y )
call vzcos( n, a, y )
```

C:

```
call vsCos( n, a, y );
call vdCos( n, a, y );
call vcCos( n, a, y );
call vzCos( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.
a	**FORTRAN:** REAL, INTENT(IN) for vscos DOUBLE PRECISION, INTENT(IN) for vdcos COMPLEX, INTENT(IN) for vccos DOUBLE PRECISION, INTENT(IN) for vzcos **C:** const float* for vsCos const double* for vdCos **C:** const MKL_Complex8* for vcCos const MKL_Complex16* for vzCos	**FORTRAN:** Array, specifies the input vector *a*. **C:** Pointer to an array that contains the input vector *a*.

Output Parameters

Name	Type	Description
y	**FORTRAN:** REAL for vscos DOUBLE PRECISION for vdcos COMPLEX for vccos	**FORTRAN:** Array, specifies the output vector *y*. **C:** Pointer to an array that contains the output vector *y*.

Name	Type	Description

	`DOUBLE COMPLEX` for `vzcos`
	C: `float*` for `vsCos`
	`double*` for `vdCos`
	C: `MKL_Complex8*` for `vcCos`
	`MKL_Complex16*` for `vzCos`	

Sin

Computes sine of vector elements.

Syntax

Fortran:

```
call vssin( n, a, y )
call vdsin( n, a, y )
call vcsin( n, a, y )
call vzsin( n, a, y )
```

C:

```
call vsSin( n, a, y );
call vdSin( n, a, y );
call vcSin( n, a, y );
call vzSin( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** `INTEGER, INTENT(IN)`. **C:** `int`.	Specifies the number of elements to be calculated.

Name	Type	Description

a	**FORTRAN:** `REAL, INTENT(IN)` for `vssin`
	`DOUBLE PRECISION, INTENT(IN)` for `vdsin`
	`COMPLEX, INTENT(IN)` for `vcsin`
	`DOUBLE PRECISION, INTENT(IN)` for `vzsin`
	C: `const MKL_Complex8*` for `vcSin`
	`const MKL_Complex16*` for `vzSin`
		FORTRAN: Array, specifies the input vector *a*.
		C: Pointer to an array that contains the input vector *a*.

Output Parameters

Name	Type	Description

y	**FORTRAN:** `REAL` for `vssin`
	`DOUBLE PRECISION` for `vdsin`
	`COMPLEX` for `vcsin`
	`DOUBLE COMPLEX` for `vzsin`
	C: `float*` for `vsSin`
	`double*` for `vdSin`
	C: `MKL_Complex8*` for `vcSin`
	`MKL_Complex16*` for `vzSin`
		FORTRAN: Array, specifies the output vector *y*.
		C: Pointer to an array that contains the output vector *y*.

SinCos

Computes sine and cosine of vector elements.

Syntax

Fortran:

```
call vssincos( n, a, y, z )
call vdsincos( n, a, y, z )
```

C:

```
vsSinCos( n, a, y, z );
vdSinCos( n, a, y, z );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.
a	**FORTRAN:** REAL, INTENT(IN) for vssincos DOUBLE PRECISION, INTENT(IN) for vdsincos **C:** const float* for vsSinCos const double* for vdSinCos	**FORTRAN:** Array, specifies the input vector *a*. **C:** Pointer to an array that contains the input vector *a*.

Output Parameters

Name	Type	Description
y, *z*	**FORTRAN:** REAL for vssincos DOUBLE PRECISION for vdsincos **C:** float* for vsSinCos	**FORTRAN:** Arrays, specify the output vectors *y* (for sine values) and *z* (for cosine values).

Name	Type	Description
	`double*` for `vdSinCos`	**C:** Pointers to arrays that contain the output vectors *y* (for sinevalues) and *z*(for cosine values).

Tan

Computes tangent of vector elements.

Syntax

Fortran:

```
call vstan( n, a, y )
call vdtan( n, a, y )
call vctan( n, a, y )
call vztan( n, a, y )
```

C:

```
call vsTan( n, a, y );
call vdTan( n, a, y );
call vcTan( n, a, y );
call vzTan( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** `INTEGER, INTENT(IN)`. **C:** `int`.	Specifies the number of elements to be calculated.
a	**FORTRAN:** `REAL, INTENT(IN)` for `vstan` `DOUBLE PRECISION, INTENT(IN)` for `vdtan`	**FORTRAN:** Array, specifies the input vector *a*. **C:** Pointer to an array that contains the input vector *a*.

Name	Type	Description

	`COMPLEX, INTENT(IN)` for `vctan`
	`DOUBLE COMPLEX, INTENT(IN)` for `vztan`
	C: `const float*` for `vsTan`
	`const double*` for `vdTan`
	C: `const MKL_Complex8*` for `vcTan`
	`const MKL_Complex16*` for `vzTan`	

Output Parameters

Name	Type	Description

y	**FORTRAN:** `REAL` for `vstan`
	`DOUBLE PRECISION` for `vdtan`
	`COMPLEX` for `vctan`
	`DOUBLE COMPLEX` for `vztan`
	C: `float*` for `vsTan`
	`double*` for `vdTan`
	`MKL_Complex8*` for `vcTan`
	`MKL_Complex16*` for `vzTan`
		FORTRAN: Array, specifies the output vector *y*.
		C: Pointer to an array that contains the output vector *y*.

Acos

Computes inverse cosine of vector elements.

Syntax

Fortran:

```
call vsacos( n, a, y )
call vdacos( n, a, y )
call vcacos( n, a, y )
call vzacos( n, a, y )
```

C:

```
call vsAcos( n, a, y );
call vdAcos( n, a, y );
call vcAcos( n, a, y );
call vzAcos( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.
a	**FORTRAN:** REAL, INTENT(IN) for vsacos DOUBLE PRECISION, INTENT(IN) for vdacos COMPLEX, INTENT(IN) for vcacos DOUBLE COMPLEX, INTENT(IN) for vzacos **C:** const float* for vsAcos const double* for vdAcos	**FORTRAN:** Array, specifies the input vector *a*. **C:** Pointer to an array that contains the input vector *a*.

Name	Type	Description
	`const MKL_Complex8*` for `vcAcos` `const MKL_Complex16*` for `vzAcos`	

Output Parameters

Name	Type	Description

y	**FORTRAN:** `REAL` for `vsacos`
	`DOUBLE PRECISION` for `vdacos`
	`COMPLEX` for `vcacos`
	`DOUBLE COMPLEX` for `vzacos`
	C: `float*` for `vsAcos`
	`double*` for `vdAcos`
	`MKL_Complex8*` for `vcAcos`
	`MKL_Complex16*` for `vzAcos`
		FORTRAN: Array, specifies the output vector *y*.
		C: Pointer to an array that contains the output vector *y*.

Asin

Computes inverse sine of vector elements.

Syntax

Fortran:

```
call vsasin( n, a, y )
call vdasin( n, a, y )
call vcasin( n, a, y )
call vzasin( n, a, y )
```

C:

```
call vsAsin( n, a, y );
call vdAsin( n, a, y );
call vcAsin( n, a, y );
call vzAsin( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.
a	**FORTRAN:** REAL, INTENT(IN) for vsasin DOUBLE PRECISION, INTENT(IN) for vdasin COMPLEX, INTENT(IN) for vcasin DOUBLE COMPLEX, INTENT(IN) for vzasin **C:** const float* for vsAsin const double* for vdAsin const MKL_Complex8* for vcAsin const MKL_Complex16* for vzAsin	**FORTRAN:** Array, specifies the input vector *a*. **C:** Pointer to an array that contains the input vector *a*.

Output Parameters

Name	Type	Description
y	**FORTRAN:** REAL for vsasin DOUBLE PRECISION for vdasin COMPLEX for vcasin	**FORTRAN:** Array, specifies the output vector *y*. **C:** Pointer to an array that contains the output vector *y*.

Name	Type	Description

	`DOUBLE COMPLEX` for `vzasin`
	C: `float*` for `vsAsin`
	`double*` for `vdAsin`
	`MKL_Complex8*` for `vcAsin`
	`MKL_Complex16*` for `vzAsin`	

Atan

Computes inverse tangent of vector elements.

Syntax

Fortran:

```
call vsatan( n, a, y )
call vdatan( n, a, y )
call vcatan( n, a, y )
call vzatan( n, a, y )
```

C:

```
call vsAtan( n, a, y );
call vdAtan( n, a, y );
call vcAtan( n, a, y );
call vzAtan( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** `INTEGER, INTENT(IN)`. **C:** `int`.	Specifies the number of elements to be calculated.

Name	Type	Description

a	**FORTRAN:** `REAL, INTENT(IN)` for `vsatan`
	`DOUBLE PRECISION, INTENT(IN)` for `vdatan`
	`COMPLEX, INTENT(IN)` for `vcatan`
	`DOUBLE COMPLEX, INTENT(IN)` for `vzatan`
	C: `const float*` for `vsAtan`
	`const double*` for `vdAsin`
	`const MKL_Complex8*` for `vcAtan`
	`const MKL_Complex16*` for `vzAsin`
		FORTRAN: Array, specifies the input vector *a*.
		C: Pointer to an array that contains the input vector *a*.

Output Parameters

Name	Type	Description

y	**FORTRAN:** `REAL` for `vsatan`
	`DOUBLE PRECISION` for `vdatan`
	`COMPLEX` for `vcatan`
	`DOUBLE COMPLEX` for `vzatan`
	C: `float*` for `vsAtan`
	`double*` for `vdAtan`
	`MKL_Complex8*` for `vcAtan`
	`MKL_Complex16*` for `vzAtan`
		FORTRAN: Array, specifies the output vector *y*.
		C: Pointer to an array that contains the output vector *y*.

Atan2

Computes four-quadrant inverse tangent of elements of two vectors.

Syntax

Fortran:

```
call vsatan2( n, a, b, y )
call vdatan2( n, a, b, y )
```

C:

```
vsAtan2( n, a, b, y );
vdAtan2( n, a, b, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.
a, *b*	**FORTRAN:** REAL, INTENT(IN) for vsatan2 DOUBLE PRECISION, INTENT(IN) for vdatan2 **C:** const float* for vsAtan2 const double* for vdAtan2	**FORTRAN:** Arrays, specify the input vectors *a* and *b*. **C:** Pointers to arrays that contain the input vectors *a* and *b*.

Output Parameters

Name	Type	Description
y	**FORTRAN:** REAL for vsatan2 DOUBLE PRECISION for vdatan2	**FORTRAN:** Array, specifies the output vector *y*.

Name	Type	Description
	C: `float*` for `vsAtan2` `double*` for `vdAtan2`	**C:** Pointer to an array that contains the output vector *y*.

Description

The elements of the output vector *y* are computed as the four-quadrant arctangent of *a[i] /b[i]*.

Cosh

Computes hyperbolic cosine of vector elements.

Syntax

Fortran:

```
call vscosh( n, a, y )
call vdcosh( n, a, y )
call vccosh( n, a, y )
call vzcosh( n, a, y )
```

C:

```
call vsCosh( n, a, y );
call vdCosh( n, a, y );
call vcCosh( n, a, y );
call vzCosh( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** `INTEGER`, `INTENT(IN)`. **C:** `int`.	Specifies the number of elements to be calculated.

Name	Type	Description

a	**FORTRAN:** `REAL, INTENT(IN)` for `vscosh`
	`DOUBLE PRECISION, INTENT(IN)` for `vdcosh`
	`COMPLEX, INTENT(IN)` for `vccosh`
	`DOUBLE COMPLEX, INTENT(IN)` for `vzcosh`
	C: `const float*` for `vsCosh`
	`const double*` for `vdCosh`
	`const MKL_Complex8*` for `vcCosh`
	`const MKL_Complex16*` for `vzCosh`
		FORTRAN: Array, specifies the input vector *a*.
		C: Pointer to an array that contains the input vector *a*.

Table 9-8 Precision Overflow Thresholds for `Cosh` Real Function

Data Type	Threshold Limitations on Input Parameters
single precision	`-Ln(FLT_MAX)-Ln2 <a[i] < Ln(FLT_MAX)+Ln2`
double precision	`-Ln(DBL_MAX)-Ln2 <a[i] < Ln(DBL_MAX)+Ln2`

NOTE. Overflow can occur also in `Cosh` complex function, but the exact formula is beyond the scope of this document.

Output Parameters

Name	Type	Description

y	**FORTRAN:** `REAL` for `vscosh`
	`DOUBLE PRECISION` for `vdcosh`
	`COMPLEX` for `vccosh`
	`DOUBLE COMPLEX` for `vzcosh`
	C: `float*` for `vsCosh`
		FORTRAN: Array, specifies the output vector *y*.
		C: Pointer to an array that contains the output vector *y*.

Name	Type	Description
	`double*` for `vdCosh`	
	`MKL_Complex8*` for `vcCosh`	
	`MKL_Complex16*` for `vzCosh`	

Sinh

Computes hyperbolic sine of vector elements.

Syntax

Fortran:

```
call vssinh( n, a, y )
call vdsinh( n, a, y )
call vcsinh( n, a, y )
call vzsinh( n, a, y )
```

C:

```
call vsSinh( n, a, y );
call vdSinh( n, a, y );
call vcSinh( n, a, y );
call vzSinh( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** `INTEGER`, `INTENT(IN)`. **C:** `int`.	Specifies the number of elements to be calculated.
a	**FORTRAN:** `REAL`, `INTENT(IN)` for `vssinh` `DOUBLE PRECISION`, `INTENT(IN)` for `vdsinh`	**FORTRAN:** Array, specifies the input vector *a*. **C:** Pointer to an array that contains the input vector *a*.

Name	Type	Description
	COMPLEX, INTENT(IN) for vcsinh DOUBLE COMPLEX, INTENT(IN) for vzsinh **C:** const float* for vsSinh const double* for vdSinh const MKL_Complex8* for vcSinh const MKL_Complex16* for vzSinh	

Table 9-9 Precision Overflow Thresholds for Sinh Real Function

Data Type	Threshold Limitations on Input Parameters
single precision	-Ln(FLT_MAX)-Ln2 <a[i] < Ln(FLT_MAX)+Ln2
double precision	-Ln(DBL_MAX)-Ln2 <a[i] < Ln(DBL_MAX)+Ln2

NOTE. Overflow can occur also in Sinh complex function, but the exact formula is beyond the scope of this document.

Output Parameters

Name	Type	Description
y	**FORTRAN:** REAL for vssinh DOUBLE PRECISION for vdsinh COMPLEX for vcsinh DOUBLE COMPLEX for vzsinh **C:** float* for vsSinh double* for vdSinh MKL_Complex8* for vcSinh MKL_Complex16* for vzSinh	**FORTRAN:** Array, specifies the output vector *y*. **C:** Pointer to an array that contains the output vector *y*.

Tanh

Computes hyperbolic tangent of vector elements.

Syntax

Fortran:

```
call vstanh( n, a, y )
call vdtanh( n, a, y )
call vctanh( n, a, y )
call vztanh( n, a, y )
```

C:

```
call vsTanh( n, a, y );
call vdTanh( n, a, y );
call vcTanh( n, a, y );
call vzTanh( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.
a	**FORTRAN:** REAL, INTENT(IN) for vstanh DOUBLE PRECISION, INTENT(IN) for vdtanh COMPLEX, INTENT(IN) for vctanh DOUBLE COMPLEX, INTENT(IN) for vztanh **C:** const float* for vsTanh const double* for vdTanh	**FORTRAN:** Array, specifies the input vector *a*. **C:** Pointer to an array that contains the input vector *a*.

Name	Type	Description
	`const MKL_Complex8*` for `vcTanh`	
	`const MKL_Complex16*` for `vzTanh`	

Output Parameters

Name	Type	Description

y	**FORTRAN:** `REAL` for `vstanh`
	`DOUBLE PRECISION` for `vdtanh`
	`COMPLEX` for `cstanh`
	`DOUBLE COMPLEX` for `zdtanh`
	C: `float*` for `vsTanh`
	`double*` for `vdTanh`
	`MKL_Complex8*` for `vcTanh`
	`MKL_Complex16*` for `vzTanh`
		FORTRAN: Array, specifies the output vector *y*.
		C: Pointer to an array that contains the output vector *y*.

Acosh

Computes inverse hyperbolic cosine (nonnegative) of vector elements.

Syntax

Fortran:

```
call vsacosh( n, a, y )
call vdacosh( n, a, y )
call vcacosh( n, a, y )
call vzacosh( n, a, y )
```

C:

```
call vsAcosh( n, a, y );
call vdAcosh( n, a, y );
call vcAcosh( n, a, y );
call vzAcosh( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.
a	**FORTRAN:** REAL, INTENT(IN) for vsacosh DOUBLE PRECISION, INTENT(IN) for vdacosh COMPLEX, INTENT(IN) for vcacosh DOUBLE COMPLEX, INTENT(IN) for vzacosh **C:** const float* for vsAcosh const double* for vdAcosh const MKL_Complex8* for vcAcosh const MKL_Complex16* for vzAcosh	**FORTRAN:** Array, specifies the input vector *a*. **C:** Pointer to an array that contains the input vector *a*.

Output Parameters

Name	Type	Description
y	**FORTRAN:** REAL for vsacosh DOUBLE PRECISION for vdacosh	**FORTRAN:** Array, specifies the output vector *y*.

Name	Type	Description

	`COMPLEX` for `vcacosh`
	`DOUBLE COMPLEX` for `vzacosh`
	C: `float*` for `vsAcosh`
	`double*` for `vdAcosh`
	`MKL_Complex8*` for `vcAcosh`
	`MKL_Complex16*` for `vzAcosh`
		C: Pointer to an array that contains the output vector *y*.

Asinh

Computes inverse hyperbolic sine of vector elements.

Syntax

Fortran:

```
call vsasinh( n, a, y )
call vdasinh( n, a, y )
call vcasinh( n, a, y )
call vzasinh( n, a, y )
```

C:

```
call vsAsinh( n, a, y );
call vdAsinh( n, a, y );
call vcAsinh( n, a, y );
call vzAsinh( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** `INTEGER, INTENT(IN)`. **C:** `int`.	Specifies the number of elements to be calculated.

Name	Type	Description

a	**FORTRAN:** `REAL, INTENT(IN)` for `vsasinh`
	`DOUBLE PRECISION, INTENT(IN)` for `vdasinh`
	`COMPLEX, INTENT(IN)` for `vcasinh`
	`DOUBLE COMPLEX, INTENT(IN)` for `vzasinh`
	C: `const float*` for `vsAsinh`
	`const double*` for `vdAsinh`
	`const MKL_Complex8*` for `vcAsinh`
	`const MKL_Complex16*` for `vzAsinh`
		FORTRAN: Array, specifies the input vector *a*.
		C: Pointer to an array that contains the input vector *a*.

Output Parameters

Name	Type	Description

y	**FORTRAN:** `REAL` for `vsasinh`
	`DOUBLE PRECISION` for `vdasinh`
	`COMPLEX` for `vcasinh`
	`DOUBLE COMPLEX` for `vzasinh`
	C: `float*` for `vsAsinh`
	`double*` for `vdAsinh`
	`MKL_Complex8*` for `vcAsinh`
	`MKL_Complex16*` for `vzAsinh`
		FORTRAN: Array, specifies the output vector *y*.
		C: Pointer to an array that contains the output vector *y*.

Atanh

Computes inverse hyperbolic tangent of vector elements.

Syntax

Fortran:

```
call vsatanh( n, a, y )
call vdatanh( n, a, y ) call vcatanh( n, a, y )  call vzatanh( n, a, y )
```

C:

```
call vsAtanh( n, a, y );
call vdAtanh( n, a, y ); call vcAtanh( n, a, y );  call vzAtanh( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.
a	**FORTRAN:** REAL, INTENT(IN) for vsatanh DOUBLE PRECISION, INTENT(IN) for vdatanh COMPLEX, INTENT(IN) for vcatanh DOUBLE COMPLEX, INTENT(IN) for vzatanh **C:** const float* for vsAtanh const double* for vdAtanh const MKL_Complex8* for vcAtanh const MKL_Complex16* for vzAtanh	**FORTRAN:** Array, specifies the input vector *a*. **C:** Pointer to an array that contains the input vector *a*.

Output Parameters

Name	Type	Description
y	**FORTRAN:** REAL for vsatanh DOUBLE PRECISION for vdatanh COMPLEX for vcatanh DOUBLE COMPLEX for vzatanh **C:** float* for vsAtanh double* for vdAtanh MKL_Complex8* for vcAtanh MKL_Complex16* for vzAtanh	**FORTRAN:** Array, specifies the output vector *y*. **C:** Pointer to an array that contains the output vector *y*.

Erf

Computes the error function value of vector elements.

Syntax

Fortran:

```
call vserf( n, a, y )
call vderf( n, a, y )
```

C:

```
call vsErf( n, a, y );
call vdErf( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.

Name	Type	Description

a	**FORTRAN:** `REAL, INTENT(IN)` for `vserf`
	`DOUBLE PRECISION, INTENT(IN)` for `vderf`
	C: `const float*` for `vsErf`
	`const double*` for `vdErf`
		FORTRAN: Array, specifies the input vector *a*.
		C: Pointer to an array that contains the input vector *a*.

Output Parameters

Name	Type	Description

y	**FORTRAN:** `REAL` for `vserf`
	`DOUBLE PRECISION` for `vderf`
	C: `float*` for `vsErf`
	`double*` for `vdErf`
		FORTRAN: Array, specifies the output vector *y*.
		C: Pointer to an array that contains the output vector *y*.

Description

The function `Erf` computes the error function values for elements of the input vector *a* and writes them to the output vector *y*.

The error function is defined as given by:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \quad .$$

Erfc

Computes the complementary error function value of vector elements.

Syntax

Fortran:

```
call vserfc( n, a, y )
call vderfc( n, a, y )
```

C:

```
call vsErfc( n, a, y );
call vdErfc( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.
a	**FORTRAN:** REAL, INTENT(IN) for vserfc DOUBLE PRECISION, INTENT(IN) for vderfc **C:** const float* for vsErfc const double* for vdErfc	**FORTRAN:** Array, specifies the input vector *a*. **C:** Pointer to an array that contains the input vector *a*.

Output Parameters

Name	Type	Description
y	**FORTRAN:** REAL for vserfc DOUBLE PRECISION for vderfc	**FORTRAN:** Array, specifies the output vector *y*.

Name	Type	Description
	C: `float*` for `vsErfc` `double*` for `vdErfc`	**C:** Pointer to an array that contains the output vector *y*.

Description

The function `Erfc` computes the complimentary error function values for elements of the input vector *a* and writes them to the output vector *y*.

The error function is defined as given by:

`erfc(x) = 1 - erf(x)`

or, in other words,

$$\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_{x}^{\infty} e^{-t^2} dt \ .$$

ErfInv

Computes inverse error function value of vector elements.

Syntax

Fortran:

```
call vserfinv( n, a, y )
call vderfinv( n, a, y )
```

C:

```
call vsErfInv( n, a, y );
call vdErfInv( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.
a	**FORTRAN:** REAL, INTENT(IN) for vserfinv DOUBLE PRECISION, INTENT(IN) for vderfinv **C:** const float* for vsErfInv const double* for vdErfInv	**FORTRAN:** Array, specifies the input vector *a*. **C:** Pointer to an array that contains the input vector *a*.

Output Parameters

Name	Type	Description
y	**FORTRAN:** REAL for vserfinv DOUBLE PRECISION for vderfinv **C:** float* for vsErfInv double* for vdErfInv	**FORTRAN:** Array, specifies the output vector *y*. **C:** Pointer to an array that contains the output vector *y*.

Description

The function ErfInv computes the inverse error function values for elements of the input vector *a* and writes them to the output vector *y*.

The inverse error function is defined as given by:

erfinv(*x*) = $\mathrm{erf}^{-1}(x)$,

where erf(*x*) denotes the error function defined as given by

$$\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_{x}^{\infty} e^{-t^2} dt \quad .$$

Floor

Computes a rounded towards minus infinity integer value for each vector element.

Syntax

Fortran:

```
call vsfloor( n, a, y )
call vdfloor( n, a, y )
```

C:

```
call vsFloor( n, a, y );
call vdFloor( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.
a	**FORTRAN:** REAL, INTENT(IN) for vsfloor DOUBLE PRECISION, INTENT(IN) for vdfloor **C:** const float* for vsFloor const double* for vdFloor	**FORTRAN:** Array, specifies the input vector *a*. **C:** Pointer to an array that contains the input vector *a*.

Output Parameters

Name	Type	Description
y	**FORTRAN:** REAL for vsfloor DOUBLE PRECISION for vdfloor	**FORTRAN:** Array, specifies the output vector *y*.

Name	Type	Description
	C: `float*` for `vsFloor` `double*` for `vdFloor`	**C:** Pointer to an array that contains the output vector *y*.

Ceil

Computes a rounded towards plus infinity integer value for each vector element.

Syntax

Fortran:

```
call vsceil( n, a, y )
call vdceil( n, a, y )
```

C:

```
call vsCeil( n, a, y );
call vdCeil( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** `INTEGER, INTENT(IN)`. **C:** `int`.	Specifies the number of elements to be calculated.

a	**FORTRAN:** `REAL, INTENT(IN)` for `vsceil`
	`DOUBLE PRECISION, INTENT(IN)` for `vdceil`
	C: `const float*` for `vsCeil`
	`const double*` for `vdCeil`
		FORTRAN: Array, specifies the input vector *a*.
		C: Pointer to an array that contains the input vector *a*.

Output Parameters

Name	Type	Description
y	**FORTRAN:** REAL for vsceil DOUBLE PRECISION for vdceil **C:** float* for vsCeil double* for vdCeil	**FORTRAN:** Array, specifies the output vector *y*. **C:** Pointer to an array that contains the output vector *y*.

Trunc

Computes a rounded towards zero integer value for each vector element.

Syntax

Fortran:

```
call vstrunc( n, a, y )
call vdtrunc( n, a, y )
```

C:

```
call vsTrunc( n, a, y );
call vdTrunc( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.
a	**FORTRAN:** REAL, INTENT(IN) for vstrunc DOUBLE PRECISION, INTENT(IN) for vdtrunc **C:** const float* for vsTrunc	**FORTRAN:** Array, specifies the input vector *a*. **C:** Pointer to an array that contains the input vector *a*.

Name	Type	Description
	`const double*` for `vdTrunc`	

Output Parameters

Name	Type	Description

y	**FORTRAN:** `REAL` for `vstrunc`
	`DOUBLE PRECISION` for `vdtrunc`
	C: `float*` for `vsTrunc`
	`double*` for `vdTrunc`
		FORTRAN: Array, specifies the output vector *y*.
		C: Pointer to an array that contains the output vector *y*.

Round

Computes a rounded to nearest integer value for each vector element.

Syntax

Fortran:

```
call vsround( n, a, y )
call vdround( n, a, y )
```

C:

```
call vsRound( n, a, y );
call vdRound( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** `INTEGER, INTENT(IN)`. **C:** `int`.	Specifies the number of elements to be calculated.
a	**FORTRAN:** `REAL, INTENT(IN)` for `vsround`	**FORTRAN:** Array, specifies the input vector *a*.

Name	Type	Description

	`DOUBLE PRECISION, INTENT(IN)` for `vdround`
	C: `const float*` for `vsRound`
	`const double*` for `vdRound`
		C: Pointer to an array that contains the input vector *a*.

Output Parameters

Name	Type	Description

y	**FORTRAN:** `REAL` for `vsround`
	`DOUBLE PRECISION` for `vdround`
	C: `float*` for `vsRound`
	`double*` for `vdRound`
		FORTRAN: Array, specifies the output vector *y*.
		C: Pointer to an array that contains the output vector *y*.

Description

Halfway values, that is, 0.5, -1.5, and the like, are rounded off away from zero. That is, 0.5 -> 1, -1.5 -> -2, etc.

NearbyInt

Computes a rounded integer value in a current rounding mode for each vector element.

Syntax

Fortran:

```
call vsnearbyint( n, a, y )
call vdnearbyint( n, a, y )
```

C:

```
call vsNearbyInt( n, a, y );
call vdNearbyInt( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.
a	**FORTRAN:** REAL, INTENT(IN) for vsnearbyint DOUBLE PRECISION, INTENT(IN) for vdnearbyint **C:** const float* for vsNearbyInt const double* for vdNearbyInt	**FORTRAN:** Array, specifies the input vector *a*. **C:** Pointer to an array that contains the input vector *a*.

Output Parameters

Name	Type	Description
y	**FORTRAN:** REAL for vsnearbyint DOUBLE PRECISION for vdnearbyint **C:** float* for vsNearbyInt double* for vdNearbyInt	**FORTRAN:** Array, specifies the output vector *y*. **C:** Pointer to an array that contains the output vector *y*.

Description

Halfway values, that is, 0.5, -1.5, and the like, are rounded off towards even values.

Rint

Computes a rounded integer value in a current rounding mode for each vector element with inexact result exception raised for each changed value.

Syntax

Fortran:

```
call vsrint( n, a, y )
call vdrint( n, a, y )
```

C:

```
call vsRint( n, a, y );
call vdRint( n, a, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.
a	**FORTRAN:** REAL, INTENT(IN) for vsrint DOUBLE PRECISION, INTENT(IN) for vdrint **C:** const float* for vsRint const double* for vdRint	**FORTRAN:** Array, specifies the input vector *a*. **C:** Pointer to an array that contains the input vector *a*.

Output Parameters

Name	Type	Description
y	**FORTRAN:** REAL for vsrint DOUBLE PRECISION for vdrint	**FORTRAN:** Array, specifies the output vector *y*.

Name	Type	Description
	C: `float*` for `vsRint` `double*` for `vdRint`	**C:** Pointer to an array that contains the output vector *y*.

Description

Halfway values, that is, 0.5, -1.5, and the like, are rounded off towards even values. For each changed value, inexact result exception is raised.

Modf

Computes a truncated integer value and remaining fraction part for each vector element.

Syntax

Fortran:

```
call vsmodf( n, a, y, z )
call vdmodf( n, a, y, z )
```

C:

```
call vsModf( n, a, y, z );
call vdModf( n, a, y, z );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** `INTEGER, INTENT(IN)`. **C:** `int`.	Specifies the number of elements to be calculated.
a	**FORTRAN:** `REAL, INTENT(IN)` for `vsmodf` `DOUBLE PRECISION, INTENT(IN)` for `vdmodf` **C:** `const float*` for `vsModf`	**FORTRAN:** Array, specifies the input vector *a*. **C:** Pointer to an array that contains the input vector *a*.

Name	Type	Description
	`const double*` for `vdModf`	

Output Parameters

Name	Type	Description

y, *z*	**FORTRAN:** `REAL` for `vsmodf`
	`DOUBLE PRECISION` for `vdmodf`
	C: `float*` for `vsModf`
	`double*` for `vdModf`
		FORTRAN: Array, specifies the output vector *y* and *z*.
		C: Pointer to an array that contains the output vector *y* and *z*.

VML Pack/Unpack Functions

This section describes VML functions which convert vectors with unit increment to and from vectors with positive increment indexing, vector indexing and mask indexing (see Appendix B for details on vector indexing methods).

Table 9-10 lists available VML Pack/Unpack functions, together with data types and indexing methods associated with them.

Table 9-10 VML Pack/Unpack Functions

Function Short Name	Data Types	Indexing Methods	Description
Pack	s, d	I,V,M	Gathers elements of arrays, indexed by different methods.
Unpack	s, d	I,V,M	Scatters vector elements to arrays with different indexing.

Pack

Copies elements of an array with specified indexing to a vector with unit increment.

Syntax

Fortran:

```
call vsPackI( n, a, inca, y )
call vsPackV( n, a, ia, y )
call vsPackM( n, a, ma, y )
call vdPackI( n, a, inca, y )
call vdPackV( n, a, ia, y )
call vdPackM( n, a, ma, y )
```

C:

```
vsPackI( n, a, inca, y );
vsPackV( n, a, ia, y );
vsPackM( n, a, ma, y );
vdPackI( n, a, inca, y );
vdPackV( n, a, ia, y );
vdPackM( n, a, ma, y );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.
a	**FORTRAN:** REAL, INTENT(IN) for vspacki, vspackv, vspackm	**FORTRAN:** Array, DIMENSION at least (1 + (*n*-1)***inca*) for vspacki/vdpacki,

Name	Type	Description
	DOUBLE PRECISION, INTENT(IN) for vdpacki, vdpackv, vdpackm. **C:** const float* for vsPackI, vsPackV, vsPackM const double* for vdPackI, vdPackV, vdPackM	Array, DIMENSION at least max(*n*,max(*ia*[j])), j=0, …, *n*-1 for vspackv/vdpackv , Array, DIMENSION at least *n* for vspackm/vdpackm. Specifies the input vector *a*. **C:** Specifies pointer to an array that contains the input vector *a*. Size of the array must be: at least (1 + (*n*-1)**inca*) for vsPackI/vdPackI, at least max(*n*,max(*ia*[j])), j=0, …, *n*-1 for vsPackV/vdPackV , at least *n* for vsPackM/vdPackM.
inca	**FORTRAN:** INTEGER, INTENT(IN) for vspacki, vdpacki. **C:** int for vsPackI, vdPackI.	Specifies the increment for the elements of *a*.
ia	**FORTRAN:** INTEGER, INTENT(IN) for vspackv, vdpackv. **C:** const int* for vsPackV, vdPackV.	**FORTRAN:** Array, DIMENSION at least *n*. Specifies the index vector for the elements of *a*. **C:** Specifies the pointer to an array of size at least *n* that contains the index vector for the elements of *a*.
ma	**FORTRAN:** INTEGER, INTENT(IN) for vspackm, vdpackm. **C:** const int* for vsPackM, vdPackM.	**FORTRAN:** Array, DIMENSION at least *n*, Specifies the mask vector for the elements of *a*. **C:** Specifies the pointer to an array of size at least *n* that contains the mask vector for the elements of *a*.

Output Parameters

Name	Type	Description
y	**FORTRAN:** REAL for vspacki, vspackv, vspackm DOUBLE PRECISION for vdpacki, vdpackv, vdpackm. **C:** float* for vsPackI, vsPackV, vsPackM double* for vdPackI, vdPackV, vdPackM	**FORTRAN:** Array, DIMENSION at least *n*. Specifies the output vector *y*. **C:** Pointer to an array of size at least *n* that contains the output vector *y*.

Unpack

Copies elements of a vector with unit increment to an array with specified indexing.

Syntax

Fortran:

```
call vsunpacki( n, a, y, incy )
call vsunpackv( n, a, y, iy )
call vsunpackm( n, a, y, my )
call vdunpacki( n, a, y, incy )
call vdunpackv( n, a, y, iy )
call vdunpackm( n, a, y, my )
```

C:

```
vsUnpackI( n, a, y, incy );
vsUnpackV( n, a, y, iy );
vsUnpackM( n, a, y, my );
vdUnpackI( n, a, y, incy );
vdUnpackV( n, a, y, iy );
vdUnpackM( n, a, y, my );
```

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the number of elements to be calculated.
a	**FORTRAN:** REAL, INTENT(IN) for vsunpacki, vsunpackv, vsunpackm DOUBLE PRECISION, INTENT(IN) for vdunpacki, vdunpackv, vdunpackm. **C:** const float* for vsUnpackI, vsUnpackV, vsUnpackM const double* for vdUnpackI, vdUnpackV, vdUnpackM	**FORTRAN:** Array, DIMENSION at least *n*. Specifies the input vector *a*. **C:** Specifies the pointer to an array of size at least *n* that contains the input vector *a*.
incy	**FORTRAN:** INTEGER, INTENT(IN) for vsunpacki, vdunpacki. **C:** int for vsUnpackI, vdUnpackI.	Specifies the increment for the elements of *y*.
iy	**FORTRAN:** INTEGER, INTENT(IN) for vsunpackv, vdunpackv. **C:** const int* for vsUnpackV, vdUnpackV.	**FORTRAN:** Array, DIMENSION at least *n*. Specifies the index vector for the elements of *y*.

Name	Type	Description
		C: Specifies the pointer to an array of size at least *n* that contains the index vector for the elements of *a*.
my	**FORTRAN:** INTEGER, INTENT(IN) for vsunpackm, vdunpackm. **C:** const int* for vsUnpackM, vdUnpackM.	**FORTRAN:** Array, DIMENSION at least *n*, Specifies the mask vector for the elements of *y*. **C:** Specifies the pointer to an array of size at least *n* that contains the mask vector for the elements of *a*.

Output Parameters

Name	Type	Description
y	**FORTRAN:** REAL for vsunpacki, vsunpackv, vsunpackm DOUBLE PRECISION for vdunpacki, vdunpackv, vdunpackm. **C:** float* for vsUnpackI, vsUnpackV, vsUnpackM double* for vdUnpackI, vdUnpackV, vdUnpackM	**FORTRAN:** Array, DIMENSION at least (1 + (*n*-1)**incy*) for vsunpacki, at least max(*n*,max(*iy*[j])),j=0,..., *n*-1, for vsunpackv, at least *n* for vsunpackm/vdunpackm **C:** Specifies the pointer to an array that contains the output vector *y*. Size of the array must be: at least (1 + (*n*-1)**incy*) for vsUnPackI, at least max(*n*,max(*ia*[j])),j=0,..., *n*-1, for vsUnPackV, at least *n* for vsUnPackM.

VML Service Functions

VML Service functions allow the user to set /get the accuracy mode, and set/get the error code. All these functions are available both in Fortran- and C- interfaces. Table 9-11 lists available VML Service functions and their short description.

Table 9-11 VML Service Functions

Function Short Name	Description
SetMode	Sets the VML mode
GetMode	Gets the VML mode
SetErrStatus	Sets the VML error status
GetErrStatus	Gets the VML error status
ClearErrStatus	Clears the VML error status
SetErrorCallBack	Sets the additional error handler callback function
GetErrorCallBack	Gets the additional error handler callback function
ClearErrorCallBack	Deletes the additional error handler callback function

SetMode

Sets a new mode for VML functions according to `mode` *parameter and stores the previous VML mode to* `oldmode`*.*

Syntax

Fortran:

```
oldmode = vmlsetmode( mode )
```

C:

```
oldmode = vmlSetMode( mode );
```

Input Parameters

Name	Type	Description
mode	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the VML mode to be set.

Output Parameters

Name	Type	Description
oldmode	**FORTRAN:** INTEGER. **C:** int.	Specifies the former VML mode.

Description

The *mode* parameter is designed to control accuracy, FPU, error handling and threading options. Table 9-12 lists values of the *mode* parameter. All other possible values of the *mode* parameter may be obtained from these values by using bitwise OR (|) operation to combine one value for accuracy, one for FPU, and one for error control options. The default value of the *mode* parameter is VML_HA | VML_ERRMODE_DEFAULT | VML_NUM_THREADES_OMP_AUTO. Thus, the current FPU control word (FPU precision and the rounding method) is used by default.

If any VML mathematical function requires different FPU precision, or rounding method, it changes these options automatically and then restores the former values. The *mode* parameter enables you to minimize switching the internal FPU mode inside each VML mathematical function that works with similar precision and accuracy settings. To accomplish this, set the *mode* parameter to VML_FLOAT_CONSISTENT for single precision real and complex functions, or to VML_DOUBLE_CONSISTENT for double precision real and complex functions. These values of the *mode* parameter are the optimal choice for the respective function groups, as they are required for most of the VML mathematical functions. After the execution is over, set the *mode* to VML_RESTORE if you need to restore the previous FPU mode.

Table 9-12 Values of the *mode* Parameter

Value of *mode*	Description
Accuracy Control	
VML_HA	High accuracy versions of VML functions will be used
VML_LA	Low accuracy versions of VML functions will be used
Additional FPU Mode Control	
VML_FLOAT_CONSISTENT	The optimal FPU mode (control word) for single precision functions is set, and the previous FPU mode is saved
VML_DOUBLE_CONSISTENT	The optimal FPU mode (control word) for double precision functions is set, and the previous FPU mode is saved
VML_RESTORE	The previously saved FPU mode is restored
Error Mode Control	

Value of *mode*	Description
VML_ERRMODE_IGNORE	No action is set for computation errors
VML_ERRMODE_ERRNO	On error, the *errno* variable is set
VML_ERRMODE_STDERR	On error, the error text information is written to *stderr*
VML_ERRMODE_EXCEPT	On error, an exception is raised
VML_ERRMODE_CALLBACK	On error, an additional error handler function is called
VML_ERRMODE_DEFAULT	On error, the *errno* variable is set, an exception is raised, and an additional error handler function is called.
Treading Mode Control	
VML_NUM_THREADS_OMP_AUTO	This is default behavior. Maximum number of threads is determined by environmental variable OMP_NUM_THREADS and can be overridden by OpenMP* function omp_set_num_threads(). For performance reasons VML threading logic can use fewer number of threads.
VML_NUM_THREADS_OMP_FIXED	Number of threads is determined by environmental variable OMP_NUM_THREADS and can be overridden by OpenMP* function omp_set_num_threads(). Use this mode to disable VML threading logic.

Examples

Several examples of calling the function vmlSetMode() with different values of the *mode* parameter are given below:

Fortran:

```
oldmode = vmlsetmode( VML_LA )

call vmlsetmode( IOR(VML_LA, IOR(VML_FLOAT_CONSISTENT,
 VML_ERRMODE_IGNORE )))

call vmlsetmode( VML_RESTORE)

call vmlsetmode( VML_NUM_THREADS_OMP_FIXED)
```

C:

```
vmlSetMode( VML_LA );

vmlSetMode( VML_LA | VML_FLOAT_CONSISTENT |
VML_ERRMODE_IGNORE );

vmlSetMode( VML_RESTORE);

vmlSetMode( VML_NUM_THREADS_OMP_FIXED);
```

GetMode

Gets the VML mode.

Syntax

Fortran:

```
mod = vmlgetmode()
```

C:

```
mod = vmlGetMode( void );
```

Output Parameters

Name	Type	Description
mod	**FORTRAN:** INTEGER.	Specifies the packed *mode* parameter.
	C: int.	

Description

The function vmlGetMode() returns the VML *mode* parameter which controls accuracy, FPUand error handling options. The *mod* variable value is some combination of the values listed in the Table 9-12 . You can obtain some of these values using the respective mask from the Table 9-13

Table 9-13 Values of Mask for the *mode* Parameter

Value of mask	Description
VML_ACCURACY_MASK	Specifies mask for accuracy *mode* selection.
VML_FPUMODE_MASK	Specifies mask for FPU *mode* selection.

Value of mask	Description
VML_ERRMODE_MASK	Specifies mask for error *mode* selection.

See example below:

Examples

Fortran:

```
mod = vmlgetmode()
accm = IAND(mod, VML_ACCURACY_MASK)
fpum = IAND(mod, VML_FPUMODE_MASK)
errm = IAND(mod, VML_ERRMODE_MASK)
```

C:

```
accm = vmlGetMode(void )& VML_ACCURACY_MASK;
fpum = vmlGetMode(void )& VML_FPUMODE_MASK;
errm = vmlGetMode(void )& VML_ERRMODE_MASK;
```

SetErrStatus

Sets the new VML error status according to `err` *and stores the previous VML error status to* `olderr`*.*

Syntax

Fortran:

```
olderr = vmlseterrstatus( err )
```

C:

```
olderr = vmlSetErrStatus( err );
```

Input Parameters

Name	Type	Description
err	**FORTRAN:** INTEGER, INTENT(IN). **C:** int.	Specifies the VML error status to be set.

Output Parameters

Name	Type	Description
olderr	**FORTRAN:** INTEGER. **C:** int.	Specifies the former VML error status.

Table 9-14 lists possible values of the *err* parameter.

Table 9-14 Values of the VML Error Status

Error Status	Description
VML_STATUS_OK	The execution was completed successfully.
VML_STATUS_BADSIZE	The array dimension is not positive.
VML_STATUS_BADMEM	NULL pointer is passed.
VML_STATUS_ERRDOM	At least one of array values is out of a range of definition.
VML_STATUS_SING	At least one of array values caused a singularity.
VML_STATUS_OVERFLOW	An overflow has happened during the calculation process.
VML_STATUS_UNDERFLOW	An underflow has happened during the calculation process.

Examples

```
vmlSetErrStatus( VML_STATUS_OK );
vmlSetErrStatus( VML_STATUS_ERRDOM );
vmlSetErrStatus( VML_STATUS_UNDERFLOW );
```

GetErrStatus

Gets the VML error status.

Syntax

Fortran:

```
err = vmlgeterrstatus( )
```

C:

```
err = vmlGetErrStatus( void );
```

Output Parameters

Name	Type	Description
err	**FORTRAN:** INTEGER. **C:** int.	Specifies the VML error status.

ClearErrStatus

Sets the VML error status to VML_STATUS_OK *and stores the previous VML error status to* *olderr*.

Syntax

Fortran:

```
olderr = vmlclearerrstatus( )
```

C:

```
olderr = vmlClearErrStatus( void );
```

Output Parameters

Name	Type	Description
olderr	**FORTRAN:** INTEGER. **C:** int.	Specifies the former VML error status.

SetErrorCallBack

Sets the additional error handler callback function and gets the old callback function.

Syntax

Fortran:

```
oldcallback = vmlseterrorcallback( callback )
```

C:

```
oldcallback = vmlSetErrorCallBack( callback );
```

Input Parameters

Name	Type	Description

callback	**FORTRAN:** Address of the callback function.	**FORTRAN:** The callback function has the following format:
		`INTEGER FUNCTION ERRFUNC(par)`
		`TYPE (ERROR_STRUCTURE) par`
		`! ...`
		`! user error processing`
		`! ...`
		`ERRFUNC = 0`
		`! if ERRFUNC= 0 - standard VML error handler`
		`! is called after the callback`
		`! if ERRFUNC != 0 - standard VML error handler`
		`! is not called`
		`END`

Name	Type	Description

		The passed error structure is defined as follows:
		`TYPE ERROR_STRUCTURE SEQUENCE`
		`INTEGER*4 ICODE`
		`INTEGER*4 IINDEX`
		`REAL*8 DBA1`
		`REAL*8 DBA2`
		`REAL*8 DBR1`
		`REAL*8 DBR2`
		`CHARACTER(64) CFUNCNAME`
		`INTEGER*4 IFUNCNAMELEN`
		`END TYPE ERROR_STRUCTURE`

callback	**C:** Pointer to the callback function.	**C:** The callback function has the following format:
		`static int __stdcall MyHandler(DefVmlErrorContext*`
		`pContext)`
		`{`
		`/* Handler body */`
		`};`

Name	Type	Description

		The passed error structure is defined as follows:
		`typedef struct _DefVmlErrorContext`
		`{`
		`int iCode;/* Error status value */`
		`int iIndex;/* Index for bad array element, or bad array dimension, or bad array pointer */`
		`double dbA1; /* Error argument 1 */`
		`double dbA2; /* Error argument 2 */`
		`double dbR1; /* Error result 1 */`
		`double dbR2; /* Error result 2 */`
		`char cFuncName[64]; /* Function name */`
		`int iFuncNameLen; /* Length of functionname*/`
		`} DefVmlErrorContext;`

Output Parameters

Name	Type	Description
oldcallback	**FORTRAN:** `INTEGER` **C:** `int`	**FORTRAN:** Address of the former callback function. **C:** Pointer to the former callback function.

Description

The callback function is called on each VML mathematical function error if `VML_ERRMODE_CALLBACK` error mode is set (see Table 9-12).

Use the `vmlSetErrorCallBack()` function if you need to define your own callback function instead of default empty callback function.

The input structure for a callback function contains the following information about the error encountered:

- the input value that caused an error
- location (array index) of this value
- the computed result value
- error code
- name of the function in which the error occurred.

You can insert your own error processing into the callback function. This may include correcting the passed result values in order to pass them back and resume computation. The standard error handler is called after the callback function only if it returns *0*.

GetErrorCallBack

Gets the additional error handler callback function.

Syntax

Fortran:

```
callback = vmlgeterrorcallback( )
```

C:

```
callback = vmlGetErrorCallBack( void );
```

Output Parameters

Name	Type	Description
callback		**FORTRAN:** Address of the callback function. **C:** Pointer to the callback function.

ClearErrorCallBack

Deletes the additional error handler callback function and retrieves the former callback function.

Syntax

Fortran:

```
oldcallback = vmlclearerrorcallback( )
```

C:

```
oldcallback = vmlClearErrorCallBack( void );
```

Output Parameters

Name	Type	Description
oldcallback	**FORTRAN:** INTEGER. **C:** int.	**FORTRAN:** Address of the former callback function. **C:** Pointer to the former callback function.

10 Statistical Functions

Statistical functions in Intel® MKL are known as Vector Statistical Library (VSL) that is designed for the purpose of

- generating vectors of pseudorandom and quasi-random numbers
- performing mathematical operations of convolution and correlation.

The corresponding functionality is described in the respective Random Number Generators and Convolution and Correlation sections.

Random Number Generators

VSL provides a set of routines implementing commonly used pseudo- or quasi-random number generators with continuous and discrete distribution. To speed up performance, all these routines were developed using the calls to the highly optimized `Basic Random Number Generators` (BRNGs) and the library of vector mathematical functions (VML, see Chapter 9, "Vector Mathematical Functions").

VSL provides interfaces both for FORTRAN and C languages. For users of the C and C++ languages the mkl_vsl.h header file is provided. For users of the FORTRAN-90 or FORTRAN-95 language the mkl_vsl.fi header file is provided. Both header files are found in the following directory:

```
${MKL}/include
```

The mkl_vsl.fi header is intended for using via the FORTRAN `include` clause and is compatible with both standard forms of F90/F95 sources — the free and 72-columns fixed forms. If you need to use the VSL interface with 80- or 132-columns fixed form sources, you may add a new file to your project. That file is formatted as a 72-columns fixed-form source and consists of a single `include` clause as follows:

```
include 'mkl_vsl.fi'
```

This `include` clause causes the compiler to generate the module files mkl_vsl.mod and mkl_vsl_type.mod, which are used to process the FORTRAN use clauses referencing to the VSL interface:

```
use mkl_vsl_type
use mkl_vsl
```

Because of this specific feature, you do not need to include the mkl_vsl.fi header into each source of your project. You only need to include the header into some of the sources. In any case, make sure that the sources that depend on the VSL interface are compiled after those that include the header so that the module files mkl_vsl.mod and mkl_vsl_type.mod are generated prior to using them.

NOTE. For FORTRAN interface, VSL provides both subroutine-style interface and function-style interface. Default interface in this case is a function-style interface. Default interface in this case is a function-style interface. Subroutine-style interface is provided for backward compatibility only. To use subroutine-style interface, manually include mkl_vsl_subroutine.fi file instead of mkl_vsl.fi by changing the line `include 'mkl_vsl.fi'` in include\mkl.fi with the line `include 'mkl_vsl_subroutine.fi'`.

Function-style interface, unlike subroutine-style interface, allows user to get error status of each routine.

All VSL routines can be classified into three major categories:

- Transformation routines for different types of statistical distributions, for example, uniform, normal (Gaussian), binomial, etc. These routines indirectly call basic random number generators, which are either pseudorandom number generators or quasi-random number generators. Detailed description of the generators can be found in Distribution Generators section.
- Service routines to handle random number streams: create, initialize, delete, copy, save to a binary file, load from a binary file, get the index of a basic generator. The description of these routines can be found in Service Subroutines section.
- Registration routines for basic pseudorandom generators and routines that obtain properties of the registered generators (see Advanced Service Subroutines section).

The last two categories are referred to as service routines.

Conventions

This document makes no specific differentiation between random, pseudorandom, and quasi-random numbers, nor between random, pseudorandom, and quasi-random number generators unless the context requires otherwise. For details, refer to 'Random Numbers' section in VSL Notes document provided with Intel® MKL.

All generators of nonuniform distributions, both discrete and continuous, are built on the basis of the uniform distribution generators, called Basic Random Number Generators (BRNGs). The pseudorandom numbers with nonuniform distribution are obtained through an appropriate

transformation of the uniformly distributed pseudorandom numbers. Such transformations are referred to as `generation methods`. For a given distribution, several generation methods can be used. See VSL Notes for the description of methods available for each generator.

The `stream descriptor` specifies which BRNG should be used in a given transformation method. See 'Random Streams and RNGs in Parallel Computation' section of VSL Notes.

The term `computational node` means a logical or physical unit that can process data in parallel.

Mathematical Notation

The following notation is used throughout the text:

N	The set of natural numbers $N = \{1, 2, 3 \ldots\}$.
Z	The set of integers $Z = \{\ldots -3, -2, -1, 0, 1, 2, 3 \ldots\}$.
R	The set of real numbers.
$\lfloor a \rfloor$	The floor of a (the largest integer less than or equal to a).
$\oplus$ or **xor**	Bitwise exclusive OR.
C_α^κ or $\binom{\alpha}{\kappa}$	Binomial coefficient or combination $(\alpha \in R,\ \alpha \geq 0;\ k \in N \cup \{0\})$.

$$C_\alpha^\kappa = 1$$

For $\alpha \geq k$ binomial coefficient is defined as

$$C_\alpha^\kappa = \frac{\alpha(\alpha - 1) \ldots (\alpha - \kappa + 1)}{\kappa!}$$

If $\alpha < k$, then

$$C_\alpha^\kappa = 0$$

$\Phi(x)$	Cumulative Gaussian distribution function

$$\Phi(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{y^2}{2}\right) dy$$

defined over $-\infty < x < +\infty$.

$\Phi(-\infty) = 0$, $\Phi(+\infty) = 1$.

$\Gamma(\alpha)$ — The complete gamma function

$$\Gamma(\alpha) = \int_{0}^{\infty} t^{\alpha-1} e^{-t} dt$$

where $\alpha > 0$.

$B(p, q)$ — The complete beta function

$$B(p,q) = \int_{0}^{1} t^{p-1}(1-t)^{q-1} dt$$

where $p>0$ and $q>0$.

LCG(a,c, m) — Linear Congruential Generator $x_{n+1} = (ax_n + c) \bmod m$, where a is called the `multiplier`, c is called the `increment`, and m is called the `modulus` of the generator.

MCG(a,m) — Multiplicative Congruential Generator $x_{n+1} = (ax_n) \bmod m$ is a special case of Linear Congruential Generator, where the increment c is taken to be 0.

GFSR(p, q) — Generalized Feedback Shift Register Generator

$x_n = x_{n-p} \oplus x_{n-q}$.

Naming Conventions

The names of all VSL functions in FORTRAN are lowercase; names in C may contain both lowercase and uppercase letters.

The names of generator routines have the following structure:

v<*type of result*>rng<*distribution*> for FORTRAN-interface

v<*type of result*>Rng<*distribution*> for C-interface,

where v is the prefix of a VSL vector function, and the field <*type of result*> is either s, d, or i and specifies one of the following types:

s	REAL for FORTRAN-interface float for C-interface
d	DOUBLE PRECISION for FORTRAN-interface double for C-interface
i	INTEGER for FORTRAN-interface int for C-interface

Prefixes s and d apply to continuous distributions only, prefix i applies only to discrete case. The prefix rng indicates that the routine is a random generator, and the <*distribution*> field specifies the type of statistical distribution.

Names of service routines follow the template below:

vsl<*name*>,

where vsl is the prefix of a VSL service function. The field <*name*> contains a short function name. For a more detailed description of service routines refer to Service Routines and Advanced Service Routines sections.

Prototype of each generator routine corresponding to a given probability distribution fits the following structure:

<*function name*>(*method*, *stream*, *n*, *r*, [<*distribution parameters* >]),

where

- *method* is the number specifying the method of generation. A detailed description of this parameter can be found in Distribution Generators section. See the next page for *method* name structure definition.
- *stream* defines the random stream descriptor and must have a nonzero value. Random streams and their usage are discussed further in Random Streams and Service Routines.
- *n* defines the number of random values to be generated. If *n* is less than or equal to zero, no values are generated. Furthermore, if *n* is negative, an error condition is set.
- *r* defines the destination array for the generated numbers. The dimension of the array must be large enough to store at least *n* random numbers.

Additional parameters included into <*distribution parameters*> field are individual for each generator routine and are described in detail in Distribution Generators section.

To invoke a distribution generator, use a call to the respective VSL routine. For example, to obtain a vector *r*, composed of *n* independent and identically distributed random numbers with normal (Gaussian) distribution, that have the mean value *a* and standard deviation *sigma*, write the following:

for FORTRAN-interface

```
status = vsrnggaussian( method, stream, n, r, a, sigma )
```

for C-interface

```
status = vsRngGaussian( method, stream, n, r, a, sigma )
```

The name of a *method* parameter has the following structure:

VSL_METHOD_<*precision*><*distribution*>_<*method*>,

VSL_METHOD_<*precision*><*distribution*>_<*method*>_ACCURATE,

where

<*precision*>
- *S* for single precision continuous distribution
- *D* for double precision continuous distribution
- *I* for discrete distribution

<*distribution*> probability distribution

<*method*> method name.

Type of name structure for *method* parameter corresponds to fast and accuarate modes of random number generation (see "Distribution Generators" section and VSL Notes for details).

Method names VSL_METHOD_<*precision*><*distribution*>_<*method*> and VSL_METHOD_<*precision*><*distribution*>_<*method*>_ACCURATE should be used with vsl<*precision*>Rng<*distribution*> function only, where

<*precision*>
- *s* for single precision continuous distribution
- *d* for double precision continuous distribution
- *i* for discrete distribution

<*distribution*> probability distribution.

Table 10-1 provides specific predefined values of the *method* name. The third column contains names of the functions that use the given method.

Table 10-1 Values of <*method*> in *method* parameter

Method	Short Description	Functions
STD	Standard method. Currently there is only one method for these functions.	Uniform (continuous), Uniform (discrete), UniformBits
BOXMULLER	BOXMULLER generates normally distributed random number x thru the pair of uniformly distributed numbers u_1 and u_2 according to the formula: $$x = \sqrt{-2 \ln u_1} \sin 2\pi u_2$$	Gaussian, GaussianMV
BOXMULLER2	BOXMULLER2 generates normally distributed random numbers x_1 and x_2 thru the pair of uniformly distributed numbers u_1 and u_2 according to the formulas: $$x_1 = \sqrt{-2 \ln u_1} \sin 2\pi u_2$$ $$x_2 = \sqrt{-2 \ln u_1} \cos 2\pi u_2$$	Gaussian, GaussianMV
ICDF	Inverse cumulative distribution function method.	Exponential, Laplace, Weibull, Cauchy, Rayleigh, Lognormal, Gumbel, Bernoulli, Geometric, Gaussian, GaussianMV
GNORM	For $\alpha > 1$, a gamma distributed random number is generated as a cube of properly scaled normal random number; for $0.6 \leq \alpha < 1$, a gamma distributed random number is generated using rejection from Weibull distribution; for $\alpha <$	Gamma

Method	Short Description	Functions
	0.6, a gamma distributed random number is obtained using transformation of exponential power distribution; for $\alpha = 1$, gamma distribution is reduced to exponential distribution.	
CJA	For $min(p, q) > 1$, Cheng method is used; for $min(p, q) < 1$, Jöhnk method is used, if $q + K \cdot p^2 + C \leq 0$ (K= 0.852..., C=-0.956...) otherwise, Atkinson switching algorithm is used; for $max(p, q) < 1$, method of Jöhnk is used; for $min(p, q) < 1$, $max(p, q) > 1$, Atkinson switching algorithm is used (CJA stands for the first letters of Cheng, Jöhnk, Atkinson); for $p = 1$ or $q = 1$, inverse cumulative distribution function method is used;for $p = 1$ and $q = 1$, beta distribution is reduced to uniform distribution.	Beta
BTPE	Acceptance/rejection method for $ntrial \cdot min(p, 1 - p) \geq 30$ with decomposition into 4 regions: – 2 parallelograms – triangle – left exponential tail – right exponential tail	Binomial
H2PE	Acceptance/rejection method for large mode of distribution with decomposition into 3 regions: – rectangular – left exponential tail – right exponential tail	Hypergeometric
PTPE	Acceptance/rejection method for $\lambda \geq 27$ with decomposition into 4 regions: – 2 parallelograms – triangle – left exponential tail	Poisson

Method	Short Description	Functions
	– right exponential tail; otherwise, table lookup method is used.	
POISNORM	for $\lambda \geq 1$, method based on Poisson inverse CDF approximation by Gaussian inverse CDF; for $\lambda < 1$, table lookup method is used.	Poisson, PoissonV
NBAR	Acceptance/rejection method for , $\frac{(a-1)\cdot(1-p)}{p} \geq 100$ with decomposition into 5 regions: – rectangular – 2 trapezoid – left exponential tail – right exponential tail	NegBinomial

Basic Generators

VSL provides the following BRNGs, which differ in speed and other properties:

- the 32-bit multiplicative congruential pseudorandom number generator MCG(1132489760, 2^{31} -1) [L'Ecuyer99]
- the 32-bit generalized feedback shift register pseudorandom number generator GFSR(250,103) [Kirkpatrick81]
- the combined multiple recursive pseudorandom number generator MRG-32k3a [L'Ecuyer99a]
- the 59-bit multiplicative congruential pseudorandom number generator MCG(13^{13}, 2^{59}) from NAG Numerical Libraries [NAG]
- Wichmann-Hill pseudorandom number generator (a set of 273 basic generators) from NAG Numerical Libraries [NAG]

- Mersenne Twister pseudorandom number generator `MT19937` [Matsumoto98] with period length 2^{19937}-1 of the produced sequence
- Set of 1024 Mersenne Twister pseudorandom number generators `MT2203` [Matsumoto98], [Matsumoto00]. Each of them generates a sequence of period length equal to 2^{2203}-1. Parameters of the generators provide mutual independence of the corresponding sequences.

Besides these pseudorandom number generators, VSL provides two basic quasi-random number generators:

- Sobol quasi-number generator [Sobol76], [Bratley88], which works in dimensions from 1 up to 40.
- Niederreiter quasi-random number generator [Bratley92], which works in dimensions from 1 up to 318.

VSL also provides opportunity to register externally defined initialization parameters of the quasi-random number generators and to work in dimensions established by user. See additional details on interface for registration of the parameters in the library in VSL Notes.

Aslo see some testing results for the generators in VSL Notes and comparative performance data at http://www.intel.com/software/products/mkl/data/vsl/vsl_performance_data.htm.

VSL provides means of registration of such user-designed generators through the steps described in Advanced Service Subroutines section.

For some basic generators, VSL provides two methods of creating independent random streams in multiprocessor computations, which are the leapfrog method and the block-splitting method. These sequence splitting methods are also useful in sequential Monte Carlo.

In addition, MT2203 pseudorandom number generator is a set of 1024 generators designed to create up to 1024 independent random sequences, which might be used in parallel Monte Carlo simulations. Another generator that has the same feature is Wichmann-Hill. It allows creating up to 273 independent random streams. The properties of the generators designed for parallel computations are discussed in detail in [Coddington94].

You may want to design and use your own basic generators. VSL provides means of registration of such user-designed generators through the steps described in Advanced Service Subroutinessection.

There is also an option to utilize externally generated random numbers in VSL distribution generator routines. For this purpose VSL provides three additional basic random number generators:

- for external random data packed in 32-bit integer array
- for external random data stored in double precision floating-point array; data is supposed to be uniformly distributed over (`a`,`b`) interval
- for external random data stored in single precision floating-point array; data is supposed to be uniformly distributed over (`a`,`b`) interval.

Such basic generators are called the abstract basic random number generators.

See VSL Notes for a more detailed description of the generator properties.

BRNG Parameter Definition

Predefined values for the *brng* input parameter are as follows:

Table 10-2 Values of *brng* parameter

Value	Short Description
VSL_BRNG_MCG31	A 31-bit multiplicative congruential generator.
VSL_BRNG_R250	A generalized feedback shift register generator.
VSL_BRNG_MRG32K3A	A combined multiple recursive generator with two components of order 3.
VSL_BRNG_MCG59	A 59-bit multiplicative congruential generator.
VSL_BRNG_WH	A set of 273 Wichmann-Hill combined multiplicative congruential generators.
VSL_BRNG_MT19937	A Mersenne Twister pseudorandom number generator.
VSL_BRNG_MT2203	A set of 1024 Mersenne Twister pseudorandom number generators.
VSL_BRNG_SOBOL	A 32-bit Gray code-based generator producing low-discrepancy sequences for dimensions $1 \le s \le 40$; user-defined dimensions are also available.
VSL_BRNG_NIEDERR	A 32-bit Gray code-based generator producing low-discrepancy sequences for dimensions $1 \le s \le 318$; user-defined dimensions are also available.
VSL_BRNG_IABSTRACT	An abstract random number generator for integer arrays.
VSL_BRNG_DABSTRACT	An abstract random number generator for double precision floating-point arrays.
VSL_BRNG_SABSTRACT	An abstract random number generator for single precision floating-point arrays.

See VSL Notes for detailed description.

Random Streams

`Random stream` (or `stream`) is an abstract source of pseudo- and quasi-random sequences of uniform distribution. Users have no direct access to these sequences and operate with stream state descriptors only. A stream state descriptor, which holds state descriptive information for a particular BRNG, is a necessary parameter in each routine of a distribution generator. Only the distribution generator routines operate with random streams directly. See VSL Notes for details.

NOTE. Random streams associated with abstract basic random number generator are called the abstract random streams. See VSL Notes for detailed description of abstract streams and their use.

User can create unlimited number of random streams by VSL Service Routines like `NewStream` and utilize them in any distribution generator to get the sequence of numbers of given probability distribution. When they are no longer needed, the streams should be deleted calling service routine `DeleteStream`.

VSL provides service functions `SaveStreamF` and `LoadStreamF` to save random stream descriptive data to a binary file and to read this data from a binary file respectively. See VSL Notes for detailed description.

Data Types

FORTRAN:

```
TYPEVSL_STREAM_STATE
INTEGER*4 descriptor1
INTEGER*4 descriptor2
ENDTYPE VSL_STREAM_STATE
```

C:

```
typedef (void*) VSLStreamStatePtr;
```

See Advanced Service Routines for the format of the stream state structure for user-designed generators.

Error Reporting

VSL routines return status codes of the performed operation to report errors and warnings to the calling program. Thus, it is up to the application to perform error-related actions and/or recover from the error. The status codes are of integer type and have the following format:

`VSL_ERROR_<ERROR_NAME>` - indicates VSL errors

`VSL_WARNING_<WARNING_NAME>` - indicates VSL warnings.

VSL errors are of negative values while warnings are of positive values. The status code of zero value indicates that the operation is completed successfully: `VSL_ERROR_OK` (or synonymic `VSL_STATUS_OK`).

Table 10-3 Status Codes and Messages

Status Code	Message
`VSL_ERROR_OK`, `VSL_STATUS_OK`	Indicates no error, execution is successful.
`VSL_ERROR_BAD_ARG`	Input argument value is not valid.
`VSL_ERROR_NULL_PTR`	Input pointer argument is NULL.
`VSL_ERROR_MEM_FAILURE`	System cannot allocate memory.
`VSL_ERROR_INVALID_BRNG_INDEX`	BRNG index is not valid.
`VSL_ERROR_BRNGS_INCOMPATIBLE`	Two BRNGs are not compatible for the operation.
`VSL_ERROR_LEAPFROG_UNSUPPORTED`	BRNG does not support Leapfrog method.
`VSL_ERROR_SKIPAHEAD_UNSUPPORTED`	BRNG does not support Skip-Ahead method.
`VSL_ERROR_BAD_STREAM`	The random stream is invalid.
`VSL_ERROR_FILE_OPEN`	Indicates an error in opening the file.
`VSL_ERROR_FILE_READ`	Indicates an error in reading the file.
`VSL_ERROR_FILE_WRITE`	Indicates an error in writing the file.
`VSL_ERROR_FILE_CLOSE`	Indicates an error in closing the file.

Status Code	Message
VSL_ERROR_BAD_FILE_FORMAT	File format is unknown.
VSL_ERROR_UNSUPPORTED_FILE_VER	File format version is not supported.
VSL_ERROR_BRNG_TABLE_FULL	Registration cannot be completed due to lack of free entries in the table of registered BRNGs.
VSL_ERROR_BAD_STREAM_STATE_SIZE	The value in StreamStateSize field is bad.
VSL_ERROR_BAD_WORD_SIZE	The value in WordSize field is bad.
VSL_ERROR_BAD_NSEEDS	The value in NSeeds field is bad.
VSL_ERROR_BAD_NBITS	The value in NBits field is bad.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or >nmax.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns zero as the number of updated entries in a buffer.
VSL_ERROR_INVALID_ABSTRACT_STREAM	The abstract random stream is invalid.

Service Routines

Stream handling comprises routines for creating, deleting, or copying the streams and getting the index of a basic generator. A random stream can also be saved to and then read from a binary file. Table 10-4 lists all available service routines

Table 10-4 Service Routines

Routine	Short Description
NewStream	Creates and initializes a random stream.
NewStreamEx	Creates and initializes a random stream for the generators with multiple initial conditions.

Routine	Short Description
iNewAbstractStream	Creates and initializes an abstract random stream for integer arrays.
dNewAbstractStream	Creates and initializes an abstract random stream for double precision floating-point arrays.
sNewAbstractStream	Creates and initializes an abstract random stream for single precision floating-point arrays.
DeleteStream	Deletes previously created stream.
CopyStream	Copies a stream to another stream.
CopyStreamState	Creates a copy of a random stream state.
SaveStreamF	Writes a stream to a binary file.
LoadStreamF	Reads a stream from a binary file.
LeapfrogStream	Initializes the stream by the leapfrog method to generate a subsequence of the original sequence.
SkipAheadStream	Initializes the stream by the skip-ahead method.
GetStreamStateBrng	Obtains the index of the basic generator responsible for the generation of a given random stream.
GetNumRegBrngs	Obtains the number of currently registered basic generators.

NOTE. In the above table, the `vsl` prefix in the function names is omitted. In the function reference this prefix is always used in function prototypes and code examples.

Most of the generator-based work comprises three basic steps:

1. Creating and initializing a stream (NewStream, NewStreamEx, CopyStream, CopyStreamState, LeapfrogStream, SkipAheadStream).
2. Generating random numbers with given distribution, see Distribution Generators.
3. Deleting the stream (DeleteStream).

Note that you can concurrently create multiple streams and obtain random data from one or several generators by using the stream state. You must use the DeleteStream function to delete all the streams afterwards.

NewStream

Creates and initializes a random stream.

Syntax

Fortran:

```
status = vslnewstream( stream, brng, seed )
```

C:

```
status = vslNewStream( &stream, brng, seed );
```

Description

For a basic generator with number *brng*, this function creates a new stream and initializes it with a 32-bit seed. The seed is an initial value used to select a particular sequence generated by the basic generator *brng*. The function is also applicable for generators with multiple initial conditions. See VSL Notes for a more detailed description of stream initialization for different basic generators.

NOTE. This function is not applicable for abstract basic random number generators. Please use vsliNewAbstractStream, vslsNewAbstractStream or vsldNewAbstractStream to utilize integer, single-precision or double-precision external random data respectively.

Input Parameters

Name	Type	Description
brng	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Index of the basic generator to initialize the stream. See Table 10-2 for specific value.

Name	Type	Description
seed	**FORTRAN:** INTEGER, INTENT(IN) **C:** unsigned int	Initial condition of the stream. In the case of a quasi-random number generator seed parameter is used to set the dimension. If the dimension is greater than the dimension that brng can support or is less than 1, then the dimension is assumed to be equal to 1.

Output Parameters

Name	Type	Description
stream	**FORTRAN:** TYPE(VSL_STREAM_STATE), INTENT(OUT) **C:** VSLStreamStatePtr*	Stream state descriptor

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_INVALID_BRNG_INDEX	BRNG index is invalid.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for *stream*.

NewStreamEx

Creates and initializes a random stream for generators with multiple initial conditions.

Syntax

Fortran:

```
status = vslnewstreamex( stream, brng, n, params )
```

C:

```
status = vslNewStreamEx( &stream, brng, n, params );
```

Description

This function provides an advanced tool to set the initial conditions for a basic generator if its input arguments imply several initialization parameters. Initial values are used to select a particular sequence generated by the basic generator *brng*. Whenever possible, use NewStream, which is analogous to vslNewStreamEx except that it takes only one 32-bit initial condition. In particular, vslNewStreamEx may be used to initialize the state table in Generalized Feedback Shift Register Generators (GFSRs). A more detailed description of this issue can be found in VSL Notes.

This function is also used to pass user-defined initialization parameters of quasi-random number generators into the library. See VSL Notes for the format for their passing and registration in VSL.

NOTE. This function is not applicable for abstract basic random number generators. Please use vsliNewAbstractStream, vslsNewAbstractStream or vsldNewAbstractStream to utilize integer, single-precision or double-precision external random data respectively.

Input Parameters

Name	Type	Description
brng	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Index of the basic generator to initialize the stream. See Table 10-2 for specific value.
n	**FORTRAN:** INTEGER, INTENT(IN) **C:** unsigned int	Number of initial conditions contained in *params*
params	**FORTRAN:** INTEGER, INTENT(IN) **C:** const unsigned int	Array of initial conditions necessary for the basic generator *brng* to initialize the stream. In the case of a quasi-random number generator only the first element in *params* parameter is used to set the dimension. If the dimension is greater than the dimension that *brng* can support or is less than 1, then the dimension is assumed to be equal to 1.

Output Parameters

Name	Type	Description
stream	**FORTRAN:** TYPE(VSL_STREAM_STATE), INTENT(OUT) **C:** VSLStreamStatePtr*	Stream state descriptor

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_INVALID_BRNG_INDEX	BRNG index is invalid.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for *stream*.

iNewAbstractStream

Creates and initializes an abstract random stream for integer arrays.

Syntax

Fortran:

```
status = vslinewabstractstream( stream, n, ibuf, icallback )
```

C:

```
status = vsliNewAbstractStream( &stream, n, ibuf, icallback );
```

Description

This function creates a new abstract stream and associates it with an integer array *ibuf* and user's callback function *icallback* that is intended for updating of *ibuf* content.

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN)	Size of the array *ibuf*

Name	Type	Description
	C: `int`	
ibuf	**FORTRAN:** `INTEGER, INTENT(IN)` **C:** `unsigned int*`	Array of *n* 32-bit integers
icallback	**FORTRAN:** See Note below **C:** See Note below	**FORTRAN:** Address of the callback function used for *ibuf* update **C:** Pointer to the callback function used for *ibuf* update

Output Parameters

Name	Type	Description
stream	**FORTRAN:** `TYPE(VSL_STREAM_STATE), TINTENT(OUT)` **C:** `VSLStreamStatePtr*`	Descriptor of the stream state structure

NOTE. Format of the callback function in Fortran:

```
INTEGER FUNCTION IUPDATEFUNC[C]( stream, n, ibuf, nmin, nmax, idx )
TYPE(VSL_STREAM_STATE),POINTER :: stream[reference]
INTEGER(KIND=4),INTENT(IN)     :: n[reference]
INTEGER(KIND=4),INTENT(OUT)    :: ibuf[reference](0:n-1)
INTEGER(KIND=4),INTENT(IN)     :: nmin[reference]
INTEGER(KIND=4),INTENT(IN)     :: nmax[reference]
INTEGER(KIND=4),INTENT(IN)     :: idx[reference]
```

Format of the callback function in C:

```
int iUpdateFunc( VSLStreamStatePtrstream, int* n, unsigned int ibuf[],
 int* nmin, int* nmax, int* idx );
```

The callback function returns the number of elements in the array actually updated by the function. Table 10-5 gives the description of the callback function parameters.

Table 10-5 `icallback` Callback Function Parameters

Parameters	Short Description
stream	Abstract stream descriptor
n	Size of *ibuf*
ibuf	Array of random numbers associated with the stream *stream*
nmin	Minimal quantity of numbers to update
nmax	Maximal quantity of numbers that can be updated
idx	Position in cyclic buffer *ibuf* to start update $0 \le idx < n$.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_BAD_ARG	Parameter *n* is not positive.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for *stream*.
VSL_ERROR_NULL_PTR	Either buffer or callback function parameter is a NULL pointer.

dNewAbstractStream

Creates and initializes an abstract random stream for double precision floating-point arrays.

Syntax

Fortran:

```
status = vsldnewabstractstream( stream, n, dbuf, a, b, dcallback )
```

C:

```
status = vsldNewAbstractStream( &stream, n, dbuf, a, b, dcallback );
```

Description

This function creates a new abstract stream for double precision floating-point arrays with random numbers of the uniform distribution over interval (a,b). The function associates the stream with a double precision array *dbuf* and user's callback function *dcallback* that is intended for updating of *dbuf* content.

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Size of the array *dbuf*
dbuf	**FORTRAN:** DOUBLE PRECISION, INTENT(IN) **C:** double*	Array of *n* double precision floating-point random numbers with uniform distribution over interval (a,b)
a	**FORTRAN:** DOUBLE PRECISION, INTENT(IN) **C:** double	Left boundary a
b	**FORTRAN:** DOUBLE PRECISION, INTENT(IN) **C:** double	Right boundary b
dcallback	**FORTRAN:** See Note below **C:** See Note below	**FORTRAN:** Address of the callback function used for update of the array *dbuf* **C:** Pointer to the callback function used for update of the array *dbuf*

Output Parameters

Name	Type	Description
stream	**FORTRAN:** `TYPE(VSL_STREAM_STATE), INTENT(OUT)` **C:** `VSLStreamStatePtr*`	Descriptor of the stream state structure

NOTE. Format of the callback function in Fortran:

```
INTEGER FUNCTION DUPDATEFUNC[C]( stream, n, dbuf, nmin, nmax, idx )
TYPE(VSL_STREAM_STATE),POINTER :: stream[reference]
INTEGER(KIND=4),INTENT(IN)     :: n[reference]
REAL(KIND=8),   INTENT(OUT)    :: dbuf[reference](0:n-1)
INTEGER(KIND=4),INTENT(IN)     :: nmin[reference]
INTEGER(KIND=4),INTENT(IN)     :: nmax[reference]
INTEGER(KIND=4),INTENT(IN)     :: idx[reference]
```

Format of the callback function in C:

```
int dUpdateFunc( VSLStreamStatePtr stream, int* n, double dbuf[], int*
 nmin, int* nmax, int* idx );
```

The callback function returns the number of elements in the array actually updated by the function. Table 10-6 gives the description of the callback function parameters.

`dcallback` Callback Function Parameters

Parameters	Short Description
stream	Abstract stream descriptor
n	Size of *dbuf*
dbuf	Array of random numbers associated with the stream *stream*

Parameters	Short Description
nmin	Minimal quantity of numbers to update
nmax	Maximal quantity of numbers that can be updated
idx	Position in cyclic buffer *dbuf* to start update 0 ≤ *idx* < *n*.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_BAD_ARG	Parameter *n* is not positive.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for *stream*.
VSL_ERROR_NULL_PTR	Either buffer or callback function parameter is a NULL pointer.

sNewAbstractStream

Creates and initializes an abstract random stream for single precision floating-point arrays.

Syntax

Fortran:

```
status = vslsnewabstractstream( stream, n, sbuf, a, b, scallback )
```

C:

```
status = vslsNewAbstractStream( &stream, n, sbuf, a, b, scallback );
```

Description

This function creates a new abstract stream for single precision floating-point arrays with random numbers of the uniform distribution over interval (a,b). The function associates the stream with a single precision array *sbuf* and user's callback function *scallback* that is intended for updating of *sbuf* content.

Input Parameters

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Size of the array *sbuf*
sbuf	**FORTRAN:** REAL, INTENT(IN) **C:** float*	Array of *n* single precision floating-point random numbers with uniform distribution over interval (a,b)
a	**FORTRAN:** REAL, INTENT(IN) **C:** float	Left boundary a
b	**FORTRAN:** REAL, INTENT(IN) **C:** float	Right boundary b
scallback	**FORTRAN:** See Note below **C:** See Note below	**FORTRAN:** Address of the callback function used for update of the array *sbuf* **C:** Pointer to the callback function used for update of the array *sbuf*

Output Parameters

Name	Type	Description
stream	**FORTRAN:** TYPE(VSL_STREAM_STATE), INTENT(OUT) **C:** VSLStreamStatePtr*	Descriptor of the stream state structure

NOTE. Format of the callback function in Fortran:

```
INTEGER FUNCTION SUPDATEFUNC[C]( stream, n, sbuf, nmin, nmax, idx )
TYPE(VSL_STREAM_STATE),POINTER :: stream[reference]
INTEGER(KIND=4),INTENT(IN)     :: n[reference]
REAL(KIND=4),   INTENT(OUT)    :: sbuf[reference](0:n-1)
INTEGER(KIND=4),INTENT(IN)     :: nmin[reference]
INTEGER(KIND=4),INTENT(IN)     :: nmax[reference]
INTEGER(KIND=4),INTENT(IN)     :: idx[reference]
```

Format of the callback function in C:

```
int sUpdateFunc( VSLStreamStatePtr stream, int* n, float sbuf[], int*
 nmin, int* nmax, int* idx );
```

The callback function returns the number of elements in the array actually updated by the function. Table 10-7 gives the description of the callback function parameters.

Table 10-7 `scallback` Callback Function Parameters

Parameters	Short Description
stream	Abstract stream descriptor
n	Size of *sbuf*
sbuf	Array of random numbers associated with the stream *stream*
nmin	Minimal quantity of numbers to update
nmax	Maximal quantity of numbers that can be updated
idx	Position in cyclic buffer *sbuf* to start update $0 \le idx < n$.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.

VSL_ERROR_BAD_ARG	Parameter *n* is not positive.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for *stream*.
VSL_ERROR_NULL_PTR	Either buffer or callback function parameter is a NULL pointer.

DeleteStream

Deletes a random stream.

Syntax

Fortran:

```
status = vsldeletestream( stream )
```

C:

```
status = vslDeleteStream( &stream );
```

Description

This function deletes the random stream created by one of the initialization functions.

Input/Output Parameters

Name	Type	Description
stream	**FORTRAN:** TYPE(VSL_STREAM_STATE), INTENT(OUT)	**FORTRAN:** Stream state descriptor. Must have non-zero value. After the stream is successfully deleted, the descriptor becomes invalid.
	C: VSLStreamStatePtr*	**C:** Stream state descriptor. Must have non-zero value. After the stream is successfully deleted, the pointer is set to NULL.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*srcstream* parameter is a NULL pointer.
VSL_ERROR_BAD_STREAM	*srcstream* is not a valid random stream.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for *newstream*.

CopyStream

Creates a copy of a random stream.

Syntax

Fortran:

```
status = vslcopystream( newstream, srcstream )
```

C:

```
status = vslCopyStream( &newstream, srcstream );
```

Description

The function creates an exact copy of *srcstream* and stores its descriptor to *newstream*.

Input Parameters

Name	Type	Description
srcstream	**FORTRAN:** TYPE(VSL_STREAM_STATE), INTENT(IN) **C:** VSLStreamStatePtr	**FORTRAN:** Descriptor of the stream to be copied **C:** Pointer to the stream state structure to be copied

Output Parameters

Name	Type	Description
newstream	**FORTRAN:** TYPE(VSL_STREAM_STATE), INTENT(OUT) **C:** VSLStreamStatePtr*	Copied stream descriptor

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*srcstream* parameter is a NULL pointer.
VSL_ERROR_BAD_STREAM	*srcstream* is not a valid random stream.

VSL_ERROR_MEM_FAILURE	System cannot allocate memory for *newstream*.

CopyStreamState

Creates a copy of a random stream state.

Syntax

Fortran:

```
status = vslcopystreamstate( deststream, srcstream )
```

C:

```
status = vslCopyStreamState( deststream, srcstream );
```

Description

The function copies a stream state from *srcstream* to the existing *deststream* stream. Both the streams should be generated by the same basic generator. En error message is generated when the index of the BRNG that produced *deststream* stream differs from the index of the BRNG that generated *srcstream* stream.

Unlike CopyStream function, which creates a new stream and copies both the stream state and other data from *srcstream*, the function CopyStreamState copies only *srcstream* stream state data to the generated *deststream* stream.

Input Parameters

Name	Type	Description
srcstream	**FORTRAN:** TYPE(VSL_STREAM_STATE), INTENT(IN) **C:** VSLStreamStatePtr	**FORTRAN:** Descriptor of the destination stream where the state of *scrstream* stream is copied **C:** Pointer to the stream state structure, from which the state structure is copied

Output Parameters

Name	Type	Description
deststream	**FORTRAN:** TYPE(VSL_STREAM_STATE), INTENT(OUT) **C:** VSLStreamStatePtr	**FORTRAN:** Descriptor of the stream with the state to be copied **C:** Pointer to the stream state structure where the stream state is copied

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	Either *srcstream* or *deststream* is a NULL pointer.
VSL_ERROR_BAD_STREAM	Either *srcstream* or *deststream* is not a valid random stream.
VSL_ERROR_BRNGS_INCOMPATIBLE	BRNG associated with *srcstream* is not compatible with BRNG associated with *deststream*.

SaveStreamF

Writes random stream descriptive data to binary file.

Syntax

Fortran:

```
errstatus = vslsavestreamf( stream, fname )
```

C:

```
errstatus = vslSaveStreamF( stream, fname );
```

Description

This function writes the random stream descriptive data to the binary file. Random stream descriptive data is saved to the binary file with the name *fname*. Random stream *stream* must be a valid stream created by NewStream-like or CopyStream-like service routines. If the stream cannot be saved to the file, *errstatus* has a non-zero value. Random stream can be read from the binary file using LoadStreamF function.

Input Parameters

Name	Type	Description
stream	**FORTRAN:** TYPE(VSL_STREAM_STATE), INTENT(IN) **C:** VSLStreamStatePtr	Random stream to be written to the file
fname	**FORTRAN:** CHARACTER(*), INTENT(IN) **C:** char*	**FORTRAN:** File name specified as a C-style null-terminated string **C:** File name specified as a Fortran-style character string

Output Parameters

Name	Type	Description
errstatus	**FORTRAN:** INTEGER **C:** int	Error status of the operation

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	Either *fname* or *stream* is a NULL pointer.
VSL_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_ERROR_FILE_OPEN	Indicates an error in opening the file.
VSL_ERROR_FILE_WRITE	Indicates an error in writing the file.
VSL_ERROR_FILE_CLOSE	Indicates an error in closing the file.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for internal needs.

LoadStreamF

Creates new stream and reads stream descriptive data from binary file.

Syntax

Fortran:

```
errstatus = vslloadstreamf( stream, fname )
```

C:

```
errstatus = vslLoadStreamF( &stream, fname );
```

Description

This function creates a new stream and reads stream descriptive data from the binary file. A new random stream is created using the stream descriptive data from the binary file with the name *fname*. If the stream cannot be read (for example, I/O error occurs or the file format is invalid), *errstatus* has a non-zero value. To save random stream to the file, use SaveStreamF function.

Input Parameters

Name	Type	Description
fname	**FORTRAN:** CHARACTER(*), INTENT(IN) **C:** char*	**FORTRAN:** File name specified as a C-style null-terminated string **C:** File name specified as a Fortran-style character string

Output Parameters

Name	Type	Description
stream	**FORTRAN:** TYPE(VSL_STREAM_STATE), INTENT(OUT) **C:** VSLStreamStatePtr*	**FORTRAN:** Descriptor of a new random stream **C:** Pointer to a new random stream

Name	Type	Description
errstatus	**FORTRAN:** INTEGER **C:** int	Error status of the operation

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*fname* is a NULL pointer.
VSL_ERROR_FILE_OPEN	Indicates an error in opening the file.
VSL_ERROR_FILE_WRITE	Indicates an error in writing the file.
VSL_ERROR_FILE_CLOSE	Indicates an error in closing the file.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for internal needs.
VSL_ERROR_BAD_FILE_FORMAT	Unknown file format.
VSL_ERROR_UNSUPPORTED_FILE_VER	File format version is unsupported.

LeapfrogStream

Initializes a stream using the leapfrog method.

Syntax

Fortran:

```
status = vslleapfrogstream( stream, k, nstreams )
```

C:

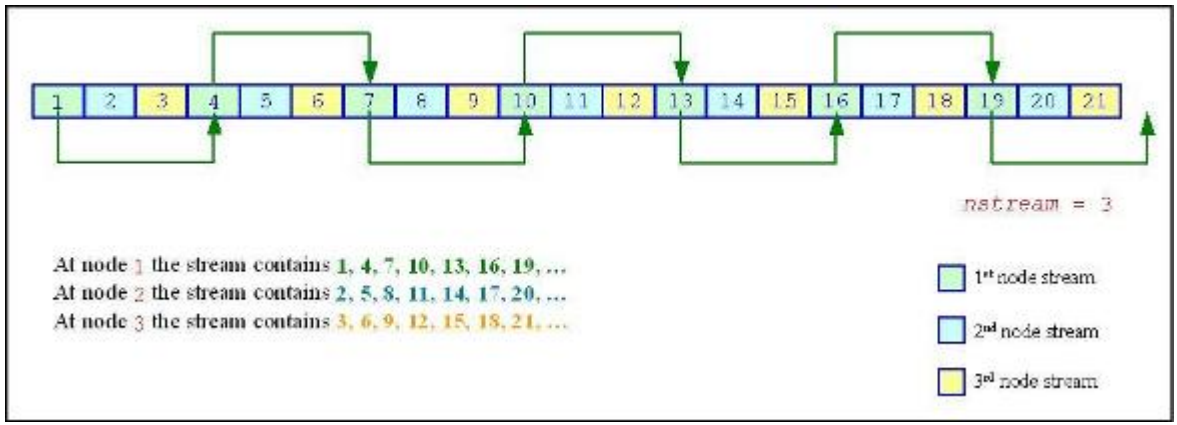
```
status = vslLeapfrogStream( stream, k, nstreams );
```

Description

The function allows generating random numbers in a random stream with non-unit stride. This feature is particularly useful in distributing random numbers from original stream across *nstreams* buffers without generating the original random sequence with subsequent manual distribution. One of the important applications of the leapfrog method is splitting the original sequence into non-overlapping subsequences across *nstreams* computational nodes. The

function initializes the original random stream (see Figure 10-1) to generate random numbers for the computational node k, $0 \le k < nstreams$, where $nstreams$ is the largest number of computational nodes used.

Figure 10-1 Leapfrog Method



The leapfrog method is supported only for those basic generators that allow splitting elements by the leapfrog method, which is more efficient than simply generating them by a generator with subsequent manual distribution across computational nodes. See VSL Notes for details.

For quasi-random basic generators the leapfrog method allows generating individual components of quasi-random vectors instead of whole quasi-random vectors. In this case $nstreams$ parameter should be equal to the dimension of the quasi-random vector while k parameter should be the index of a component to be generated ($0 \le k < nstreams$). Other parameters values are not allowed.

The following code examples illustrate the initialization of three independent streams using the leapfrog method:

Example 10-1 FORTRAN Code for Leapfrog Method

```
...
TYPE(VSL_STREAM_STATE)   ::stream1
TYPE(VSL_STREAM_STATE)   ::stream2
TYPE(VSL_STREAM_STATE)   ::stream3
! Creating 3 identical streams
status = vslnewstream(stream1, VSL_BRNG_MCG31, 174)
status = vslcopystream(stream2, stream1)
status = vslcopystream(stream3, stream1)
! Leapfrogging the streams

status = vslleapfrogstream(stream1, 0, 3)
status = vslleapfrogstream(stream2, 1, 3)
status = vslleapfrogstream(stream3, 2, 3)
! Generating random numbers

...
! Deleting the streams

status = vsldeletestream(stream1)
status = vsldeletestream(stream2)
status = vsldeletestream(stream3)
...
```

Example 10-2 C Code for Leapfrog Method

```
...
VSLStreamStatePtr stream1;
VSLStreamStatePtr stream2;
VSLStreamStatePtr stream3;
/* Creating 3 identical streams */
status = vslNewStream(&stream1, VSL_BRNG_MCG31, 174);
status = vslCopyStream(&stream2, stream1);
status = vslCopyStream(&stream3, stream1);
/* Leapfrogging the streams
*/
status = vslLeapfrogStream(stream1, 0, 3);
status = vslLeapfrogStream(stream2, 1, 3);
status = vslLeapfrogStream(stream3, 2, 3);
/* Generating random numbers
*/
...
/* Deleting the streams
*/
status = vslDeleteStream(&stream1);
status = vslDeleteStream(&stream2);
status = vslDeleteStream(&stream3);
...
```

Input Parameters

Name	Type	Description
stream	**FORTRAN:** `TYPE(VSL_STREAM_STATE), INTENT(IN)`	**FORTRAN:** Descriptor of the stream to which leapfrog method is applied

Name	Type	Description
	C: VSLStreamStatePtr	**C:** Pointer to the stream state structure to which leapfrog method is applied
k	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Index of the computational node, or stream number
nstreams	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Largest number of computational nodes, or stride

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_ERROR_LEAPFROG_UNSUPPORTED	BRNG does not support Leapfrog method.

SkipAheadStream

Initializes a stream using the block-splitting method.

Syntax

Fortran:

```
status = vslskipaheadstream( stream, nskip )
```

C:

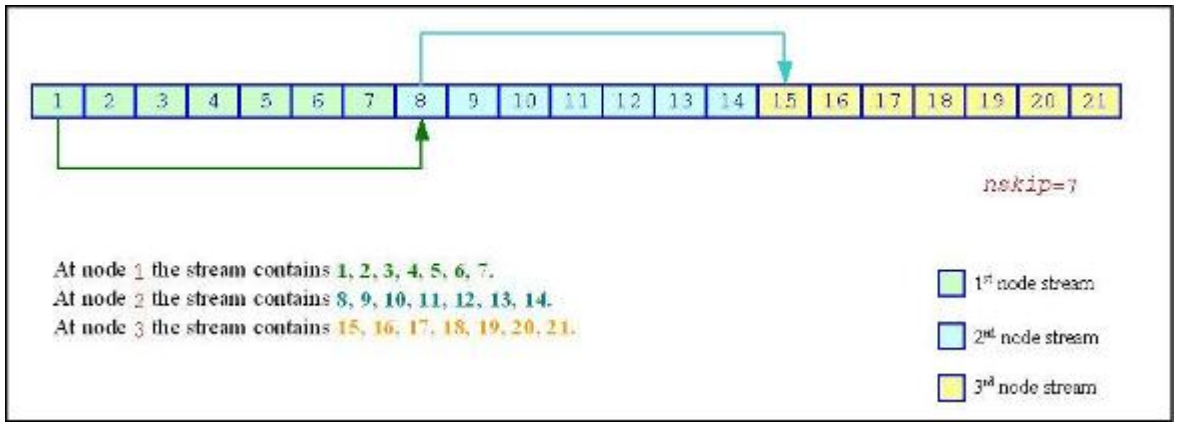
```
status = vslSkipAheadStream( stream, nskip);
```

Description

This function skips a given number of elements in a random stream. This feature is particularly useful in distributing random numbers from original random stream across different computational nodes. If the largest number of random numbers used by a computational node is *nskip*, then the original random sequence may be split by SkipAheadStream into

non-overlapping blocks of *nskip* size so that each block corresponds to the respective computational node. The number of computational nodes is unlimited. This method is known as the block-splitting method or as the skip-ahead method. (see Figure 10-2).

Figure 10-2 Block-Splitting Method



The skip-ahead method is supported only for those basic generators that allow skipping elements by the skip-ahead method, which is more efficient than simply generating them by generator with subsequent manual skipping. See VSL Notes for details.

Please note that for quasi-random basic generators the skip-ahead method works with components of quasi-random vectors rather than with whole quasi-random vectors. Thus to skip `NS` quasi-random vectors, set *nskip* parameter equal to the `NS*DIMEN`, where `DIMEN` is the dimension of quasi-random vector.

The following code examples illustrate how to initialize three independent streams using `SkipAheadStream` function:

Example 10-3 FORTRAN Code for Block-Splitting Method

```
...
type(VSL_STREAM_STATE)  ::stream1
type(VSL_STREAM_STATE)  ::stream2
type(VSL_STREAM_STATE)  ::stream3
! Creating the 1st stream
status = vslnewstream(stream1, VSL_BRNG_MCG31, 174)
! Skipping ahead by 7 elements the 2nd stream

status = vslcopystream(stream2, stream1);
status = vslskipaheadstream(stream2, 7);
! Skipping ahead by 7 elements the 3rd stream

status = vslcopystream(stream3, stream2);
status = vslskipaheadstream(stream3, 7);
! Generating random numbers

...
! Deleting the streams

status = vsldeletestream(stream1)
status = vsldeletestream(stream2)
status = vsldeletestream(stream3)
...
```

Example 10-4 C Code for Block-Splitting Method

```
VSLStreamStatePtr stream1;
VSLStreamStatePtr stream2;
VSLStreamStatePtr stream3;
/* Creating the 1st stream
*/
status = vslNewStream(&stream1, VSL_BRNG_MCG31, 174);
/* Skipping ahead by 7 elements the 2nd stream */
status = vslCopyStream(&stream2, stream1);
status = vslSkipAheadStream(stream2, 7);
/* Skipping ahead by 7 elements the 3rd stream */
status = vslCopyStream(&stream3, stream2);
status = vslSkipAheadStream(stream3, 7);
/* Generating random numbers
*/
...
/* Deleting the streams
*/
status = vslDeleteStream(&stream1);
status = vslDeleteStream(&stream2);
status = vslDeleteStream(&stream3);
...
```

Input Parameters

Name	Type	Description
stream	**FORTRAN:** `TYPE(VSL_STREAM_STATE), INTENT(IN)` **C:** `VSLStreamStatePtr`	**FORTRAN:** Descriptor of the stream to which block-splitting method is applied **C:** Pointer to the stream state structure to which block-splitting method is applied

Name	Type	Description
nskip	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Number of skipped elements

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_ERROR_SKIPAHEAD_UNSUPPORTED	BRNG does not support Skip-Ahead method.

GetStreamStateBrng

Returns index of a basic generator used for generation of a given random stream.

Syntax

Fortran:

```
brng = vslgetstreamstatebrng( stream )
```

C:

```
brng = vslGetStreamStateBrng( stream );
```

Description

This function retrieves the index of a basic generator used for generation of a given random stream.

Input Parameters

Name	Type	Description
stream	**FORTRAN:** TYPE(VSL_STREAM_STATE), TINTENT(IN)	**FORTRAN:** Descriptor of the stream state **C:** Pointer to the stream state structure

Name	Type	Description
	C: VSLStreamStatePtr	

Output Parameters

Name	Type	Description
brng	**FORTRAN:** INTEGER **C:** int	Index of the basic generator assigned for the generation of *stream* ; negative in case of an error

Return Values

VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_ERROR_BAD_STREAM	*stream* is not a valid random stream.

GetNumRegBrngs

Obtains the number of currently registered basic generators.

Syntax

Fortran:

```
nregbrngs = vslgetnumregbrngs( )
```

C:

```
nregbrngs = vslGetNumRegBrngs( void );
```

Description

This function obtains the number of currently registered basic generators. Whenever user registers a user-designed basic generator, the number of registered basic generators is incremented. The maximum number of basic generators that can be registered is determined by VSL_MAX_REG_BRNGS parameter.

Output Parameters

Name	Type	Description
nregbrngs	**FORTRAN:** INTEGER **C:** int	Number of basic generators registered at the moment of the function call

Distribution Generators

VSL routines are used to generate random numbers with different types of distribution. Each function group is introduced below by the type of underlying distribution and contains a short description of its functionality, as well as specifications of the call sequence for both FORTRAN and C-interface and the explanation of input and output parameters. Table 10-8 and Table 10-9 list the random number generator routines, together with used data types, output distributions, and sets correspondence between data types of the generator routines and called basic random number generators.

Table 10-8 Continuous Distribution Generators

Type of Distribution	Data Types	BRNG Data Type	Description
Uniform	s, d	s, d	Uniform continuous distribution on the interval [*a*,*b*].
Gaussian	s, d	s, d	Normal (Gaussian) distribution.
GaussianMV	s, d	s, d	Multivariate normal (Gaussian) distribution.
Exponential	s, d	s, d	Exponential distribution.
Laplace	s, d	s, d	Laplace distribution (double exponential distribution).
Weibull	s, d	s, d	Weibull distribution.
Cauchy	s, d	s, d	Cauchy distribution.
Rayleigh	s, d	s, d	Rayleigh distribution.
Lognormal	s, d	s, d	Lognormal distribution.

Type of Distribution	Data Types	BRNG Data Type	Description
Gumbel	s, d	s, d	Gumbel (extreme value) distribution.
Gamma	s, d	s, d	Gamma distribution.
Beta	s, d	s, d	Beta distribution.

Table 10-9 Discrete Distribution Generators

Type of Distribution	Data Types	BRNG Data Type	Description
Uniform	i	d	Uniform discrete distribution on the interval [*a*,*b*).
UniformBits	i	i	Generator of integer random values with uniform bit distribution.
Bernoulli	i	s	Bernoulli distribution.
Geometric	i	s	Geometric distribution.
Binomial	i	d	Binomial distribution.
Hypergeometric	i	d	Hypergeometric distribution.
Poisson	i	s (for VSL_METHOD_IPOISSON_POISNORM) s (for distribution parameter $\lambda \geq 27$) and d (for $\lambda < 27$) (for VSL_METHOD_IPOISSON_PTPE)	Poisson distribution.
PoissonV	i	s	Poisson distribution with varying mean.

Type of Distribution	Data Types	BRNG Data Type	Description
NegBinomial	i	d	Negative binomial distribution, or Pascal distribution.

The library provides two modes of random number generation, accurate and fast. Accurate generation mode is intended for the applications that are highly demanding to accuracy of calculations. When used in this mode, the generators produce random numbers lying completely within definitional domain for all values of the distribution parameters. For example, random numbers obtained from the generator of continuous distribution that is uniform on interval [*a*,*b*] belong to this interval irrespective of what *a* and *b* values may be. Fast mode provides high performance of generation and also guaranties that generated random numbers belong to the definitional domain except for some specific values of distribution parameters. The generation mode is set by specifying relevant value of the method parameter in generator routines. List of distributions that support accurate mode of generation is given in the table below.

Distribution Generators Supporting Accurate Mode

Type of Distribution	Data Types
Uniform	s, d
Exponential	s, d
Weibull	s, d
Rayleigh	s, d
Lognormal	s, d
Gamma	s, d
Beta	s, d

See additional details about accurate and fast mode of random number generation in VSL Notes.

Continuous Distributions

This section describes routines for generating random numbers with continuous distribution.

Uniform

Generates random numbers with uniform distribution.

Syntax

Fortran:

```
status = vsrnguniform( method, stream, n, r, a, b )
status = vdrnguniform( method, stream, n, r, a, b )
```

C:

```
status = vsRngUniform( method, stream, n, r, a, b );
status = vdRngUniform( method, stream, n, r, a, b );
```

Description

This function generates random numbers uniformly distributed over the interval `[a, b]`, where `a, b` are the left and right bounds of the interval, respectively, and `a, b`$\in R$; `a < b`.

The probability density function is given by:

$$f_{a,b}(x) = \begin{cases} \dfrac{1}{b-a}, & x \in [a,b] \\ 0, & x \notin [a,b] \end{cases}, \quad -\infty < x < +\infty$$

The cumulative distribution function is as follows:

$$f_{a,b}(x) = \begin{cases} 0, & x < a \\ \dfrac{x-a}{b-a}, & a \le x \le b \\ 1, & x \ge b \end{cases}, \quad -\infty < x < +\infty .$$

Input Parameters

Name	Type	Description
method	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Generation method; the specific values are as follows: VSL_METHOD_SUNIFORM_STD VSL_METHOD_DUNIFORM_STD VSL_METHOD_SUNIFORM_STD_ACCURATE VSL_METHOD_DUNIFORM_STD_ACCURATE Standard method.
stream	**FORTRAN:** TYPE (VSL_STREAM_STATE), INTENT(IN) **C:** VSLStreamStatePtr	**FORTRAN:** Descriptor of the stream state structure. **C:** Pointer to the stream state structure
n	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Number of random values to be generated
a	**FORTRAN:** REAL, INTENT(IN) for vsrnguniform DOUBLE PRECISION, INTENT(IN) for vdrnguniform **C:** float for vsRngUniform double for vdRngUniform	Left bound a
b	**FORTRAN:** REAL, INTENT(IN) for vsrnguniform	Right bound b

Name	Type	Description
	DOUBLE PRECISION, INTENT(IN) for vdrnguniform **C:** float for vsRngUniform double for vdRngUniform	

Output Parameters

Name	Type	Description
r	**FORTRAN:** REAL, INTENT(OUT) for vsrnguniform DOUBLE PRECISION, INTENT(OUT) for vdrnguniform **C:** float* for vsRngUniform double* for vdRngUniform	Vector of n random numbers uniformly distributed over the interval [*a*,*b*]

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Gaussian

Generates normally distributed random numbers.

Syntax

Fortran:

```
status = vsrnggaussian( method, stream, n, r, a, sigma )
status = vdrnggaussian( method, stream, n, r, a, sigma )
```

C:

```
status = vsRngGaussian( method, stream, n, r, a, sigma );
status = vdRngGaussian( method, stream, n, r, a, sigma );
```

Description

This function generates random numbers with normal (Gaussian) distribution with mean value `a` and standard deviation σ, where

$a,\ \sigma \in R\ ;\ \sigma > 0.$

The probability density function is given by:

$$f_{a,\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-a)^2}{2\sigma^2}\right),\quad -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\sigma}(x) = \int_{-\infty}^{x} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y-a)^2}{2\sigma^2}\right) \mathbf{dy},\quad -\infty < x < +\infty.$$

The cumulative distribution function $F_{a,\sigma}(x)$ can be expressed in terms of standard normal distribution $\Phi(x)$ as

$F_{a,\sigma}(x) = \Phi((x - a)/\sigma)$

Input Parameters

Name	Type	Description
method	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Generation method. The specific values are as follows: VSL_METHOD_SGAUSSIAN_BOXMULLER VSL_METHOD_SGAUSSIAN_BOXMULLER2 VSL_METHOD_SGAUSSIAN_ICDF VSL_METHOD_DGAUSSIAN_BOXMULLER VSL_METHOD_DGAUSSIAN_BOXMULLER2 VSL_METHOD_DGAUSSIAN_ICDF See brief description of the methods BOXMULLER, BOXMULLER2, and ICDF in Table 10-1
stream	**FORTRAN:** TYPE (VSL_STREAM_STATE), INTENT(IN) **C:** VSLStreamStatePtr	**FORTRAN:** Descriptor of the stream state structure. **C:** Pointer to the stream state structure
n	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Number of random values to be generated
a	**FORTRAN:** REAL, INTENT(IN) for vsrnggaussian DOUBLE PRECISION, INTENT(IN) for vdrnggaussian **C:** float for vsRngGaussian	Mean value a

Name	Type	Description
	`double` for `vdRngGaussian`	

sigma	**FORTRAN:** `REAL`, `INTENT(IN)` for `vsrnggaussian`
	`DOUBLE PRECISION`, `INTENT(IN)` for `vdrnggaussian`
	C: `float` for `vsRngGaussian`
	`double` for `vdRngGaussian`
		Standard deviation σ

Output Parameters

Name	Type	Description

r	**FORTRAN:** `REAL`, `INTENT(OUT)` for `vsrnggaussian`
	`DOUBLE PRECISION`, `INTENT(OUT)` for `vdrnggaussian`
	C: `float*` for `vsRngGaussian`
	`double*` for `vdRngGaussian`
		Vector of `n` normally distributed random numbers

Return Values

`VSL_ERROR_OK`, `VSL_STATUS_OK`	Indicates no error, execution is successful.
`VSL_ERROR_NULL_PTR`	*stream* is a `NULL` pointer.
`VSL_ERROR_BAD_STREAM`	*stream* is not a valid random stream.
`VSL_ERROR_BAD_UPDATE`	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, `< 0` or `> nmax`.

VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

GaussianMV

Generates random numbers from multivariate normal distribution.

Syntax

Fortran:

```
status = vsrnggaussianmv( method, stream, n, r, dimen, mstorage, a, t )
status = vdrnggaussianmv( method, stream, n, r, dimen, mstorage, a, t )
```

C:

```
status = vsRngGaussianMV( method, stream, n, r, dimen, mstorage, a, t );
status = vdRngGaussianMV( method, stream, n, r, dimen, mstorage, a, t );
```

Description

This function generates random numbers with d-variate normal (Gaussian) distribution with mean value a and variance-covariance matrix C, where $a \in R^d$; C is a $d \times d$ symmetric positive-definite matrix.

The probability density function is given by:

$$f_{a,C}(x) = \frac{1}{\sqrt{\det(2\pi C)}} \exp\left(-1/2(x-a)^T C^{-1}(x-a)\right),$$

where $x \in R^d$.

Matrix C can be represented as $C = TT^T$, where T is a lower triangular matrix - Cholesky factor of C.

Instead of variance-covariance matrix C the generation routines require Cholesky factor of C in input. To compute Cholesky factor of matrix C, the user may call MKL LAPACK routines for matrix factorization: ?potrf or ?pptrf for v?RngGaussianMV/v?rnggaussianmv routines (? means either s or d for single and double precision respectively). See Application Notes for more details.

Input Parameters

Name	Type	Description
method	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Generation method. The specific values are as follows: VSL_METHOD_SGAUSSIANMV_BOXMULLER VSL_METHOD_SGAUSSIANMV_BOXMULLER2 VSL_METHOD_SGAUSSIANMV_ICDF VSL_METHOD_DGAUSSIANMV_BOXMULLER VSL_METHOD_DGAUSSIANMV_BOXMULLER2 VSL_METHOD_DGAUSSIANMV_ICDF See brief description of the methods BOXMULLER, BOXMULLER2, and ICDF in Table 10-1
stream	**FORTRAN:** TYPE (VSL_STREAM_STATE), INTENT(IN) **C:** VSLStreamStatePtr	**FORTRAN:** Descriptor of the stream state structure. **C:** Pointer to the stream state structure
n	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Number of random values to be generated
dimen	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Dimension d ($d \geq 1$) of output random vectors
mstorage	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	**FORTRAN:** Matrix storage scheme for upper triangular matrix T^T. The routine supports three matrix storage schemes:

Name	Type	Description
		• VSL_MATRIX_STORAGE_FULL— all d x d elements of the matrix T^T are passed, however, only the upper triangle part is actually used in the routine. • VSL_MATRIX_STORAGE_PACKED— upper triangle elements of T^T are packed by rows into a one-dimensional array. • VSL_MATRIX_STORAGE_DIAGONAL— only diagonal elements of T^T are passed. **C:** Matrix storage scheme for lower triangular matrix T. The routine supports three matrix storage schemes: • VSL_MATRIX_STORAGE_FULL— all d x d elements of the matrix T are passed, however, only the lower triangle part is actually used in the routine. • VSL_MATRIX_STORAGE_PACKED— lower triangle elements of T are packed by rows into a one-dimensional array. • VSL_MATRIX_STORAGE_DIAGONAL— only diagonal elements of T are passed.
a	**FORTRAN:** REAL, INTENT(IN) for vsrnggaussianmv DOUBLE PRECISION, INTENT(IN) for vdrnggaussianmv **C:** float* for vsRngGaussianMV double* for vdRngGaussianMV	Mean vector a of dimension d

Name	Type	Description

t	**FORTRAN:** `REAL, INTENT(IN)` for `vsrnggaussianmv`
	`DOUBLE PRECISION, INTENT(IN)` for `vdrnggaussianmv`
	C: `float*` for `vsRngGaussianMV`
	`double*` for `vdRngGaussianMV`
		FORTRAN: Elements of the upper triangular matrix passed according to the matrix T^T storage scheme *mstorage*.
		C: Elements of the lower triangular matrix passed according to the matrix T storage scheme *mstorage*.

Output Parameters

Name	Type	Description

r	**FORTRAN:** `REAL, INTENT(OUT)` for `vsrnggaussianmv`
	`DOUBLE PRECISION, INTENT(OUT)` for `vdrnggaussianmv`
	C: `float*` for `vsRngGaussianMV`
	`double*` for `vdRngGaussianMV`
		Array of n random vectors of dimension *dimen*

Application Notes

Since matrices are stored in Fortran by columns, while in C they are stored by rows, the usage of MKL factorization routines (assuming Fortran matrices storage) in combination with multivariate normal RNG (assuming C matrix storage) is slightly different in C and Fortran. The following tables help in using these routines in C and Fortran. For further information please refer to the appropriate VSL example file.

Using Cholesky Factorization Routines in Fortran

Matrix Storage Scheme	Variance-Covariance Matrix Argument	Factorization Routine	UPLO Parameter in Factorization Routine	Result of Factorization as Input Argument for RNG
VSL_MATRIX_STORAGE_FULL	C in Fortran two-dimensional array	spotrf for vsrnggaussianmv dpotrf for vdrnggaussianmv	'U'	Upper triangle of T^T. Lower triangle is not used.
VSL_MATRIX_STORAGE_PACKED	Lower triangle of C packed by columns into one-dimensional array	spptrf for vsrnggaussianmv dpptrf for vdrnggaussianmv	'L'	Upper triangle of T^T packed by rows into one-dimensional array.

Using Cholesky Factorization Routines in C

Matrix Storage Scheme	Variance-Covariance Matrix Argument	Factorization Routine	UPLO Parameter in Factorization Routine	Result of Factorization as Input Argument for RNG
VSL_MATRIX_STORAGE_FULL	C in C two-dimensional array	spotrf for vsRngGaussianMV dpotrf for vdRngGaussianMV	'U'	Upper triangle of T^T. Lower triangle is not used.
VSL_MATRIX_STORAGE_PACKED	Lower triangle of C packed by columns into one-dimensional array	spptrf for vsRngGaussianMV dpptrf for vdRngGaussianMV	'L'	Upper triangle of T^T packed by

Matrix Storage Scheme	Variance-Covariance Matrix Argument	Factorization Routine	UPLO Parameter in Factorization Routine	Result of Factorization as Input Argument for RNG
				rows into one-dimensional array.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Exponential

Generates exponentially distributed random numbers.

Syntax

Fortran:

```
status = vsrngexponential( method, stream, n, r, a, beta )
status = vdrngexponential( method, stream, n, r, a, beta )
```

C:

```
status = vsRngExponential( method, stream, n, r, a, beta );
status = vdRngExponential( method, stream, n, r, a, beta );
```

Description

This function generates random numbers with exponential distribution that has displacement `a` and scalefactor β, where $a,\ \beta \in R\ ;\ \beta > 0$.

The probability density function is given by:

$$f_{a,\beta}(x) = \begin{cases} \frac{1}{\beta} \exp\left((-(x-a)) / \beta \right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - \exp\left((-(x-a)) / \beta \right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

Input Parameters

Name	Type	Description

method	**FORTRAN:** `INTEGER, INTENT(IN)`
	C: `int`
		Generation method. The specific values are as follows:
		`VSL_METHOD_SEXPONENTIAL_ICDF`
		`VSL_METHOD_DEXPONENTIAL_ICDF`
		`VSL_METHOD_SEXPONENTIAL_ICDF_ACCURATE`
		`VSL_METHOD_DEXPONENTIAL_ICDF_ACCURATE`
		Inverse cumulative distribution function method

stream	**FORTRAN:** `TYPE (VSL_STREAM_STATE), INTENT(IN)` **C:** `VSLStreamStatePtr`	**FORTRAN:** Descriptor of the stream state structure. **C:** Pointer to the stream state structure

Name	Type	Description
n	**FORTRAN:** `INTEGER, INTENT(IN)` **C:** `int`	Number of random values to be generated
a	**FORTRAN:** `REAL, INTENT(IN)` for `vsrngexponential` `DOUBLE PRECISION, INTENT(IN)` for `vdrngexponential` **C:** `float` for `vsRngExponential` **C:** `double` for `vdRngExponential`	Displacement *a*

beta	**FORTRAN:** `REAL, INTENT(IN)` for `vsrngexponential`
	`DOUBLE PRECISION, INTENT(IN)` for `vdrngexponential`
	C: `float` for `vsRngExponential`
	`double` for `vdRngExponential`
		Scalefactor β

Output Parameters

Name	Type	Description
r	**FORTRAN:** `REAL, INTENT(OUT)` for `vsrngexponential`	Vector of *n* exponentially distributed random numbers

Name	Type	Description
	DOUBLE PRECISION, INTENT(OUT) for vdrngexponential **C:** float* for vsRngExponential double* for vdRngExponential	

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Laplace

Generates random numbers with Laplace distribution.

Syntax

Fortran:

```
status = vsrnglaplace( method, stream, n, r, a, beta )
status = vdrnglaplace( method, stream, n, r, a, beta )
```

C:

```
status = vsRngLaplace( method, stream, n, r, a, beta );
status = vdRngLaplace( method, stream, n, r, a, beta );
```

Description

This function generates random numbers with Laplace distribution with mean value (or average) a and scalefactor β, where a, $\beta \in R$; $\beta > 0$. The scalefactor value determines the standard deviation as

$$\sigma = \beta\sqrt{2}$$

The probability density function is given by:

$$f_{a,\beta}(x) = \frac{1}{\sqrt{2}\beta} \exp\left(-\frac{|x-a|}{\beta}\right), -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$f_{a,\beta}(x) = \begin{cases} \frac{1}{2}\exp\left(-\frac{|x-a|}{\beta}\right), & x \geq a \\ 1 - \frac{1}{2}\exp\left(-\frac{|x-a|}{\beta}\right), & x < a \end{cases}, -\infty < x < +\infty.$$

Input Parameters

Name	Type	Description
method	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Generation method. The specific values are as follows: VSL_METHOD_SLAPLACE_ICDF VSL_METHOD_DLAPLACE_ICDF Inverse cumulative distribution function method

Name	Type	Description
stream	**FORTRAN:** `TYPE (VSL_STREAM_STATE), INTENT(IN)` **C:** `VSLStreamStatePtr`	**FORTRAN:** Descriptor of the stream state structure. **C:** Pointer to the stream state structure
n	**FORTRAN:** `INTEGER, INTENT(IN)` **C:** `int`	Number of random values to be generated

a	**FORTRAN:** `REAL, INTENT(IN)` for `vsrnglaplace`
	`DOUBLE PRECISION, INTENT(IN)` for `vdrnglaplace`
	C: `float` for `vsRngLaplace`
	`double` for `vdRngLaplace`
		Mean value *a*

beta	**FORTRAN:** `REAL, INTENT(IN)` for `vsrnglaplace`
	`DOUBLE PRECISION, INTENT(IN)` for `vdrnglaplace`
	C: `float` for `vsRngLaplace`
	`double` for `vdRngLaplace`
		Scalefactor β

Output Parameters

Name	Type	Description
r	**FORTRAN:** `REAL, INTENT(OUT)` for `vsrnglaplace`	Vector of *n* Laplace distributed random numbers

Name	Type	Description
	DOUBLE PRECISION, INTENT(OUT) for vdrnglaplace **C:** float* for vsRngLaplace double* for vdRngLaplace	

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Weibull

Generates Weibull distributed random numbers.

Syntax

Fortran:

```
status = vsrngweibull( method, stream, n, r, alpha, a, beta )
status = vdrngweibull( method, stream, n, r, alpha, a, beta )
```

C:

```
status = vsRngWeibull( method, stream, n, r, alpha, a, beta );
status = vdRngWeibull( method, stream, n, r, alpha, a, beta );
```

Description

This function generates Weibull distributed random numbers with displacement a, scalefactor β, and shape α, where α, β, $a \in R$; $\alpha > 0$, $\beta > 0$.

The probability density function is given by:

$$f_{a,\alpha,\beta}(x) = \begin{cases} \frac{\alpha}{\beta^{\alpha}}(x-a)^{\alpha-1}\exp\left(-\left(\frac{x-a}{\beta}\right)^{\alpha}\right), & x \geq a \\ 0, & x < a \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\alpha,\beta}(x) = \begin{cases} 1-\exp\left(-\left(\frac{x-a}{\beta}\right)^{\alpha}\right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

Input Parameters

Name	Type	Description

method	**FORTRAN:** `INTEGER, INTENT(IN)`
	C: `int`
		Generation method. The specific values are as follows:
		`VSL_METHOD_SWEIBULL_ICDF`
		`VSL_METHOD_DWEIBULL_ICDF`
		`VSL_METHOD_SWEIBULL_ICDF_ACCURATE`
		`VSL_METHOD_DWEIBULL_ICDF_ACCURATE`
		Inverse cumulative distribution function method

stream	**FORTRAN:** `TYPE (VSL_STREAM_STATE), INTENT(IN)` **C:** `VSLStreamStatePtr`	**FORTRAN:** Descriptor of the stream state structure. **C:** Pointer to the stream state structure

Name	Type	Description
n	**FORTRAN:** `INTEGER, INTENT(IN)` **C:** `int`	Number of random values to be generated

alpha	**FORTRAN:** `REAL, INTENT(IN)` for `vsrngweibull`
	`DOUBLE PRECISION, INTENT(IN)` for `vdrngweibull`
	C: `float` for `vsRngWeibull`
	`double` for `vdRngWeibull`
		Shape α

a	**FORTRAN:** `REAL, INTENT(IN)` for `vsrngweibull`
	`DOUBLE PRECISION, INTENT(IN)` for `vdrngweibull`
	C: `float` for `vsRngWeibull`
	`double` for `vdRngWeibull`
		Displacement *a*

beta	**FORTRAN:** `REAL, INTENT(IN)` for `vsrngweibull`
	`DOUBLE PRECISION, INTENT(IN)` for `vdrngweibull`
	C: `float` for `vsRngWeibull`
	`double` for `vdRngWeibull`
		Scalefactor β

Output Parameters

Name	Type	Description
r	**FORTRAN:** REAL, INTENT(OUT) for vsrngweibull DOUBLE PRECISION, INTENT(OUT) for vdrngweibull **C:** float* for vsRngWeibull double* for vdRngWeibull	Vector of *n* Weibull distributed random numbers

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Cauchy

Generates Cauchy distributed random values.

Syntax

Fortran:

```
status = vsrngcauchy( method, stream, n, r, a, beta )
status = vdrngcauchy( method, stream, n, r, a, beta )
```

C:

```
status = vsRngCauchy( method, stream, n, r, a, beta );
status = vdRngCauchy( method, stream, n, r, a, beta );
```

Description

This function generates Cauchy distributed random numbers with displacement a and scalefactor β, where a, $\beta \in R$; $\beta > 0$.

The probability density function is given by:

$$f_{a,\beta}(x) = \frac{1}{\pi\beta\left(1 + \left(\frac{x-a}{\beta}\right)^2\right)}, -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \frac{1}{2} + \frac{1}{\pi}\arctan\left(\frac{x-a}{\beta}\right), -\infty < x < +\infty.$$

Input Parameters

Name	Type	Description
method	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Generation method. The specific values are as follows: VSL_METHOD_SCAUCHY_ICDF VSL_METHOD_DCAUCHY_ICDF Inverse cumulative distribution function method

Name	Type	Description
stream	**FORTRAN:** `TYPE (VSL_STREAM_STATE), INTENT(IN)` **C:** `VSLStreamStatePtr`	**FORTRAN:** Descriptor of the stream state structure. **C:** Pointer to the stream state structure
n	**FORTRAN:** `INTEGER, INTENT(IN)` **C:** `int`	Number of random values to be generated

a	**FORTRAN:** `REAL, INTENT(IN)` for `vsrngcauchy`
	`DOUBLE PRECISION, INTENT(IN)` for `vdrngcauchy`
	C: `float` for `vsRngCauchy`
	`double` for `vdRngCauchy`
		Displacement *a*

beta	**FORTRAN:** `REAL, INTENT(IN)` for `vsrngcauchy`
	`DOUBLE PRECISION, INTENT(IN)` for `vdrngcauchy`
	C: `float` for `vsRngCauchy`
	`double` for `vdRngCauchy`
		Scalefactor β

Output Parameters

Name	Type	Description
r	**FORTRAN:** `REAL, INTENT(OUT)` for `vsrngcauchy` `DOUBLE PRECISION, INTENT(OUT)` for `vdrngcauchy`	Vector of *n* Cauchy distributed random numbers

Name	Type	Description
	C: `float*` for `vsRngCauchy` `double*` for `vdRngCauchy`	

Return Values

`VSL_ERROR_OK, VSL_STATUS_OK`	Indicates no error, execution is successful.
`VSL_ERROR_NULL_PTR`	*`stream`* is a `NULL` pointer.
`VSL_ERROR_BAD_STREAM`	*`stream`* is not a valid random stream.
`VSL_ERROR_BAD_UPDATE`	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, `< 0` or `> nmax`.
`VSL_ERROR_NO_NUMBERS`	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Rayleigh

Generates Rayleigh distributed random values.

Syntax

Fortran:

```
status = vsrngrayleigh( method, stream, n, r, a, beta )
status = vdrngrayleigh( method, stream, n, r, a, beta )
```

C:

```
status = vsRngRayleigh( method, stream, n, r, a, beta );
status = vdRngRayleigh( method, stream, n, r, a, beta );
```

Description

This function generates Rayleigh distributed random numbers with displacement a and scalefactor β, where a, $\beta \in R$; $\beta > 0$.

Rayleigh distribution is a special case of Weibull distribution, where the shape parameter $\alpha = 2$.

The probability density function is given by:

$$f_{a,\beta}(x) = \begin{cases} \frac{2(x-a)}{\beta^2} \exp\left(-\frac{(x-a)^2}{\beta^2}\right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - \exp\left(-\frac{(x-a)^2}{\beta^2}\right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

Input Parameters

Name	Type	Description
method	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Generation method. The specific values are as follows: VSL_METHOD_SRAYLEIGH_ICDF VSL_METHOD_DRAYLEIGH_ICDF VSL_METHOD_SRAYLEIGH_ICDF_ACCURATE VSL_METHOD_DRAYLEIGH_ICDF_ACCURATE Inverse cumulative distribution function method
stream	**FORTRAN:** TYPE (VSL_STREAM_STATE), INTENT(IN) **C:** VSLStreamStatePtr	**FORTRAN:** Descriptor of the stream state structure. **C:** Pointer to the stream state structure

Name	Type	Description
n	**FORTRAN:** `INTEGER, INTENT(IN)` **C:** `int`	Number of random values to be generated

a	**FORTRAN:** `REAL, INTENT(IN)` for `vsrngrayleigh`
	`DOUBLE PRECISION, INTENT(IN)` for `vdrngrayleigh`
	C: `float` for `vsRngRayleigh`
	`double` for `vdRngRayleigh`
		Displacement *a*

beta	**FORTRAN:** `REAL, INTENT(IN)` for `vsrngrayleigh`
	`DOUBLE PRECISION, INTENT(IN)` for `vdrngrayleigh`
	C: `float` for `vsRngRayleigh`
	`double` for `vdRngRayleigh`
		Scalefactor β

Output Parameters

Name	Type	Description
r	**FORTRAN:** `REAL, INTENT(OUT)` for `vsrngrayleigh` `DOUBLE PRECISION, INTENT(OUT)` for `vdrngrayleigh`	Vector of *n* Rayleigh distributed random numbers

Name	Type	Description
	C: `float*` for `vsRngRayleigh` `double*` for `vdRngRayleigh`	

Return Values

`VSL_ERROR_OK, VSL_STATUS_OK`	Indicates no error, execution is successful.
`VSL_ERROR_NULL_PTR`	*`stream`* is a `NULL` pointer.
`VSL_ERROR_BAD_STREAM`	*`stream`* is not a valid random stream.
`VSL_ERROR_BAD_UPDATE`	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, `< 0` or `> nmax`.
`VSL_ERROR_NO_NUMBERS`	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Lognormal

Generates lognormally distributed random numbers.

Syntax

Fortran:

```
status = vsrnglognormal( method, stream, n, r, a, sigma, b, beta )
status = vdrnglognormal( method, stream, n, r, a, sigma, b, beta )
```

C:

```
status = vsRngLognormal( method, stream, n, r, a, sigma, b, beta );
status = vdRngLognormal( method, stream, n, r, a, sigma, b, beta );
```

Description

This function generates lognormally distributed random numbers with average of distribution *a* and standard deviation σ of subject normal distribution, displacement *b*, and scalefactor β, where *a*, σ, *b*, β∈*R* ; σ > 0 , β > 0.

The probability density function is given by:

$$f_{a,\sigma,b,\beta}(x) = \begin{cases} \dfrac{1}{\sigma(x-b)\sqrt{2\pi}} \exp\left(-\dfrac{[\ln((x-b)/\beta)-a]^2}{2\sigma^2}\right), & x > b \\ 0, & x \le b \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\sigma,b,\beta}(x) = \begin{cases} \Phi((\ln((x-b)/\beta)-a)/\sigma), & x > b \\ 0, & x \le b \end{cases}$$

Input Parameters

Name	Type	Description
method	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Generation method. The specific values are as follows: VSL_METHOD_SLOGNORMAL_ICDF VSL_METHOD_DLOGNORMAL_ICDF VSL_METHOD_SLOGNORMAL_ICDF_ACCURATE VSL_METHOD_DLOGNORMAL_ICDF_ACCURATE Inverse cumulative distribution function method
stream	**FORTRAN:** TYPE (VSL_STREAM_STATE), INTENT(IN) **C:** VSLStreamStatePtr	**FORTRAN:** Descriptor of the stream state structure. **C:** Pointer to the stream state structure
n	**FORTRAN:** INTEGER, INTENT(IN)	Number of random values to be generated

Name	Type	Description
	C: `int`	

a	**FORTRAN:** `REAL, INTENT(IN)` for `vsrnglognormal`
	`DOUBLE PRECISION, INTENT(IN)` for `vdrnglognormal`
	C: `float` for `vsRngLognormal`
	`double` for `vdRngLognormal`
		Average *a* of the subject normal distribution

sigma	**FORTRAN:** `REAL, INTENT(IN)` for `vsrnglognormal`
	`DOUBLE PRECISION, INTENT(IN)` for `vdrnglognormal`
	C: `float` for `vsRngLognormal`
	`double` for `vdRngLognormal`
		Standard deviation σ of the subject normal distribution

b	**FORTRAN:** `REAL, INTENT(IN)` for `vsrnglognormal`
	`DOUBLE PRECISION, INTENT(IN)` for `vdrnglognormal`
	C: `float` for `vsRngLognormal`
	`double` for `vdRngLognormal`
		Displacement *b*

Name	Type	Description

beta	**FORTRAN:** `REAL, INTENT(IN)` for `vsrnglognormal`
	`DOUBLE PRECISION, INTENT(IN)` for `vdrnglognormal`
	C: `float` for `vsRngLognormal`
	`double` for `vdRngLognormal`
		Scalefactor β

Output Parameters

Name	Type	Description

r	**FORTRAN:** `REAL, INTENT(OUT)` for `vsrnglognormal`
	`DOUBLE PRECISION, INTENT(OUT)` for `vdrnglognormal`
	C: `float*` for `vsRngLognormal`
	`double*` for `vdRngLognormal`
		Vector of *n* lognormally distributed random numbers

Return Values

`VSL_ERROR_OK`, `VSL_STATUS_OK`	Indicates no error, execution is successful.
`VSL_ERROR_NULL_PTR`	*stream* is a `NULL` pointer.
`VSL_ERROR_BAD_STREAM`	*stream* is not a valid random stream.
`VSL_ERROR_BAD_UPDATE`	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, `< 0` or `> nmax`.

VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Gumbel

Generates Gumbel distributed random values.

Syntax

Fortran:

```
status = vsrnggumbel( method, stream, n, r, a, beta )
status = vdrnggumbel( method, stream, n, r, a, beta )
```

C:

```
status = vsRngGumbel( method, stream, n, r, a, beta );
status = vdRngGumbel( method, stream, n, r, a, beta );
```

Description

This function generates Gumbel distributed random numbers with displacement a and scalefactor β, where $a, \beta \in R$; $\beta > 0$.

The probability density function is given by:

$$f_{a,\beta}(x) = \left\{\frac{1}{\beta}\exp\left(\frac{x-a}{\beta}\right)\exp(-\exp((x-a)/\beta)), -\infty < x < +\infty\right.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = 1 - \exp(-\exp((x-a)/\beta)), -\infty < x < +\infty$$

Input Parameters

Name	Type	Description
method	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Generation method. The specific values are as follows: VSL_METHOD_SGUMBEL_ICDF VSL_METHOD_DGUMBEL_ICDF Inverse cumulative distribution function method
stream	**FORTRAN:** TYPE (VSL_STREAM_STATE), INTENT(IN) **C:** VSLStreamStatePtr	**FORTRAN:** Descriptor of the stream state structure **C:** Pointer to the stream state structure
n	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Number of random values to be generated
a	**FORTRAN:** REAL, INTENT(IN) for vsrnggumbel DOUBLE PRECISION, INTENT(IN) for vdrnggumbel **C:** float for vsRngGumbel double for vdRngGumbel	Displacement *a*
beta	**FORTRAN:** REAL, INTENT(IN) for vsrnggumbel DOUBLE PRECISION, INTENT(IN) for vdrnggumbel **C:** float for vsRngGumbel double for vdRngGumbel	Scalefactor β

Output Parameters

Name	Type	Description
r	**FORTRAN:** REAL, INTENT(OUT) for vsrnggumbel DOUBLE PRECISION, INTENT(OUT) for vdrnggumbel **C:** float* for vsRngGumbel double* for vdRngGumbel	Vector of *n* random numbers with Gumbel distribution

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Gamma

Generates gamma distributed random values.

Syntax

Fortran:

```
status = vsrnggamma( method, stream, n, r, alpha, a, beta )
status = vdrnggamma( method, stream, n, r, alpha, a, beta )
```

C:

```
status = vsRngGamma( method, stream, n, r, alpha, a, beta );
status = vdRngGamma( method, stream, n, r, alpha, a, beta );
```

Description

This function generates random numbers with gamma distribution that has shape parameter α, displacement a, and scale parameter β, where α, β, and $a \in R$; $\alpha > 0$, $\beta > 0$.

The probability density function is given by:

$$f_{\alpha,a,\beta}(x) = \begin{cases} \frac{1}{\Gamma(\alpha)\beta^{\alpha}}(x-a)^{\alpha-1}e^{-(x-a)/\beta}, & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty$$

where $\Gamma(\alpha)$ is the complete gamma function.

The cumulative distribution function is as follows:

$$F_{\alpha,a,\beta}(x) = \begin{cases} \int_a^x \frac{1}{\Gamma(\alpha)\beta^{\alpha}}(y-a)^{\alpha-1}e^{-(y-a)/\beta}dy, & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty$$

Input Parameters

Name	Type	Description
method	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Generation method. The specific values are as follows: VSL_METHOD_SGAMMA_GNORM VSL_METHOD_DGAMMA_GNORM VSL_METHOD_SGAMMA_GNORM_ACCURATE VSL_METHOD_DGAMMA_GNORM_ACCURATE Acceptance/rejection method using random numbers with Gaussian distribution. See brief description of the method GNORM in Table 10-1
stream	**FORTRAN:** TYPE (VSL_STREAM_STATE), INTENT(IN) **C:** VSLStreamStatePtr	**FORTRAN:** Descriptor of the stream state structure **C:** Pointer to the stream state structure
n	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Number of random values to be generated
alpha	**FORTRAN:** REAL, INTENT(IN) for vsrnggamma DOUBLE PRECISION, INTENT(IN) for vdrnggamma **C:** float for vsRngGamma double for vdRngGamma	Shape α
a	**FORTRAN:** REAL, INTENT(IN) for vsrnggamma DOUBLE PRECISION, INTENT(IN) for vdrnggamma **C:** float for vsRngGamma	Displacement *a*

Name	Type	Description
	`double` for `vdRngGamma`	

beta	**FORTRAN:** `REAL`, `INTENT(IN)` for `vsrnggamma`
	`DOUBLE PRECISION`, `INTENT(IN)` for `vdrnggamma`
	C: `float` for `vsRngGamma`
	`double` for `vdRngGamma`
		Scalefactor β

Output Parameters

Name	Type	Description

r	**FORTRAN:** `REAL`, `INTENT(OUT)` for `vsrnggamma`
	`DOUBLE PRECISION`, `INTENT(OUT)` for `vdrnggamma`
	C: `float*` for `vsRngGamma`
	`double*` for `vdRngGamma`
		Vector of *n* random numbers with gamma distribution

Return Values

`VSL_ERROR_OK, VSL_STATUS_OK`	Indicates no error, execution is successful.
`VSL_ERROR_NULL_PTR`	*stream* is a `NULL` pointer.
`VSL_ERROR_BAD_STREAM`	*stream* is not a valid random stream.
`VSL_ERROR_BAD_UPDATE`	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, `< 0` or `> nmax`.
`VSL_ERROR_NO_NUMBERS`	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Beta

Generates beta distributed random values.

Syntax

Fortran:

```
status = vsrngbeta( method, stream, n, r, p, q, a, beta )
status = vdrngbeta( method, stream, n, r, p, q, a, beta )
```

C:

```
status = vsRngBeta( method, stream, n, r, p, q, a, beta );
status = vdRngBeta( method, stream, n, r, p, q, a, beta );
```

Description

This function generates random numbers with beta distribution that has shape parameters `p` and `q`, displacement `a`, and scale parameter β, where `p`, `q`, `a`, and β∈`R ; p > 0, q > 0, β > 0`.

The probability density function is given by:

$$f_{p,q,a,\beta}(x) = \begin{cases} \dfrac{1}{B(p,q)\beta^{p+q-1}}(x-a)^{p-1}(\beta+a-x)^{q-1}, & a \le x < a+\beta \\ 0, & x < a, x \ge a+\beta \end{cases},$$

where $B(p, q)$ is the complete beta function.

The cumulative distribution function is as follows:

$$F_{p,q,a,\beta}(x) = \begin{cases} 0, & x < a \\ \int_a^x \frac{1}{B(p,q)\beta^{p+q-1}} (y-a)^{p-1}(\beta + a - y)^{q-1}, & a \le x < a+\beta, - \\ 1, & x \ge a+\beta \end{cases}$$

Input Parameters

Name	Type	Description
method	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Generation method. The specific values are as follows: VSL_METHOD_SBETA_CJA VSL_METHOD_DBETA_CJA VSL_METHOD_SBETA_CJA_ACCURATE VSL_METHOD_DBETA_CJA_ACCURATE See brief description of the method CJA in Table 10-1
stream	**FORTRAN:** TYPE (VSL_STREAM_STATE), INTENT(IN) **C:** VSLStreamStatePtr	**FORTRAN:** Descriptor of the stream state structure **C:** Pointer to the stream state structure
n	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Number of random values to be generated
p	**FORTRAN:** REAL, INTENT(IN) for vsrngbeta DOUBLE PRECISION, INTENT(IN) for vdrngbeta **C:** float for vsRngBeta double for vdRngBeta	Shape *p*

Name	Type	Description

q	**FORTRAN:** `REAL`, `INTENT(IN)` for `vsrngbeta`
	`DOUBLE PRECISION`, `INTENT(IN)` for `vdrngbeta`
	C: `float` for `vsRngBeta`
	`double` for `vdRngBeta`
		Shape *q*

a	**FORTRAN:** `REAL`, `INTENT(IN)` for `vsrngbeta`
	`DOUBLE PRECISION`, `INTENT(IN)` for `vdrngbeta`
	C: `float` for `vsRngBeta`
	`double` for `vdRngBeta`
		Displacement *a*

beta	**FORTRAN:** `REAL`, `INTENT(IN)` for `vsrngbeta`
	`DOUBLE PRECISION`, `INTENT(IN)` for `vdrngbeta`
	C: `float` for `vsRngBeta`
	`double` for `vdRngBeta`
		Scalefactor β

Output Parameters

Name	Type	Description

r	**FORTRAN:** `REAL`, `INTENT(OUT)` for `vsrngbeta`
	`DOUBLE PRECISION`, `INTENT(OUT)` for `vdrngbeta`
	C: `float*` for `vsRngBeta`
	`double*` for `vdRngBeta`
		Vector of *n* random numbers with beta distribution

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Discrete Distributions

This section describes routines for generating random numbers with discrete distribution.

Uniform

Generates random numbers uniformly distributed over the interval [a, b).

Syntax

Fortran:

```
status = virnguniform( method, stream, n, r, a, b )
```

C:

```
status = viRngUniform( method, stream, n, r, a, b );
```

Description

This function generates random numbers uniformly distributed over the interval [a, b), where a, b are the left and right bounds of the interval respectively, and a, b∈Z; a < b.

The probability distribution is given by:

$$P(X = k) = \frac{1}{b - a}, k \in \{a, a + 1, \ldots, b - 1\}.$$

The cumulative distribution function is as follows:

$$F_{a,b}(x) = \begin{cases} 0, & x < a \\ \dfrac{\lfloor x - a + 1 \rfloor}{b - a}, & a \le x < b, x \in R \\ 1, & x \ge b \end{cases}$$

Input Parameters

Name	Type	Description
method	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Generation method; dummy and set to 0 in case of uniform distribution. The specific value is as follows: VSL_METHOD_IUNIFORM_STD Standard method. Currently there is only one method for this distribution generator.
stream	**FORTRAN:** TYPE (VSL_STREAM_STATE), INTENT(IN) **C:** VSLStreamStatePtr	**FORTRAN:** Descriptor of the stream state structure. **C:** Pointer to the stream state structure
n	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Number of random values to be generated
a	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Left interval bound a
b	**FORTRAN:** INTEGER, INTENT(IN)	Right interval bound b

Name	Type	Description
	C: `int`	

Output Parameters

Name	Type	Description
r	**FORTRAN:** `INTEGER, INTENT(OUT)` **C:** `int*`	Vector of *n* random numbers uniformly distributed over the interval `[a,b)`

Return Values

`VSL_ERROR_OK`, `VSL_STATUS_OK`	Indicates no error, execution is successful.
`VSL_ERROR_NULL_PTR`	*stream* is a `NULL` pointer.
`VSL_ERROR_BAD_STREAM`	*stream* is not a valid random stream.
`VSL_ERROR_BAD_UPDATE`	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, `< 0` or `> nmax`.
`VSL_ERROR_NO_NUMBERS`	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

UniformBits

Generates integer random values with uniform bit distribution.

Syntax

Fortran:

```
status = virnguniformbits( method, stream, n, r )
```

C:

```
status = viRngUniformBits( method, stream, n, r );
```

Description

This function generates integer random values with uniform bit distribution.The generators of uniformly distributed numbers can be represented as recurrence relations over integer values in modular arithmetic. Apparently, each integer can be treated as a vector of several bits. In a truly random generator, these bits are random, while in pseudorandom generators this randomness can be violated. For example, a well known drawback of linear congruential generators is that lower bits are less random than higher bits (for example, see [Knuth81]). For this reason, care should be taken when using this function. Typically, in a 32-bit *LCG* only 24 higher bits of an integer value can be considered random. See VSL Notes for details.

Input Parameters

Name	Type	Description
method	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Generation method; dummy and set to 0. The specific value is as follows: VSL_METHOD_IUNIFORMBITS_STD Standard method. Currently there is only one method for this distribution generator.
stream	**FORTRAN:** TYPE (VSL_STREAM_STATE), INTENT(IN) **C:** VSLStreamStatePtr	**FORTRAN:** Descriptor of the stream state structure. **C:** Pointer to the stream state structure
n	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Number of random values to be generated

Output Parameters

Name	Type	Description
r	**FORTRAN:** INTEGER, INTENT(OUT) **C:** unsigned int*	**FORTRAN:** Vector of *n* random integer numbers. If the *stream* was generated by a 64 or a 128-bit generator, each integer value is represented by two or four elements of *r* respectively. The number of bytes occupied by each integer is contained in the

Name	Type	Description
		field *wordsize* of the structure VSL_BRNG_PROPERTIES. The total number of bits that are actually used to store the value are contained in the field *nbits* of the same structure. See Advanced Service Routines for a more detailed discussion of VSLBrngProperties.
		C: Vector of *n* random integer numbers. If the *stream* was generated by a 64 or a 128-bit generator, each integer value is represented by two or four elements of *r* respectively. The number of bytes occupied by each integer is contained in the field *WordSize* of the structure VSLBrngProperties. The total number of bits that are actually used to store the value are contained in the field *NBits* of the same structure. See Advanced Service Routines for a more detailed discussion of VSLBrngProperties.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Bernoulli

Generates Bernoulli distributed random values.

Syntax

Fortran:

```
status = virngbernoulli( method, stream, n, r, p )
```

C:

```
status = viRngBernoulli( method, stream, n, r, p );
```

Description

This function generates Bernoulli distributed random numbers with probability p of a single trial success, where

$p \in R;\ 0 \le p \le 1$.

A variate is called Bernoulli distributed, if after a trial it is equal to 1 with probability of success p, and to 0 with probability $1 - p$.

The probability distribution is given by:

$P(X = 1) = p$

$P(X = 0) = 1 - p$

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, x < 0 \\ 1 - p, 0 \le x < 1, x \in R. \\ 1, x \ge 1 \end{cases}$$

Input Parameters

Name	Type	Description
method	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Generation method. The specific value is as follows: VSL_METHOD_IBERNOULLI_ICDF Inverse cumulative distribution function method.
stream	**FORTRAN:** TYPE (VSL_STREAM_STATE), INTENT(IN) **C:** VSLStreamStatePtr	**FORTRAN:** Descriptor of the stream state structure. **C:** Pointer to the stream state structure

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Number of random values to be generated
p	**FORTRAN:** DOUBLE PRECISION, INTENT(IN) **C:** double	Success probability *p* of a trial

Output Parameters

Name	Type	Description
r	**FORTRAN:** INTEGER, INTENT(OUT) **C:** int*	Vector of *n* Bernoulli distributed random values

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Geometric

Generates geometrically distributed random values.

Syntax

Fortran:

```
status = virnggeometric( method, stream, n, r, p )
```

C:

```
status = viRngGeometric( method, stream, n, r, p );
```

Description

This function generates geometrically distributed random numbers with probability p of a single trial success, where $p \in R;\ 0 < p < 1$.

A geometrically distributed variate represents the number of independent Bernoulli trials preceding the first success. The probability of a single Bernoulli trial success is p.

The probability distribution is given by:

$P(X = k) = p \cdot (1 - p)^k$, $k \in \{0,1,2, \ldots\}$.

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, & x < 0 \\ 1 - (1 - p)^{\lfloor x+1 \rfloor}, & 0 \geq x \end{cases} \qquad x \in R.$$

Input Parameters

Name	Type	Description
method	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Generation method. The specific value is as follows: VSL_METHOD_IGEOMETRIC_ICDF Inverse cumulative distribution function method.
stream	**FORTRAN:** TYPE (VSL_STREAM_STATE), INTENT(IN) **C:** VSLStreamStatePtr	**FORTRAN:** Descriptor of the stream state structure. **C:** Pointer to the stream state structure
n	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Number of random values to be generated

Name	Type	Description
p	**FORTRAN:** DOUBLE PRECISION, INTENT(IN) **C:** double	Success probability *p* of a trial

Output Parameters

Name	Type	Description
r	**FORTRAN:** INTEGER, INTENT(OUT) **C:** int*	Vector of *n* geometrically distributed random values

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Binomial

Generates binomially distributed random numbers.

Syntax

Fortran:

```
status = virngbinomial( method, stream, n, r, ntrial, p )
```

C:

```
status = viRngBinomial( method, stream, n, r, ntrial, p );
```

Description

This function generates binomially distributed random numbers with number of independent Bernoulli trials m, and with probability p of a single trial success, where $p \in R;\ 0 \le p \le 1$, $m \in N$.

A binomially distributed variate represents the number of successes in m independent Bernoulli trials with probability of a single trial success p.

The probability distribution is given by:

$$P(X = k) = C_m^k p^k (1 - p)^{m-k}, k \in \{0, 1, \ldots, m\}.$$

The cumulative distribution function is as follows:

$$F_{m,p}(x) = \begin{cases} 0, & x < 0 \\ \sum_{k=0}^{\lfloor x \rfloor} C_m^k p^k (1 - p)^{m-k}, & 0 \le x < m, x \in R \\ 1, & x \ge m \end{cases}$$

Input Parameters

Name	Type	Description
method	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Generation method. The specific value is as follows: VSL_METHOD_IBINOMIAL_BTPE See brief description of the BTPE method in Table 10-1 .

Name	Type	Description
stream	**FORTRAN:** TYPE (VSL_STREAM_STATE), INTENT(IN) **C:** VSLStreamStatePtr	**FORTRAN:** Descriptor of the stream state structure. **C:** Pointer to the stream state structure
n	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Number of random values to be generated
ntrials	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Number of independent trials m
p	**FORTRAN:** DOUBLE PRECISION, INTENT(IN) **C:** double	Success probability *p* of a single trial

Output Parameters

Name	Type	Description
r	**FORTRAN:** INTEGER, INTENT(OUT) **C:** int*	Vector of *n* binomially distributed random values

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.

VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Hypergeometric

Generates hypergeometrically distributed random values.

Syntax

Fortran:

```
status = virnghypergeometric( method, stream, n, r, l, s, m )
```

C:

```
status = viRngHypergeometric( method, stream, n, r, l, s, m );
```

Description

This function generates hypergeometrically distributed random values with lot size l, size of sampling s, and number of marked elements in the lot m, where l, m, $s \in \mathbb{N} \cup \{0\}$; $l \geq \max(s, m)$.

Consider a lot of l elements comprising m "marked" and $l-m$ "unmarked" elements. A trial sampling without replacement of exactly s elements from this lot helps to define the hypergeometric distribution, which is the probability that the group of s elements contains exactly k marked elements.

The probability distribution is given by:)

$$P(X = k) = \frac{C_m^k C_{l-m}^{s-k}}{C_l^s}$$

, $k \in$ {max(0, s + m - l), ..., min(s, m)}

The cumulative distribution function is as follows:

$$F_{l,s,m}(x) = \begin{cases} 0, & x < \max(0, s + m - l) \\ \sum_{k=\max(0,s+m-l)}^{\lfloor x \rfloor} \frac{C_m^k C_{l-m}^{s-k}}{C_l^s}, & \max(0, s + m - l) \le x \le \min(s, m) \\ 1, & x > \min(s, m) \end{cases}$$

Input Parameters

Name	Type	Description
method	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Generation method. The specific value is as follows: VSL_METHOD_IHYPERGEOMETRIC_H2PE See brief description of the H2PE method in Table 10-1
stream	**FORTRAN:** TYPE (VSL_STREAM_STATE), INTENT(IN) **C:** VSLStreamStatePtr	**FORTRAN:** Descriptor of the stream state structure. **C:** Pointer to the stream state structure
n	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Number of random values to be generated
l	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Lot size l
s	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Size of sampling without replacement s
m	**FORTRAN:** INTEGER, INTENT(IN)	Number of marked elements m

Name	Type	Description
	C: `int`	

Output Parameters

Name	Type	Description
r	**FORTRAN:** `INTEGER, INTENT(OUT)`	Vector of *n* hypergeometrically distributed random values
	C: `int*`	

Return Values

`VSL_ERROR_OK`, `VSL_STATUS_OK`	Indicates no error, execution is successful.
`VSL_ERROR_NULL_PTR`	*stream* is a `NULL` pointer.
`VSL_ERROR_BAD_STREAM`	*stream* is not a valid random stream.
`VSL_ERROR_BAD_UPDATE`	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, `< 0` or `> nmax`.
`VSL_ERROR_NO_NUMBERS`	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Poisson

Generates Poisson distributed random values.

Syntax

Fortran:

```
status = virngpoisson( method, stream, n, r, lambda )
```

C:

```
status = viRngPoisson( method, stream, n, r, lambda );
```

Description

This function generates Poisson distributed random numbers with distribution parameter λ, where $\lambda \in R;\ \lambda > 0$.

The probability distribution is given by:

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!},$$

$k \in \{0, 1, 2, \ldots\}$.

The cumulative distribution function is as follows:

$$F_\lambda(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda^k e^{-\lambda}}{k!}, & x \geq 0 \\ 0, & x < 0 \end{cases}, x \in R$$

Input Parameters

Name	Type	Description
method	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Generation method. The specific values are as follows: VSL_METHOD_IPOISSON_PTPE VSL_METHOD_IPOISSON_POISNORM See brief description of the PTPE and POISNORM methods in Table 10-1 .
stream	**FORTRAN:** TYPE (VSL_STREAM_STATE), INTENT(IN) **C:** VSLStreamStatePtr	**FORTRAN:** Descriptor of the stream state structure. **C:** Pointer to the stream state structure

Name	Type	Description
n	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Number of random values to be generated
lambda	**FORTRAN:** DOUBLE PRECISION, INTENT(IN) **C:** double	Distribution parameter λ

Output Parameters

Name	Type	Description
r	**FORTRAN:** INTEGER, INTENT(OUT) **C:** int*	Vector of *n* Poisson distributed random values

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

PoissonV

Generates Poisson distributed random values with varying mean.

Syntax

Fortran:

```
status = virngpoissonv( method, stream, n, r, lambda )
```

C:

```
status = viRngPoissonV( method, stream, n, r, lambda );
```

Description

This function generates n Poisson distributed random numbers x_i(i = 1, ..., n) with distribution parameter λ_i, where $\lambda_i \in R$; $\lambda_i > 0$.

The probability distribution is given by:

$$P(X_i = k) = \frac{\lambda_i^k \exp(-\lambda_i)}{k!}, k \in \{0, 1, 2, \ldots\}.$$

The cumulative distribution function is as follows:

$$F_{\lambda_i}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda_i^k e^{-\lambda_i}}{k!}, & x \geq 0 \\ 0, & x < 0 \end{cases}, x \in R$$

Input Parameters

Name	Type	Description
method	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Generation method. The specific value is as follows: VSL_METHOD_IPOISSONV_POISNORM See brief description of the POISNORM method in Table 10-1
stream	**FORTRAN:** TYPE (VSL_STREAM_STATE), INTENT(IN) **C:** VSLStreamStatePtr	**FORTRAN:** Descriptor of the stream state structure. **C:** Pointer to the stream state structure
n	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Number of random values to be generated
lambda	**FORTRAN:** DOUBLE PRECISION, INTENT(IN) **C:** double*	Array of n distribution parameters λ_i

Output Parameters

Name	Type	Description
r	**FORTRAN:** INTEGER, INTENT(OUT) **C:** int*	Vector of n Poisson distributed random values

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_ERROR_BAD_STREAM	*stream* is not a valid random stream.

VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

NegBinomial

Generates random numbers with negative binomial distribution.

Syntax

Fortran:

```
status = virngnegbinomial( method, stream, n, r, a, p )
```

C:

```
status = viRngNegbinomial( method, stream, n, r, a, p );
```

Description

This function generates random numbers with negative binomial distribution and distribution parameters a and p, where p, $a \in R$; $0 < p < 1$; $a > 0$.

If the first distribution parameter $a \in N$, this distribution is the same as Pascal distribution. If $a \in N$, the distribution can be interpreted as the expected time of a-th success in a sequence of Bernoulli trials, when the probability of success is p.

The probability distribution is given by:

$$P(X = k) = C_{a+k-1}^{k} p^{a}(1 - p)^{k}, k \in \{0, 1, 2, \ldots\}.$$

The cumulative distribution function is as follows:

$$F_{a,p}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} C_{a+k-1}^{k} p^{a}(1-p)^{k}, & x \geq 0 \\ 0, & x < 0 \end{cases}, x \in R$$

Input Parameters

Name	Type	Description
method	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Generation method. The specific value is as follows: VSL_METHOD_INEGBINOMIAL_NBAR See brief description of the NBAR method in Table 10-1
stream	**FORTRAN:** TYPE (VSL_STREAM_STATE), INTENT(IN) **C:** VSLStreamStatePtr	**FORTRAN:** descriptor of the stream state structure. **C:** pointer to the stream state structure
n	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Number of random values to be generated
a	**FORTRAN:** DOUBLE PRECISION, INTENT(IN) **C:** double	The first distribution parameter *a*
p	**FORTRAN:** DOUBLE PRECISION, INTENT(IN) **C:** double	The second distribution parameter p

Output Parameters

Name	Type	Description
r	**FORTRAN:** INTEGER, INTENT(OUT) **C:** int*	Vector of *n* random values with negative binomial distribution.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	*stream* is a NULL pointer.
VSL_ERROR_BAD_STREAM	*stream* is not a valid random stream.
VSL_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

Advanced Service Routines

This section describes service routines for registering a user-designed basic generator (RegisterBrng) and for obtaining properties of the previously registered basic generators (GetBrngProperties). See VSL Notes ("Basic Generators" section of VSL Structure chapter) for substantiation of the need for several basic generators including user-defined BRNGs.

Data types

The Avdanced Service routines refer to a structure defining the properties of the basic generator. This structure is described in Fortran as follows:

```
TYPE VSL_BRNG_PROPERTIES
    INTEGER streamstatesize
    INTEGER nseeds
    INTEGER includeszero
    INTEGER wordsize
    INTEGER nbits
    INTEGER nitstream
    INTEGER sbrng
    INTEGER dbrng
    INTEGER ibrng
END TYPE VSL_BRNG_PROPERTIES
```

The C version is as follows:

```
typedef struct _VSLBRngProperties {
       int StreamStateSize;
       int NSeeds;
       int IncludesZero;
       int WordSize;
       int NBits;
       InitStreamPtr InitStream;
       sBRngPtr sBRng;
       dBRngPtr dBRng;
       iBRngPtr iBRng;
} VSLBRngProperties;
```

The following table provides brief descriptions of the fields engaged in the above structure:

Table 10-12 Field Descriptions

Field	Short Description
FORTRAN: `streamstatesize` **C:** `StreamStateSize`	The size, in bytes, of the stream state structure for a given basic generator.
FORTRAN: `nseeds` **C:** `NSeeds`	The number of 32-bit initial conditions (seeds) necessary to initialize the stream state structure for a given basic generator.
FORTRAN: `includeszero` **C:** `IncludesZero`	Flag value indicating whether the generator can produce a random 0.
FORTRAN: `wordsize` **C:** `WordSize`	Machine word size, in bytes, used in integer-value computations. Possible values: 4, 8, and 16 for 32, 64, and 128-bit generators, respectively.
FORTRAN: `nbits` **C:** `NBits`	The number of bits required to represent a random value in integer arithmetic. Note that, for instance, 48-bit random values are stored to 64-bit (8 byte) memory locations. In this case, `wordsize`/`WordSize` is equal to `8` (number of bytes used to store the random value), while `nbits`/`NBits` contains the actual number of bits occupied by the value (in this example, `48`).
FORTRAN: `initstream` **C:** `InitStream`	Contains the pointer to the initialization routine of a given basic generator.
FORTRAN: `sbrng` **C:** `sBRng`	Contains the pointer to the basic generator of single precision real numbers uniformly distributed over the interval (*a*,*b*) (`real` in FORTRAN and `float` in C).
FORTRAN: `dbrng` **C:** `dBRng`	Contains the pointer to the basic generator of double precision real numbers uniformly distributed over the interval (*a*,*b*) (`double PRECISION` in FORTRAN and `double` in C).
FORTRAN: `ibrng` **C:** `iBRng`	Contains the pointer to the basic generator of integer numbers with uniform bit distribution[1] (`INTEGER` in FORTRAN and `unsigned int` in C).

[1]A specific generator that permits operations over single bits and bit groups of random numbers.

RegisterBrng

Registers user-defined basic generator.

Syntax

Fortran:

```
brng = vslregisterbrng( properties )
```

C:

```
brng = vslRegisterBrng( &properties );
```

Description

An example of a registration procedure can be found in the respective directory of VSL examples.

Input Parameters

Name	Type	Description
properties	**FORTRAN:** `TYPE(VSL_BRNG_PROPERTIES), INTENT(IN)` **C:** `VSLBrngProperties*`	Pointer to the structure containing properties of the basic generator to be registered

Output Parameters

Name	Type	Description
brng	**FORTRAN:** `INTEGER, INTENT(OUT)` **C:** `int`	Number (index) of the registered basic generator; used for identification. Negative values indicate the registration error.

Return Values

`VSL_ERROR_OK`, `VSL_STATUS_OK`	Indicates no error, execution is successful.
`VSL_ERROR_BRNG_TABLE_FULL`	Registration cannot be completed due to lack of free entries in the table of registered BRNGs.
`VSL_ERROR_BAD_STREAM_STATE_SIZE`	Bad value in `StreamStateSize` field.

VSL_ERROR_BAD_WORD_SIZE	Bad value in WordSize field.
VSL_ERROR_BAD_NSEEDS	Bad value in NSeeds field.
VSL_ERROR_BAD_NBITS	Bad value in NBits field.
VSL_ERROR_NULL_PTR	At least one of the fields iBrng, dBrng, sBrng or InitStream is a NULL pointer.

GetBrngProperties

Returns structure with properties of a given basic generator.

Syntax

Fortran:

```
status = vslgetbrngproperties( brng, properties )
```

C:

```
status = vslGetBrngProperties( brng, &properties );
```

Input Parameters

Name	Type	Description
brng	**FORTRAN:** INTEGER, INTENT(IN) **C:** int	Number (index) of the registered basic generator; used for identification. See specific values in Table 10-2 . Negative values indicate the registration error.

Output Parameters

Name	Type	Description
properties	**FORTRAN:** TYPE(VSL_BRNG_PROPERTIES), INTENT(OUT) **C:** VSLBrngProperties*	Pointer to the structure containing properties of the generator with number *brng*

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_INVALID_BRNG_INDEX	BRNG index is invalid.

Formats for User-Designed Generators

To register a user-designed basic generator using RegisterBrng function, you need to pass the pointer iBrng to the integer-value implementation of the generator; the pointers sBrng and dBrng to the generator implementations for single and double precision values, respectively; and pass the pointer InitStream to the stream initialization routine. See below recommendations on defining such functions with input and output arguments. An example of the registration procedure for a user-designed generator can be found in the respective directory of VSL examples.

The respective pointers are defined as follows:

```
typedef   int (*InitStreamPtr)( int method, void * stream, int n, const
unsigned int params[] );

typedef   int (*sBRngPtr)( void * stream, int n, float r[], float a, float
b );

typedef   int (*dBRngPtr)( void * stream, int n, double r[], double a, double
 b );

typedef    int (*iBRngPtr)( void * stream, int n, unsigned int r[] );
```

InitStream

C:

```
int MyBrngInitStream( int method, VSLStreamStatePtr stream,

   int n, const unsigned int params[] );

{

      /* Initialize the stream */

      ...

} /* MyBrngInitStream */
```

Description

The initialization routine of a user-designed generator must initialize *stream* according to the specified initialization *method*, initial conditions *params* and the argument *n*. The value of *method* determines the initialization method to be used.

- If *method* is equal to 1, the initialization is by the standard generation method, which must be supported by all basic generators. In this case the function assumes that the *stream* structure was not previously initialized. The value of *n* is used as the actual number of 32-bit values passed as initial conditions through *params*. Note, that the situation when the actual number of initial conditions passed to the function is not sufficient to initialize the generator is not an error. Whenever it occurs, the basic generator must initialize the missing conditions using default settings.
- If *method* is equal to 2, the generation is by the leapfrog method, where *n* specifies the number of computational nodes (independent streams). Here the function assumes that the *stream* was previously initialized by the standard generation method. In this case *params* contains only one element, which identifies the computational node. If the generator does not support the leapfrog method, the function must return the error code VSL_ERROR_LEAPFROG_UNSUPPORTED.
- If *method* is equal to 3, the generation is by the block-splitting method. Same as above, the *stream* is assumed to be previously initialized by the standard generation method; *params* is not used, *n* identifies the number of skipped elements. If the generator does not support the block-splitting method, the function must return the error code VSL_ERROR_SKIPAHEAD_UNSUPPORTED.

For a more detailed description of the leapfrog and the block-splitting methods, refer to the description of LeapfrogStream and SkipAheadStream, respectively.

Stream state structure is individual for every generator. However, each structure has a number of fields that are the same for all the generators:

C:

```
typedef struct
{
    unsigned int Reserved1[2];
    unsigned int Reserved2[2];
    [fields specific for the given generator]
} MyStreamState;
```

The fields *Reserved1* and *Reserved2* are reserved for private needs only, and must not be modified by the user. When including specific fields into the structure, follow the rules below:

- The fields must fully describe the current state of the generator. For example, the state of a linear congruential generator can be identified by only one initial condition;
- If the generator can use both the leapfrog and the block-splitting methods, additional fields should be introduced to identify the independent streams. For example, in *LCG*(*a*, *c*, *m*), apart from the initial conditions, two more fields should be specified: the value of the multiplier a^k and the value of the increment $(a^k-1)c/(a-1)$.

For a more detailed discussion, refer to [Knuth81], and [Gentle98]. An example of the registration procedure can be found in the respective directory of VSL examples.

iBRng

C:

```
void iMyBrng( VSLStreamStatePtr stream, int n,

              unsigned int r[] )
{
 int i; /* Loop variable */
 /* Generating integer random numbers */
 /* Pay attention to word size needed to
 store only random number */
 for( i = 0; i < n; i++)
 {
r[i] = ...;
 }
 /* Update stream state */
 ...
 return errcode;
} /* iMyBrng */
```

NOTE. When using 64 and 128-bit generators, consider digit capacity to store the numbers to the random vector *r* correctly. For example, storing one 64-bit value requires two elements of *r*, the first to store the lower 32 bits and the second to store the higher 32 bits. Similarly, use 4 elements of *r* to store a 128-bit value.

sBRng

C:

```
void sMyBrng( VSLStreamStatePtr stream, int n, float r[],
              float a, float b )
{
 int   i;    /* Loop variable */
 /* Generating float (a,b) random numbers */
 for ( i = 0; i < n; i++ )
 {
r[i] = ...;
 }
 /* Update stream state */
 ...
 return errcode;
} /* sMyBrng */
```

dBRng

C:

```
void dMyBrng( VSLStreamStatePtr stream, int n, double r[],
              double a, double b )
{
     int i;   /* Loop variable */
 /* Generating double (a,b) random numbers */
 for ( i = 0; i < n; i++ )
 {
r[i] = ...;
 }
 /* Update stream state */
 ...
 return errcode;
} /* dMyBrng */
```

Convolution and Correlation

Overview

VSL provides a set of routines intended to perform linear convolution and correlation transformations for single and double precision data.

For correct definition of implemented operations, see Mathematical Notation and Definitions section.

The current implementation provides:

- Fourier algorithms for one-dimensional single and double precision data
- Fourier algorithms for multi-dimensional single and double precision data
- Direct algorithms for one-dimensional single and double precision data.
- Direct algorithms for multi-dimensional single and double precision data.

One-dimensional algorithms cover the following functions from the IBM* ESSL library:

```
SCONF, SCORF

SCOND, SCORD

SDCON, SDCOR

DDCON, DDCOR

SDDCON, SDDCOR.
```

Special wrappers are designed to simulate these ESSL functions. The wrappers are provided as sample sources for FORTRAN and C. To reuse them, use the following directories:

```
${MKL}/examples/vslc/essl/vsl_wrappers

${MKL}/examples/vslf/essl/vsl_wrappers
```

Additionally, you can browse the examples demonstrating the calculation of the ESSL functions through the wrappers. You can find the examples in the following directories:

```
${MKL}/examples/vslc/essl

${MKL}/examples/vslf/essl
```

Convolution and correlation API provides interfaces for FORTRAN-90 and C/89 languages. You may use the C/89 interface also with later versions of C or C++, or FORTRAN-90 interface with programs written in FORTRAN-95. Note that there is no FORTRAN-77 support.

For users of the C/C++ and FORTRAN languages, the mkl_vsl.h and mkl_vsl.fi headers are provided. Both header files are found under the directory:

```
${MKL}/include
```

See more details about the FORTRAN header in Random Number Generators section of this chapter.

Convolution and correlation API is implemented through task objects, or tasks. Task object is a data structure, or descriptor, which holds parameters that determine the specific convolution or correlation operation. Such parameters may be precision, type, and number of dimensions of user data, an identifier of the computation algorithm to be used, shapes of data arrays, and so on.

All Intel MKL convolution and correlation routines process task objects in one way or another: either create a new task descriptor, change the parameter settings, compute mathematical results of the convolution or correlation using the stored parameters, or perform other operations. Accordingly, all routines are split into the following groups:

Task Constructors - routines that create a new task object descriptor and set up most common parameters.

Task Editors - routines that can set or modify some parameter settings in the existing task descriptor.

Task Execution Routines - compute results of the convolution or correlation operation over the actual input data, using the operation parameters held in the task descriptor.

Task Copy - routines used to make several copies of the task descriptor.

Task Destructors - routines that delete task objects and free the memory.

When the task is executed or copied for the first time, a special process runs which is called task commitment. During this process, consistency of task parameters is checked and the required work data are prepared. If the parameters are consistent, the task is tagged as committed successfully. The task remains committed until you edit its parameters. Hence, the task can be executed multiple times after a single commitment process. Since the task commitment process may include costly intermediate calculations such as preparation of Fourier transform of input data, launching the process only once can help speed up overall performance.

Naming Conventions

The names of FORTRAN routines in the convolution and correlation API are written in lowercase, while the names of FORTRAN types and constants are written in uppercase. The names are not case-sensitive.

In C, the names of routines, types, and constants are case-sensitive and can be lowercase and uppercase.

The names of routines have the following structure:

`vsl[`*`datatype`*`]{Conv|Corr}<base name>` for C-interface

`vsl[`*`datatype`*`]{conv|corr}<base name>` for FORTRAN-interface

where `vsl` is a prefix indicating that the routine belongs to Vector Statistical Library of Intel® MKL.

The field `[`*`datatype`*`]` is optional. If present, the symbol specifies the type of the input and output data and can be either `s` (for single precision real type) or `d` (for double precision real type).

The prefix `Conv` or `Corr` specifies whether the routine refers to convolution or correlation task, respectively.

The `<base name>` field specifies a particular functionality that the routine is designed for, for example, `NewTask`, `DeleteTask`.

NOTE. In this document, routines are often referred to by their base name when this does not lead to ambiguity. In the routine reference, the full name is always used in prototypes and code examples.

Data Types

All convolution or correlation routines use the following types for specifying data objects:

Type	Data Object
FORTRAN: `TYPE(VSL_CONV_TASK)` **C:** `VSLConvTaskPtr`	Pointer to a task descriptor for convolution
FORTRAN: `TYPE(VSL_CORR_TASK)` **C:** `VSLCorrTaskPtr`	Pointer to a task descriptor for correlation
FORTRAN: `REAL(KIND=4)` **C:** `float`	Input/output user data in single precision
FORTRAN: `REAL(KIND=8)` **C:** `double`	Input/output user data in double precision
FORTRAN: `INTEGER` **C:** `int`	All other data

Generic integer type (without specifying the byte size) is used for all integer data.

NOTE. The actual size of the generic integer type is platform-dependent. The appropriate byte size for integers must be chosen at the stage of compiling your software.

Parameters

Basic parameters held by the task descriptor are assigned values when the task object is created, copied, or modified by task editors. Parameters of the correlation or convolution task are initially set up by task constructors when the task object is created. Parameter changes or additional settings are made by task editors. More parameters which define location of the data being convolved need to be specified when the task execution routine is invoked.

According to how the parameters are passed or assigned values, all of them can be categorized as either explicit (directly passed as routine parameters when a task object is created or executed) or optional (assigned some default or implicit values during task construction).

The following table lists all applicable parameters used in the Intel MKL convolution and correlation API.

Table 10-13 Convolution and Correlation Task Parameters

Name	Category	Type	Default Value Label	Description
job	explicit	integer	Implied by the constructor name	Specifies whether the task relates to convolution or correlation
type	explicit	integer	Implied by the constructor name	Specifies the type (real or complex) of the input/output data. Set to real in the current version.
precision	explicit	integer	Implied by the constructor name	Specifies precision (single or double) of the input/output data to be provided in arrays *x*,*y*,*z*.
kind	optional	integer	"linear"	Specifies whether the task relates to computing linear or circular convolution/correlation
mode	explicit	integer	None	Specifies whether the convolution/correlation computation should be done via Fourier transforms, or by a direct method, or by automatically choosing between the two. See SetMode for the list of named constants for this parameter.
method	optional	integer	"auto"	Hints at a particular computation method if several methods are available for the given *mode*. Setting this parameter to "auto" means that software will choose the best available method.

Name	Category	Type	Default Value Label	Description
internal_precision	optional	integer	Set equal to the value of *precision*	Specifies precision of internal calculations. Can enforce double precision calculations even when input/output data are single precision. See `SetInternalPrecision` for the list of named constants for this parameter.
dims	explicit	integer	None	Specifies the rank (number of dimensions) of the user data provided in arrays *x*,*y*,*z*. Can be in the range from 1 to 7.
x,*y*	explicit	real arrays	None	Specify input data arrays. See Data Allocation for more information.
z	explicit	real array	None	Specifies output data array. See Data Allocation for more information.
xshape, *yshape*, *zshape*	explicit	integer arrays	None	Define shapes of the arrays *x*, *y*, *z*. See Data Allocation for more information.
xstride, *ystride*, *zstride*	explicit	integer arrays	None	Define strides within arrays *x*, *y*, *z*, that is specify the physical location of the input and output data in these arrays. See Data Allocation for more information.
start	optional	integer array	Undefined	Defines the first element of the mathematical result that will be stored to output array *z*. See `SetStart` and Data Allocation for more information.
decimation	optional	integer array	Undefined	Defines how to thin out the mathematical result that will be stored to output array *z*. See `SetDecimation` and Data Allocation for more information.

Users of the C or C++ language may pass the NULL pointer instead of either or all of the parameters *xstride*, *ystride*, or *zstride* for multi-dimensional calculations. In this case, the software assumes the dense data allocation for the arrays *x*, *y*, or *z* due to the FORTRAN-style "by columns" representation of multi-dimensional arrays.

Task Status and Error Reporting

Task status is an integer value which is zero if no error has been detected while processing the task, or a specific non-zero error code otherwise. Negative status values are used for errors, and positive values are reserved for warnings.

An error can be caused by bad parameter values, system fault like memory allocation failure, or can be an internal error self-detected by the software.

Each task descriptor contains the current status of the task. When creating a task object, constructor assigns the `VSL_STATUS_OK` status to the task. When processing the task afterwards, other routines such as editors of executors can change the task status if an error occurs and write a corresponding error code into the task status field.

Note that at the stage of creating a task or editing its parameters, the set of parameters may be inconsistent. The parameter consistency check is only performed during the task commitment operation which is implicitly invoked before task execution or task copying. If an error is detected at this stage, task execution or task copying is terminated and the task descriptor saves the corresponding error code. Once an error occurs, any further attempts to process that task descriptor will be terminated and the task will keep the same error code.

Normally, every convolution or correlation function (except `DeleteTask`) returns the status assigned to the task while performing the function operation.

The status codes are given symbolic names defined in the respective header files. For C/C++, these names are defined as macros via `#define` statements, and for FORTRAN as integer constants via `PARAMETER` operators.

If there is no error, the `VSL_STATUS_OK` is returned, which is defined as zero:

```
C/C++:              #define, VSL_STATUS_OK 0
F90/F95:            INTEGER,PARAMETER:: VSL_STATUS_OK = 0
```

In case of an error, a non-zero error code is returned, which signals about the origin of the failure. The following status codes for the convolution/correlation error codes are pre-defined in the header files for both C/C++ and FORTRAN languages.

Convolution/Correlation Status Codes and Messages

Status Code	Message
`VSL_CC_ERROR_NOT_IMPLEMENTED`,	Requested functionality is not implemented.

Status Code	Message
VSL_CC_ERROR_ALLOCATION_FAILURE	Memory allocation failure.
VSL_CC_ERROR_BAD_DESCRIPTOR	Task descriptor is corrupted.
VSL_CC_ERROR_SERVICE_FAILURE	A service function has failed.
VSL_CC_ERROR_EDIT_FAILURE	Failure while editing the task.
VSL_CC_ERROR_EDIT_PROHIBITED	You cannot edit this parameter.
VSL_CC_ERROR_COMMIT_FAILURE	Task committment has failed.
VSL_CC_ERROR_COPY_FAILURE	Failure while copying the task.
VSL_CC_ERROR_DELETE_FAILURE	Failure while deleting the task.
VSL_CC_ERROR_BAD_ARGUMENT	Bad argument or task parameter.
VSL_CC_ERROR_JOB	Bad parameter: *job*.
SL_CC_ERROR_KIND	Bad parameter: *kind*.
VSL_CC_ERROR_MODE	Bad parameter: *mode*.
VSL_CC_ERROR_METHOD	Bad parameter: *method*.
VSL_CC_ERROR_TYPE	Bad parameter: *type*.
VSL_CC_ERROR_EXTERNAL_PRECISION	Bad parameter: *external_precision*.
VSL_CC_ERROR_INTERNAL_PRECISION	Bad parameter: *internal_precision*.
VSL_CC_ERROR_PRECISION	Incompatible external/internal precisions.
VSL_CC_ERROR_DIMS	Bad parameter: *dims*.
VSL_CC_ERROR_XSHAPE	Bad parameter: *xshape*.
VSL_CC_ERROR_YSHAPE	Bad parameter: *yshape*.

Status Code	Message
	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or >nmax.
VSL_CC_ERROR_ZSHAPE	Bad parameter: *zshape*.
VSL_CC_ERROR_XSTRIDE	Bad parameter: *xstride*.
VSL_CC_ERROR_YSTRIDE	Bad parameter: *ystride*.
VSL_CC_ERROR_ZSTRIDE	Bad parameter: *zstride*.
VSL_CC_ERROR_X	Bad parameter: *x*.
VSL_CC_ERROR_Y	Bad parameter: *y*.
VSL_CC_ERROR_Z	Bad parameter: *z*.
VSL_CC_ERROR_START	Bad parameter: *start*.
VSL_CC_ERROR_DECIMATION	Bad parameter: *decimation*.
VSL_CC_ERROR_OTHER	Some other error.

Task Constructors

Task constructors are routines intended for creating a new task descriptor and setting up basic parameters. This means that no additional parameter adjustment is typically required and other routines can use the task object.

Intel® MKL implementation of the convolution and correlation API provides two different forms of constructors: a general form and an X-form. X-form constructors work in the same way as the general form but also assign particular data to the first operand vector used in convolution or correlation operation (stored in array *x*).

Using X-form constructors is recommended when you need to compute multiple convolutions or correlations with the same data vector held in array *x* against different vectors held in array *y*. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

For each constructor routine there is also an associated one-dimensional version which exploits the algorithmic and computational benefits provided by the simplicity of the data structures for one-dimensional case.

NOTE. If constructor fails to create a task descriptor, it returns `NULL` task pointer.

The Table 10-14 lists available task constructors:

Table 10-14 Task Constructors

Routine	Description
`NewTask`	Creates a new convolution or correlation task descriptor for a multidimensional case.
`NewTask1D`	Creates a new convolution or correlation task descriptor for a one-dimensional case.
`NewTaskX`	Creates a new convolution or correlation task descriptor as an X-form for a multidimensional case.
`NewTaskX1D`	Creates a new convolution or correlation task descriptor as an X-form for a one-dimensional case.

NewTask

Creates a new convolution or correlation task descriptor for multidimensional case.

Syntax

Fortran:

```
status = vslsconvnewtask(task, mode, dims, xshape, yshape, zshape)
status = vsldconvnewtask(task, mode, dims, xshape, yshape, zshape)
status = vslscorrnewtask(task, mode, dims, xshape, yshape, zshape)
status = vsldcorrnewtask(task, mode, dims, xshape, yshape, zshape)
```

C:

```
status = vslsConvNewTask(task, mode, dims, xshape, yshape, zshape);
status = vsldConvNewTask(task, mode, dims, xshape, yshape, zshape);
status = vslsCorrNewTask(task, mode, dims, xshape, yshape, zshape);
status = vsldCorrNewTask(task, mode, dims, xshape, yshape, zshape);
```

Description

Each `NewTask` constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see Table 10-13).

The parameters *xshape*, *yshape*, and *zshape* define the shapes of the input and output data provided by the arrays *x*, *y*, and *z*, respectively. Each shape parameter is an array of integers with its length equal to the value of *dims*. You explicitly assign the shape parameters when calling the constructor. If the value of the parameter *dims* is 1, then *xshape*, *yshape*, *zshape* are equal to the number of elements read from the arrays *x* and *y* or stored to the array *z*. Note that values of shape parameters may differ from physical shapes of arrays *x*, *y*, and *z* if non-trivial strides are assigned.

If constructor fails to create a task descriptor, it returns `NULL` task pointer.

Input Parameters

Name	Type	Description
mode	**FORTRAN:** `INTEGER` **C:** `int`	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table 10-16 for a list of possible values.
dims	**FORTRAN:** `INTEGER` **C:** `int`	Rank of user data. Specifies number of dimensions for the input and output arrays *x*, *y*, and *z* used during the execution stage. Must be in the range from 1 to 7. The value is explicitly assigned by the constructor.
xshape	**FORTRAN:** `INTEGER, DIMENSION(*)` **C:** `int[]`	Defines the shape of the input data for the source array *x*. See Data Allocation for more information.

Name	Type	Description
yshape	**FORTRAN:** INTEGER, DIMENSION(*) **C:** int[]	Defines the shape of the input data for the source array *y*. See Data Allocation for more information.
zshape	**FORTRAN:** INTEGER, DIMENSION(*) **C:** int[]	Defines the shape of the output data to be stored in array *z*. See Data Allocation for more information.

Output Parameters

Name	Type	Description
task	**FORTRAN:** TYPE(VSL_CONV_TASK) for vslsconvnewtask, vsldconvnewtask TYPE(VSL_CORR_TASK) for vslscorrnewtask, vsldcorrnewtask **C:** VSLConvTaskPtr* for vslsConvNewTask, vsldConvNewTask VSLCorrTaskPtr* for vslsCorrNewTask, vsldCorrNewTask	Pointer to the task descriptor if created successfully or NULL pointer otherwise.
status	**FORTRAN:** INTEGER **C:** int	Set to VSL_STATUS_OK if the task is created successfully or set to non-zero error code otherwise.

NewTask1D

Creates a new convolution or correlation task descriptor for one-dimensional case.

Syntax

Fortran:

```
status = vslsconvnewtask1d(task, mode, xshape, yshape, zshape)
status = vsldconvnewtask1d(task, mode, xshape, yshape, zshape)
status = vslscorrnewtask1d(task, mode, xshape, yshape, zshape)
status = vsldcorrnewtask1d(task, mode, xshape, yshape, zshape)
```

C:

```
status = vslsConvNewTask1D(task, mode, xshape, yshape, zshape);
status = vsldConvNewTask1D(task, mode, xshape, yshape, zshape);
status = vslsCorrNewTask1D(task, mode, xshape, yshape, zshape);
status = vsldCorrNewTask1D(task, mode, xshape, yshape, zshape);
```

Description

Each `NewTask1D` constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see Table 10-13). Unlike `NewTask`, these routines represent a special one-dimensional version of the constructor which assumes that the value of the parameter *dims* is 1. The parameters *xshape*, *yshape*, and *zshape* are equal to the number of elements read from the arrays *x* and *y* or stored to the array *z*. You explicitly assign the shape parameters when calling the constructor.

Input Parameters

Name	Type	Description
mode	**FORTRAN:** `INTEGER` **C:** `int`	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table 10-16 for a list of possible values.

Name	Type	Description
xshape	**FORTRAN:** INTEGER **C:** int	Defines the length of the input data sequence for the source array *x*. See Data Allocation for more information.
yshape	**FORTRAN:** INTEGER **C:** int	Defines the length of the input data sequence for the source array *y*. See Data Allocation for more information.
zshape	**FORTRAN:** INTEGER **C:** int	Defines the length of the output data sequence to be stored in array *z*. See Data Allocation for more information.

Output Parameters

Name	Type	Description
task	**FORTRAN:** TYPE(VSL_CONV_TASK) for vslsconvnewtask1d, vsldconvnewtask1d TYPE(VSL_CORR_TASK) for vslscorrnewtask1d, vsldcorrnewtask1d **C:** VSLConvTaskPtr* for vslsConvNewTask1D, vsldConvNewTask1D VSLCorrTaskPtr* for vslsCorrNewTask1D, vsldCorrNewTask1D	Pointer to the task descriptor if created successfully or NULL pointer otherwise.
status	**FORTRAN:** INTEGER **C:** int	Set to VSL_STATUS_OK if the task is created successfully or set to non-zero error code otherwise.

NewTaskX

Creates a new convolution or correlation task descriptor for multidimensional case and assigns source data to the first operand vector.

Syntax

Fortran:

```
status = vslsconvnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vsldconvnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vslscorrnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vsldcorrnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
```

C:

```
status = vslsConvNewTaskX(task, mode, dims, xshape, yshape, zshape, x,
xstride);
status = vsldConvNewTaskX(task, mode, dims, xshape, yshape, zshape, x,
xstride);
status = vslsCorrNewTaskX(task, mode, dims, xshape, yshape, zshape, x,
xstride);
status = vsldCorrNewTaskX(task, mode, dims, xshape, yshape, zshape, x,
xstride);
```

Description

Each `NewTaskX` constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see Table 10-13).

Unlike `NewTask`, these routines represent the so called X-form version of the constructor, which means that in addition to creating the task descriptor they assign particular data to the first operand vector in array *x* used in convolution or correlation operation. The task descriptor created by the `NewTaskX` constructor keeps the pointer to the array *x* all the time, that is, until the task object is deleted by one of the destructor routines (see `DeleteTask`).

Using this form of constructors is recommended when you need to compute multiple convolutions or correlations with the same data vector in array *x* against different vectors in array *y*. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

The parameters *xshape*, *yshape*, and *zshape* define the shapes of the input and output data provided by the arrays *x*, *y*, and *z*, respectively. Each shape parameter is an array of integers with its length equal to the value of *dims*. You explicitly assign the shape parameters when calling the constructor. If the value of the parameter *dims* is 1, then *xshape*, *yshape*, and *zshape* are equal to the number of elements read from the arrays *x* and *y* or stored to the array *z*. Note that values of shape parameters may differ from physical shapes of arrays *x*, *y*, and *z* if non-trivial strides are assigned.

The stride parameter *xstride* specifies the physical location of the input data in the array *x*. In a one-dimensional case, stride is an interval between locations of consecutive elements of the array. For example, if the value of the parameter *xstride* is `s`, then only every s^{th} element of the array *x* will be used to form the input sequence. The stride value must be positive or negative but not zero.

Input Parameters

Name	Type	Description
mode	**FORTRAN:** `INTEGER` **C:** `int`	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table 10-16 for a list of possible values.
dims	**FORTRAN:** `INTEGER` **C:** `int`	Rank of user data. Specifies number of dimensions for the input and output arrays *x*, *y*, and *z* used during the execution stage. Must be in the range from 1 to 7. The value is explicitly assigned by the constructor.
xshape	**FORTRAN:** `INTEGER, DIMENSION(*)` **C:** `int[]`	Defines the shape of the input data for the source array *x*. See Data Allocation for more information.
yshape	**FORTRAN:** `INTEGER, DIMENSION(*)` **C:** `int[]`	Defines the shape of the input data for the source array *y*. See Data Allocation for more information.

Name	Type	Description
zshape	**FORTRAN:** `INTEGER, DIMENSION(*)` **C:** `int[]`	Defines the shape of the output data to be stored in array *z*. See Data Allocation for more information.
x	**FORTRAN:** `REAL(KIND=4), DIMENSION (*)` for single precision flavors, `REAL(KIND=8), DIMENSION (*)` for double precision flavors **C:** `float[]` for single precision flavors `double[]` for double precision flavors	Pointer to the array containing input data for the first operand vector. See Data Allocation for more information.
xstride	**FORTRAN:** `INTEGER, DIMENSION (*)` **C:** `int[]`	Strides for input data in the array *x*.

Output Parameters

Name	Type	Description
task	**FORTRAN:** `TYPE(VSL_CONV_TASK)` for `vslsconvnewtaskx`, `vsldconvnewtaskx` `TYPE(VSL_CORR_TASK)` for `vslscorrnewtaskx`, `vsldcorrnewtaskx` **C:** `VSLConvTaskPtr*` for `vslsConvNewTaskX`, `vsldConvNewTaskX`	Pointer to the task descriptor if created successfully or `NULL` pointer otherwise.

Name	Type	Description
	`VSLCorrTaskPtr*` for `vslsCorrNewTaskX`, `vsldCorrNewTaskX`	
status	**FORTRAN:** `INTEGER` **C:** `int`	Set to `VSL_STATUS_OK` if the task is created successfully or set to non-zero error code otherwise.

NewTaskX1D

Creates a new convolution or correlation task descriptor for one-dimensional case and assigns source data to the first operand vector.

Syntax

Fortran:

```
status = vslsconvnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
status = vsldconvnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
status = vslscorrnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
status = vsldcorrnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
```

C:

```
status = vslsConvNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vsldConvNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vslsCorrNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vsldCorrNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
```

Description

Each `NewTaskX1D` constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see Table 10-13).

These routines represent a special one-dimensional version of the so called X-form of the constructor. This assumes that the value of the parameter *dims* is 1 and that in addition to creating the task descriptor, constructor routines assign particular data to the first operand vector in array *x* used in convolution or correlation operation. The task descriptor created by the `NewTaskX1D` constructor keeps the pointer to the array *x* all the time, that is, until the task object is deleted by one of the destructor routines (see `DeleteTask`).

Using this form of constructors is recommended when you need to compute multiple convolutions or correlations with the same data vector in array *x* against different vectors in array *y*. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

The parameters *xshape*, *yshape*, and *zshape* are equal to the number of elements read from the arrays *x* and *y* or stored to the array *z*. You explicitly assign the shape parameters when calling the constructor.

The stride parameters *xstride* specifies the physical location of the input data in the array *x* and is an interval between locations of consecutive elements of the array. For example, if the value of the parameter *xstride* is `s`, then only every s^{th} element of the array *x* will be used to form the input sequence. The stride value must be positive or negative but not zero.

Input Parameters

Name	Type	Description
mode	**FORTRAN:** `INTEGER` **C:** `int`	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table 10-16 for a list of possible values.
xshape	**FORTRAN:** `INTEGER` **C:** `int`	Defines the length of the input data sequence for the source array *x*. See Data Allocation for more information.
yshape	**FORTRAN:** `INTEGER` **C:** `int`	Defines the length of the input data sequence for the source array *y*. See Data Allocation for more information.
zshape	**FORTRAN:** `INTEGER` **C:** `int`	Defines the length of the output data sequence to be stored in array *z*. See Data Allocation for more information.

Name	Type	Description
x	**FORTRAN:** `REAL(KIND=4), DIMENSION (*)` for single precision flavors, `REAL(KIND=8),DIMENSION (*)` for double precision flavors **C:** `float[]` for single precision flavors `double[]` for double precision flavors	Pointer to the array containing input data for the first operand vector. See Data Allocation for more information.
xstride	**FORTRAN:** `INTEGER` **C:** `int`	Stride for input data sequence in the array *x*.

Output Parameters

Name	Type	Description

task	**FORTRAN:**
	`TYPE(VSL_CONV_TASK)` for `vslsconvnewtaskx1d`, `vsldconvnewtaskx1d`
	`TYPE(VSL_CORR_TASK)` for `vslscorrnewtaskx1d`, `vsldcorrnewtaskx1d`
	C: `VSLConvTaskPtr*` for `vslsConvNewTaskX1D`, `vsldConvNewTaskX1D`
	`VSLCorrTaskPtr*` for `vslsCorrNewTaskX1D`, `vsldCorrNewTaskX1D`
		Pointer to the task descriptor if created successfully or `NULL` pointer otherwise.

Name	Type	Description
status	**FORTRAN:** INTEGER **C:** int	Set to VSL_STATUS_OK if the task is created successfully or set to non-zero error code otherwise.

Task Editors

Task editors in convolution and correlation API of the Intel MKL are routines intended for setting up or changing the following task parameters (see Table 10-13):

- *mode*
- *internal_precision*
- *start*
- *decimation*.

For setting up or changing each of the above parameters, a separate routine exists.

NOTE. Fields of the task descriptor structure are accessible only through the set of task editor routines provided with the software.

After applying any of the editor routines to change the task descriptor settings, the task loses its commitment status and will go through the full commitment process again during the next execution or copy operation. This is motivated by the fact that the currently stored work data computed during the last commitment process may become invalid with respect to new parameter settings. For more information about task commitment, see Overview.

Table 10-15 lists available task editors.

Table 10-15 Task Editors

Routine	Description
SetMode	Changes the value of the parameter *mode* for the operation of convolution or correlation.
SetInternalPrecision	Changes the value of the parameter *internal_precision* for the operation of convolution or correlation.
SetStart	Sets the value of the parameter *start* for the operation of convolution or correlation.

Routine	Description
SetDecimation	Sets the value of the parameter *decimation* for the operation of convolution or correlation.

NOTE. You can use the NULL task pointer in calls to editor routines. In this case, the routine will be terminated and no system crash will occur.

SetMode

Changes the value of the parameter mode in the convolution or correlation task descriptor.

Syntax

Fortran:

```
status = vslconvsetmode(task, newmode)
status = vslcorrsetmode(task, newmode)
```

C:

```
status = vslConvSetMode(task, newmode);
status = vslCorrSetMode(task, newmode);
```

Description

The routine changes the value of the parameter mode for the operation of convolution or correlation. This parameter defines whether the computation should be done via Fourier transforms of the input/output data or using a direct algorithm. Initial value for mode is assigned by a task constructor.

Predefined values for the mode parameter are as follows:

Table 10-16 Values of *mode* parameter

Value	Purpose
VSL_CONV_MODE_FFT	Compute convolution by using fast Fourier transform.

Value	Purpose
VSL_CORR_MODE_FFT	Compute correlation by using fast Fourier transform.
VSL_CONV_MODE_DIRECT	Compute convolution directly.
VSL_CORR_MODE_DIRECT	Compute correlation directly.
VSL_CONV_MODE_AUTO	Automatically choose direct or Fourier mode for convolution.
VSL_CORR_MODE_AUTO	Automatically choose direct or Fourier mode for correlation.

Input Parameters

Name	Type	Description
task	**FORTRAN:** TYPE(VSL_CONV_TASK) for vslconvsetmode TYPE(VSL_CORR_TASK) for vslcorrsetmode **C:** VSLConvTaskPtr for vslConvSetMode VSLCorrTaskPtr for vslCorrSetMode	Pointer to the task descriptor.
newmode	**FORTRAN:** INTEGER **C:** int	New value of the parameter mode.

Output Parameters

Name	Type	Description
status	**FORTRAN:** INTEGER **C:** int	Current status of the task.

SetInternalPrecision

Changes the value of the parameter `internal_precision` *in the convolution or correlation task descriptor.*

Syntax

Fortran:

```
status = vslconvsetinternalprecision(task, precision)
status = vslcorrsetinternalprecision(task, precision)
```

C:

```
status = vslConvSetInternalPrecision(task, precision);
status = vslCorrSetInternalPrecision(task, precision);
```

Description

The routine changes the value of the parameter `internal_precision` for the operation of convolution or correlation. This parameter defines whether the internal computations of the convolution or correlation result should be done in single or double precision. Initial value for `internal_precision` is assigned by a task constructor and set to either "single" or "double" according to the particular flavor of the constructor used.

Changing the `internal_precision` can be useful if the default setting of this parameter was "single" but you want to calculate the result with double precision even if input and output data are represented in single precision.

Predefined values for the `internal_precision` input parameter are as follows:

Table 10-17 Values of `internal_precision` Parameter

Value	Purpose
VSL_CONV_PRECISION_SINGLE	Compute convolution with single precision.
VSL_CORR_PRECISION_SINGLE	Compute correlation with single precision.
VSL_CONV_PRECISION_DOUBLE	Compute convolution with double precision.
VSL_CORR_PRECISION_DOUBLE	Compute correlation with double precision.

Input Parameters

Name	Type	Description

task	**FORTRAN:** `TYPE(VSL_CONV_TASK)` for `vslconvsetinternalprecision`
	`TYPE(VSL_CORR_TASK)` for `vslcorrsetinternalprecision`
	C: `VSLConvTaskPtr` for `vslConvSetInternalPrecision`
	`VSLCorrTaskPtr` for `vslCorrSetInternalPrecision`
		Pointer to the task descriptor.

precision	**FORTRAN:** `INTEGER` **C:** `int`	New value of the parameter *internal_precision*.

Output Parameters

Name	Type	Description
status	**FORTRAN:** `INTEGER` **C:** `int`	Current status of the task.

SetStart

Changes the value of the parameter `start` in the convolution or correlation task descriptor.

Syntax

Fortran:

```
status = vslconvsetstart(task, start)
status = vslcorrsetstart(task, start)
```

C:

```
status = vslConvSetStart(task, start);
status = vslCorrSetStart(task, start);
```

Description

The routine sets the value of the parameter *start* for the operation of convolution or correlation. In a one-dimensional case, this parameter points to the first element in the mathematical result that should be stored in the output array. In a multidimensional case, *start* is an array of indices and its length is equal to the number of dimensions specified by the parameter *dims*. For more information about the definition and effect of this parameter, see Data Allocation.

During the initial task descriptor construction, the default value for *start* is undefined and this parameter is not used. Hence, the only way to set and use the *start* parameter is via assigning it some value by one of the `SetStart` routines.

Input Parameters

Name	Type	Description

task	**FORTRAN:** `TYPE(VSL_CONV_TASK)` for `vslconvsetstart`
	`TYPE(VSL_CORR_TASK)` for `vslcorrsetstart`
	C: `VSLConvTaskPtr` for `vslConvSetStart`
	`VSLCorrTaskPtr` for `vslCorrSetStart`
		Pointer to the task descriptor.

start	**FORTRAN:** `INTEGER, DIMENSION (*)` **C:** `int[]`	New value of the parameter *start*.

Output Parameters

Name	Type	Description
status	**FORTRAN:** INTEGER **C:** int	Current status of the task.

SetDecimation

Changes the value of the parameter `decimation` *in the convolution or correlation task descriptor.*

Syntax

Fortran:

```
status = vslconvsetdecimation(task, decimation)
status = vslcorrsetdecimation(task, decimation)
```

C:

```
status = vslConvSetDecimation(task, decimation);
status = vslCorrSetDecimation(task, decimation);
```

Description

The routine sets the value of the parameter `decimation` for the operation of convolution or correlation.

This parameter determines how to thin out the mathematical result of convolution or correlation before writing it into the output data array. For example, in a one-dimensional case, if `decimation` = d >1, only every d-th element of the mathematical result is written to the output array `z`. In a multidimensional case, `decimation` is an array of indices and its length is equal to the number of dimensions specified by the parameter `dims`. For more information about the definition and effect of this parameter, see Data Allocation.

During the initial task descriptor construction, the default value for `decimation` is undefined and this parameter is not used. Hence, the only way to set and use the `decimation` parameter is via assigning it some value by one of the `SetDecimation` routines.

Input Parameters

Name	Type	Description

task	**FORTRAN:** `TYPE(VSL_CONV_TASK)` for `vslconvsetdecimation`
	`TYPE(VSL_CORR_TASK)` for `vslcorrsetdecimation`
	C: `VSLConvTaskPtr` for `vslConvSetDecimation`
	`VSLCorrTaskPtr` for `vslCorrSetDecimation`
		Pointer to the task descriptor.

decimation	**FORTRAN:** `INTEGER, DIMENSION (*)` **C:** `int[]`	New value of the parameter *decimation*.

Output Parameters

Name	Type	Description
status	**FORTRAN:** `INTEGER` **C:** `int`	Current status of the task.

Task Execution Routines

Task execution routines compute convolution or correlation results based on parameters held by the task descriptor and on the supplied user data for input vectors.

Once created and adjusted, the task can be executed multiple times by applying to different input/output data of the same type, precision, and shape.

Intel MKL implementation of the convolution and correlation API provides two different forms of execution routines: a general form and an X-form. General form executors use the task descriptor created by the general form constructor and expect to get two source data arrays *x* and *y* on input. Alternatively, X-form executors use the task descriptor created by the X-form constructor and expect to get only one source data array *y* on input because the first array *x* has been already specified on the construction stage.

When the task is executed for the first time, the execution routine includes task commitment operation, which involves two basic steps: parameters consistency check and preparation of auxiliary data (for example, this might be the calculation of Fourier transform for input data).

For each execution routine there is also an associated one-dimensional version which exploits the algorithmic and computational benefits provided by the simplicity of the data structures for one-dimensional case.

NOTE. You can use the `NULL` task pointer in calls to execution routines. In this case, the routine will be terminated and no system crash will occur.

If the task is executed successfully, the execution routine returns zero status code. If an error is detected, the execution routine returns an error code which signals that a specific error has occurred. In particular, an error status code is returned in the following cases:

- if the task pointer is `NULL`,
- if the task descriptor is corrupted,
- if calculation has failed for some other reason.

NOTE. Intel® MKL does not control floating-point erros, like overflow or gradual underflow, or operations with NaNs, etc.

If an error occurs, the task descriptor stores the error code.

The table below lists all task execution routines.

Table 10-18 Task Execution Routines

Routine	Description
`Exec`	Computes convolution or correlation for a multidimensional case.
`Exec1D`	Computes convolution or correlation for a one-dimensional case.
`ExecX`	Computes convolution or correlation as X-form for a multidimensional case.
`ExecX1D`	Computes convolution or correlation as X-form for a one-dimensional case.

Exec

Computes convolution or correlation for multidimensional case.

Syntax

Fortran:

```
status = vslsconvexec(task, x, xstride, y, ystride, z, zstride)
status = vsldconvexec(task, x, xstride, y, ystride, z, zstride)
status = vslscorrexec(task, x, xstride, y, ystride, z, zstride)
status = vsldcorrexec(task, x, xstride, y, ystride, z, zstride)
```

C:

```
status = vslsConvExec(task, x, xstride, y, ystride, z, zstride);
status = vsldConvExec(task, x, xstride, y, ystride, z, zstride);
status = vslsCorrExec(task, x, xstride, y, ystride, z, zstride);
status = vsldCorrExec(task, x, xstride, y, ystride, z, zstride);
```

Description

Each of the `Exec` routines computes convolution or correlation of the data provided by the arrays *x* and *y* and then stores the results in the array *z*. Parameters of the operation are read from the task descriptor created previously by a corresponding `NewTask` constructor and pointed to by *task*. If *task* is `NULL`, no operation is done.

The stride parameters *xstride*, *ystride*, and *zstride* specify the physical location of the input and output data in the arrays *x*, *y*, and *z*, respectively. In a one-dimensional case, stride is an interval between locations of consecutive elements of the array. For example, if the value of the parameter *zstride* is `s`, then only every s^{th} element of the array *z* will be used to store the output data. The stride value must be positive or negative but not zero.

Input Parameters

Name	Type	Description

task	**FORTRAN:** `TYPE(VSL_CONV_TASK)` for `vslsconvexec` and `vsldconvexec`
	`TYPE(VSL_CORR_TASK)` for `vslscorrexec` and `vsldcorrexec`
	C: `VSLConvTaskPtr` for `vslsConvExec` and `vsldConvExec`
	`VSLCorrTaskPtr` for `vslsCorrExec` and `vsldCorrExec`
		Pointer to the task descriptor.

x, *y*	**FORTRAN:** `REAL(KIND=4), DIMENSION(*)` for `vslsconvexec` and `vslscorrexec`
	`REAL(KIND=8), DIMENSION(*)` for `vsldconvexec` and `vsldcorrexec`
	C: `float[]` for `vslsConvExec` and `vslsCorrExec`
	`double[]` for `vsldConvExec` and `vsldCorrExec`
		Pointers to arrays containing input data. See Data Allocation for more information.

xstride, *ystride*, *zstride*	**FORTRAN:** `INTEGER, DIMENSION(*)` **C:** `int[]`	Strides for input and output data. For more information, see Data Allocation.

Output Parameters

Name	Type	Description

z	**FORTRAN:** `REAL(KIND=4), DIMENSION(*)` for `vslsconvexec` and `vslscorrexec`
	`REAL(KIND=8), DIMENSION(*)` for `vsldconvexec` and `vsldcorrexec`
	C: `float[]` for `vslsConvExec` and `vslsCorrExec`
	`double[]` for `vsldConvExec` and `vsldCorrExec`
		Pointer to the array that stores output data. See Data Allocation for more information.

status	**FORTRAN:** `INTEGER` **C:** `int`	Set to `VSL_STATUS_OK` if the task is executed successfully or set to non-zero error code otherwise.

Exec1D

Computes convolution or correlation for one-dimensional case.

Syntax

Fortran:

```
status = vslsconvexec1d(task, x, xstride, y, ystride, z, zstride)
status = vsldconvexec1d(task, x, xstride, y, ystride, z, zstride)
status = vslscorrexec1d(task, x, xstride, y, ystride, z, zstride)
status = vsldcorrexec1d(task, x, xstride, y, ystride, z, zstride)
```

C:

```
status = vslsConvExec1D(task, x, xstride, y, ystride, z, zstride);
status = vsldConvExec1D(task, x, xstride, y, ystride, z, zstride);
status = vslsCorrExec1D(task, x, xstride, y, ystride, z, zstride);
status = vsldCorrExec1D(task, x, xstride, y, ystride, z, zstride);
```

Description

Each of the `Exec1D` routines computes convolution or correlation of the data provided by the arrays *x* and *y* and then stores the results in the array *z*. These routines represent a special one-dimensional version of the operation, assuming that the value of the parameter *dims* is 1. Using this version of execution routines can help speed up performance in case of one-dimensional data.

Parameters of the operation are read from the task descriptor created previously by a corresponding `NewTask1D` constructor and pointed to by *task*. If *task* is `NULL`, no operation is done.

Input Parameters

Name	Type	Description

task	**FORTRAN:**
	`TYPE(VSL_CONV_TASK)` for `vslsconvexec1d` and `vsldconvexec1d`
	`TYPE(VSL_CORR_TASK)` for `vslscorrexec1d` and `vsldcorrexec1d`
	C: `VSLConvTaskPtr` for `vslsConvExec1D` and `vsldConvExec1D`
	`VSLCorrTaskPtr` for `vslsCorrExec1D` and `vsldCorrExec1D`
		Pointer to the task descriptor.

Name	Type	Description

x, *y*	**FORTRAN:** `REAL(KIND=4), DIMENSION(*)` for `vslsconvexec1d` and `vslscorrexec1d`
	`REAL(KIND=8), DIMENSION(*)` for `vsldconvexec1d` and `vsldcorrexec1d`
	C: `float[]` for `vslsConvExec1D` and `vslsCorrExec1D`
	`double[]` for `vsldConvExe1D` and `vsldCorrExec1D`
		Pointers to arrays containing input data. See Data Allocation for more information.

xstride, *ystride*, *zstride*	**FORTRAN:** `INTEGER` **C:** `int`	Strides for input and output data. For more information, see stride parameters.

Output Parameters

Name	Type	Description
z	**FORTRAN:** `REAL(KIND=4), DIMENSION(*)` for `vslsconvexec1d` and `vslscorrexec1d` `REAL(KIND=8), DIMENSION(*)` for `vsldconvexec1d` and `vsldcorrexec1d` **C:** `float[]` for `vslsConvExec1D` and `vslsCorrExec1D`	Pointer to the array that stores output data. See Data Allocation for more information.

Name	Type	Description
	`double[]` for `vsldConvExec1D` and `vsldCorrExec1D`	
status	**FORTRAN:** `INTEGER` **C:** `int`	Set to `VSL_STATUS_OK` if the task is executed successfully or set to non-zero error code otherwise.

ExecX

Computes convolution or correlation for multidimensional case with the fixed first operand vector.

Syntax

Fortran:

```
status = vslsconvexecx(task, y, ystride, z, zstride)
status = vsldconvexecx(task, y, ystride, z, zstride)
status = vslscorrexecx(task, y, ystride, z, zstride)
status = vsldcorrexecx(task, y, ystride, z, zstride)
```

C:

```
status = vslsConvExecX(task, y, ystride, z, zstride);
status = vsldConvExecX(task, y, ystride, z, zstride);
status = vslsCorrExecX(task, y, ystride, z, zstride);
status = vsldCorrExecX(task, y, ystride, z, zstride);
```

Description

Each of the `ExecX` routines computes convolution or correlation of the data provided by the arrays *x* and *y* and then stores the results in the array *z*. These routines represent a special version of the operation, which assumes that the first operand vector was set on the task construction stage and the task object keeps the pointer to the array *x*.

Parameters of the operation are read from the task descriptor created previously by a corresponding NewTaskX constructor and pointed to by *task*. If *task* is NULL, no operation is done.

Using this form of execution routines is recommended when you need to compute multiple convolutions or correlations with the same data vector in array *x* against different vectors in array *y*. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

Input Parameters

Name	Type	Description
task	**FORTRAN:** TYPE(VSL_CONV_TASK) for vslsconvexecx and vsldconvexecx TYPE(VSL_CORR_TASK) for vslscorrexecx and vsldcorrexecx **C:** VSLConvTaskPtr for vslsConvExecX and vsldConvExecX VSLCorrTaskPtr for vslsCorrExecX and vsldCorrExecX	Pointer to the task descriptor.
x ,*y*	**FORTRAN:** REAL(KIND=4), DIMENSION(*) for vslsconvexecx and vslscorrexecx REAL(KIND=8), DIMENSION(*) for vsldconvexecx and vsldcorrexecx **C:** float[] for vslsConvExecX and vslsCorrExecX	Pointer to array containing input data (for the second operand vector). See Data Allocation for more information.

Name	Type	Description
	`double[]` for `vsldConvExeX` and `vsldCorrExecX`	
ystride ,*zstride*	**FORTRAN:** `INTEGER, DIMENSION(*)` **C:** `int[]`	Strides for input and output data. For more information, see stride parameters.

Output Parameters

Name	Type	Description

z	**FORTRAN:** `REAL(KIND=4), DIMENSION(*)` for `vslsconvexecx` and `vslscorrexecx`
	`REAL(KIND=8), DIMENSION(*)` for `vsldconvexecx` and `vsldcorrexecx`
	C: `float[]` for `vslsConvExecX` and `vslsCorrExecX`
	`double[]` for `vsldConvExecX` and `vsldCorrExecX`
		Pointer to the array that stores output data. See Data Allocation for more information.

status	**FORTRAN:** `INTEGER` **C:** `int`	Set to `VSL_STATUS_OK` if the task is executed successfully or set to non-zero error code otherwise.

ExecX1D

Computes convolution or correlation for one-dimensional case with the fixed first operand vector.

Syntax

Fortran:

```
status = vslsconvexecx1d(task, y, ystride, z, zstride)
status = vsldconvexecx1d(task, y, ystride, z, zstride)
status = vslscorrexecx1d(task, y, ystride, z, zstride)
status = vsldcorrexecx1d(task, y, ystride, z, zstride)
```

C:

```
status = vslsConvExecX1D(task, y, ystride, z, zstride);
status = vsldConvExecX1D(task, y, ystride, z, zstride);
status = vslsCorrExecX1D(task, y, ystride, z, zstride);
status = vsldCorrExecX1D(task, y, ystride, z, zstride);
```

Description

Each of the ExecX1D routines computes convolution or correlation of one-dimensional (assuming that *dims* =1)data provided by the arrays *x* and *y* and then stores the results in the array *z*. These routines represent a special version of the operation, which expects that the first operand vector was set on the task construction stage.

Parameters of the operation are read from the task descriptor created previously by a corresponding NewTaskX1D constructor and pointed to by *task*. If *task* is NULL, no operation is done.

Using this form of execution routines is recommended when you need to compute multiple one-dimensional convolutions or correlations with the same data vector in array *x* against different vectors in array *y*. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

Input Parameters

Name	Type	Description

task	**FORTRAN:** `TYPE(VSL_CONV_TASK)` for `vslsconvexecx1d` and `vsldconvexecx1d`
	`TYPE(VSL_CORR_TASK)` for `vslscorrexecx1d` and `vsldcorrexecx1d`
	C: `VSLConvTaskPtr` for `vslsConvExecX1D` and `vsldConvExecX1D`
	`VSLCorrTaskPtr` for `vslsCorrExecX1D` and `vsldCorrExecX1D`
		Pointer to the task descriptor.

x, *y*	**FORTRAN:** `REAL(KIND=4), DIMENSION(*)` for `vslsconvexecx1d` and `vslscorrexecx1d`
	`REAL(KIND=8), DIMENSION(*)` for `vsldconvexecx1d` and `vsldcorrexecx1d`
	C: `float[]` for `vslsConvExecX1D` and `vslsCorrExecX1D`
	`double[]` for `vsldConvExeX1D` and `vsldCorrExecX1D`
		Pointer to array containing input data (for the second operand vector). See Data Allocation for more information.

ystride, *zstride*	**FORTRAN:** `INTEGER` **C:** `int`	Strides for input and output data. For more information, see stride parameters.

Output Parameters

Name	Type	Description

z	**FORTRAN:** `REAL(KIND=4), DIMENSION(*)` for `vslsconvexecx1d` and `vslscorrexecx1d`
	`REAL(KIND=8), DIMENSION(*)` for `vsldconvexecx1d` and `vsldcorrexec1d`
	C: `float[]` for `vslsConvExecX1D` and `vslsCorrExecX1D`
	`double[]` for `vsldConvExecX1D` and `vsldCorrExecX1D`
		Pointer to the array that stores output data. See Data Allocation for more information.

status	**FORTRAN:** `INTEGER` **C:** `int`	Set to `VSL_STATUS_OK` if the task is executed successfully or set to non-zero error code otherwise.

Task Destructors

Task destructors are routines designed for deleting task objects and deallocating memory.

DeleteTask

Destroys the task object and frees the memory.

Syntax

Fortran:

```
errcode = vslconvdeletetask(task)
errcode = vslcorrdeletetask(task)
```

C:

```
errcode = vslConvDeleteTask(task);
errcode = vslCorrDeleteTask(task);
```

Description

Given a pointer to a task descriptor, this routine deletes the task descriptor object and frees the memory allocated for the data structure. If the task holds a work memory, the latter is also freed. The task pointer is set to `NULL`.

Note that if by some reason the task was not deleted successfully, the routine returns an error code. This error code has no relation to the task status code and does not change it.

NOTE. You can use the `NULL` task pointer in calls to destructor routines. In this case, the routine will be terminated and no system crash will occur.

Input Parameters

Name	Type	Description

task	**FORTRAN:** `TYPE(VSL_CONV_TASK)` for `vslconvdeletetask`
	`TYPE(VSL_CORR_TASK)` for `vslcorrdeletetask`
	C: `VSLConvTaskPtr*` for `vslConvDeleteTask`
	`VSLCorrTaskPtr*` for `vslCorrDeleteTask`
		Pointer to the task descriptor.

Output Parameters

Name	Type	Description
errcode	**FORTRAN:** `INTEGER` **C:** `int`	Contains 0 if the task object is deleted successfully. Contains error code if an error occurred.

Task Copy

The routines are designed for copying convolution and correlation task descriptors.

CopyTask

Copies a descriptor for convolution or correlation task.

Syntax

Fortran:

```
status = vslconvcopytask(newtask, srctask)
status = vslcorrcopytask(newtask, srctask)
```

C:

```
status = vslConvCopyTask(newtask, srctask);
status = vslCorrCopyTask(newtask, srctask);
```

Description

If a task object *srctask* already exists, you can use an appropriate `CopyTask` routine to make its copy in *newtask*. After the copy operation, both source and new task objects will become committed (see Overview for information about task commitment). If the source task was not previously committed, the commitment operation for this task is implicitly invoked before copying starts. If an error occurs during source task commitment, the task stores the error code in the status field. If an error occurs during copy operation, the routine returns a `NULL` pointer instead of a reference to a new task object.

Input Parameters

Name	Type	Description

srctask	**FORTRAN:**
	`TYPE(VSL_CONV_TASK)` for `vslconvcopytask`
	`TYPE(VSL_CORR_TASK)` for `vslcorrcopytask`
		Pointer to the source task descriptor.

Name	Type	Description
	C: `VSLConvTaskPtr` for `vslConvCopyTask` `VSLCorrTaskPtr` for `vslCorrCopyTask`	

Output Parameters

Name	Type	Description

newtask	**FORTRAN:** `TYPE(VSL_CONV_TASK)` for `vslconvcopytask`
	`TYPE(VSL_CORR_TASK)` for `vslcorrcopytask`
	C: `VSLConvTaskPtr*` for `vslConvCopyTask`
	`VSLCorrTaskPtr*` for `vslCorrCopyTask`
		Pointer to the new task descriptor.

status	**FORTRAN:** `INTEGER` **C:** `int`	Current status of the source task.

Usage Examples

This section demonstrates how you can use the Intel MKL routines to perform some common convolution and correlation operations both for single threaded and multiple threaded calculations. The following two sample functions `scond1` and `sconf1` simulate the convolution and correlation functions `SCOND` and `SCONF` found in IBM ESSL* library. The functions assume single threaded calculations and can be used with C or C++ compilers.

Example 10-5 Function scond1 for Single Threaded Calculations

```
#include "mkl_vsl.h"
int scond1(
    float h[], int inch,
    float x[], int incx,
    float y[], int incy,
    int nh, int nx, int iy0, int ny)
{
    int status;
    VSLConvTaskPtr task;
    vslsConvNewTask1D(&task,VSL_CONV_MODE_DIRECT,nh,nx,ny);
    vslConvSetStart(task, &iy0);
    status = vslsConvExec1D(task, h,inch, x,incx, y,incy);
    vslConvDeleteTask(&task);
    return status;
}
```

Example 10-6 Function sconf1 for Single Threaded Calculations

```
#include "mkl_vsl.h"
int sconf1(
    int init,
    float h[], int inc1h,
    float x[], int inc1x, int inc2x,
    float y[], int inc1y, int inc2y,
    int nh, int nx, int m, int iy0, int ny,
    void* aux1, int naux1, void* aux2, int naux2)
{
    int status;
    /* assume that aux1!=0 and naux1 is big enough */
```

```
    VSLConvTaskPtr* task = (VSLConvTaskPtr*)aux1;
    if (init != 0)
        /* initialization: */
        status = vslsConvNewTaskX1D(task,VSL_CONV_MODE_FFT,
         nh,nx,ny, h,inc1h);
    if (init == 0) {
        /* calculations: */
        int i;
        vslConvSetStart(*task, &iy0);
        for (i=0; i<m; i++) {
         float* xi = &x[inc2x * i];
         float* yi = &y[inc2y * i];
         /* task is implicitly committed at i==0 */
         status = vslsConvExecX1D(*task, xi, inc1x, yi, inc1y);
        };
    };
    vslConvDeleteTask(task);
    return status;
}
```

Using Multiple Threads

For functions such as `sconf1` described in the previous example, parallel calculations may be more preferable instead of cycling. If `m>1`, you can use multiple threads for invoking the task execution against different data sequences. For such cases, use task copy routines to create `m` copies of the task object before the calculations stage and then run these copies with different threads. Ensure that you make all necessary parameter adjustments for the task (using Task Editors) before copying it.

The sample code for that can look like following:

```
if (init == 0) {
    int i, status, ss[M];
    VSLConvTaskPtr tasks[M];
    /* assume that M is big enough */
    . . .
    vslConvSetStart(*task, &iy0);
    . . .
    for (i=0; i<m; i++)
        /* implicit commitment at i==0 */
        vslConvCopyTask(&tasks[i],*task);
    . . .
```

Then, `m` threads may be started to execute different copies of the task:

```
. . .
        float* xi = &x[inc2x * i];
        float* yi = &y[inc2y * i];
        ss[i]=vslsConvExecX1D(tasks[i], xi,inc1x, yi,inc1y);
    . . .
```

And finally, after all threads have finished the calculations, overall status ought to be collected from all task objects. The following code assumes signaling the first error found, if any:

```
    . . .
    for (i=0; i<m; i++) {
        status = ss[i];
        if (status != 0) /* 0 means "OK" */
            break;
    };
    return status;
}; /* end if init==0 */
```

Execution routines modify the task internal state (fields of the task structure). Such modifications may conflict with each other if different threads work with the same task object simultaneously. This is the reason why different threads must use different copies of the task.

Mathematical Notation and Definitions

The following notation is necessary to explain the underlying mathematical definitions used in the text:

$\mathbf{R} = (-\infty, +\infty)$	The set of real numbers.
$\mathbf{Z} = \{0, \pm1, \pm2, \ldots\}$	The set of integer numbers.
$\mathbf{Z}^N = \mathbf{Z} \times \ldots \times \mathbf{Z}$	The set of N-dimensional series of integer numbers.
$p = (p_1, \ldots, p_N) \in \mathbf{Z}^N$	N-dimensional series of integers.
$u: \mathbf{Z}^N \rightarrow \mathbf{R}$	Function u with arguments from $\mathbf{Z}^N$ and values from $\mathbf{R}$.
$u(p) = u(p_1, \ldots, p_N)$	The value of the function u for the argument $(p_1, \ldots, p_N)$.
$w = u * v$	Function w is the convolution of the functions u, v.
$w = u \bullet v$	Function w is the correlation of the functions u, v.

Given series $p, q \in \mathbf{Z}^N$:

- series $r = p + q$ is defined as $r^n = p^n + q^n$ for every $n=1,\ldots,N$
- series $r = p - q$ is defined as $r^n = p^n - q^n$ for every $n=1,\ldots,N$
- series $r = \sup\{p, q\}$ is defines as $r^n = \max\{p^n, q^n\}$ for every $n=1,\ldots,N$
- series $r = \inf\{p, q\}$ is defined as $r^n = \min\{p^n, q^n\}$ for every $n=1,\ldots,N$
- inequality $p \le q$ means that $p^n \le q^n$ for every $n=1,\ldots,N$.

A function $u(p)$ is called a finite function if there exist series P^{min}, $P^{max} \in \mathbf{Z}^N$ such that:

$u(p) \neq 0$

implies

$P^{min} \le p \le P^{max}$.

Operations of convolution and correlation are only defined for finite functions.

Consider functions u, v and series P^{min}, $P^{max}Q^{min}$, $Q^{max} \in \mathbf{Z}^N$ such that:

$u(p) \neq 0$ implies $P^{min} \le p \le P^{max}$.

$v(q) \neq 0$ implies $Q^{min} \leq q \leq Q^{max}$.

Definitions of linear correlation and linear convolution for functions u and v are given below.

Linear Convolution

If function $w = u*v$ is the convolution of u and v, then:

$w(r) \neq 0$ implies $\mathbf{R}^{min} \leq r \leq \mathbf{R}^{max}$,

where $\mathbf{R}^{min} = P^{min} + Q^{min}$ and $\mathbf{R}^{max} = P^{max} + Q^{max}$.

If $\mathbf{R}^{min} \leq r \leq \mathbf{R}^{max}$, then:

$w(r) = \sum u(t) \cdot v(r-t)$ is the sum for all $t \in \mathbf{Z}^N$ such that $\mathbf{T}^{min} \leq t \leq \mathbf{T}^{max}$,

where $\mathbf{T}^{min} = \sup\{P^{min}, r - Q^{max}\}$ and $\mathbf{T}^{max} = \inf\{P^{max}, r - Q^{min}\}$.

Linear Correlation

If function $w = u \bullet v$ is the correlation of u and v, then:

$w(r) \neq 0$ implies $\mathbf{R}^{min} \leq r \leq \mathbf{R}^{max}$,

where $\mathbf{R}^{min} = Q^{min} - P^{max}$ and $\mathbf{R}^{max} = Q^{max} - P^{min}$.

If $\mathbf{R}^{min} \leq r \leq \mathbf{R}^{max}$, then:

$w(r) = \sum u(t) \cdot v(r+t)$ is the sum for all $t \in \mathbf{Z}^N$ such that $\mathbf{T}^{min} \leq t \leq \mathbf{T}^{max}$,

where $\mathbf{T}^{min} = \sup\{P^{min}, Q^{min} - r\}$ and $\mathbf{T}^{max} = \inf\{P^{max}, Q^{max} - r\}$.

Representation of the functions u, v, w as the input/output data for the Intel MKL convolution and correlation functions is described in the Data Allocation section below.

Data Allocation

This section explains the relation between:

- mathematical finite functions u, v, w introduced in the section Mathematical Notation and Definitions;

- multi-dimensional input and output data vectors representing the functions u, v, w;
- arrays u, v, w used to store the input and output data vectors in computer memory

The convolution and correlation routine parameters that determine the allocation of input and output data are the following:

- Data arrays *x*, *y*, *z*
- Shape arrays *xshape*, *yshape*, *zshape*
- Strides within arrays *xstride*, *ystride*, *zstride*
- Parameters *start*, *decimation*

Finite Functions and Data Vectors

The finite functions $u(p)$, $v(q)$, and $w(r)$ introduced above are represented as multi-dimensional vectors of input and output data:

$inputu(i_1,\ldots,i_{dims})$ for $u(p_1,\ldots,p_N)$

$inputv(j_1,\ldots,j_{dims})$ for $v(q_1,\ldots,q_N)$

$output(k_1,\ldots,k_{dims})$ for $w(r_1,\ldots,r_N)$.

Parameter *dims* represents the number of dimensions and is equal to N.

The parameters *xshape*, *yshape*, and *zshape* define the shapes of input/output vectors:

$inputu(i_1,\ldots,i_{dims})$ is defined if $1 \le i_n \le xshape(n)$ for every $n=1,\ldots,dims$

$inputv(j_1,\ldots,j_{dims})$ is defined if $1 \le j_n \le yshape(n)$ for every $n=1,\ldots,dims$

$output(k_1,\ldots,k_{dims})$ is defined if $1 \le k_n \le zshape(n)$ for every $n=1,\ldots,dims$.

Relation between the input vectors and the functions u and v is defined by the following formulas:

$inputu(i_1,\ldots,i_{dims}) = u(p_1,\ldots,p_N)$, where $p_n = P_n^{min} + (i_n-1)$ for every n

$inputv(j_1,\ldots,j_{dims}) = v(q_1,\ldots,q_N)$, where $q_n=Q_n^{min} + (j_n-1)$ for every n.

Relation between the output vector and the function $w(r)$ is similar (but only in the case when parameters *start* and *decimation* are not defined):

$output(k_1,\ldots,k_{dims}) = w(r_1,\ldots,r_N)$, where $r_n=R_n^{min} + (k_n-1)$ for every n.

If the parameter *start* is defined, it must belong to the interval $R_n^{min} \le start(n) \le R_n^{max}$. If defined, the *start* parameter replaces R^{min} in the formula:

output(k_1,...,k_{dims})=w(r_1,...,r_N), where r_n=start(n) + (k_n-1)

If the parameter *decimation* is defined, it changes the relation according to the following formula:

output(k_1,...,k_{dims})=w(r_1,...,r_N), where $r_n = R_n^{min}$ + (k_n-1)*decimation(n)

If both parameters *start* and *decimation* are defined, the formula is as follows:

output(k_1,...,k_{dims})=w(r_1,...,r_N), where r_n=start(n) + (k_n-1)*decimation(n)

The convolution and correlation software checks the values of *zshape*, *start*, and *decimation* during task commitment. If r_n exceeds R_n^{max} for some k_n, n=1,...,dims, an error is raised.

Allocation of Data Vectors

Both parameter arrays *x* and *y* contain input data vectors in memory, while array *z* is intended for storing output data vector. To access the memory, the convolution and correlation software uses only pointers to these arrays and ignores the array shapes.

For parameters *x*, *y*, and *z*, you can provide one-dimensional arrays with the requirement that actual length of these arrays be sufficient to store the data vectors.

The allocation of the input and output data inside the arrays *x*, *y*, and *z* is described below assuming that the arrays are one-dimensional. Given multi-dimensional indices $i, j, k \in \mathbf{Z}^N$, one-dimensional indices $e, f, g \in \mathbf{Z}$ are defined such that:

inputu(i_1,...,i_{dims}) is allocated at *x(e)*

inputv(j_1,...,j_{dims}) is allocated at *y(f)*

output(k_1,...,k_{dims}) is allocated at *z(g)*.

The indices *e*, *f*, and *g* are defined as follows:

$e = 1 + \sum xstride(n) \cdot dx(n)$ (the sum is for all n=1,...,dims)

$f = 1 + \sum ystride(n) \cdot dy(n)$ (the sum is for all n=1,...,dims)

$g = 1 + \sum zstride(n) \cdot dz(n)$ (the sum is for all n=1,...,dims)

The distances dx(n), dy(n), and dz(n) depend on the signum of the stride:

dx(n) = i_n-1 if xstride(n)>0, or dx(n) = i_n-xshape(n) if xstride(n)<0

dy(n) = j_n-1 if ystride(n)>0, or dy(n) = j_n-yshape(n) if ystride(n)<0

dz(n) = k_n-1 if zstride(n)>0, or dz(n) = k_n-zshape(n) if zstride(n)<0

The definitions of indices *e*, *f*, and *g* assume that indexes for arrays *x*, *y*, and *z* are started from unity:

x(e) is defined for e=1,...,length(x)

y(f) is defined for f=1,...,length(y)

z(g) is defined for g=1,...,length(z)

Below is a detailed explanation about how elements of the multi-dimensional output vector are stored in the array *z* for one-dimensional and two-dimensional cases.

One-dimensional case. If *dims*=1, then *zshape* is the number of the output values to be stored in the array *z*. The actual length of array *z* may be greater than *zshape* elements.

If zstride>1, output values are stored with the stride: output(1) is stored to z(1), output(2) is stored to z(1+zstride), and so on. Hence, the actual length of *z* must be at least 1+zstride*(zshape-1) elements or more.

If zstride<0, it still defines the stride between elements of array *z*. However, the order of the used elements is the opposite. For the *k*-th output value, output(k) is stored in z(1+|zstride|*(zshape-k)), where |zstride| is the absolute value of zstride. The actual length of the array *z* must be at least 1+|zstride|*(zshape - 1) elements.

Two-dimensional case. If *dims*=2, the output data is a two-dimensional matrix. The value zstride(1) defines the stride inside matrix columns, that is, the stride between the output(k_1, k_2) and output(k_1+1, k_2) for every pair of indices k_1, k_2. On the other hand, zstride(2) defines the stride between columns, that is, the stride between output(k_1,k_2) and output(k_1,k_2+1).

If zstride(2) is greater than zshape(1), this causes sparse allocation of columns. If the value of zstride(2) is smaller than zshape(1), this may result in the transposition of the output matrix. For example, if zshape = (2,3), you can define zstride = (3,1) to allocate output values like transposed matrix of the shape 3x2.

Whether zstride assumes this kind of transformations or not, you need to ensure that different elements output (k_1, ...,k_{dims}) will be stored in different locations *z(g)*.

Fourier Transform Functions 11

This chapter describes the following implementations of Discrete Fourier transform functions available in Intel® MKL:

- Discrete Fourier transform (DFT) functions for single-processor or shared-memory systems (see DFT Functions below)
- Cluster DFT Functions for distributed-memory architectures (available with Intel® MKL Cluster Edition only).

Both these groups of DFT functions present a uniform and easy-to-use Applications Programmer Interface providing fast computation of DFT via the Fast Fourier Transform (FFT) algorithm.

NOTE. DFT functions support arbitrary lengths.

These routines offer broad functionality and high performance not only for radix 2 but also for 3, 5, 7, 11, and other radices.

DFT Functions

The Discrete Fourier Transform function library of Intel MKL provides one-dimensional, two-dimensional, and multi-dimensional (up to the order of 7) routines and both Fortran and C interfaces for all transform functions.

The full list of DFT functions implemented in Intel MKL is given in the table below:

Table 11-1 DFT Functions in Intel MKL

Function Name	Operation
Descriptor Manipulation Functions	
DftiCreateDescriptor	Allocates memory for the descriptor data structure and instantiates it with default configuration settings.
DftiCommitDescriptor	Performs all initialization that facilitates the actual DFT computation.
DftiCopyDescriptor	Copies an existing descriptor.

Function Name	Operation
DftiFreeDescriptor	Frees memory allocated for a descriptor.
DFT Computation Functions	
DftiComputeForward	Computes the forward DFT.
DftiComputeBackward	Computes the backward DFT.
Descriptor Configuration Functions	
DftiSetValue	Sets one particular configuration parameter with the specified configuration value.
DftiGetValue	Gets the configuration value of one particular configuration parameter.
Status Checking Functions	
DftiErrorClass	Checks if the status reflects an error of a predefined class.
DftiErrorMessage	Generates an error message.

Description of DFT functions is followed by discussion of configuration settings (see Configuration Settings) and various configuration parameters used.

Computing DFT

DFT functions described later in this chapter are implemented in Fortran and C interface. Fortran stands for Fortran 95. DFT interface relies critically on many modern features offered in Fortran 95 that have no counterpart in Fortran 77

NOTE. Following the explicit function interface in Fortran, data array must be defined as one-dimensional for any transformation type.

The materials presented in this chapter assume the availability of native complex types in C as they are specified in C9X.

You can find example code that uses DFT interface functions to compute transform results in Fourier Transform Functions Code Examples section in the Appendix C.

For most common situations, we expect a DFT computation can be effected by four function calls. The approach adopted in Intel MKL for DFT computation uses one single data structure, the descriptor, to record flexible configuration whose parameters can be changed independently. This results in enhanced functionality and ease of use.

The record of type `DFTI_DESCRIPTOR`, when created, contains information about the length and domain of the DFT to be computed, as well as the setting of a rather large number of configuration parameters. The default settings for all of these parameters include, for example, the following:

- the DFT to be computed does not have a scale factor;
- there is only one set of data to be transformed;
- the data is stored contiguously in memory;
- the computed result overwrites (in place) the input data; etc.

Should any one of these many default settings be inappropriate, they can be changed one-at-a-time through the function `DftiSetValue` as illustrated in the Example C-20 and Example C-21.

DFT Interface

To use the DFT functions, you need to access the module `MKL_DFTI` through the "`use`" statement in Fortran; or access the header file mkl_dfti.h through "`include`" in C.

The Fortran interface provides a derived type `DFTI_DESCRIPTOR`; a number of named constants representing various names of configuration parameters and their possible values; and a number of overloaded functions through the generic functionality of Fortran 95.

The C interface provides a structure type `DFTI_DESCRIPTOR`, a macro definition

```
#define DFTI_DESCRIPTOR_HANDLE DFTI_DESCRIPTOR *;
```

a number of named constants of two enumeration types `DFTI_CONFIG_PARAM` and `DFTI_CONFIG_VALUE`; and a number of functions, some of which accept different number of input arguments.

NOTE. Some of the DFT functions and/or functionality described in the subsequent sections of this chapter may not be supported by the currently available implementation of the library. You can find the complete list of the implementation-specific exceptions in the release notes to your version of the library.

There are four main categories of DFT functions in Intel MKL:

1. **Descriptor Manipulation.** There are four functions in this category. The first one, DftiCreateDescriptor, creates a DFT descriptor whose storage is allocated dynamically by the routine. This function configures the descriptor with default settings corresponding to a few input values supplied by the user.

   The second, DftiCommitDescriptor, "commits" the descriptor to all its setting. In practice, this usually means that all the necessary precomputation will be performed. This may include factorization of the input length and computation of all the required twiddle factors. The third function, DftiCopyDescriptor, makes an extra copy of a descriptor, and the fourth function, DftiFreeDescriptor, frees up all the memory allocated for the descriptor information.
2. **DFT Computation.** There are two functions in this category. The first, DftiComputeForward, effects a forward DFT computation, and the second function, DftiComputeBackward, performs a backward DFT computation.
3. **Descriptor configuration.** There are two functions in this category. One function, DftiSetValue, sets one specific value to one of the many configuration parameters that are changeable (a few are not); the other, DftiGetValue, gets the current value of any one of these configuration parameters (all are readable). These parameters, though many, are handled one-at-a-time.
4. **Status Checking.** The functions described in the three categories above return an integer value denoting the status of the operation. In particular, a non-zero return value always indicates a problem of some sort. Envisioned to be further enhanced in later releases of Intel MKL, DFT interface at present provides for one logical status class function, DftiErrorClass, and a simple status message generation function, DftiErrorMessage.

Status Checking Functions

All of the descriptor manipulation, DFT computation, and descriptor configuration functions return an integer value denoting the status of the operation. Two functions serve to check the status. The first function is a logical function that checks if the status reflects an error of a predefined class, and the second is an error message function that returns a character string.

ErrorClass

Checks if the status reflects an error of a predefined class.

Syntax

Fortran:

```
Predicate = DftiErrorClass( Status, Error_Class )
```

C:

```
predicate = DftiErrorClass( status, error_class );
```

Description

DFT interface in Intel MKL provides a set of predefined error class listed in Table 11-2. These are named constants and have the type `INTEGER` in Fortran and `long` in C.

Table 11-2 Predefined Error Class

Named Constants	Comments
DFTI_NO_ERROR	No error
DFTI_MEMORY_ERROR	Usually associated with memory allocation
DFTI_INVALID_CONFIGURATION	Invalid settings of one or more configuration parameters
DFTI_INCONSISTENT_CONFIGURATION	Inconsistent configuration or input parameters
DFTI_NUMBER_OF_THREADS_ERROR	Number of OMP threads in the computation function is not equal to the number of OMP threads in the initialization stage (commit function)
DFTI_MULTITHREADED_ERROR	Usually associated with OMP routine's error return value
DFTI_BAD_DESCRIPTOR	Descriptor is unusable for computation
DFTI_UNIMPLEMENTED	Unimplemented legitimate settings; implementation dependent
DFTI_MKL_INTERNAL_ERROR	Internal library error

Named Constants	Comments
DFTI_1D_LENGTH_EXCEEDS_INT32	Length of one of dimensions exceeds 2^{32} -1 (4 bytes).

Note that the correct usage is to check if the status returns .TRUE. or .FALSE. through the use of DftiErrorClass with a specific error class. Direct comparison of a status with the predefined class is an incorrect usage. see Example C-22 on a correct use of the status checking functions.

Interface and Prototype

```
//Fortran interface
INTERFACE DftiErrorClass
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
 FUNCTION some_actual_function_8( Status, Error_Class )
 LOGICAL some_actual_function_8
 INTEGER, INTENT(IN) :: Status, Error_Class
 END FUNCTION some_actual_function_8
END INTERFACE DftiErrorClass


/* C prototype */
long DftiErrorClass( long , long );
```

ErrorMessage

Generates an error message.

Syntax

Fortran:

```
ERROR_MESSAGE = DftiErrorMessage( Status )
```

C:

```
error_message = DftiErrorMessage( status );
```

Description

The error message function generates an error message character string. The maximum length of the string in Fortran is given by the named constant DFTI_MAX_MESSAGE_LENGTH. The actual error message is implementation dependent. In Fortran, the user needs to use a character string of length DFTI_MAX_MESSAGE_LENGTH as the target. In C, the function returns a pointer to a character string, that is, a character array with the delimiter ' 0'.

Example C-22 shows how this function can be implemented.

Interface and Prototype

```
//Fortran interface
INTERFACE DftiErrorMessage
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
 FUNCTION some_actual_function_9( Status, Error_Class )
 CHARACTER(LEN=DFTI_MAX_MESSAGE_LENGTH) some_actual_function_9( Status )
 INTEGER, INTENT(IN) :: Status
 END FUNCTION some_actual_function_9
END INTERFACE DftiErrorMessage
```

```
/* C prototype */
char *DftiErrorMessage( long );
```

Descriptor Manipulation Functions

There are four functions in this category: create a descriptor, commit a descriptor, copy a descriptor, and free a descriptor.

CreateDescriptor

Allocates memory for the descriptor data structure and instantiates it with default configuration settings.

Syntax

Fortran:

```
Status = DftiCreateDescriptor( Desc_Handle, &
Precision, &
Forward_Domain, &
Dimension, &
Length )
```

C:

```
status = DftiCreateDescriptor( &desc_handle, precision, forward_domain,
dimension, length );
```

Description

This function allocates memory for the descriptor data structure and instantiates it with all the default configuration settings with respect to the precision, domain, dimension, and length of the desired transform. The domain is understood to be the domain of the forward transform. Since memory is allocated dynamically, the result is actually a pointer to the created descriptor. This function is slightly different from the "initialization" routine in more traditional software packages or libraries used for computing DFT. In all likelihood, this function will not perform

any significant computation work such as twiddle factors computation, as the default configuration settings can still be changed upon user's request through the value setting function DftiSetValue.

The precision and (forward) domain are specified through named constants provided in DFT interface for the configuration values. The choices for precision are DFTI_SINGLE and DFTI_DOUBLE; and the choices for (forward) domain are DFTI_COMPLEX and DFTI_REAL. See Table 11-5 for the complete table of named constants for configuration values.

Dimension is a simple positive integer indicating the dimension of the transform. Length is either a simple positive integer for one-dimensional transform, or an integer array (pointer in C) containing the positive integers corresponding to the lengths dimensions for multi-dimensional transform.

The function returns DFTI_NO_ERROR when completes successfully. See Status Checking Functions for more information on returned status.

Interface and Prototype

```
!Fortran interface.
INTERFACE DftiCreateDescriptor
!Note that the body provided here is to illustrate the different
!argument list and types of dummy arguments. The interface
!does not guarantee what the actual function names are.
!Users can only rely on the function name following the keyword INTERFACE
 FUNCTION some_actual_function_1D( Desc_Handle, Prec, Dom, Dim, Length )
 INTEGER :: some_actual_function_1D

 TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 INTEGER, INTENT(IN) :: Prec, Dom
 INTEGER, INTENT(IN) :: Dim, Length
 END FUNCTION some_actual_function_1D
```

```
 FUNCTION some_actual_function_HIGHD( Desc_Handle, Prec, Dom, Dim, Length )
 INTEGER :: some_actual_function_HIGHD
 TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 INTEGER, INTENT(IN) :: Prec, Dom
 INTEGER, INTENT(IN) :: Dim, Length(*)
 END FUNCTION some_actual_function_HIGHD
END INTERFACE DftiCreateDescriptor
```

Note that the function is overloaded as the actual argument for `Length` can be a scalar or a a rank-one array.

```
/* C prototype */
long DftiCreateDescriptor( DFTI_DESCRIPTOR_HANDLE *,
      DFTI_CONFIG_PARAM ,
      DFTI_CONFIG_PARAM ,
      long , ... );
```

The variable arguments facility is used to cope with the argument for lengths that can be a scalar (`long`), or an array (`long *`).

CommitDescriptor

Performs all initialization that facilitates the actual DFT computation.

Syntax

Fortran:

```
Status = DftiCommitDescriptor( Desc_Handle )
```

C:

```
status = DftiCommitDescriptor( desc_handle );
```

Description

The interface requires a function that commits a previously created descriptor be invoked before the descriptor can be used for DFT computations. Typically, this committal performs all initialization that facilitates the actual DFT computation. For a modern implementation, it may involve exploring many different factorizations of the input length to search for highly efficient computation method.

Any changes of configuration parameters of a committed descriptor via the set value function (see Descriptor Configuration) requires a re-committal of the descriptor before a computation function can be invoked. Typically, this committal function call is immediately followed by a computation function call (see DFT Computation).

The function returns `DFTI_NO_ERROR` when completes successfully. See Status Checking Functions for more information on returned status.

Interface and Prototype

```
! Fortran interface
INTERFACE DftiCommitDescriptor
!Note that the body provided here is to illustrate the different
!argument list and types of dummy arguments. The interface
!does not guarantee what the actual function names are.
!Users can only rely on the function name following the
!keyword INTERFACE
 FUNCTION some_actual function_1 ( Desc_Handle )
 INTEGER :: some_actual function_1
 TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 END FUNCTION some_actual function_1
END INTERFACE DftiCommitDescriptor


/* C prototype */
long DftiCommitDescriptor( DFTI_DESCRIPTOR_HANDLE );
```

CopyDescriptor

Copies an existing descriptor.

Syntax

Fortran:

```
Status = DftiCopyDescriptor( Desc_Handle_Original, Desc_Handle_Copy )
```

C:

```
status = DftiCopyDescriptor( desc_handle_original, &desc_handle_copy );
```

Description

This function makes a copy of an existing descriptor and provides a pointer to it. The purpose is that all information of the original descriptor will be maintained even if the original is destroyed via the free descriptor function `DftiFreeDescriptor`.

The function returns `DFTI_NO_ERROR` when completes successfully. See Status Checking Functions for more information on returned status.

Interface and Prototype

```
! Fortran interface
INTERFACE DftiCopyDescriptor
! Note that the body provided here is to illustrate the different
!argument list and types of dummy arguments. The interface
!does not guarantee what the actual function names are.
!Users can only rely on the function name following the
!keyword INTERFACE
 FUNCTION some_actual_function_2( Desc_Handle_Original,
 Desc_Handle_Copy )
 INTEGER :: some_actual_function_2
 TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle_Original, Desc_Handle_Copy
 END FUNCTION some_actual_function_2
END INTERFACE DftiCopyDescriptor
```

```
/* C prototype */
long DftiCopyDescriptor( DFTI_DESCRIPTOR_HANLDE, DFTI_DESCRIPTOR_HANDLE *
);
```

FreeDescriptor

Frees memory allocated for a descriptor.

Syntax

Fortran:

```
Status = DftiFreeDescriptor( Desc_Handle )
```

C:

```
status = DftiFreeDescriptor( &desc_handle );
```

Description

This function frees up all memory space allocated for a descriptor.

NOTE. Memory allocation/deallocation inside Intel MKL is managed by Intel MKL Memory Manager. So, even after successful completion of `FreeDescriptor`, the memory space may continue being allocated for the application because the Memory Manager sometimes doesn't return the memory space to OS but considers the space free and can reuse it for future memory allocation. See Example "MKL_FreeBuffers Usage with DFT Functions" in the description of the service function `FreeBuffers` on how to use Intel MKL Memory Manager and actually release memory.

The function returns `DFTI_NO_ERROR` when completes successfully. See Status Checking Functions for more information on returned status.

Interface and Prototype

```
! Fortran interface
INTERFACE DftiFreeDescriptor
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
 FUNCTION some_actual_function_3( Desc_Handle )
 INTEGER :: some_actual_function_3
 TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 END FUNCTION some_actual_function_3
END INTERFACE DftiFreeDescriptor


/* C prototype */
long DftiFreeDescriptor( DFTI_DESCRIPTOR_HANDLE * );
```

DFT Computation Functions

There are two functions in this category: compute the forward transform, and compute the backward transform.

ComputeForward

Computes the forward DFT.

Syntax

Fortran:

```
Status = DftiComputeForward( Desc_Handle, X_inout )
Status = DftiComputeForward( Desc_Handle, X_in, X_out )
```

C:

```
status = DftiComputeForward( desc_handle, x_inout );
status = DftiComputeForward( desc_handle, x_in, x_out );
```

Description

As soon as a descriptor is configured and committed successfully, actual computation of DFT can be performed. The `DftiComputeForward` function computes the forward DFT.

This is the transform using the factor $e^{-i2\pi/n}$. Because of the flexibility in configuration, input data can be represented in various ways as well as output result can be placed differently. Consequently, the number of input parameters as well as their type vary. This variation is accommodated by the generic function facility of Fortran 95. Data and result parameters are all declared as assumed-size rank-1 array `DIMENSION(0:*)`.

The function returns `DFTI_NO_ERROR` when completes successfully. See Status Checking Functions for more information on returned status.

Interface and Prototype

```
//Fortran interface.
INTERFACE DftiComputeFoward
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
 // One argument single precision complex
 FUNCTION some_actual_function_4_C( Desc_Handle, X )
 INTEGER :: some_actual_function_4_C
 TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 COMPLEX, INTENT(INOUT) :: X(*)
 END FUNCTION some_actual_function_4_C
 // One argument double precision complex
 FUNCTION some_actual_function_4_Z( Desc_Handle, X )
 INTEGER :: some_actual_function_4_Z
 TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 COMPLEX (Kind((0D0,0D0))), INTENT(INOUT) :: X(*)
 END FUNCTION some_actual_function_4_Z
 // One argument single precision real
 FUNCTION some_actual_function_4_R( Desc_Handle, X )
 INTEGER :: some_actual_function_4_R
 TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 REAL, INTENT(INOUT) :: X(*)
 END FUNCTION some_actual_function_4_R
 // One argument double precision real
 ...
```

```
 // Two argument single precision complex
 ...
 ...
 FUNCTION some_actual_function_4_CC( Desc_Handle, X_In, Y_Out )
 INTEGER :: some_actual_function_4_CC
 TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 COMPLEX, INTENT(IN) :: X_In(*)
 COMPLEX, INTENT(OUT) :: Y_Out(*)
 END FUNCTION some_actual_function_4_CC
END INTERFACE DftiComputeFoward


/* C prototype */
long DftiComputeForward( DFTI_DESCRIPTOR_HANDLE,
            void *,
       ... );
```

The implementations of DFT interface expect the data be treated as data stored linearly in memory with a regular "stride" pattern (discussed more fully in Strides, see also [3]). The function expects the starting address of the first element. Hence we use the assume-size declaration in Fortran.

The descriptor by itself contains sufficient information to determine exactly how many arguments and of what type should be present. The implementation could use this information to check against possible input inconsistency.

ComputeBackward

Computes the backward DFT.

Syntax

Fortran:

```
Status = DftiComputeBackward( Desc_Handle, X_inout )
Status = DftiComputeBackward( Desc_Handle, X_in, X_out )
```

C:

```
status = DftiComputeBackward( desc_handle, x_inout );
status = DftiComputeBackward( desc_handle, x_in, x_out );
```

Description

As soon as a descriptor is configured and committed successfully, actual computation of DFT can be performed. The `DftiComputeBackward` function computes the backward DFT.

This is the transform using the factor $e^{i2\pi/n}$. Because of the flexibility in configuration, input data can be represented in various ways as well as output result can be placed differently. Consequently, the number of input parameters as well as their type vary. This variation is accommodated by the generic function facility of Fortran 95. Data and result parameters are all declared as assumed-size rank-1 array `DIMENSION(0:*)`. The function returns `DFTI_NO_ERROR` when completes successfully. See Status Checking Functions for more information on returned status.

Interface and Prototype

```
INTERFACE DftiComputeBackward
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
 // One argument single precision complex
 FUNCTION some_actual_function_5_C( Desc_Handle, X )
 INTEGER :: some_actual_function_5_C
 TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 COMPLEX, INTENT(INOUT) :: X(*)
 END FUNCTION some_actual_function_5_C
 // One argument double precision complex
 FUNCTION some_actual_function_5_Z( Desc_Handle, X )
 INTEGER :: some_actual_function_5_Z
 TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 COMPLEX (Kind((0D0,0D0))), INTENT(INOUT) :: X(*)
 END FUNCTION some_actual_function_5_Z
 // One argument single precision real
 FUNCTION some_actual_function_5_R( Desc_Handle, X )
 INTEGER :: some_actual_function_5_R
 TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 REAL, INTENT(INOUT) :: X(*)
 END FUNCTION some_actual_function_5_R
 // One argument double precision real
 ...
 // Two argument single precision complex
```

```
...
...
FUNCTION some_actual_function_5_CC( Desc_Handle, X_In, Y_Out )
INTEGER :: some_actual_function_5_CC
TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
COMPLEX, INTENT(IN) :: X_In(*)
COMPLEX, INTENT(OUT) :: Y_Out(*)
END FUNCTION some_actual_function_5_CCEND INTERFACE DftiComputeBackward END
INTERFACE DftiComputeBackward
```

```
/* C prototype */
long DftiComputeBackward( DFTI_DESCRIPTOR_HANDLE,
      void *,
      ... );
```

The implementations of DFT interface expect the data be treated as data stored linearly in memory with a regular "stride" pattern (discussed more fully in Strides, see also [3]). The function expects the starting address of the first element. Hence we use the assume-size declaration in Fortran.

The descriptor by itself contains sufficient information to determine exactly how many arguments and of what type should be present. The implementation could use this information to check against possible input inconsistency.

Descriptor Configuration Functions

There are two functions in this category: the value setting function `DftiSetValue` sets one particular configuration parameter to an appropriate value, and the value getting function `DftiGetValue` reads the values of one particular configuration parameter. While all configuration parameters are readable, a few of them cannot be set by user. Some of these contain fixed information of a particular implementation such as version number, or dynamic information, but nevertheless are derived by the implementation during execution of one of the functions. See Configuration Settings for details.

SetValue

Sets one particular configuration parameter with the specified configuration value.

Syntax

Fortran:

```
Status = DftiSetValue( Desc_Handle, &
Config_Param, &
Config_Val )
```

C:

```
status = DftiSetValue( desc_handle, config_param, config_val );
```

Description

This function sets one particular configuration parameter with the specified configuration value. The configuration parameter is one of the named constants listed in Table 11-3, and the configuration value is the corresponding appropriate type, which can be a named constant or a native type. See Configuration Settings for details of the meaning of the setting.

The function returns `DFTI_NO_ERROR` when completes successfully. See Status Checking Functions for more information on returned status.

Interface and Prototype

```
! Fortran interface
INTERFACE DftiSetValue
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
 FUNCTION some_actual_function_6_INTVAL( Desc_Handle, Config_Param, INTVAL
)
 INTEGER :: some_actual_function_6_INTVAL
 Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 INTEGER, INTENT(IN) :: Config_Param
 INTEGER, INTENT(IN) :: INTVAL
 END FUNCTION some_actual_function_6_INTVAL

 FUNCTION some_actual_function_6_SGLVAL( Desc_Handle, Config_Param, SGLVAL
)
 INTEGER :: some_actual_function_6_SGLVAL
 Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 INTEGER, INTENT(IN) :: Config_Param
 REAL, INTENT(IN) :: SGLVAL
END FUNCTION some_actual_function_6_SGLVAL
```

```
 FUNCTION some_actual_function_6_DBLVAL( Desc_Handle, Config_Param, DBLVAL
)
 INTEGER :: some_actual_function_6_DBLVAL
 Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 INTEGER, INTENT(IN) :: Config_Param
 REAL (KIND(0D0)), INTENT(IN) :: DBLVAL
 END FUNCTION some_actual_function_6_DBLVAL

 FUNCTION some_actual_function_6_INTVEC( Desc_Handle, Config_Param, INTVEC
)
 INTEGER :: some_actual_function_6_INTVEC
 Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 INTEGER, INTENT(IN) :: Config_Param
 INTEGER, INTENT(IN) :: INTVEC(*)
 END FUNCTION some_actual_function_6_INTVEC

 FUNCTION some_actual_function_6_CHARS( Desc_Handle, Config_Param, CHARS )
 INTEGER :: some_actual_function_6_CHARS
 Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 INTEGER, INTENT(IN) :: Config_Param
 CHARCTER(*), INTENT(IN) :: CHARS
 END FUNCTION some_actual_function_6_CHARS
END INTERFACE DftiSetValue


/* C prototype */
long DftiSetValue( DFTI_DESCRIPTOR_HANDLE, DFTI_CONFIG_PARAM , ... );
```

GetValue

Gets the configuration value of one particular configuration parameter.

Syntax

Fortran:

```
Status = DftiGetValue( Desc_Handle, &
Config_Param, &
Config_Val )
```

C:

```
status = DftiGetValue( desc_handle, config_param, &config_val );
```

Description

This function gets the configuration value of one particular configuration parameter. The configuration parameter is one of the named constants listed in Table 11-3 and Table 11-4, and the configuration value is the corresponding appropriate type, which can be a named constant or a native type.

The function returns `DFTI_NO_ERROR` when completes successfully. See Status Checking Functions for more information on returned status.

Interface and Prototype

```
! Fortran interface

INTERFACE DftiGetValue

//Note that the body provided here is to illustrate the different

//argument list and types of dummy arguments. The interface

//does not guarantee what the actual function names are.

//Users can only rely on the function name following the

//keyword INTERFACE

 FUNCTION some_actual_function_7_INTVAL( Desc_Handle, Config_Param, INTVAL
)

 INTEGER :: some_actual_function_7_INTVAL

 Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle

 INTEGER, INTENT(IN) :: Config_Param

 INTEGER, INTENT(OUT) :: INTVAL

 END FUNCTION DFTI_GET_VALUE_INTVAL


 FUNCTION some_actual_function_7_SGLVAL( Desc_Handle, Config_Param, SGLVAL
)

 INTEGER :: some_actual_function_7_SGLVAL

 Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle

 INTEGER, INTENT(IN) :: Config_Param

 REAL, INTENT(OUT) :: SGLVAL

 END FUNCTION some_actual_function_7_SGLVAL
```

```
 FUNCTION some_actual_function_7_DBLVAL( Desc_Handle, Config_Param, DBLVAL
)
 INTEGER :: some_actual_function_7_DBLVAL
 Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 INTEGER, INTENT(IN) :: Config_Param
 REAL (KIND(0D0)), INTENT(OUT) :: DBLVAL
 END FUNCTION some_actual_function_7_DBLVAL

 FUNCTION some_actual_function_7_INTVEC( Desc_Handle, Config_Param, INTVEC
)
 INTEGER :: some_actual_function_7_INTVEC
 Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 INTEGER, INTENT(IN) :: Config_Param
 INTEGER, INTENT(OUT) :: INTVEC(*)
 END FUNCTION some_actual_function_7_INTVEC

 FUNCTION some_actual_function_7_INTPNT( Desc_Handle, Config_Param, INTPNT
)
 INTEGER :: some_actual_function_7_INTPNT
 Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 INTEGER, INTENT(IN) :: Config_Param
 INTEGER, DIMENSION(*), POINTER :: INTPNT
 END FUNCTION some_actual_function_7_INTPNT

 FUNCTION some_actual_function_7_CHARS( Desc_Handle, Config_Param, CHARS )
 INTEGER :: some_actual_function_7_CHARS
 Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
 INTEGER, INTENT(IN) :: Config_Param
 CHARCTER(*), INTENT(OUT):: CHARS
 END FUNCTION some_actual_function_7_CHARS
END INTERFACE DftiGetValue
```

```
/* C prototype */
long DftiGetValue( DFTI_DESCRIPTOR_HANDLE,
      DFTI_CONFIG_PARAM ,
      ... );
```

Configuration Settings

Each of the configuration parameters is identified by a named constant in the `MKL_DFTI` module. In C, these named constants have the enumeration type `DFTI_CONFIG_PARAM`. The list of configuration parameters whose values can be set by user is given in Table 11-3; the list of configuration parameters that are read-only is given in Table 11-4. All parameters are readable. Most of these parameters are self-explanatory, while some others are discussed more fully in the description of the relevant functions.

Table 11-3 Settable Configuration Parameters

Named Constants	Value Type	Comments
Most common configurations, no default, must be set explicitly		
`DFTI_PRECISION`	Named constant	Precision of computation
`DFTI_FORWARD_DOMAIN`	Named constant	Domain for the forward transform
`DFTI_DIMENSION`	Integer scalar	Dimension of the transform
`DFTI_LENGTHS`	Integer scalar/array	Lengths of each dimension
Common configurations including multiple transform and data representation		
`DFTI_NUMBER_OF_TRANSFORMS`	Integer scalar	For multiple number of transforms
`DFTI_FORWARD_SCALE`	Floating-point scalar	Scale factor for forward transform
`DFTI_BACKWARD_SCALE`	Floating-point scalar	Scale factor for backward transform

Named Constants	Value Type	Comments
DFTI_PLACEMENT	Named constant	Placement of the computation result
DFTI_COMPLEX_STORAGE	Named constant	Storage method, complex domain data
DFTI_REAL_STORAGE	Named constant	Storage method, real domain data
DFTI_CONJUGATE_EVEN_STORAGE	Named constant	Storage method, conjugate even domain data
DFTI_DESCRIPTOR_NAME	Character string	No longer than DFTI_MAX_NAME_LENGTH
DFTI_PACKED_FORMAT	Named constant	Packed format, real domain data
DFTI_NUMBER_OF_USER_THREADS	Integer scalar	Number of user threads employing the same descriptor for DFT computation
Configurations regarding stride of data		
DFTI_INPUT_DISTANCE	Integer scalar	Multiple transforms, distance of first elements
DFTI_OUTPUT_DISTANCE	Integer scalar	Multiple transforms, distance of first elements
DFTI_INPUT_STRIDES	Integer array	Stride information of input data
DFTI_OUTPUT_STRIDES	Integer array	Stride information of output data
Advanced configuration		
DFTI_ORDERING	Named constant	Scrambling of data order
DFTI_TRANSPOSE	Named constant	Scrambling of dimension

Table 11-4Read-Only Configuration Parameters

Named Constants	Value Type	Comments
DFTI_COMMIT_STATUS	Name constant	Whether descriptor has been committed
DFTI_VERSION	String	Intel MKL library version number

The configuration parameters are set by various values. Some of these values are specified by native data types such as an integer value (for example, number of simultaneous transforms requested), or a single-precision number (for example, the scale factor one would like to apply on a forward transform).

Other configuration values are discrete in nature (for example, the domain of the forward transform) and are thus provided in the DFTI module as named constants. In C, these named constants have the enumeration type DFTI_CONFIG_VALUE. The complete list of named constants used for this kind of configuration values is given in Table 11-5.

Table 11-5 Named Constant Configuration Values

Named Constant	Comments
DFTI_SINGLE	Single precision
DFTI_DOUBLE	Double precision
DFTI_COMPLEX	Complex domain
DFTI_REAL	Real domain
DFTI_INPLACE	Output overwrites input
DFTI_NOT_INPLACE	Output does not overwrite input
DFTI_COMPLEX_COMPLEX	Storage method (see Storage schemes)
DFTI_REAL_REAL	Storage method (see Storage schemes)
DFTI_COMPLEX_REAL	Storage method (see Storage schemes)
DFTI_REAL_COMPLEX	Storage method (see Storage schemes)
DFTI_COMMITTED	Committal status of a descriptor
DFTI_UNCOMMITTED	Committal status of a descriptor

Named Constant	Comments
DFTI_ORDERED	Data ordered in both forward and backward domains
DFTI_BACKWARD_SCRAMBLED	Data scrambled in backward domain (by forward transform)
DFTI_NONE	Used to specify no transposition
DFTI_CCS_FORMAT	Packed format, real data (see Packed formats)
DFTI_PACK_FORMAT	Packed format, real data (see Packed formats)
DFTI_PERM_FORMAT	Packed format, real data (see Packed formats)
DFTI_CCE_RORMAT	Packed format, real data (see Packed formats)
DFTI_VERSION_LENGTH	Number of characters for library version length
DFTI_MAX_NAME_LENGTH	Maximum descriptor name length
DFTI_MAX_MESSAGE_LENGTH	Maximum status message length

Table 11-6 lists the possible values for those configuration parameters that are discrete in nature.

Table 11-6 Settings for Discrete Configuration Parameters

Named Constant	Possible Values
DFTI_PRECISION	DFTI_SINGLE, or DFTI_DOUBLE (no default)
DFTI_FORWARD_DOMAIN	DFTI_COMPLEX, or DFTI_REAL
DFTI_PLACEMENT	DFTI_INPLACE (default), or DFTI_NOT_INPLACE
DFTI_COMPLEX_STORAGE	DFTI_COMPLEX_COMPLEX (default), or
DFTI_REAL_STORAGE	DFTI_REAL_REAL (default), or

Named Constant	Possible Values
	DFTI_REAL_COMPLEX
DFTI_CONJUGATE_EVEN_STORAGE	DFTI_COMPLEX_COMPLEX, or DFTI_COMPLEX_REAL (default)
DFTI_PACKED_FORMAT	DFTI_CCS_FORMAT (default), or DFTI_PACK_FORMAT, or DFTI_PERM_FORMAT, or DFTI_CCE_FORMAT

Table 11-7 lists the default values of the settable configuration parameters.

Table 11-7 Default Configuration Values of Settable Parameters

Named Constants	Default Value
DFTI_NUMBER_OF_TRANSFORMS	1
DFTI_NUMBER_OF_USER_THREADS	1
DFTI_FORWARD_SCALE	1.0
DFTI_BACKWARD_SCALE	1.0
DFTI_PLACEMENT	DFTI_INPLACE
DFTI_COMPLEX_STORAGE	DFTI_COMPLEX_COMPLEX
DFTI_REAL_STORAGE	DFTI_REAL_REAL
DFTI_CONJUGATE_EVEN_STORAGE	DFTI_COMPLEX_REAL
DFTI_PACKED_FORMAT	DFTI_CCS_FORMAT
DFTI_DESCRIPTOR_NAME	no name, string of zero length
DFTI_INPUT_DISTANCE	0
DFTI_OUTPUT_DISTANCE	0

Named Constants	Default Value
DFTI_INPUT_STRIDES	Tightly packed according to dimension and FFT lengths
DFTI_OUTPUT_STRIDES	Same as above. see Strides for details
DFTI_ORDERING	DFTI_ORDERED
DFTI_TRANSPOSE	DFTI_NONE

Precision of transform

The configuration parameter DFTI_PRECISION denotes the floating-point precision in which the transform is to be carried out. A setting of DFTI_SINGLE stands for single precision, and a setting of DFTI_DOUBLE stands for double precision. The data is meant to be presented in this precision; the computation will be carried out in this precision; and the result will be delivered in this precision. This is one of the four settable configuration parameters that do not have default values. The user must set them explicitly, most conveniently at the call to descriptor creation function DftiCreateDescriptor.

Forward domain of transform

The general form of the discrete Fourier transform is

$$z_{k_1, k_2, \ldots, k_d} = \sigma \times \sum_{j_d=0}^{n_d-1} \ldots \sum_{j_2=0}^{n_2-1} \sum_{j_1=0}^{n_1-1} w_{j_1, j_2, \ldots, j_d} \exp\left(\delta i 2\pi \sum_{l=1}^{d} j_l k_l / n_l \right)$$

for $k_1 = 0, \pm 1, \pm 2, \ldots$, where σ is an arbitrary real-valued scale factor and $\delta = \pm 1$. The forward transform is defined by $\sigma = 1$ and $\delta = -1$. In most common situations, the domain of the forward transform, that is, the set where the input (periodic) sequence $\{w_{j_1, j_2, \ldots, j_d}\}$ belongs, can be either the set of complex-valued sequences, real-valued sequences, and complex-valued conjugate even sequences. The configuration parameter DFTI_FORWARD_DOMAIN indicates the domain for the forward transform. Note that this implicitly specifies the domain for the backward transform because of mathematical property of the DFT. See Table 11-8 for details.

Table 11-8 Correspondence of Forward and Backward Domain

Forward Domain		Implied Backward Domain
Complex	(DFTI_COMPLEX)	Complex
Real	(DFTI_REAL)	Conjugate Even

On transforms in the real domain, some software packages only offer one "real-to-complex" transform. This in essence omits the conjugate even domain for the forward transform. The forward domain configuration parameter DFTI_FORWARD_DOMAIN is the second of four configuration parameters without default value.

Transform dimension and lengths

The dimension of the transform is a positive integer value represented in an integer scalar of Integer data type in Fortran and long data type in C. For one-dimensional transform, the transform length is specified by a positive integer value represented in an integer scalar of Integer data type in Fortran and long data type in C. For multi-dimensional (≥ 2) transform, the lengths of each of the dimension is supplied in an integer array (Integer data type in Fortran and long data type in C). DFTI_DIMENSION and DFTI_LENGTHS are the remaining two of four configuration parameters without default.

As mentioned, these four configuration parameters do not have default value. They are most conveniently set at the descriptor creation function. They can only be set in the descriptor creation function, and not by the function DftiSetValue.

Number of transforms

In some situations, the user may need to perform a number of DFT transforms of the same dimension and lengths. The most common situation would be to transform a number of one-dimensional data of the same length. This parameter has the default value of 1, and can be set to positive integer value by an Integer data type in Fortran and long data type in C. Data sets have no common elements. The distance parameter is obligatory if multiple number is more than one.

Scale

The forward transform and backward transform are each associated with a scale factor σ of its own with default value of 1. The user can set one or both of them via the two configuration parameters DFTI_FORWARD_SCALE and DFTI_BACKWARD_SCALE. For example, for a

one-dimensional transform of length n, one can use the default scale of 1 for the forward transform while setting the scale factor for backward transform to be $1/n$, making the backward transform the inverse of the forward transform.

The scale factor configuration parameter should be set by a real floating-point data type of the same precision as the value for `DFTI_PRECISION`.

Placement of result

By default, the computational functions overwrite the input data with the output result. That is, the default setting of the configuration parameter `DFTI_PLACEMENT` is `DFTI_INPLACE`. The user can change that by setting it to `DFTI_NOT_INPLACE`. Data sets have no common elements.

Packed formats

The result of the forward transform (i.e. in the frequency-domain) of real data is represented in several possible packed formats: ***Pack***, ***Perm***, ***CCS***, or ***CCE***. The data can be packed due to the symmetry property of the DFT transform of a real data.

The ***CCE*** format stores the values of the first half of the output complex conjugate-even signal resulted from the forward DFT. Note that the one-dimensional signal stored in `CCE` format is one complex element longer. For multi-dimensional real transform, `n1 * n2 * n3 * ... * nk` the size of complex matrix in `CCE` format is `(n1/2+1)* n2 * n3 * ... * nk` for Fortran and `n1 * n2 * ... * (nk/2+1)` for C.

The ***CCS*** format looks like the `CCE` format. It is the same format as `CCE` for one-dimensional transform. The `CCS` format is slightly different for multi-dimensional real transform. In `CCS` format, the output samples of the DFT are arranged as shown in Table 11-9 for one-dimensional DFT and in Table 11-10 for two-dimensional DFT.

The ***Pack*** format is a compact representation of a complex conjugate-symmetric sequence. The disadvantage of this format is that it is not the natural format used by the real DFT algorithms ("natural" in the sense that array is natural for complex DFTs). In `Pack` format, the output samples of the DFT are arranged as shown in Table 11-9 for one-dimensional DFT and in Table 11-11 for two-dimensional DFT.

The ***Perm*** format is an arbitrary permutation of the `Pack` format for even lengths and one is the same as the `Pack` format for odd lengths. In `Perm` format, the output samples of the DFT are arranged as shown in Table 11-9 for one-dimensional DFT and in Table 11-12 for two-dimensional DFT.

Table 11-9 Packed Format Output Samples

For n = s*2									
DFT Real	**0**	**1**	**2**	**3**	**...**	**n-2**	**n-1**	**n**	**n+1**

For n = s*2									
CCS	R_0	0	R_1	I_1	...	$R_{n/2-1}$	$I_{n/2-1}$	$R_{n/2}$	0
Pack	R_0	R_1	I_1	R_2	...	$I_{n/2-1}$	$R_{n/2}$		
Perm	R_0	$R_{n/2}$	R_1	I_1	...	$R_{n/2-1}$	$I_{n/2-1}$		

For n = s*2 + 1										
DFT Real	**0**	**1**	**2**	**3**	**...**	**n-4**	**n-3**	**n-2**	**n-1**	**n**	**n+1**
CCS	R_0	0	R_1	I_1	...	I_{s-2}	R_{s-1}	I_{s-1}	R_s	I_s	
Pack	R_0	R_1	I_1	R_2	...	R_{s-1}	I_{s-1}	R_s	I_s		
Perm	R_0	R_1	I_1	R_2	...	R_{s-1}	I_{s-1}	R_s	I_s		

Note that Table 11-9 uses the following notation for complex data entries:

R_j = Re z_j

I_j = Im z_j

See also Table 11-13 and Table 11-14.

Table 11-10 CCS Format Output Samples (Two-Dimensional Matrix (*m*+2)-by-(*n*+2))

For m = s*2, n = k*2								
z(1,1)	0	REz(1,2)	IMz(1,2)	...	REz(1,k)	IMz(1,k)	z(1,k+1)	0
0	0	0	0	...	0	0	0	0
REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)	n/u*	n/u
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)	n/u	n/u
...	...	...	...	...	...	...	n/u	n/u
REz(m/2,1)	REz(m/2,2)	REz(m/2,3)	REz(m/2,4)	...	REz(m/2,n-1)	REz(m/2,n)	n/u	n/u
IMz(m/2,1)	IMz(m/2,2)	IMz(m/2,3)	IMz(m/2,4)	...	IMz(m/2,n-1)	IMz(m/2,n)	n/u	n/u

For m = s*2, n = k*2								
z(m/2+1,1)	0	REz(m/2+1,2)	IMz(m/2+1,2)	...	REz(m/2+1,k)	IMz(m/2+1,k)	z(m/2+1,k+1)	0
0	0	0	0	...	0	0	n/u	n/u

For m = s*2+1, n = k*2								
z(1,1)	0	REz(1,2)	IMz(1,2)	...	REz(1,k)	IMz(1,k)	z(1,k+1)	0
0	0	0	0	...	0	0	0	0
REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)	n/u	n/u
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)	n/u	n/u
...	...	...	...	...	...	...	n/u	n/u
REz(s,1)	REz(s,2)	REz(s,3)	REz(s,4)	...	REz(s,n-1)	REz(s,n)	n/u	n/u
IMz(s,1)	IMz(s,2)	IMz(s,3)	IMz(s,4)	...	IMz(s,n-1)	IMz(s,n)	n/u	n/u

For m = s*2, n = k*2+1							
z(1,1)	0	REz(1,2)	IMz(1,2)	...	IMz(1,k-1)	REz(1,k)	IM z(1,k)
0	0	0	0	...	0	0	0
REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)	n/u*
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)	n/u
...	...	...	...	...	...	...	n/u
REz(m/2,1)	REz(m/2,2)	REz(m/2,3)	REz(m/2,4)	...	REz(m/2,n-1)	REz(m/2,n)	n/u
IMz(m/2,1)	IMz(m/2,2)	IMz(m/2,3)	IMz(m/2,4)	...	IMz(m/2,n-1)	IMz(m/2,n)	n/u
z(m/2+1,1)	0	REz(m/2+1,2)	IMz(m/2+1,2)	...	IMz(m/2+1,k-1)	REz(m/2+1,k)	IMz(m/2+1,k)
0	0	0	0	...	0	0	n/u

For m = s*2+1, n = k*2+1							
z(1,1)	0	REz(1,2)	IMz(1,2)	...	IMz(1,k-1)	REz(1,k)	IMz(1,k)
0	0	0	0	...	0	0	0
REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)	n/u
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)	n/u
...	...	...	...	...	...	...	n/u
REz(s,1)	REz(s,2)	REz(s,3)	REz(s,4)	...	REz(s,n-1)	REz(s,n)	n/u
IMz(s,1)	IMz(s,2)	IMz(s,3)	IMz(s,4)	...	IMz(s,n-1)	IMz(s,n)	n/u

* n/u - not used.

Note that in the Table 11-10, (`n`+2) columns are used for even `n = k*2`, while `n` columns are used for odd `n = k*2+1`. In the latter case, the first row is

```
z(1,1) 0 REz(1,2) IMz(1,2) ... REz(1,k) IMz(1,k)
```

If `m` is even, the (`m`+1)-th row is

```
z(m/2+1,1) 0 REz(m/2+1,2) IMz(m/2+1,2) ... REz(m/2+1,k) IMz(m/2+1,k)
```

Table 11-11 Pack Format Output Samples (Two-Dimensional Matrix *m*-by-*n*)

For m = s*2						
z(1,1)	REz(1,2)	IMz(1,2)	REz(1,3)	...	IMz(1,k)	z(1,k+1)
REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)
...	...	...	...	...	...	...
REz(m/2,1)	REz(m/2,2)	REz(m/2,3)	REz(m/2,4)	...	REz(m/2,n-1)	REz(m/2,n)
IMz(m/2,1)	IMz(m/2,2)	IMz(m/2,3)	IMz(m/2,4)	...	IMz(m/2,n-1)	IMz(m/2,n)
z(m/2+1,1)	REz(m/2+1,2)	IMz(m/2+1,2)	REz(m/2+1,3)	...	IMz(m/2+1,k)	z(m/2+1,k+1)

For m = s*2+1						
z(1,1)	REz(1,2)	IMz(1,2)	REz(1,3)	...	IMz(1,k)	z(1,n/2+1)
REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)
...	...	...	...	...	...	...
REz(s,1)	REz(s,2)	REz(s,3)	REz(s,4)	...	REz(s,n-1)	REz(s,n)
IMz(s,1)	IMz(s,2)	IMz(s,3)	IMz(s,4)	...	IMz(s,n-1)	IMz(s,n)

Table 11-12 Perm Format Output Samples (Two-Dimensional Matrix *m*-by-*n*)

For m = s*2						
z(1,1)	z(1,k+1)	REz(1,2)	IMz(1,2)	...	REz(1,k)	IMz(1,k)
z(m/2+1,1)	z(m/2+1,k+1)	REz(m/2+1,2)	IMz(m/2+1,2)	...	REz(m/2+1,k)	IMz(m/2+1,k)
REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)
...	...	...	...	...	...	...
REz(m/2,1)	REz(m/2,2)	REz(m/2,3)	REz(m/2,4)	...	REz(m/2,n-1)	REz(m/2,n)
IMz(m/2,1)	IMz(m/2,2)	IMz(m/2,3)	IMz(m/2,4)	...	IMz(m/2,n-1)	IMz(m/2,n)

For m = s*2+1						
z(1,1)	z(1,k+1)	REz(1,2)	IMz(1,2)	...	REz(1,k)	IMz(1,k)
REz(2,1)	REz(2,2)	REz(2,3)	REz(2,4)	...	REz(2,n-1)	REz(2,n)
IMz(2,1)	IMz(2,2)	IMz(2,3)	IMz(2,4)	...	IMz(2,n-1)	IMz(2,n)
...	...	...	...	...	...	...

For m = s*2+1						
REz(s,1)	REz(s,2)	REz(s,3)	REz(s,4)	...	REz(s,n-1)	REz(s,n)
IMz(s,1)	IMz(s,2)	IMz(s,3)	IMz(s,4)	...	IMz(s,n-1)	IMz(s,n)

Note that in the Table 11-11 and Table 11-12, for even number of columns `n = k*2`, while for odd number of columns `n = k*2+1` and the first row is

```
z(1,1) REz(1,2) IMz(1,2) ... REz(1,k) IMz(1,k)
```

If `m` is even, the last row in `Pack` format and the second row in `Perm` format is

```
z(m/2+1,1) REz(m/2+1,2) IMz(m/2+1,2) ... REz(m/2+1,k) IMz(m/2+1,k)
```

The tables for two-dimensional DFT use Fortran-interface conventions. For C-interface specifics in storing packed data, see Storage schemes section below. See also Table 11-15 and Table 11-16 for examples of Fortran-interface and C-interface formats.

Storage schemes

For each of the three domains `DFTI_COMPLEX`, `DFTI_REAL`, and `DFTI_CONJUGATE_EVEN` (for the forward as well as the backward operator), a subset of the four storage schemes `DFTI_COMPLEX_COMPLEX`, `DFTI_COMPLEX_REAL`, `DFTI_REAL_COMPLEX`, and `DFTI_REAL_REAL` is provided. Specific examples are presented here to illustrate the storage schemes. See the document [3] for the rationale behind this definition of the storage schemes.

NOTE. The data is stored in the Fortran style only, that is, the real and imaginary parts are stored side by side.

Storage scheme for complex domain. This setting is recorded in the configuration parameter `DFTI_COMPLEX_STORAGE`. The three values that can be set are `DFTI_COMPLEX_COMPLEX`, `DFTI_COMPLEX_REAL`, and `DFTI_REAL_REAL`. Consider a one-dimensional `n`-length transform of the form

$$z_k = \sum_{j=0}^{n-1} w_j e^{-i2\pi jk/n}, \qquad w_j, z_k \in \mathbb{C}.$$

Assume the stride has default value (unit stride) and DFTI_PLACEMENT has the default in-place setting.

DFTI_COMPLEX_COMPLEX storage scheme (by default). A typical usage will be as follows.

```
COMPLEX :: X(0:n-1)
...some other code...
Status = DftiComputeForward( Desc_Handle, X )
```

On input,

$X(j) = w_j$, $j = 0,1,\ldots,n-1$.

On output,

$X(k) = z_k$, $k = 0,1,\ldots,n-1$.

Storage scheme for the real and conjugate even domains. This setting for the storage schemes for these domains is recorded in the configuration parameters DFTI_REAL_STORAGE and DFTI_CONJUGATE_EVEN_STORAGE. Since a forward real domain corresponds to a conjugate even backward domain, they are considered together. The example uses one-, two- and three-dimensional real to conjugate even transforms. In-place computation is assumed whenever possible (that is, when the input data type matches the output data type).

One-Dimensional Transform

Consider a one-dimensional n-length transform of the form

$$z_k = \sum_{j=0}^{n-1} w_j e^{-i2\pi jk/n}, \qquad w_j \in R, z_k \in C.$$

There is a symmetry:

For even n: $z(n/2+i) = conjg(z(n/2-i))$, $1 \le i \le n/2-1$, and moreover z(0) and z(n/2) are real values.

For odd n: $z(m+i) = conjg(z(m-i+1))$, $1 \le i \le m$, and moreover z(0) is real value.

$m = floor(n/2)$.

Table 11-13 Comparison of the Storage Effects of Complex-to-Complex and Real-to-Complex DFTs for Forward Transform

N=8

Input Vectors			**Output Vectors**				
Complex DFT		**Real DFT**	**complex DFT**		**real DFT**		
Complex Data		**Real Data**	**Complex Data**		**Real Data**		
Real	**Imaginary**		**Real**	**Imaginary**	**CCS**	**Pack**	**Perm**
w0	0.000000	w0	z0	0.000000	z0	z0	z0
w1	0.000000	w1	Re(z1)	Im(z1)	0.000000	Re(z1)	z4
w2	0.000000	w2	Re(z2)	Im(z2)	Re(z1)	Im(z1)	Re(z1)
w3	0.000000	w3	Re(z3)	Im(z3)	Im(z1)	Re(z2)	Im(z1)
w4	0.000000	w4	z4	0.000000	Re(z2)	Im(z2)	Re(z2)
w5	0.000000	w5	Re(z3)	-Im(z3)	Im(z2)	Re(z3)	Im(z2)
w6	0.000000	w6	Re(z2)	-Im(z2)	Re(z3)	Im(z3)	Re(z3)
w7	0.000000	w7	Re(z1)	-Im(z1)	Im(z3)	z4	Im(z3)
					z4		
					0.000000		

N=7

Input Vectors			**Output Vectors**		
Complex DFT		**Real DFT**	**complex DFT**		**real DFT**

N=7							
Complex Data		**Real Data**	**Complex Data**		**Real Data**		
Real	**Imaginary**		**Real**	**Imaginary**	**CCS**	**Pack**	**Perm**
w0	0.000000	w0	z0	0.000000	z0	z0	z0
w1	0.000000	w1	Re(z1)	Im(z1)	0.000000	Re(z1)	Re(z1)
w2	0.000000	w2	Re(z2)	Im(z2)	Re(z1)	Im(z1)	Im(z1)
w3	0.000000	w3	Re(z3)	Im(z3)	Im(z1)	Re(z2)	Re(z2)
w4	0.000000	w4	Re(z3)	-Im(z3)	Re(z2)	Im(z2)	Im(z2)
w5	0.000000	w5	Re(z2)	-Im(z2)	Im(z2)	Re(z3)	Re(z3)
w6	0.000000	w6	Re(z1)	-Im(z1)	Re(z3)	Im(z3)	Im(z3)
					Im(z3)		

Table 11-14 Comparison of the Storage Effects of Complex-to-Complex and Complex-to-Real DFTs for Backward Transform

N=8							
Input Vectors			**Output Vectors**				
Complex DFT		**Real DFT**	**complex DFT**		**real DFT**		
Complex Data		**Real Data**	**Complex Data**		**Real Data**		
Real	**Imaginary**		**Real**	**Imaginary**	**CCS**	**Pack**	**Perm**
w0	0.000000	w0	z0	0.000000	z0	z0	z0

N=8							
w1	0.000000	w1	Re(z1)	Im(z1)	0.000000	Re(z1)	z4
w2	0.000000	w2	Re(z2)	Im(z2)	Re(z1)	Im(z1)	Re(z1)
w3	0.000000	w3	Re(z3)	Im(z3)	Im(z1)	Re(z2)	Im(z1)
w4	0.000000	w4	z4		Re(z2)	Im(z2)	Re(z2)
w5	0.000000	w5	Re(z3)	-Im(z3)	Im(z2)	Re(z3)	Im(z2)
w6	0.000000	w6	Re(z2)	-Im(z2)	Re(z3)	Im(z3)	Re(z3)
w7	0.00000	w7	Re(z1)	-Im(z1)	Im(z3)	z4	Im(z3)
					z4		
					0.000000		

N=7							
Input Vectors			**Output Vectors**				
Complex DFT		**Real DFT**	**complex DFT**		**real DFT**		
Complex Data		**Real Data**	**Complex Data**		**Real Data**		
Real	**Imaginary**		**Real**	**Imaginary**	**CCS**	**Pack**	**Perm**
w0	0.000000	w0	z0	0.000000	z0	z0	z0
w1	0.000000	w1	Re(z1)	Im(z1)	0.000000	Re(z1)	Re(z1)
w2	0.000000	w2	Re(z2)	Im(z2)	Re(z1)	Im(z1)	Im(z1)
w3	0.000000	w3	Re(z3)	Im(z3)	Im(z1)	Re(z2)	Re(z2)

N=7							
w4	0.000000	w4	Re(z3)	-Im(z3)	Re(z2)	Im(z2)	Im(z2)
w5	0.000000	w5	Re(z2)	-Im(z2)	Im(z2)	Re(z3)	Re(z3)
w6	0.000000	w6	Re(z1)	-Im(z1)	Re(z3)	Im(z3)	Im(z3)
					Im(z3)		

Assume that the stride has the default value (unit stride).

This complex conjugate-symmetric vector can be stored in the complex array of size m+1 or in the real array of size 2m+2 or 2m depending on packed format.

Two-Dimensional Transform

Each of the real-to-complex routines computes the forward DFT of a two-dimensional real matrix according to the mathematical equation

$$z_{i,j} = \sum_{k=0}^{m-1} \sum_{l=0}^{n-1} t_{k,l} * w_m^{-i*k} * w_n^{-j*l}, 0 \le i \le m\text{-}1, 0 \le j \le n\text{-}1$$

$t_{k,l} = cmplx(r_{k,l},0)$, where $r_{k,l}$ is a real input matrix, $0 \le k \le m-1$, $0 \le l \le n-1$. The mathematical result $z_{i,j}$, $0 \le i \le m-1$, $0 \le j \le n-1$, is the complex matrix of size (m,n). Each column is the complex conjugate-symmetric vector as follows:

For even m:

for $0 \le j \le n-1$,

$z(m/2+i,j) = conjg(z(m/2-i,j)), 1 \le i \le m/2-1$.

Moreover, $z(0,j)$ and $z(m/2,j)$ are real values for $j=0$ and $j=n/2$.

For odd m:

for $0 \le j \le n-1$,

$z(s+i,j) = conjg(z(s-i,j)), 1 \le i \le s-1$,

where `s = floor(m/2)`.

Moreover, `z(0,j)` are real values for `j=0` and `j=n/2`.

This mathematical result can be stored in the real two-dimensional array of size:

`(m+2,n+2)` (`CCS` format), or

`(m,n)` (`Pack` or `Perm` formats), or

`(2*(m/1+1), n)` (`CCE` format, Fortran-interface),

`((m, 2*(n/2+1))` (`CCE` format, C-interface)

or in the complex two-dimensional array of size:

`(m/2+1, n)` (`CCE` format, Fortran-interface),

`(m, n/2+1)` (`CCE` format, C-interface)

Since the multidimensional array data are arranged differently in Fortran and C (see Strides), the output array that holds the computational result contains complex conjugate-symmetric columns (for Fortran) or complex conjugate-symmetric rows (for C).

The following tables give examples of output data layout in `Pack` format for a forward two-dimensional real-to-complex DFT of a 6-by-4 real matrix. Note that the same layout is used for the input data of the corresponding backward complex-to-real DFT.

Table 11-15 Fortran-interface Data Layout for a 6-by-4 Matrix

z(1,1)	Re z(1,2)	Im z(1,2)	z(1,3)
Re z(2,1)	Re z(2,2)	Re z(2,3)	Re z(2,4)
Im z(2,1)	Im z(2,2)	Im z(2,3)	Im z(2,4)
Re z(3,1)	Re z(3,2)	Re z(3,3)	Re z(3,4)
Im z(3,1)	Im z(3,2)	Im z(3,3)	Im z(3,4)
z(4,1)	Re z(4,2)	Im z(4,2)	z(4,3)

For the above example, the stride array is taken to be (0, 1, 6).

Table 11-16 C-interface Data Layout for a 6-by-4 Matrix

z(1,1)	Re z(1,2)	Im z(1,2)	z(1,3)

Re z(2,1)	Re z(2,2)	Im z(2,2)	Re z(2,3)
Im z(2,1)	Re z(3,2)	Im z(3,2)	Im z(2,3)
Re z(3,1)	Re z(4,2)	Im z(4,2)	Re z(3,3)
Im z(3,1)	Re z(5,2)	Im z(5,2)	Im z(3,3)
z(4,1)	Re z(6,2)	Im z(6,2)	z(4,3)

For the second example, the stride array is taken to be (0, 4, 1).

See also Packed formats.

Three-Dimensional Transform

Each of the real-to-complex routines computes the forward DFT of a three-dimensional real matrix according to the mathematical equation

$$z_{i,j,q} = \sum_{k=0}^{m-1} \sum_{l=0}^{n-1} \sum_{s=0}^{k-1} t_{p,l,s} * W_m^{-i*p} * W_n^{-j*l} * W_k^{-q*s},$$
$$0 \le i \le m\text{-}1, 0 \le j \le n\text{-}1, 0 \le s \le k\text{-}1$$

$t_{p,l,s} = \mathrm{cmplx}(r_{p,l,s},0)$, where $r_{p,l,s}$ is a real input matrix, $0 \le k \le m-1$, $0 \le l \le n-1$, $0 \le s \le k-1$. The mathematical result $z_{i,j,q}$, $0 \le i \le m-1$, $0 \le j \le n-1$, $0 \le q \le k-1$ is the complex matrix of size (m,n,k), which is a complex conjugate-symmetric, or conjugate even, matrix as follows:

$z_{m1,n1,k1} = \mathrm{conjg}(z_{m-m1,n-n1,k-k1})$, where each dimension is periodic.

This mathematical result can be stored in the real three-dimensional array of size:

- $(m/2+1,n,k)$ (CCE format, Fortran-interface),
- $(m,n,k/2+1)$ (CCE format, C-interface).

Since the multidimensional array data are arranged differently in Fortran and C (see Strides), the output array that holds the computational result contains complex conjugate-symmetric columns (for Fortran) or complex conjugate-symmetric rows (for C).

NOTE. There is one packed format for 3D REAL DFT - `CCE` format. In both in-place and out-of-place REAL DFT, for real data, the stride and distance parameters are in `REAL` units and for complex data, they are in `COMPLEX` units. So, elements of input and output data can be placed in different elements of input-output array of the in-place FFT.

1. **`DFTI_REAL_REAL`** for real domain, **`DFTI_COMPLEX_REAL`** for conjugate even domain (by default). It is used for 1D and 2D REAL DFT.

   - A typical usage of in-place transform is as follows:

     ```
     // m = floor( n/2 )
     REAL :: X(0:2*m+1)
     ...some other code...
     ...assuming inplace transform...
     Status = DftiComputeForward( Desc_Handle, X )
     ```

     On input,

     `X(j) = wj, j = 0,1,...,n-1.`

     On output,

     Output data stored in one of formats: `Pack`, `Perm` or `CCS` (see Packed formats).

     CCS format: $X(2*k) = Re(z_k)$, $X(2*k+1) = Im(z_k)$, $k = 0,1,...,m$.

     Pack format:

     even n: $X(0) = Re(z_0)$, $X(2*k-1) = Re(z_k)$, $X(2*k) = Im(z_k)$, $k = 1,...,m-1$, and $X(n-1) = Re(z_m)$

     odd n: $X(0) = Re(z_0)$, $X(2*k-1) = Re(z_k)$, $X(2*k) = Im(z_k)$, $k = 1,...,m$

     Perm format:

     even n: $X(0) = Re(z_0)$, $X(1) = Re(z_m)$, $X(2*k) = Re(z_k)$, $X(2*k+1) = Im(z_k)$, $k = 1,...,m-1$,

     odd n: $X(0) = Re(z_0)$, $X(2*k-1) = Re(z_k)$, $X(2*k) = Im(z_k)$, $k = 1,...,m$.

     See Example C-16, Example C-17, Example C-18, and Example C-19.

     Input and output data exchange the roles in the backward transform.

- A typical usage of out-of-place transform is as follows:

```
// m = floor( n/2 )
REAL :: X(0:n-1)
REAL :: Y(0:2*m+1)
...some other code...
...assuming out-of-place transform...
Status = DftiComputeForward( Desc_Handle, X, Y )
```

On input, `X(j) = wj, j = 0,1,...,n-1.`

On output,

Output data stored in one of formats: `Pack`, `Perm` or `CCS` (see Packed formats).

CCS format: `Y(2*k) = Re(`z_k`) , Y(2*k+1) = Im(`z_k`) , k = 0,1,...,m.`

Pack format:

even n: `Y(0) = Re(`z_0`), Y(2*k-1) = Re(`z_k`), Y(2*k) = Im(`z_k`), k = 1,...,m-1, and Y(n-1) = Re(`z_m`)`

odd n: `Y(0) = Re(`z_0`), Y(2*k-1) = Re(`z_k`), Y(2*k) = Im(`z_k`), k = 1,...,m`

Perm format:

even n: `Y(0) = Re(`z_0`), Y(1) = Re(`z_m`), Y(2*k) = Re(`z_k`) , Y(2*k+1) = Im(`z_k`) , k = 1,...,m-1,`

odd n: `Y(0) = Re(`z_0`), Y(2*k-1) = Re(`z_k`), Y(2*k) = Im(`z_k`), k = 1,...,m.`

Notice that if the stride of the output array is not set to the default value unit stride, the real and imaginary parts of one complex element will be placed with this stride.

For example:

CCS format: `Y(2*k*s) = Re(`z_k`) , Y((2*k+1)*s) = Im(`z_k`) , k = 0,1, ..., m, s - stride.`

See Example C-16a and Example C-17a.

Input and output data exchange the roles in the backward transform.

2. **`DFTI_REAL_REAL`** for real domain, **`DFTI_COMPLEX_COMPLEX`** for conjugate even domain. It is used for 1D, 2D and 3D REAL DFT. The CCE format is set by default. You must explicitly set the storage scheme in this case, because its value is not the default one.

- A typical usage of in-place transform is as follows:

```
// m = floor( n/2 )

REAL :: X(0:m*2)

...some other code...

...assuming in-place transform...

Status = DftiSetValue( Desc_Handle,  DFTI_CONJUGATE_EVEN_STORAGE,
DFTI_COMPLEX_COMPLEX)

...

Status = DftiComputeForward( Desc_Handle, X)
```

  On input,

  $X(j) = wj$, $j = 0,1,...,n-1$.

  On output,

  $X(2*k) = Re(z_k)$, $X(2*k+1) = Im(z_k)$, $k = 0,1,...,m$.

  See Example C-24.

  Input and output data exchange the roles in the backward transform.

- A typical usage of out-of-place transform is as follows:

```
// m = floor( n/2 )

REAL :: X(0:n-1)

COMPLEX :: Y(0:m)

...some other code...

...assuming out-of-place transform...

Status = DftiSetValue( Desc_Handle,  DFTI_CONJUGATE_EVEN_STORAGE,
DFTI_COMPLEX_COMPLEX)

...

Status = DftiComputeForward( Desc_Handle, X, Y )
```

  On input,

  $X(j) = wj$, $j = 0,1,...,n-1$.

  On output,

  $Y(k) = zk$, $k = 0,1,...,m$.

See Example C-24a and Example C-25

Input and output data exchange the roles in the backward transform.

3. **DFTI_REAL_COMPLEX** for real domain, **DFTI_COMPLEX_COMPLEX** for conjugate even domain. It is not used in the current version. See Note in the "DFT Interface" section for details. A typical usage is as follows:

```
// m = floor( n/2 )
COMPLEX :: X(0:m)
...some other code...
...inplace transform...
Status = DftiComputeForward( Desc_Handle, X )
```

On input,

`X(j) = wj, j = 0,1,...,n-1.`

That is, the imaginary parts of `X(j)` are zero.

On output,

`Y(k) = zk, k = 0,1,...,m,`

where `m` is `floor( n/2 )`.

Number of user threads

Customer application can be parallelized by using the following techniques:

1. You do not create threads in your application but specify the parallel mode within the DFT module of Intel MKL. See Intel MKL User's Guide document for more information on how to do this.
2. You create threads in application yourself and have each thread perform all stages of DFT implementation including descriptor initialization, DFT computation, and descriptor deallocation. In this case each descriptor is used only within its corresponding thread.
3. You create threads after initializing the DFT descriptor. This implies that threading is employed for parallel DFT computation only, and the descriptor is freed after return from the parallel region. In this case each thread uses the same descriptor.

For the first and second cases listed above, set the parameter `DFTI_NUMBER_OF_USER_THREADS` to 1 (its default value), since each particular descriptor instance is used only in a single thread.

In case 3, you must use the `DftiSetValue()` function to set the `DFTI_NUMBER_OF_USER_THREADS` to the actual number of DFT computation threads, because multiple threads will be using the same descriptor. If this setting is not done, your program will work incorrectly or fail, since the descriptor contains individual data for each thread.

1. It is not recommended to simultaneously parallelize your program and employ the Intel MKL internal threading because this will slow down performance. Note that in case 3 above, DFT computation is automatically initiated in a single threading mode.
2. You must not change the number of threads after the `DftiCommitDescriptor()` function completed DFT initialization.

See Example C-26, Example C-27, and Example C-28 in Appendix C.

Input and output distances

DFT interface in Intel MKL allows the computation of multiple number of transforms. Consequently, the user needs to be able to specify the data distribution of these multiple sets of data. This is accomplished by the distance between the first data element of the consecutive data sets. This parameter is obligatory if multiple number is more than one. The parameter is a value of `Integer` data type in Fortran and `long` data type in C. Data sets don't have any common elements. The following example illustrates the specification. Consider computing the forward DFT on three 32-length complex sequences stored in `X(0:31, 1)`, `X(0:31, 2)`, and `X(0:31, 3)`. Suppose the results are to be stored in the locations `Y(0:31, k)`, `k` = 1, 2, 3, of the array `Y(0:63, 3)`. Thus the input distance is 32, while the output distance is 64. Notice that the data and result parameters in computation functions are all declared as assumed-size rank-1 array `DIMENSION(0:*)`. Therefore two-dimensional array must be transformed to one-dimensional array by `EQUIVALENCE` statement or other facilities of Fortran. Here is the code fragment:

```
Complex :: X_2D(0:31,3), Y_2D(0:63, 3)

Complex :: X(96), Y(192)
Equivalence (X_2D, X)
Equivalence (Y_2D, Y)
.....................
Status = DftiCreateDescriptor(Desc_Handle, DFTI_SINGLE,
```

```
 DFTI_COMPLEX, 1, 32)
Status = DftiSetValue(Desc_Handle, DFTI_NUMBER_OF_TRANSFORMS, 3)
Status = DftiSetValue(Desc_Handle, DFTI_INPUT_DISTANCE, 32)
Status = DftiSetValue(Desc_Handle, DFTI_OUTPUT_DISTANCE, 64)
Status = DftiSetValue(Desc_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE)
Status = DftiCommitDescriptor(Desc_Handle)
Status = DftiComputeForward(Desc_Handle, X, Y)
Status = DftiFreeDescriptor(Desc_Handle)
```

Strides

In addition to supporting transforms of multiple number of datasets, DFT interface supports non-unit stride distribution of data within each data set. The parameter is an array of values of `Integer` data type in Fortran and `long` data type in C. Consider the following situation where a 32-length DFT is to be computed on the sequence `xj, 0≤j<32`. The actual location of these values are in `X(5), X(7), ..., X(67)` of an array `X(1:68)`. The stride accommodated by DFT interface consists of a displacement from the first element of the data array *`L0`*, (4 in this case), and a constant distance of consecutive elements *`L1`* (2 in this case). Thus for the Fortran array `X`

`xj = X(1 + L0 + L1 * j) = X(5 + L1 * j).`

This stride vector (4,2) is provided by a length-2 rank-1 integer array:

```
COMPLEX :: X(68)
INTEGER :: Stride(2)
.....................
Status = DftiCreateDescriptor(Desc_Handle, DFTI_SINGLE,
 DFTI_COMPLEX, 1, 32)
Stride = (/ 4, 2 /)
Status = DftiSetValue(Desc_Handle, DFTI_INPUT_STRIDES, Stride)
Status = DftiSetValue(Desc_Handle, DFTI_OUTPUT_STRIDES, Stride)
Status = DftiCommitDescriptor(Desc_Handle)
Status = DftiComputeForward(Desc_Handle, X)
Status = DftiFreeDescriptor(Desc_Handle)
```

In general, for a d-dimensional transform, the stride is provided by a d+1-length integer vector ($L0$, $L1$, $L2$, ..., Ld) with the meaning:

$L0$ = displacement from the first array element

$L1$ = distance between consecutive data elements in the first dimension

$L2$ = distance between consecutive data elements in the second dimension

... = ...

Ld = distance between consecutive data elements in the d-th dimension.

A d-dimensional data sequence

$X_{j_1, j_2, \ldots, j_d}$, $0 \le j_i \le J_i$, $1 \le i \le d$

will be stored in the rank-1 array X by the mapping

$X_{j_1, j_2, \ldots, j_d} = X$ (first index + $L0 + j_1 L1 + j_2 L2 + \ldots + j_d Ld$) .

For multiple transforms, the value $L0$ applies to the first data sequence, and Lj, j = 1, 2,..., d apply to all the data sequences.

In the case of a single one-dimensional sequence, $L1$ is simply the usual stride. The default setting of strides in the general multi-dimensional situation corresponds to the case where the sequences are distributed tightly into the array:

$$L1 = 1, L2 = J_1, L3 = J_1 J_2, \ldots, Ld = \prod_{i=1}^{d-1} J_i$$

Both the input data and output data have a stride associated with it. The default is set in accordance with the data to be stored contiguously in memory in a way that is natural to the language.

Note that in case of a real FFT, where different formats are available, the default value is not the one that seems most natural for certain formats. For example, with the CCE format, strides are set by default to *L1* = 1, *L2* = *J1* for a real transform regardless. However, for a complex matrix, slightly over half of the matrix is actually stored, and you should set strides to *L1* = 1, *L2* = *J1*/2+1. In case of an in-place transform with the CCE data format, even for a real array, you should set strides explicitly: *L1* = 1, *L2* = (*J1*/2+1)*2.

See Example C-23 as an illustration of how to use the configuration parameters discussed above. See Storage schemes and Packed formats on how to define arrays for different formats.

Ordering

It is well known that a number of FFT algorithms apply an explicit permutation stage that is time consuming [4]. The exclusion of this step is similar to applying DFT to input data whose order is scrambled, or allowing a scrambled order of the DFT results. In applications such as convolution and power spectrum calculation, the order of result or data is unimportant and thus permission of scrambled order is attractive if it leads to higher performance. The following options are available in Intel MKL:

- DFTI_ORDERED: Forward transform data ordered, backward transform data ordered (default option).
- DFTI_BACKWARD_SCRAMBLED: Forward transform data ordered, backward transform data scrambled.

Table 11-17 tabulates the effect on this configuration setting.

Table 11-17 Scrambled Order Transform

	DftiComputeForward	**DftiComputeBackward**
DFTI_ORDERING	Input → Output	Input → Output
DFTI_ORDERED	ordered → ordered	ordered → ordered
DFTI_BACKWARD_SCRAMBLED	ordered → scrambled	scrambled → ordered

Note that meaning of the latter two options are "allow scrambled order if practical." There are situations where in fact allowing out of order data gives no performance advantage, and thus an implementation may choose to ignore the suggestion. Strictly speaking, the normal order is also a scrambled order, the trivial one.

Transposition

This is an option that allows for the result of a high-dimensional transform to be presented in a transposed manner. The default setting is `DFTI_NONE` and can be set to `DFTI_ALLOW`. Similar to that of scrambled order, sometimes in higher dimension transform, performance can be gained if the result is delivered in a transposed manner. DFT interface offers an option for the output be returned in a transposed form if performance gain is possible. Since the generic stride specification is naturally suited for representation of transposition, this option allows the strides for the output to be possibly different from those originally specified by the user. Consider an example where a two-dimensional result

$$Y_{j_1,j_2},\ 0 \le j_i < n_i,$$

is expected. Originally the user specified that the result be distributed in the (flat) array Y in with generic strides $L_1 = 1$ and $L_2 = n_1$. With the transposition option, the computation may actually return the result into Y with stride $L_1 = n_2$ and $L_2 = 1$. These strides can be obtained from an appropriate inquiry function. Note also that in dimension 3 and above, transposition means an arbitrary permutation of the dimension.

Cluster DFT Functions

This section describes the cluster Discrete Fourier Transform (DFT) functions implemented in Intel MKL (available with Intel® MKL Cluster Edition for Linux* and Windows*).

Starting from version 9.0, Intel MKL provides two versions of cluster DFT interface:

- The new version, which is incompatible with the interfaces implemented in Intel MKL versions lower than 9.0.
- The interface introduced in Intel MKL 8.1, which is supported for backward compatibility.

This section describes the new version of the cluster DFT interface. Documentation on the elder version is available on the Web at http://www.intel.com/cd/software/products/asmo-na/eng/perflib/mkl/219843.htm.

The cluster DFT function library was designed to perform Discrete Fourier Transform on a cluster, that is, a group of computers interconnected via a network. Each computer (node) in the cluster has its own memory and processor(s). Data interchanges between the nodes are provided by the network.

One or more processes may be running in parallel on each cluster node. To organize communication between different processes, the cluster DFT function library uses Message Passing Interface (MPI). Given the number of available MPI implementations (for example, MPICH, Intel® MPI and others), Cluster DFT works with MPI via a message-passing library for linear algebra, called BLACS, to avoid dependence on a specific MPI implementation.

The cluster Discrete Fourier Transform function library of Intel MKL provides one-dimensional, two-dimensional, and multi-dimensional (up to the order of 7) routines and both Fortran and C interfaces for all transform functions.

To develop applications using cluster DFT, you should have basic knowledge of and skills in MPI programming.

The interfaces for Intel Cluster MKL DFT functions are very similar to the corresponding interfaces for conventional MKL DFT Functions described earlier in this chapter. You can refer there for details not explained in this section.

The full list of cluster DFT functions implemented in Intel MKL is given in Table 11-18:

Table 11-18 Cluster DFT Functions in Intel MKL

Function Name	Operation
Descriptor Manipulation Functions	
DftiCreateDescriptorDM	Allocates memory for the descriptor data structure and instantiates it with default configuration settings.
DftiCommitDescriptorDM	Performs all initialization that facilitates the actual DFT computation.
DftiFreeDescriptorDM	Frees memory allocated for a descriptor.
DFT Computation Functions	
DftiComputeForwardDM	Computes the forward DFT.
DftiComputeBackwardDM	Computes the backward DFT.
Descriptor Configuration Functions	
DftiSetValueDM	Sets one particular configuration parameter with the specified configuration value.
DftiGetValueDM	Gets the configuration value of one particular configuration parameter.

Computing Cluster DFT

The cluster DFT functions described later in this section are implemented in Fortran and C interface. Fortran stands for Fortran 95.

Cluster DFT computation is performed by `DftiComputeForwardDM` and `DftiComputeBackwardDM` functions, called in a program using MPI, which will be referred to as MPI program. After an MPI program starts, a number of processes is created. MPI identifies each process by its rank. The processes are independent of one another and communicate via MPI. A function called in an MPI program is invoked in all the processes. Each process figures out what to do using its rank. Input or output data for a cluster DFT transform is a sequence of complex values. A cluster DFT computation function operates local part of the input data, i.e. some part of the data to be operated in a particular process, as well as generates local part of the output data. Each process performs its part of computations. Running in parallel and communicating through MPI, the processes perform the entire DFT computation. DFT computations using Intel MKL cluster DFT functions, should typically be effected by a number of steps listed below:

1. Initiate MPI by calling `MPI_Init` in C/C++ or `MPI_INIT` in Fortran (the function must be called prior to calling any DFT function and any MPI function).
2. Allocate memory for the descriptor by calling `DftiCreateDescriptorDM`.
3. Specify a value(s) of configuration parameters by a call(s) to `DftiSetValueDM`.
4. Obtain values of configuration parameters needed to create local data arrays; the values are retrieved by calls to `DftiGetValueDM`.
5. Perform initialization that facilitates DFT computation by a call to `DftiCommitDescriptorDM`.
6. Create arrays for local parts of input and output data and fill the local part of input data with values (For more information, see Distributing Data among Processes.)
7. Compute the transform by calling `DftiComputeForwardDM` or `DftiComputeBackwardDM`.
8. Gather local output data into the global array using MPI functions or employ the data otherwise.
9. Release memory allocated for a descriptor by a call to `DftiFreeDescriptorDM`.
10. Finalize communication through MPI by calling `MPI_Finalize` in C/C++ or `MPI_FINALIZE` in Fortran (the function must be called after the last call to a cluster DFT function and the last call to an MPI function).

Several code examples of using the cluster DFT interface functions in the "Examples for Cluster DFT Functions" section in Appendix C illustrate cluster DFT computations.

Distributing Data among Processes

Intel MKL cluster DFT stores all input and output multi-dimensional arrays (matrices) in one-dimensional arrays (vectors). The arrays are stored in the row-major order in C/C++ and in the column-major order in Fortran. For example, a two-dimensional matrix `A` of size (*m*,*n*) will be stored in a vector `B` of size *m***n* so that

- `B[i*n+j]=A[i][j]` in C/C++ (*i=0, ... , m-1, j=0, ... , n-1*)
- `B(j*m+i)=A(i,j)` in Fortran (*i=1, ... , m, j=1, ... , n*).

NOTE. Order of FFT dimensions is the same as the order of array dimensions in the programming language. For example, a 3-dimensional FFT with Lengths=(*m*,*n*,*l*) can be computed over an array `Ar[m][n][l]` in C/C++ or `AR(m,n,l)` in Fortran.

All MPI processes involved in cluster DFT computation operate their own portions of data. These local arrays make up the virtual global array that the fast Fourier transform is applied to. It is your responsibility to properly allocate local arrays (if needed), fill them with initial data and gather resulting data into an actual global array or process the resulting data otherwise. To be able do this, you should grasp how the virtual global array is composed of the local ones.

Multi-dimensional transforms

If the dimension of transform is greater than one, cluster DFT splits data in the dimension whose index changes most slowly, so that the parts contain all elements with several consecutive values of this index. It is the first dimension in C and the last one in Fortran. If the global array is two-dimensional, in C, it gives each process several consecutive rows. The term “rows” will be used regardless of the array dimension and programming language. Local arrays are placed in memory allocated for the virtual global array consecutively, in the order determined by process ranks. For example, in case of two processes, during the computation of a three-dimensional transform whose matrix has size (11,15,12), the processes may store local arrays of sizes (6,15,12) and (5,15,12), respectively.

Let *p* be the number of MPI processes and the matrix of a transform to be computed have size (*m*,*n*,*l*). Then, in C, each MPI process will work with local data array of size (m_q , *n*, *l*), where $\Sigma m_q=m$, *q=0, ... , p-1*. Local input arrays should contain appropriate parts of the actual global input array. Then local output arrays will contain appropriate parts of the actual global output array. You can figure out which particular rows of the global array the local array should contain

from the following configuration parameters of the cluster DFT interface: CDFT_LOCAL_NX, CDFT_LOCAL_START_X, and CDFT_LOCAL_SIZE. To retrieve values of the parameters, you should use the DftiGetValueDM function:

- CDFT_LOCAL_NX specifies how many rows of the global array the current process receives.
- CDFT_LOCAL_START_X specifies which row of the global input or output array corresponds to the first row of the local input or output array. If A is a global array and L is the appropriate local array, then
  - L[i][j][k]=A[i+cdft_local_start_x][j][k], where $i=0, \ldots, m_q-1, j=0, \ldots, n-1, k=0, \ldots, l-1$ for C/C++
  - L(i,j,k)=A(i,j,k+cdft_local_start_x-1), where $i=1, \ldots, m_q, j=1, \ldots, n, k=1, \ldots, l$ for Fortran.

Example C-29 in Appendix C shows how cluster DFT distributes data among processes for a two-dimensional FFT computation.

One-dimensional transforms

In this case, initial and resulting data are distributed among processes differently and even the number of elements stored in a particular process after the transform may vary compared with the one before the transform. Each local array stores a segment of consecutive elements of the appropriate global array. Such segment may be determined by the number of elements and a shift with respect to the first array element. So, to specify segments of the global input and output arrays that a particular process receives, four configuration parameters are needed. In addition to CDFT_LOCAL_NX, and CDFT_LOCAL_START_X, the parameters CDFT_LOCAL_OUT_NX and CDFT_LOCAL_OUT_START_X have meaningful values, which can be also retrieved using DftiGetValueDM. The meaning of the four configuration parameters depends upon the type of the transform, as shown in Table 11-19:

Table 11-19 Data Distribution Configuration Parameters for 1D Transforms

Meaning of the Parameter	Forward Transform	Backward Transform
Number of elements in input array	CDFT_LOCAL_NX	CDFT_LOCAL_OUT_NX
Elements shift in input array	CDFT_LOCAL_START_X	CDFT_LOCAL_OUT_START_X
Number of elements in output array	CDFT_LOCAL_OUT_NX	CDFT_LOCAL_NX

Meaning of the Parameter	Forward Transform	Backward Transform
Elements shift in output array	CDFT_LOCAL_OUT_START_X	CDFT_LOCAL_START_X

Memory size for local data

The memory size needed for local arrays cannot be just calculated from CDFT_LOCAL_NX (CDFT_LOCAL_OUT_NX), as cluster DFT sometimes requires allocating a little bit more memory for local data than just the size of the appropriate sub-array. The configuration parameter CDFT_LOCAL_SIZE specifies the size of the local input and output array in data elements. In the current implementation of cluster DFT interface, data elements are complex values, consisting of a real and imaginary parts. Each of the local input and output arrays must have size not less than CDFT_LOCAL_SIZE*size_of_element. If you employ a user-defined workspace for in-place transforms (for more information, refer to Table 11-20), it must have the same size. Example C-30 in Appendix C illustrates how cluster DFT distributes data among processes in case of a one-dimensional FFT computation effected with a user-defined workspace.

Available Auxiliary Functions

If a global input array is located on one MPI process or you want to gather the global output array on one MPI process, you can use functions MKL_CDFT_ScatterData and MKL_CDFT_GatherData to distribute or gather data among processes, respectively. These functions are defined in a file that is delivered with the Intel MKL Cluster Edition product and located in the following subdirectory of the Intel MKL installation directory: examples/cdftc/examples_support.c for C/C++ and examples/cdftf/examples_support.f90 for Fortran.

Restriction on Lengths of Transforms

The algorithm that cluster DFT uses to distribute data among processes imposes a restriction on lengths of transforms with respect to the number of MPI processes used for the DFT computation:

- For a multi-dimensional transform, lengths of the first two dimensions in C/C++ or of the last two dimensions in Fortran must be not less than the number of MPI processes.
- Length of a one-dimensional transform must be the product of two integers each of which is not less than the number of MPI processes.

Non-compliance with the restriction causes an error `CDFT_SPREAD_ERROR` (refer to Error Codes for details). To achieve the compliance, you can change the transform lengths and/or the number of MPI processes, which is specified at start of an MPI program. MPI-2 enables changing the number of processes during execution of an MPI program.

NOTE. The best performance of a one-dimensional FFT is reached when the transform length L is a square of an integer number: $L=N^2$.

Cluster DFT Interface

NOTE. Cluster DFT interface implemented in Intel MKL 9.0 is incompatible with previous versions. For details, refer to Intel MKL Release Notes.

To use the cluster DFT functions, you need to access the module `MKL_CDFT` through the "`use`" statement in Fortran; or access the header file mkl_cdfti.h through "`include`" in C/C++.

The Fortran interface provides a derived type `DFTI_DESCRIPTOR_DM`; a number of named constants representing various names of configuration parameters and their possible values; and a number of overloaded functions through the generic functionality of Fortran 95.

The C interface provides a structure type `DFTI_DESCRIPTOR_DM_HANDLE` and a number of functions, some of which accept a different number of input arguments.

To provide communication between parallel processes through MPI, the following include statement must be also present in your code:

- Fortran:

  ```
  INCLUDE "mpif.h"
  ```

  (for some MPI versions, "`mpif90.h`" header may be used instead).

- C/C++:

  ```
  #include "mpi.h"
  ```

There are three main categories of the cluster DFT functions in Intel MKL:

1. **Descriptor Manipulation**. There are three functions in this category. The first one, DftiCreateDescriptorDM, creates a DFT descriptor whose storage is allocated dynamically by the routine. The second, DftiCommitDescriptorDM, "commits" the descriptor to all its settings. The third function, DftiFreeDescriptorDM, frees up all the memory allocated for the descriptor information.
2. **DFT Computation**. There are two functions in this category. The first one, DftiComputeForwardDM, effects a forward DFT computation, and the second function, DftiComputeBackwardDM, performs a backward DFT computation.
3. **Descriptor Configuration**. There are two functions in this category. One function, DftiSetValueDM, sets one specific configuration value to one of the many configuration parameters. The other, DftiGetValueDM, gets the current value of any of these configuration parameters, all of which are readable. These parameters, though many, are handled one at a time.

Descriptor Manipulation Functions

There are three functions in this category: create a descriptor, commit a descriptor, and free a descriptor.

CreateDescriptorDM

Allocates memory for the descriptor data structure and instantiates it with default configuration settings.

Syntax

Fortran:

```
Status = DftiCreateDescriptorDM(comm, handle, v1, v2, dim, size)
Status = DftiCreateDescriptorDM(comm, handle, v1, v2, dim, sizes)
```

C/C++:

```
status = DftiCreateDescriptorDM(comm, &handle, v1, v2, dim, size );
status = DftiCreateDescriptorDM(comm, &handle, v1, v2, dim, sizes );
```

Input Parameters

comm	MPI communicator, e.g. MPI_COMM_WORLD.

v1	Precision.
v2	Type of forward domain. Must be DFTI_COMPLEX for the current version.
dim	Dimension of transform.
size	Length of transform in a one-dimensional case.
sizes	Lengths of transform in a multi-dimensional case.

Output Parameters

handle	Pointer to the handle of transform. If the function completes successfully, the pointer to the created handle is stored in the variable.

Description

This function allocates memory in a particular MPI process for the descriptor data structure and instantiates it with default configuration settings with respect to the precision, domain, dimension, and length of the desired transform. The domain is understood to be the domain of the forward transform. The result is a pointer to the created descriptor. This function is slightly different from the "initialization" routine DftiCommitDescriptorDM in more traditional software packages or libraries used for computing DFT. In all likelihood, this function will not perform any significant computation work such as twiddle factors computation, as the default configuration settings can still be changed using the function DftiSetValueDM.

The precision is specified through named constants provided in the interface for the configuration values. The choices for precision are DFTI_SINGLE and DFTI_DOUBLE. It corresponds to precision of input data, output data, and computation. A setting of DFTI_SINGLE indicates single-precision floating-point data type and a setting of DFTI_DOUBLE indicates double-precision floating-point data type.

Dimension is a simple positive integer indicating the dimension of the transform. In C/C++ context, length is a single integer value of type long for one-dimensional transforms or an array of integers of type long for multi-dimensional transforms. In Fortran context, length is an integer or an array of integers.

Return Values

The function returns DFTI_NO_ERROR when completes successfully. In this case, the pointer to the created handle is stored in *handle*. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the Error Codes section).

Interface and Prototype

```
! Fortran Interface
INTERFACE DftiCreateDescriptorDM
    INTEGER(4) FUNCTION DftiCreateDescriptorDMn(C,H,P1,P2,D,L)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: H
        INTEGER(4) C,P1,P2,D,L(*)
    END FUNCTION
    INTEGER(4) FUNCTION DftiCreateDescriptorDM1(C,H,P1,P2,D,L)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: H
        INTEGER(4) C,P1,P2,D,L
    END FUNCTION
END INTERFACE

/* C/C++ prototype */
long DftiCreateDescriptorDM(MPI_Comm,DFTI_DESCRIPTOR_DM_HANDLE*,
   enum DFTI_CONFIG_VALUE,enum DFTI_CONFIG_VALUE,long,...);
```

CommitDescriptorDM

Performs all initialization that facilitates the actual DFT computation.

Syntax

Fortran:

```
Status = DftiCommitDescriptorDM(handle)
```

C/C++:

```
status = DftiCommitDescriptorDM(handle);
```

Input Parameters

handle	Valid handle obtained from DftiCreateDescriptorDM.

Description

The cluster DFT interface requires a function that commits a previously created descriptor be invoked before the descriptor can be used for DFT computations in a particular MPI process. The DftiCommitDescriptorDM function performs all initialization that facilitates the actual DFT computation. For a modern implementation, it may involve exploring many different factorizations of the input length to search for highly efficient computation method.

Any changes of configuration parameters of a committed descriptor via the set value function (see Descriptor Configuration) requires a re-committal of the descriptor before a computation function can be invoked. Typically, this committal function call is immediately followed by a computation function call (see DFT Computation).

Return Values

The function returns DFTI_NO_ERROR when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the Error Codes section).

Interface and Prototype

```
! Fortran Interface
INTERFACE DftiCommitDescriptorDM
   INTEGER(4) FUNCTION DftiCommitDescriptorDM(handle);
     TYPE(DFTI_DESCRIPTOR_DM), POINTER :: handle
 END FUNCTION
END INTERFACE

/* C/C++ prototype */

long DftiCommitDescriptorDM(DFTI_DESCRIPTOR_DM_HANDLE handle);
```

FreeDescriptorDM

Frees memory allocated for a descriptor.

Syntax

Fortran:

```
Status = DftiFreeDescriptorDM(handle)
```

C/C++:

```
status = DftiFreeDescriptorDM(&handle);
```

Input Parameters

handle	Valid handle obtained from DftiCreateDescriptorDM.

Output Parameters

handle	Descriptor handle. Memory allocated for the handle is released on output.

Description

This function frees up all memory allocated for a descriptor in a particular MPI process. Call the DftiFreeDescriptorDM function to delete the descriptor handle. Upon successful completion of DftiFreeDescriptorDM the descriptor handle is no longer valid.

Return Values

The function returns DFTI_NO_ERROR when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the Error Codes section).

Interface and Prototype

```
! Fortran Interface
INTERFACE DftiFreeDescriptorDM
 INTEGER(4) FUNCTION DftiFreeDescriptorDM(handle)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: handle
   END FUNCTION
END INTERFACE

/* C/C++ prototype */

long DftiFreeDescriptorDM(DFTI_DESCRIPTOR_DM_HANDLE *handle);
```

DFT Computation Functions

There are two functions in this category: compute the forward transform and compute the backward transform.

ComputeForwardDM

Computes the forward Discrete Fourier Transform.

Syntax

Fortran:

```
Status = DftiComputeForwardDM(handle, in_X, out_X)
Status = DftiComputeForwardDM(handle, in_out_X)
```

C/C++:

```
status = DftiComputeForwardDM(handle, in_X, out_X);
status = DftiComputeForwardDM(handle, in_out_X);
```

Input Parameters

handle	Valid descriptor handle.
in_X, in_out_X	Local part of input data. Array of complex values. Refer to the Distributing Data among Processes section on how to allocate and initialize the array.

Output Parameters

out_X, in_out_X	Local part of output data. Array of complex values. Refer to the Distributing Data among Processes section on how to allocate the array.

Description

As soon as a descriptor is configured and committed successfully, actual computation of DFT can be performed. The `DftiComputeForwardDM` function computes the forward DFT.

Forward DFT is the transform using the factor $e^{-i2\pi/n}$. The computation is carried out by calling the `DftiComputeForward` function. So, the functions have very much in common and details not explicitly mentioned below, can be found in the description of `DftiComputeForward`.

The valid descriptor handle is created by `DftiCreateDescriptorDM` and committed by `DftiCommitDescriptorDM`. Configuration parameters that the descriptor handle passes to the function are listed in Table 11-21.

Local part of input data, as well as local part of the output data, is an appropriate sequence of complex values (each complex value consists of two real numbers: real part and imaginary part) that a particular process stores. See the Distributing Data among Processes section for details.

The choices for precision of input and output data are the same as those for precision of transform: a setting of `DFTI_SINGLE` indicates single-precision floating-point data type and a setting of `DFTI_DOUBLE` indicates double-precision floating-point data type.

The configuration parameter `DFTI_PLACEMENT` informs the function whether the computation should be in-place. If the value of this parameter is `DFTI_INPLACE` (default), you should call the function with two parameters, otherwise you should supply three parameters. If `DFTI_PLACEMENT = DFTI_INPLACE` and three parameters are supplied, then the third parameter will be ignored.

CAUTION. Even in case of an out-of-place transform, local array of input data *in_X* may be changed. To save data, make its copy before calling `DftiComputeForwardDM`.

In case of an in-place transform, `DftiComputeForwardDM` will dynamically allocate and then deallocate a work buffer of the same size as the local input/output array requires.

NOTE. You can specify your own workspace of the same size through the configuration parameter `CDFT_WORKSPACE` to avoid redundant memory allocation.

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the Error Codes section).

Interface and Prototype

```
! Fortran Interface
INTERFACE DftiComputeForwardDM
   INTEGER(4) FUNCTION DftiComputeForwardDM(h, in_X, out_X)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
   COMPLEX(8), DIMENSION(*) :: in_x, out_X
   END FUNCTION DftiComputeForwardDM
   INTEGER(4) FUNCTION DftiComputeForwardDMi(h, in_out_X)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      COMPLEX(8), DIMENSION(*) :: in_out_X
   END FUNCTION DftiComputeForwardDMi
   INTEGER(4) FUNCTION DftiComputeForwardDMs(h, in_X, out_X)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      COMPLEX(4), DIMENSION(*) :: in_x, out_X
   END FUNCTION DftiComputeForwardDMs
   INTEGER(4) FUNCTION DftiComputeForwardDMis(h, in_out_X)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      COMPLEX(4), DIMENSION(*) :: in_out_X
   END FUNCTION DftiComputeForwardDMis
END INTERFACE

/* C/C++ prototype */
long DftiComputeForwardDM(DFTI_DESCRIPTOR_DM_HANDLE handle, void *in_X,...);
```

ComputeBackwardDM

Computes the backward Discrete Fourier Transform.

Syntax

Fortran:

```
Status = DftiComputeBackwardDM(handle, in_X, out_X)
Status = DftiComputeBackwardDM(handle, in_out_X)
```

C/C++:

```
status = DftiComputeBackwardDM(handle, in_X, out_X);
status = DftiComputeBackwardDM(handle, in_out_X);
```

Input Parameters

handle	Valid descriptor handle.
in_X, in_out_X	Local part of input data. Array of complex values. Refer to the Distributing Data among Processes section on how to allocate and initialize the array.

Output Parameters

out_X, in_out_X	Local part of output data. Array of complex values. Refer to the Distributing Data among Processes section on how to allocate the array.

Description

As soon as a descriptor is configured and committed successfully, actual computation of DFT can be performed. The `DftiComputeBackwardDM` function computes the backward DFT.

Backward DFT is the transform using the factor $e^{i2\pi/n}$. The computation is carried out by calling the `DftiComputeBackward` function. So, the functions have very much in common and details not explicitly mentioned below, can be found in the description of `DftiComputeBackward`.

The valid descriptor handle is created by `DftiCreateDescriptorDM` and committed by `DftiCommitDescriptorDM`. Configuration parameters that the descriptor handle passes to the function are listed in Table 11-21.

Local part of input data, as well as local part of the output data, is an appropriate sequence of complex values (each complex value consists of two real numbers: real part and imaginary part) that a particular process stores. See the Distributing Data among Processes section for details.

The choices for precision of input and output data are the same as those for precision of transform: a setting of DFTI_SINGLE indicates single-precision floating-point data type and a setting of DFTI_DOUBLE indicates double-precision floating-point data type.

The configuration parameter DFTI_PLACEMENT informs the function whether the computation should be in-place. If the value of this parameter is DFTI_INPLACE (default), you should call the function with two parameters, otherwise you should supply three parameters. If DFTI_PLACEMENT = DFTI_INPLACE and three parameters are supplied, then the third parameter will be ignored.

CAUTION. Even in case of an out-of-place transform, local array of input data *in_X* may be changed. To save data, make its copy before calling DftiComputeBackwardDM.

In case of an in-place transform, DftiComputeBackwardDM will dynamically allocate and then deallocate a work buffer of the same size as the local input/output array requires.

NOTE. You can specify your own workspace of the same size through the configuration parameter CDFT_WORKSPACE to avoid redundant memory allocation.

Return Values

The function returns DFTI_NO_ERROR when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the Error Codes section).

Interface and Prototype

```
! Fortran Interface
INTERFACE DftiComputeBackwardDM
   INTEGER(4) FUNCTION DftiComputeBackwardDM(h, in_X, out_X)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      COMPLEX(8), DIMENSION(*) :: in_x, out_X
   END FUNCTION DftiComputeBackwardDM
   INTEGER(4) FUNCTION DftiComputeBackwardDMi(h, in_out_X)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      COMPLEX(8), DIMENSION(*) :: in_out_X
   END FUNCTION DftiComputeBackwardDMi
   INTEGER(4) FUNCTION DftiComputeBackwardDMs(h, in_X, out_X)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      COMPLEX(4), DIMENSION(*) :: in_x, out_X
   END FUNCTION DftiComputeBackwardDMs
   INTEGER(4) FUNCTION DftiComputeBackwardDMis(h, in_out_X)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      COMPLEX(4), DIMENSION(*) :: in_out_X
   END FUNCTION DftiComputeBackwardDMis
END INTERFACE

/* C/C++ prototype */
long DftiComputeBackwardDM(DFTI_DESCRIPTOR_DM_HANDLE handle, void *in_X,...);
```

Descriptor Configuration Functions

There are two functions in this category: the value setting function DftiSetValueDM sets one particular configuration parameter to an appropriate value, the value getting function DftiGetValueDM reads the values of one particular configuration parameter.

Some configuration parameters used by cluster DFT functions originate from the conventional DFT interface (see Configuration Settings subsection in the "DFT Functions" section for details).

Other parameters are specific for cluster DFT. Integer values of these configuration parameters have type `long` in C/C++ and `INTEGER(4)` in Fortran. The exact type of the configuration parameters being floating-point scalars is `float` or `double` in C/C++ and `REAL(4)` or `REAL(8)` in Fortran. The configuration parameters whose values are identified by named constants have the `enum` type in C/C++ and `INTEGER` in Fortran. They are defined in the mkl_cdft.h header file in C/C++ and `MKL_CDFT` module in Fortran.

Names the CDFT-specific configuration parameters have prefix `CDFT`.

SetValueDM

Sets one particular configuration parameter with the specified configuration value.

Syntax

Fortran:

```
Status = DftiSetValueDM (handle, param, value)
```

C/C++:

```
status = DftiSetValueDM (handle, param, value);
```

Input Parameters

handle	Valid descriptor handle.
param	Name of a parameter to be set up in the descriptor handle. See Table 11-20 for the list of available names.
value	Value of a parameter.

Description

This function sets one particular configuration parameter with the specified configuration value. The configuration parameter is one of the named constants listed in the table below, and the configuration value is the corresponding appropriate type. See Configuration Settings for details of the meaning of the setting.

Table 11-20 Settable Configuration Parameters

Named constant	Type of parameter	Description	Default value
DFTI_FORWARD_SCALE	Floating-point scalar	Scale factor of forward transform.	1.0
DFTI_BACKWARD_SCALE	Floating-point scalar	Scale factor of backward transform.	1.0
DFTI_PLACEMENT	Named constant	Placement of the computation result.	DFTI_INPLACE
DFTI_ORDERING	Named constant	Scrambling of data order.	DFTI_ORDERED
DFTI_WORKSPACE	Array of an appropriate type	Auxiliary buffer, a user-defined workspace. Enables saving memory during in-place computations.	NULL (allocate workspace dynamically).

Return Values

The function returns DFTI_NO_ERROR when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the Error Codes section).

Interface and Prototype

```
! Fortran Interface
INTERFACE DftiSetValueDM
   INTEGER(4) FUNCTION DftiSetValueDM(h, p, v)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      INTEGER(4) :: p, v
   END FUNCTION
   INTEGER(4) FUNCTION DftiSetValueDMd(h, p, v)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      INTEGER(4) :: p
      REAL(8) :: v
   END FUNCTION
   INTEGER(4) FUNCTION DftiSetValueDMs(h, p, v)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      INTEGER(4) :: p
      REAL(4) :: v
   END FUNCTION
   INTEGER(4) FUNCTION DftiSetValueDMsw(h, p, v)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      INTEGER(4) :: p
      COMPLEX(4) :: v(*)
   END FUNCTION
   INTEGER(4) FUNCTION DftiSetValueDMdw(h, p, v)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      INTEGER(4) :: p
      COMPLEX(8) :: v(*)
   END FUNCTION
END INTERFACE
```

```
/* C/C++ prototype */
long DftiSetValueDM(DFTI_DESCRIPTOR_DM_HANDLE handle, int param,...);
```

GetValueDM

Gets the configuration value of one particular configuration parameter.

Syntax

Fortran:

```
Status = DftiGetValueDM(handle, param, value)
```

C/C++:

```
status = DftiGetValueDM(handle, param, &value);
```

Input Parameters

handle	Valid descriptor handle.
param	Name of a parameter to be retrieved from the descriptor handle. See Table 11-21 for the list of available names.

Output Parameters

value	Value of the parameter.

Description

This function gets the configuration value of one particular configuration parameter. The configuration parameter is one of the named constants listed in the table below, and the configuration value is the corresponding appropriate type, which can be a named constant or a native type. Possible values of the named constants can be found in Table 11-6.

Table 11-21 Retrievable Configuration Parameters

Named Constant	Type of parameter	Description
DFTI_PRECISION	Named constant	Precision of computation, input data and output data.
DFTI_DIMENSION	Integer scalar	Dimension of the transform
DFTI_LENGTHS	Array of integer values	Array of lengths of the transform. Number of lengths corresponds to the dimension of the transform.
DFTI_FORWARD_SCALE	Floating-point scalar	Scale factor of forward transform.
DFTI_BACKWARD_SCALE	Floating-point scalar	Scale factor of backward transform.
DFTI_PLACEMENT	Named constant	Placement of the computation result.
DFTI_COMMIT_STATUS	Named constant	Shows whether descriptor has been committed.
DFTI_FORWARD_DOMAIN	Named constant	Forward domain of transforms (In the current implementation of the cluster DFT interface, the value is always DFTI_COMPLEX.).
DFTI_ORDERING	Named constant	Scrambling of data order.
CDFT_MPI_COMM	Type of MPI communicator	MPI communicator used for transforms.
CDFT_LOCAL_SIZE	Integer scalar	Necessary size of input, output, and buffer arrays in data elements.
CDFT_LOCAL_X_START	Integer scalar	Row/element number of the global array that corresponds to the first row/element of the local array. For more information, see Distributing Data among Processes.

Named Constant	Type of parameter	Description
CDFT_LOCAL_NX	Integer scalar	The number of rows/elements of the global array stored in the local array. For more information, see Distributing Data among Processes.
CDFT_LOCAL_OUT_X_START	Integer scalar	Element number of the appropriate global array that corresponds to the first element of the input or output local array in a 1D case. For details, see Distributing Data among Processes.
CDFT_LOCAL_OUT_NX	Integer scalar	The number of elements of the appropriate global array that are stored in the input or output local array in a 1D case. For details, see Distributing Data among Processes.

Return Values

The function returns DFTI_NO_ERROR when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the Error Codes section).

Interface and Prototype

```
! Fortran Interface
INTERFACE DftiGetValueDM
   INTEGER(4) FUNCTION DftiGetValueDM(h, p, v)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      INTEGER(4) :: p, v
   END FUNCTION
   INTEGER(4) FUNCTION DftiGetValueDMar(h, p, v)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      INTEGER(4) :: p, v(*)
   END FUNCTION
   INTEGER(4) FUNCTION DftiGetValueDMd(h, p, v)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      INTEGER(4) :: p
      REAL(8) :: v
   END FUNCTION
   INTEGER(4) FUNCTION DftiGetValueDMs(h, p, v)
      TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
      INTEGER(4) :: p
      REAL(4) :: v
   END FUNCTION
END INTERFACE

/* C/C++ prototype */
long DftiGetValueDM(DFTI_DESCRIPTOR_DM_HANDLE handle, int param,...);
```

Error Codes

All the cluster DFT functions return an integer value denoting the status of the operation. These values are identified by named constants. Each function returns `DFTI_NO_ERROR` if no errors were encountered during execution. Otherwise, a function generates an error code. In addition to DFT error codes, CDFT has its own ones. Names of the CDFT-specific named constants have prefix "`CDFT`". Table 11-22 lists error codes that cluster DFT functions may return.

Table 11-22 Error Codes that Cluster DFT Functions Return

Named Constants	Comments
`DFTI_NO_ERROR`	No error.
`DFTI_MEMORY_ERROR`	Usually associated with memory allocation.
`DFTI_INVALID_CONFIGURATION`	Invalid settings of one or more configuration parameters.
`DFTI_INCONSISTENT_CONFIGURATION`	Inconsistent configuration or input parameters.
`DFTI_NUMBER_OF_THREADS_ERROR`	Number of OMP threads in the computation function is not equal to the number of OMP threads in the initialization stage (commit function).
`DFTI_MULTITHREADED_ERROR`	Usually associated with OMP routine's error return value.
`DFTI_BAD_DESCRIPTOR`	Descriptor is unusable for computation.
`DFTI_UNIMPLEMENTED`	Unimplemented legitimate settings; implementation dependent.
`DFTI_MKL_INTERNAL_ERROR`	Internal library error.
`DFTI_1D_LENGTH_EXCEEDS_INT32`	Length of one of dimensions exceeds 2^{32} -1 (4 bytes).
`CDFT_SPREAD_ERROR`	Data cannot be distributed (For more information, see Distributing Data among Processes.)
`CDFT_MPI_ERROR`	MPI error. Occurs when calling MPI.

12 Interval Linear Solvers

Intel® MKL Interval Linear solver routines that can be used for:

- solving systems of interval linear equations $Ax = b$ with an interval matrix $A = (a_{ij})$ and interval right-hand side vector $b = (b_i)$;
- checking properties of interval matrices.

For more information on key concepts of interval linear systems, see Appendix A, "Linear Solvers Basics".

Routines described below are subdivided according to the problems they solve into the following groups:

Routines for Fast Solution of Interval Systems

Routines for Sharp Solution of Interval Systems

Routines for Inverting Interval Matrices

Routines for Checking Properties of Interval Matrices

Auxiliary and Utility Routines

Table 12-1 contains the full list of Intel MKL routines for solving interval linear systems.

Table 12-1 Intel MKL Interval Linear Solver Routines

Routine Name	Description
?trtrs	Solves a triangular system of interval linear equations by backward substitution procedure.
?gegas	Solves a system of interval linear equations by interval Gauss method.
?gehss	Solves a system of interval linear equations by interval Householder method.
?gekws	Solves a system of interval linear equations by Krawczyk iteration method.
?gegss	Solves a system of interval linear equations by interval Gauss-Seidel iteration.
?gehbs	Solves a system of interval linear equations by Hansen-Bliek-Rohn procedure.
?gepps	Solves a system of interval linear equations by a parameter partitioning method.
?gepss	Solves a system of interval linear equations by a solution partitioning method.
?trtri	Computes inverse interval matrix to a triangular interval matrix.
?geszi	Computes inverse interval matrix by Schulz interval iterative procedure.
?gerbr	Tests regularity of an interval matrix by Ris-Beeck and Rex-Rohn criteria
?gesvr	Tests regularity/singularity of an interval matrix by Rump and Rex-Rohn singular value criteria.
?gemip	Performs midpoint-inverse preconditioning of an interval linear system.

Routine Naming Conventions

For the routines introduced below, the LAPACK-like naming conventions are used. Specifically, all the routine names have the structure `xxyyzzz`, where the first letters `xx` indicate the data types:

`si`	real interval, single precision
`di`	real interval, double precision
`cr`	complex rectangular interval, single precision
`zr`	complex rectangular interval, double precision
`cc`	complex circular interval, single precision
`zc`	complex circular interval, double precision

The third and fourth letters `yy` indicate the matrix type:

`ge`	general
`tr`	triangular

The last three letters `zzz` indicate the computational procedure performed by the routine:

`trs`	backward substitution solver for triangular interval linear systems
`gas`	interval Gauss solver for interval linear systems
`hss`	interval Householder solver for interval linear systems
`kws`	iterative Krawczyk solver for interval linear systems
`gss`	interval Gauss-Seidel iteration solver for interval linear systems
`hbs`	Hansen-Bliek-Rohn solver for interval linear systems
`pps`	parameter partitioning method-based solver for interval linear systems
`pss`	solution partitioning method-based solver for interval linear systems
`tri`	inverting triangular interval matrix based on backward substitution
`szi`	inverting general interval matrix by Schulz iterative method
`rbr`	testing regularity/singularity of interval matrix by Ris-Beeck criterion
`svr`	testing regularity/singularity of interval matrix by Rump and Rex-Rohn singular value criteria
`mip`	midpoint-inverse preconditioning of interval linear system

The question mark in the routine group name corresponds to different character codes indicating the data type (si, di, cr, zr, cc, or zc). For example, ?trtri denotes the group name for either of the routines sitrtri, ditrtri, crtrtri, zrtrtri, cctrtri, or zctrtri.

Routines for Fast Solution of Interval Systems

?trtrs

Solves a triangular system of interval linear equations by backward substitution procedure.

Syntax

```
call sitrtrs(uplo, trans, diag, n, nrhs, a, lda, b, ldb, info)
call ditrtrs(uplo, trans, diag, n, nrhs, a, lda, b, ldb, info)
call crtrtrs(uplo, trans, diag, n, nrhs, a, lda, b, ldb, info)
call zrtrtrs(uplo, trans, diag, n, nrhs, a, lda, b, ldb, info)
call cctrtrs(uplo, trans, diag, n, nrhs, a, lda, b, ldb, info)
call zctrtrs(uplo, trans, diag, n, nrhs, a, lda, b, ldb, info)
```

Description

The routine ?trtrs solves for X the following systems of interval linear equations with a triangular matrix *A* and multiple right-hand sides stored in *B*:

AX = *B*, if *trans* = 'N'

A^TX = *B*, if *trans* = 'T'

A^HX = *B*, if *trans* = 'C' (for complex matrices only).

The routine implements backward substitution algorithm and produces optimal enclosures of the solution sets to interval linear systems, which is due to the simple structure of the matrix *A*.

Input Parameters

uplo	CHARACTER(1). Must be one of 'U' , 'L' , 'u', or 'l'. Indicates whether *A* is upper or lower triangular. If *uplo* = 'U' or 'u', then *A* is upper triangular.

If *uplo* = 'L' or 'l', then *A* is lower triangular.

trans — CHARACTER(1). Must be one of 'N', 'T', 'C', 'n', 't', or 'c'.
If *trans* = 'N' or 'n', then $AX = B$ is solved for X.
If *trans* = 'T' or 't', then $A^TX = B$ is solved for X.
If *trans* = 'C' or 'c', then $A^HX = B$ is solved for X.

diag — CHARACTER(1). Must be one of 'N', 'U', 'n', or 'u'.
If *diag* = 'N' or 'n', then *A* is not a unit triangular matrix.
If *diag* = 'U' or 'u', then *A* is unit triangular: diagonal elements of *A* are assumed to be 1 and not referenced in the array *a*.

n — INTEGER. The order of *A*, the number of rows in *B* ($n \geq 0$).

nrhs — INTEGER. The number of right-hand sides ($nrhs \geq 0$).

a, *b* — S_INTERVAL for sitrtrs.
D_INTERVAL for ditrtrs.
CR_INTERVAL for crtrtrs.
ZR_INTERVAL for zrtrtrs.
CC_INTERVAL for cctrtrs.
ZC_INTERVAL for zctrtrs.
Arrays: *a* (*lda*,*), *b* (*ldb*,*).
The array *a* contains the matrix *A*.
The array *b* contains the matrix *B*, whose columns are the right-hand sides for the systems of equations. The second dimension of *a* must be at least max(1,*n*) and the second dimension of *b* must be at least max(1,*nrhs*).

lda — INTEGER. The first dimension of *a*, *lda* ≥ max(1, *n*).

ldb — INTEGER. The first dimension of *b*, *ldb* ≥ max(1, *n*).

Output Parameters

b — Overwritten by the solution matrix X.

info — INTEGER.
If *info* = 0, the execution is successful.
If *info* > 0, the execution is not successful.
If *info* = -*i*, the *i*-th parameter has an illegal value.

?gegas

Solves a system of interval linear equations by interval Gauss method.

Syntax

```
call sigegas(trans, n, nrhs, a, lda, b, ldb, info)
call digegas(trans, n, nrhs, a, lda, b, ldb, info)
call crgegas(trans, n, nrhs, a, lda, b, ldb, info)
call zrgegas(trans, n, nrhs, a, lda, b, ldb, info)
call ccgegas(trans, n, nrhs, a, lda, b, ldb, info)
call zcgegas(trans, n, nrhs, a, lda, b, ldb, info)
```

Description

The routine `?gegas` uses the interval Gauss method to compute an enclosure of the solution set to the following interval linear system of equations:

$AX = B$, if *trans* = 'N'

$A^TX = B$, if *trans* = 'T'

$A^HX = B$, if *trans* = 'C' (for complex matrices only).

Input Parameters

trans	CHARACTER(1). Must be one of 'N', 'T', 'C', 'n', 't', or 'c'. Indicates the form of the equations system: If *trans* = 'N' or 'n', then $AX = B$ is solved for X. If *trans* = 'T' or 't', then $A^TX = B$ is solved for X. If *trans* = 'C' or 'c', then $A^HX = B$ is solved for X.
n	INTEGER. The order of A, the number of rows in B ($n \geq 0$).
nrhs	INTEGER. The number of right-hand sides ($nrhs \geq 0$).
a, *b*	REAL for sigegas. DOUBLE PRECISION for digegas. Arrays: *a* (*lda*,*), *b* (*ldb*,*). The array *a* contains the matrix A.

	The array *b* contains the matrix *B*, whose columns are the right-hand sides for the systems of equations. The second dimension of *a* must be at least max(1,*n*) and the second dimension of *b* must be at least max(1,*nrhs*).
lda	INTEGER. The first dimension of *a*, *lda* ≥ max(1, *n*).
ldb	INTEGER. The first dimension of *b*, *ldb* ≥ max(1, *n*).

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* > 0, the execution is not successful. If *info* = -*i*, the *i*-th parameter has an illegal value.

Example 12-1 Fortran 90 Code for Interval Gauss Method

The following piece of Fortran code presents an example of using the routine digegas to compute, by an interval Gauss method, an enclosure of the solution set to the interval linear system of equations:

$$\begin{pmatrix} [2,3] & [0,1] \\ [1,2] & [2,3] \end{pmatrix} x = \begin{pmatrix} [0,120] \\ [60,240] \end{pmatrix}$$

```
-------------------------------------------------------------------------------------------
. . . . . .
USE INTERVAL_ARITHMETIC
. . . . . .
TYPE(D_INTERVAL)          :: A(2,2), B(2)
INTEGER          :: N, INFO
CHARACTER(1)           :: TRANS = 'n'
. . . . . .
N = 2
A(1,1) = DINTERVAL(2.,3.);     A(1,2) = DINTERVAL(0.,1.)
A(2,1) = DINTERVAL(1.,2.);     A(2,2) = DINTERVAL(2.,3.)
B(1,1) = DINTERVAL(0.,120.); B(2,1) = DINTERVAL(60.,240.)
. . . . . .
CALL DIGEGAS( TRANS, N, 1, A, 2, B, 2, INFO )
-------------------------------------------------------------------------------------------
```

Note that assigning double-precision intervals to the entries of the matrix *A* and right-hand side vector *B* is carried out by `DINTERVAL` function supplied by `INTERVAL_ARITHMETIC` module.

?gehss

Solves a system of interval linear equations by interval Householder method.

Syntax

```
call sigehss(trans, n, nrhs, a, lda, b, ldb, info)
call digehss(trans, n, nrhs, a, lda, b, ldb, info)
```

Description

The routine `?gehss` uses the interval Householder method to compute an enclosure of the solution set to the following interval linear system of equations:

AX = *B*, if *trans* = 'N',

$A^TX = B$, if *trans* = 'T' or 'C'.

Input Parameters

trans	CHARACTER(1). Must be one of 'N', 'T', 'C', 'n', 't', or 'c'. Indicates the form of the equations system: If *trans* = 'N' or 'n', then $AX = B$ is solved for X. If *trans* = 'T' or 'C' or 't'or 'c', then $A^TX = B$ is solved for X.
n	INTEGER. The order of *A*, the number of rows in *B* ($n \geq 0$).
nrhs	INTEGER. The number of right-hand sides ($nrhs \geq 0$).
a, *b*	REAL for sigehss. DOUBLE PRECISION for digehss. Arrays: *a* (*lda*,*), *b* (*ldb*,*). The array *a* contains the matrix *A*. The array *b* contains the matrix *B*, whose columns are the right-hand sides for the systems of equations. The second dimension of *a* must be at least max(1,*n*) and the second dimension of *b* must be at least max(1,*nrhs*).
lda	INTEGER. The first dimension of *a*, *lda* ≥ max(1, *n*).
ldb	INTEGER. The first dimension of b, *ldb* ≥ max(1, *n*).

Output Parameters

b	Overwritten by an enclosure of the solution matrix X.
info	INTEGER. If *info* = 0, the execution is successful. If *info* > 0, the execution is not successful. If *info* = -*i*, the *i*-th parameter has an illegal value.

?gekws

Solves a system of interval linear equations by Krawczyk iteration method.

Syntax

```
call sigekws(trans, n, nrhs, a, lda, b, ldb, epsilon, info)
call digekws(trans, n, nrhs, a, lda, b, ldb, epsilon, info)
call crgekws(trans, n, nrhs, a, lda, b, ldb, epsilon, info)
call zrgekws(trans, n, nrhs, a, lda, b, ldb, epsilon, info)
call ccgekws(trans, n, nrhs, a, lda, b, ldb, epsilon, info)
call zcgekws(trans, n, nrhs, a, lda, b, ldb, epsilon, info)
```

Description

The routine ?gekws uses the Krawczyk interval iteration to compute an enclosure of the solution set to the following interval linear system of equations:

$AX = B$, if *trans* = 'N'

$A^TX = B$, if *trans* = 'T'

$A^HX = B$, if *trans* = 'C' (for complex matrices only).

Input Parameters

trans	CHARACTER(1). Must be one of 'N', 'T', 'C', 'n', 't', or 'c'. Indicates the form of the equations system: If *trans* = 'N' or 'n', then $AX = B$ is solved for X. If *trans* = 'T' or 't', then $A^TX = B$ is solved for X. If *trans* = 'C' or 'c', then $A^HX = B$ is solved for X.
n	INTEGER. The order of A, the number of rows in B ($n \geq 0$).
nrhs	INTEGER. The number of right-hand sides ($nrhs \geq 0$).
a, *b*	S_INTERVAL for sigekws. D_INTERVAL for digekws. CR_INTERVAL for crgekws. ZR_INTERVAL for zrgekws.

	CC_INTERVAL for ccgekws. ZC_INTERVAL for zcgekws. Arrays: *a* (*lda*,*), *b* (*ldb*,*). The array *a* contains the matrix *A*. The array *b* contains the matrix *B*, whose columns are the right-hand sides for the systems of equations.
lda	INTEGER. The first dimension of *a*, *lda* ≥ max(1, *n*).
ldb	INTEGER. The first dimension of b, *ldb* ≥ max(1, *n*).
epsilon	REAL for sigekws, crgekws, and ccgekws. DOUBLE PRECISION for digekws, zrgekws, and zcgekws. The prescribed accuracy of the estimate.

Output Parameters

b	Overwritten by an enclosure of the solution matrix X.
info	INTEGER. If *info* = 0, the execution is successful. If *info* > 0, the execution is not successful. If *info* = -*i*, the *i*-th parameter has an illegal value.

Application Notes

Krawczyk interval iteration already incorporates midpoint inverse preconditioning, so that additional application of ?gemip routines is not necessary and does not improve the overall efficiency.

?gegss

Solves a system of interval linear equations by interval Gauss-Seidel iteration.

Syntax

```
call sigegss(trans, n, nrhs, a, lda, b, ldb, encl, epsilon, nits, info)
call digegss(trans, n, nrhs, a, lda, b, ldb, encl, epsilon, nits, info)
call crgegss(trans, n, nrhs, a, lda, b, ldb, encl, epsilon, nits, info)
call zrgegss(trans, n, nrhs, a, lda, b, ldb, encl, epsilon, nits, info)
call ccgegss(trans, n, nrhs, a, lda, b, ldb, encl, epsilon, nits, info)
call zcgegss(trans, n, nrhs, a, lda, b, ldb, encl, epsilon, nits, info)
```

Description

The routine `?gegss` uses the interval Gauss-Seidel iteration to compute an enclosure of a portion of the solution set to the following interval linear system of equations:

$AX = B$, if *trans* = 'N'

$A^TX = B$, if *trans* = 'T'

$A^HX = B$, if *trans* = 'C' (for complex matrices only).

See in Appendix C of this manual for example code on using this routine.

Input Parameters

trans	CHARACTER(1). Must be one of 'N', 'T', 'C', 'n', 't', or 'c'. Indicates the form of the equations system: If *trans* = 'N' or 'n', then $AX = B$ is solved for X. If *trans* = 'T' or 't', then $A^TX = B$ is solved for X. If *trans* = 'C' or 'c', then $A^HX = B$ is solved for X.
n	INTEGER. The order of A, the number of rows in B ($n \geq 0$).
nrhs	INTEGER. The number of right-hand sides ($nrhs \geq 0$).
a, *b*	S_INTERVAL for sigegss. D_INTERVAL for digegss. CR_INTERVAL for crgegss.

	ZR_INTERVAL for zrgegss. CC_INTERVAL for ccgegss. ZC_INTERVAL for zcgegss. Arrays: *a* (*lda*,*), *b* (*ldb*,*). The array *a* contains the matrix *A*. The array *b* contains the matrix *B*, whose columns are the right-hand sides for the systems of equations.
lda	INTEGER. The first dimension of *a*, *lda*≥ max(1, *n*).
ldb	INTEGER. The first dimension of b, *ldb*≥ max(1, *n*).
encl	S_INTERVAL for sigegss. D_INTERVAL for digegss. CR_INTERVAL for crgegss. ZR_INTERVAL for zrgegss. CC_INTERVAL for ccgegss. ZC_INTERVAL for zcgegss. Array: *encl* (*ldb*,*). The array *encl* defines the interval box bounding the portion of the solution set that the routine estimates.
epsilon	REAL for sigegss, crgegss, and ccgegss. DOUBLE PRECISION for digegss, zrgegss, and zcgegss. The prescribed accuracy of the esimate.
nits	INTEGER. The number of Gauss-Seidel iterations alloted, *nits* ≥ 0.

Output Parameters

b	Overwritten by an enclosure of the solution matrix *X*.
info	INTEGER. If *info* = 0, the execution is successful. If *info*= *i* > 0, then the diagonal element *a*(*i*,*i*) of the matrix contains zero. The execution of the routine did not fail, but it is recommended to interchange the rows and/or columns of the matrix so as to exclude zero-containing elements from its main diagonal. If *info* = -*i*, the *i*-th parameter has an illegal value.

Application Notes

Interval Gauss-Seidel iteration is a `local solver` of interval linear systems, which means that it is mainly intended for computing enclosures of portions of the solution set bounded by a given interval box in the space R^n.

If the goal is to compute, by `?gegss`, an enclosure of the entire solution set to an interval linear system of equations, then its initial (crude) enclosure should be provided through *encl*argument.

?gehbs

Solves a system of interval linear equations by Hansen-Bliek-Rohn procedure.

Syntax

```
call sigehbs(trans, n, a, lda, b, ldb, info)
call digehbs(trans, n, a, lda, b, ldb, info)
```

Description

The routine `?gehbs` uses the Hansen-Bliek-Rohn procedure to compute an enclosure of the solution set to the following interval linear system of equations:

$AX = B$, if *trans* = 'N',

$A^TX = B$, if *trans* = 'T' or 'C'.

See in Appendix C of this manual for example code on using this routine.

Input Parameters

trans	CHARACTER(1). Must be one of 'N', 'T', 'C', 'n', 't', or 'c'. Indicates the form of the equations system: If *trans* = 'N' or 'n', then $AX = B$ is solved for X. If *trans* = 'T' or 'C' or 't'or 'c', then $A^TX = B$ is solved for X.
n	INTEGER. The order of A, the number of rows in B ($n \geq 0$).
a, *b*	REAL for sigehbs. DOUBLE PRECISION for digehbs. Arrays: *a* (*lda*,*), *b* (*ldb*,*). The array *a* contains the matrix A.

	The array *b* contains the matrix *B*, whose columns are the right-hand sides for the systems of equations.
lda	INTEGER. The first dimension of *a*, *lda* ≥ max(1, *n*).
ldb	INTEGER. The first dimension of b, *ldb* ≥ max(1, *n*).

Output Parameters

b	Overwritten by the solution matrix *X*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* > 0, the execution is not successful. If *info* = -*i*, the *i*-th parameter has an illegal value.

Application Notes

If the middle matrix of *A* is not close to a diagonal matrix, then the midpoint inverse preconditioning by ?gemip routine may be necessary to correct the interval linear system and yield better results.

Routines for Sharp Solution of Interval Systems

?gepps

Solves a system of interval linear equations by a parameter partitioning method.

Syntax

```
call sigepps( trans, n, a, lda, b, ldb, cmpt, mode, estm, epsilon, nits, info)
call digepps( trans, n, a, lda, b, ldb, cmpt, mode, estm, epsilon, nits, info)
```

Description

The routine ?gepps uses the parameter partitioning (PPS) method to compute some (or all) of the sharp outer component-wise estimates of the solution set to the following interval linear system of equations:

AX = *B*, if *trans* = 'N',

$A^TX = B$, if *trans* = 'T' or 'C'.

Input Parameters

trans	CHARACTER(1). Must be one of 'N', 'T', 'C', 'n', 't', or 'c'. Indicates the form of the equations system: If *trans* = 'N' or 'n', then $AX = B$ is solved for X. If *trans* = 'T' or 'C' or 't'or 'c', then $A^TX = B$ is solved for X.
n	INTEGER. The order of *A*, the number of rows in *B* ($n \geq 0$).
a, *b*	REAL for sigepps. DOUBLE PRECISION for digepps. Arrays: *a* (*lda*,*), *b* (*ldb*). The array *a* contains the matrix *A*. The array *b* contains the vector *B* of the right-hand sides for the system of equations.
lda	INTEGER. The first dimension of *a*, $lda \geq \max(1, n)$.
ldb	INTEGER. The first dimension of *b*, $ldb \geq \max(1, n)$.
cmpt	INTEGER. The number of the component of the solution set to be estimated.
mode	CHARACTER(1). Must be either 'L' or 'U'(or the corresponding lowercase letters). Indicates how to estimate the solution set along the coordinate direction specified by the parameter *cmpt*: if *mode* = 'L' or 'l', then the routine computes the lower estimate of the solution set over the *cmpt*-th coordinate; if *mode* = 'U' or 'u', then the routine computes the upper estimate of the solution set over the *cmpt*-th coordinate.
epsilon	REAL for sigepps. DOUBLE PRECISION for digepps. The prescribed accuracy of the estimate.
nits	INTEGER. The number of iterations of the PPS algorithm alloted, $nits \geq 0$.

Output Parameters

estm	REAL for sigepps.

	DOUBLE PRECISION for digepps. Estimate of the solution set along the coordinate axis with the number *cmpt*. if *mode* = 'L', then *estm* represents the lower estimate of the solution set. if *mode* = 'U', then *estm* is equal to the upper estimate of the solution set.
epsilon	The actual precision of the estimate.
nits	The number of iterations that the algorithm actually executed.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *i* > 0, the execution is not successful. If *info* = -*i*, the *i*-th parameter has an illegal value.

Application Notes

Routines ?gepps and ?gepss implement two mutually dual algorithms for computing optimal estimates of the solution sets to interval linear systems, which use adaptive partitioning of either parameter set or solution set of the system. The choice of either of these methods should be based on the specific features of the problem under solution such as the number of essentially interval parameters, the shape of the solution set, and so on.

For example, if the interval system has few interval parameters, then PPS-method is preferable, while PSS-method works better for systems that have simple shape of the solution set.

Computing optimal (or sharp) enclosures of the solution sets to interval linear systems, as well as enclosures that are guaranteed to be sharp within a prescribed accuracy, is known to be an NP-hard problem.

Therefore, a good choice of the parameters *epsilon* and *nits* becomes crucial for effective work of ?gepps and ?gepss routines and for producing desired results.

With this in mind, the recommendation is to organize the whole solution process with ?gepps or ?gepss interactively as a sequence of routine calls, starting from a moderate *nits* and a rough *epsilon* and then increasing *nits* and reducing *epsilon* until ?gepps or ?gepss still complete their execution.

?gepss

Solves a system of interval linear equations by a solution partitioning method.

Syntax

```
call sigepss( trans, n, a, lda, b, ldb, cmpt, mode, estm, epsilon, nits, info)
call digepss( trans, n, a, lda, b, ldb, cmpt, mode, estm, epsilon, nits, info)
```

Description

The routine ?gepss uses the (PSS) method to compute a sharp outer component-wise estimate, along a prescribed coordinate axis, of the solution set to the following interval linear system of equations:

$AX = B$, if *trans* = 'N',

$A^TX = B$, if *trans* = 'T' or 'C'.

Input Parameters

trans	CHARACTER(1). Must be one of 'N', 'T', 'C', 'n', 't', or 'c'. Indicates the form of the equations system: If *trans* = 'N' or 'n', then $AX = B$ is solved for X. If *trans* = 'T' or 'C' or 't'or 'c', then $A^TX = B$ is solved for X.
n	INTEGER. The order of A, the number of rows in B ($n \geq 0$).
a, *b*	REAL for sigepss. DOUBLE PRECISION for digepss. Arrays: *a* (*lda*,*), *b* (*ldb*). The array *a* contains the matrix A. The array *b* contains the vector B of the right-hand sides for the system of equations to be solved.
lda	INTEGER. The first dimension of *a*, *lda* ≥ max(1, *n*).
ldb	INTEGER. The first dimension of b, *ldb* ≥ max(1, *n*).
cmpt	INTEGER. The number of the component of the solution set to be estimated.
mode	CHARACTER(1). Must be either 'L' or 'U'(or the corresponding lowercase letters).

	Indicates how to estimate the solution set along the coordinate direction specified by the parameter *cmpt*: if *mode* = 'L' or 'l', then the routine computes the lower estimate of the solution set over the *cmpt*-th coordinate; if *mode* = 'U' or 'u', then the routine computes the upper estimate of the solution set over the *cmpt*-th coordinate.
epsilon	REAL for sigepss. DOUBLE PRECISION for digepss. The prescribed accuracy of the esimate for the solution set.
nits	INTEGER. The number of iterations of the PSS algorithm alloted, *nits*≥0.

Output Parameters

estm	REAL for sigepss. DOUBLE PRECISION for digepss. Estimate of the solution set along the coordinate axis with the number *cmpt*. if *mode* = 'L', then *estm* represents the lower estimate of the solution set. if *mode* = 'U', then *estm* is equal to the upper estimate of the solution set.
epsilon	The actual precision of the estimate *estm*.
nits	INTEGER. The number of iterations that the algorithm actually executed.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *i* > 0, the execution is not successful. If *info* = -*i*, the *i*-th parameter has an illegal value and the routine outputs the corresponding message.

Application Notes

Routines ?gepps and ?gepss implement two mutually dual algorithms for computing optimal estimates of the solution sets to interval linear systems, which use adaptive partitioning of either parameter set or solution set of the system. The choice of either of these methods should be based on the specific features of the problem under solution such as the number of essentially interval parameters, the shape of the solution set, and so on.

For example, if the interval system has few interval parameters, then PPS-method is preferable, while PSS-method works better for systems that have simple shape of the solution set.

Computing optimal (or sharp) enclosures of the solution sets to interval linear systems, as well as enclosures that are guaranteed to be sharp within a prescribed accuracy, is known to be an NP-hard problem.

Therefore, a good choice of the parameters *epsilon* and *nits* becomes crucial for effective work of ?gepps and ?gepss routines and for producing desired results.

With this in mind, the recommendation is to organize the whole solution process with ?gepps or ?gepss interactively as a sequence of routine calls, starting from a moderate *nits* and a rough *epsilon* and then increasing *nits* and reducing *epsilon* until ?gepps or ?gepss still complete their execution.

Example 12-2 Fortran 90 Code for Parameter Partitioning (PPS) Method

Consider a sample problem that requires computing a sharp lower estimate (to within the accuracy of, say, 1.E-4), along the first coordinate direction, of the solution set to the interval linear system

$$\begin{pmatrix} 3.5 & [0,2] & [0,2] \\ [0,2] & 3.5 & [0,2] \\ [0,2] & [0,2] & 3.5 \end{pmatrix} x = \begin{pmatrix} [-1,1] \\ [-1,1] \\ [-1,1] \end{pmatrix}$$

The problem can be solved by the following Fortran code that implements parameter partitioning method (PPS-method) and uses sigepps routine:

```
-----------------------------------------------------------------------------------
. . . . . .
USE INTERVAL_ARITHMETIC
. . . . . .
INTEGER, PARAMETER          :: LDA = 3, LDB = 3
INTEGER          :: NITS, CMPT, INFO, I, J
CHARACTER(1)          :: MODE = 'L'
REAL(4)          :: EPS, ESTM
TYPE(S_INTERVAL)          :: A(3,3), B(3)
. . . . . .
DO I = 1, 3
   DO J = 1, 3
      IF( I/=J ) THEN
          A(I,J) = SINTERVAL(0.,2.)
      ELSE
          A(I,J) = SINTERVAL(3.5)
      END IF
      B(I) = SINTERVAL(-1.,1.)
   END DO
END DO
CMPT = 1
NITS = 100
EPS= 1.E-4
. . . . . .
CALL SIGEPPS( 'n', 3, A, LDA, B, LDB, CMPT, MODE, ESTM, EPS, NITS, INFO )
-----------------------------------------------------------------------------------
```

To guarantee the completion of the algorithm, the value of the parameter NITS is set equal to 100 (iterations), which is enough for the above specific example. Note that assigning single-precision intervals to the entries of the matrix *A* and right-hand side vector *B* is carried out by SINTERVAL function supplied by INTERVAL_ARITHMETIC module.

Routines for Inverting Interval Matrices

?trtri

Computes inverse interval matrix to a triangular interval matrix.

Syntax

```
call sitrtri(uplo, diag, n, a, lda, info)
call ditrtri(uplo, diag, n, a, lda, info)
call crtrtri(uplo, diag, n, a, lda, info)
call zrtrtri(uplo, diag, n, a, lda, info)
call cctrtri(uplo, diag, n, a, lda, info)
call zctrtri(uplo, diag, n, a, lda, info)
```

Description

The routine ?trtri computes an interval enclosure of the inverse A^{-1} of an interval triangular matrix *A*.

This routine implements a backward substitution algorithm and produces optimal enclosures of the inverse interval matrix, which is due to the simple structure of the matrix to be inverted.

Input Parameters

uplo	CHARACTER(1). Must be one of 'U' , 'L' , 'u', or 'l'. Indicates whether *A* is upper or lower triangular. If *uplo* = 'U' or 'u', then *A* is upper triangular. If *uplo* = 'L' or 'l', then *A* is lower triangular.
diag	CHARACTER(1). Must be one of 'N', 'U', 'n', or 'u'. If *diag* = 'N' or 'n', then *A* is not a unit triangular matrix.

If *diag* = 'U' or 'u', then *A* is unit triangular: diagonal elements of *A* are assumed to be 1 and not referenced in the array *a*.

n — INTEGER. The order of the matrix *A* (*n* ≥ 0).

a — S_INTERVAL for sitrtri.
D_INTERVAL for ditrtri.
CR_INTERVAL for crtrtri.
ZR_INTERVAL for zrtrtri.
CC_INTERVAL for cctrtri.
ZC_INTERVAL for zctrtri.
Array: DIMENSION (*lda*,*).
Contains the matrix *A*.
The second dimension of *a* must be at least max(1, *n*).

lda — INTEGER. The first dimension of *a*, *lda* ≥ max(1, *n*).

Output Parameters

a — Overwritten by an interval *n*-by-*n* matrix that encloses the inverse matrix A^{-1}.

info — INTEGER.
If *info* = 0, the execution was successful.
If *info* = -*i*, the *i*-th parameter has an illegal value.
If *info* = *i*, the *i*-th diagonal element of *A* contains zero, *A* is singular, and its inversion could not be completed.

?geszi

Computes inverse interval matrix by Schulz iterative method.

Syntax

```
call sigeszi(n, a, lda, info)
call digeszi(n, a, lda, info)
```

Description

For a general interval square matrix *A*, the routine ?geszi computes an enclosure of the inverse interval matrix A^{-1} by the Schulz iterative method.

See in Appendix C of this manual for example code on using this routine.

Input Parameters

n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
a	REAL for sigeszi. DOUBLE PRECISION for digeszi. Array: DIMENSION (*lda*,*). Contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*).
lda	INTEGER. The first dimension of *a*, *lda* ≥ max(1, *n*).

Output Parameters

a	Overwritten by an enclosure of the inverse interval matrix A^{-1}.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *i* > 0, the execution is not successful. If *info* = -*i*, the *i*-th parameter has an illegal value.

Application Notes

Schulz iteration implemented in ?geszi routine converges only provided that the interval matrix *A* is not "too wide". Otherwise, when Schulz iteration diverges, the result is set equal to the interval matrix with the elements [-*infty*,+*infty*], where *infty* is computer infinity of the corresponding kind.

Routines for Checking Properties of Interval Matrices

?gerbr

Tests regularity of an interval matrix by Ris-Beeck and Rex-Rohn criteria.

Syntax

```
call sigerbr( n, a, lda, sr, reg, info )
call digerbr( n, a, lda, sr, reg, info )
```

Description

The routine ?gerbr checks whether a general interval square matrix *A* is regular or singular by using a combination of Ris-Beeck spectral criterion and Rex-Rohn test.

Input Parameters

n	INTEGER. The order of the matrix *A* (*n* ≥ 0).
a	REAL for sigerbr. DOUBLE PRECISION for digerbr. Array: DIMENSION (*lda*,*). Contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*).
lda	INTEGER. The first dimension of *a*, *lda* ≥ max(1, *n*).

Output Parameters

sr	REAL for sigerbr. DOUBLE PRECISION for digerbr. An upper estimate of the spectral radius of the matrix ($\lvert(\text{mid } A)^{-1}\rvert \cdot \text{rad } A$). This is an additional information about the matrix *A*, which is crucial for the so-called strong regularity of *A*.
reg	INTEGER. Displays the results of the singularity test. If *reg* > 0, then *A* is regular. If *reg* < 0, then *A* is singular. If *reg* = 0, then the result is undetermined, that is, the test was not sufficiently sensitive to detect whether the matrix *A* is regular or singular.
info	INTEGER. If *info* = 0, the execution is successful. If *info* = *i* > 0, the execution is not successful. If *info* = -*i*, the *i*-th parameter has an illegal value.

Application Notes

The test implemented in the routine ?gerbr is rather crude, and in critical cases further investigation of the matrix *A* is recommended. However, the routine may help to determine (by comparing the value of *sr* with 1) whether an interval matrix is strongly regular or not.

?gesvr

Tests regularity/singularity of an interval matrix by Rump and Rex-Rohn singular value criteria.

Syntax

```
call sigesvr(n, a, lda, msr, rsr, reg, info)
call digesvr(n, a, lda, msr, rsr, reg, info)
```

Description

The routine ?gesvr checks whether a general interval square matrix *A* is regular or singular by using Rump and Rex-Rohn singular value criteria.

Input Parameters

n	INTEGER. The order of the matrix *A* ($n \geq 0$).
a	REAL for sigesvr. DOUBLE PRECISION for digesvr. Array: DIMENSION (*lda*,*). Contains the matrix *A*. The second dimension of *a* must be at least max(1, *n*).
lda	INTEGER. The first dimension of *a*, $lda \geq \max(1, n)$.

Output Parameters

msr, *rsr*	S_INTERVAL for sigesvr. D_INTERVAL for digesvr. Additional information about the matrix *A*. The intervals represent the ranges of the singular spectra of the midpoint matrix and radius matrix, respectively.
reg	INTEGER. Displays results of the singularity test. If $reg > 0$, then *A* is regular. If $reg < 0$, then *A* is singular. If $reg = 0$, then the result is undetermined, that is, the test was not sufficiently sensitive to detect whether the matrix *A* is regular or singular.
info	INTEGER.

If *info* = 0, the execution is successful.
If *info* = *i* > 0, the execution is not successful.
If *info* = -*i*, the *i*-th parameter has an illegal value.

Application Notes

The routine ?gesvr implements a test that is only a sufficient condition for a matrix to be either regular or singular. This means that in some boundary cases the test may prove not sensitive enough to determine whether a given matrix is regular or singular, and the routine returns *reg* = 0 on output.

Example 12-3 Fortran 90 Code for Testing Regularity of Interval Matrix by Singular Value Criteria

To test regularity of the interval matrix

$$\begin{pmatrix} [2, 4] & [-1, 2] \\ [-2, 1] & [2, 4] \end{pmatrix}$$

by singular value criteria, the following piece of Fortran 90 code may be helpful:

```
-----------------------------------------------------------------------------------------
. . . . . .
USE INTERVAL_ARITHMETIC
. . . . . .
INTEGER, PARAMETER           :: LDA = 2, N = 2
TYPE(D_INTERVAL)           :: A(LDA,N), MSR, RSR
INTEGER           :: REG, INFO
. . . . . .
A(1,1) = DINTERVAL(2.,4.);  A(1,2) = DINTERVAL(-2.,1.)
A(2,1) = DINTERVAL(-1.,2.); A(2,2) = DINTERVAL(2.,4.)
. . . . . .
CALL DIGESVR( N, A, LDA, MSR, RSR, REG, INFO )
-----------------------------------------------------------------------------------------
```

Mutual disposition of the intervals MSR and RSR on the real axis can serve to some extent as a measure of how large the regularity margin is (in case of MSR > RSR), or how far the matrix is from the regular ones (in case of MSR < RSR).

Auxiliary and Utility Routines

?gemip

Performs midpoint-inverse preconditioning of an interval linear system.

Syntax

```
call sigemip(n, nrhs, a, lda, b, ldb, info)
call digemip(n, nrhs, a, lda, b, ldb, info)
call crgemip(n, nrhs, a, lda, b, ldb, info)
call zrgemip(n, nrhs, a, lda, b, ldb, info)
call ccgemip(n, nrhs, a, lda, b, ldb, info)
call zcgemip(n, nrhs, a, lda, b, ldb, info)
```

Description

The routine ?gemip performs midpoint-inverse preconditioning of the interval linear system $AX = B$. This is done through multiplying both matrices A and B by the midpoint-inverse matrix $(\text{mid } A)^{-1}$ in computer (rounded) interval arithmetic.

Input Parameters

n	INTEGER. The order of the matrix A.
nrhs	INTEGER. The number of right-hand sides ($nrhs \geq 0$).
a, *b*	S_INTERVAL for sigemip. D_INTERVAL for digemip. CR_INTERVAL for crgemip. ZR_INTERVAL for zrgemip. CC_INTERVAL for ccgemip. ZC_INTERVAL for zcgemip.

	Arrays: *a* (*lda*,*), *b* (*ldb*,*). The array *a* contains the matrix *A*. The array *b* contains the matrix *B*, whose columns are the right-hand sides for the systems of equations. The second dimension of *a* must be at least max(1,*n*) and the second dimension of *b* must be at least max(1,*nrhs*).
lda	INTEGER. The first dimension of *a*, *lda* ≥ max(1, *n*).
ldb	INTEGER. The first dimension of b, *ldb* ≥ max(1, *n*).

Output Parameters

a	Overwritten by the preconditioned matrix *A*.
b	Overwritten by the preconditioned matrix *B*.
info	INTEGER. If *info* = 0, the execution is successful. If *info* > 0, the execution is not successful. If *info* = -*i*, the *i*-th parameter has an illegal value.

Application Notes

Preconditioning may sometimes extend applicability of the algorithms for the solution of interval linear systems and/or improve the quality of the results produced by these algorithms.

In particular, interval Gauss method, interval Householder method and interval Gauss-Seidel iteration applied to interval linear systems with "wide" matrices that are not diagonally dominant should be preceded by preconditioning to yield better results. For Hansen-Bliek-Rohn procedure, the midpoint-inverse preconditioning is recommended if the middle matrix of the system is far from diagonal.

Example 12-4 Fortran 90 Code for Preconditioning Interval Linear System

The following piece of Fortran code presents an example of how you can perform preconditioning of the interval linear system

$$\begin{pmatrix} [2,4] & [-2,1] \\ [-1,2] & [2,4] \end{pmatrix} x = \begin{pmatrix} [0,2] & [-2,2] \\ [0,2] & [-2,2] \end{pmatrix}$$

and then solve it by using the interval Gauss method:

```
-----------------------------------------------------------------------------------------
. . . . . .
USE INTERVAL_ARITHMETIC
. . . . . .
TYPE(D_INTERVAL)           :: A(2,2), B(2,2)
INTEGER           :: N = 2, NRHS = 2, LDA = 2, LDB = 2, INFO
CHARACTER(1)           :: TRANS = 'n'
. . . . . .
A(1,1) = DINTERVAL(2.,4.);     A(1,2) = DINTERVAL(-2.,1.)
A(2,1) = DINTERVAL(-1.,2.);    A(2,2) = DINTERVAL(2.,4.)
B(1,1) = DINTERVAL(0.,2.);     B(2,1) = DINTERVAL(0.,2.)
B(1,2) = DINTERVAL(-2.,2.);    B(2,2) = DINTERVAL(-2.,2.)
. . . . . .
CALL DIGEMIP( N, NRHS, A, LDA, B, LDB, INFO )
CALL DIGEGAS( TRANS, N, NRHS, A, LDA, B, LDB, INFO )
-----------------------------------------------------------------------------------------
```

For more code examples on using this routine, see in Appendix C of this manual .

Partial Differential Equations Support 13

Intel® Math Kernel Library (Intel® MKL) provides tools for solving Partial Differential Equations (PDE). These tools are Trigonometric Transform interface routines (see Trigonometric Transform Routines) and Poisson Library (see Poisson Library Routines).

Poisson Library is designed for fast solving of simple Helmholtz, Poisson, and Laplace problems. The Trigonometric Transform interface, which underlies the solver, is, in turn, based on Intel MKL DFT interface (refer to Fourier Transform Functions), optimized for Intel® processors.

Direct use of the Trigonometric Transform routines may be helpful to those who have already implemented their own solvers similar to the one that the Poisson Library provides. As it may be hard enough to modify the original code so as to make it work with Poisson Library, you are encouraged to use fast sine, cosine, and staggered cosine transforms implemented in the Trigonometric Transform interface to improve performance of your solver.

Both Trigonometric Transform and Poisson Library routines can be called from C and Fortran-90, although the interfaces description uses C convention. Fortran-90 users can find routine calls specifics in the "Calling PDE Support Routines from Fortran-90" section.

Trigonometric Transform Routines

In addition to Discrete Fourier Transform (DFT) interface, described in chapter " Fast Fourier Transforms", Intel® MKL supports the Real Discrete Trigonometric Transforms interface referred to as TT interface. The interface implements a group of routines (TT routines) used to compute sine, cosine, and staggered cosine transforms. TT interface provides much flexibility of use: you can adjust routines to your particular needs at the cost of manual tuning routine parameters or just call routines with default parameter values. Current Intel MKL implementation of TT interface can be used in solving partial differential equations and contains routines that are helpful for Fast Poisson and similar solvers.

To describe Intel MKL TT interface, C convention will be used. Fortran users should refer to Calling PDE Support Routines from Fortran-90.

For the list of Trigonometric Transforms currently implemented in Intel MKL TT interface, see Transforms Implemented.

Transforms Implemented

TT routines allow computing the following transforms:

Forward sine transform

$$F(k) = \frac{2}{n}\sum_{i=1}^{n-1} f(i)\sin\frac{ki\pi}{n}, k = 1,...,n-1$$

Backward sine transform

$$f(i) = \sum_{k=1}^{n-1} F(k)\sin\frac{ki\pi}{n}, i = 1,...,n-1$$

Forward cosine transform

$$F(k) = \frac{1}{n}\left[f(0) + \cos\frac{k\pi}{n} f(n)\right] + \frac{2}{n}\sum_{i=1}^{n-1} f(i)\cos\frac{ki\pi}{n}, k = 0,...,n$$

Backward cosine transform

$$f(i) = \frac{1}{2}\left[F(0) + \cos\frac{i\pi}{n} F(n)\right] + \sum_{k=1}^{n-1} F(k)\cos\frac{ki\pi}{n}, i = 0,...,n$$

Forward staggered cosine transform

$$F(k) = \frac{1}{n} f(0) + \frac{2}{n}\sum_{i=1}^{n-1} f(i)\cos\frac{(2k+1)i\pi}{2n}, k = 0,...,n-1$$

Backward staggered cosine transform

$$f(i) = \sum_{k=0}^{n-1} F(k)\cos\frac{(2k+1)i\pi}{2n}, i = 0,...,n-1$$

NOTE. The size of the transform n must be even. Current implementation of Trigonometric Transforms does not support transforms of odd size.

Sequence of Invoking TT Routines

Computation of a transform using TT interface is conceptually divided into four steps each of which is performed via a dedicated routine. Table 13-1 lists names of the routines and briefly describes their purpose and use.

Most of TT routines have versions operating with single-precision and double-precision data. Names of such routines begin respectively with "s" and "d". The wildcard "?" stands for either of these symbols in routine names.

Table 13-1 TT Interface Routines

Routine	Description
?_init_trig_transform	Initializes basic data structures of Trigonometric Transforms.
?_commit_trig_transform	Checks consistency and correctness of user-defined data as well as creates a data structure to be used by Intel MKL DFT interface[1].
?_forward_trig_transform ?_backward_trig_transform	Computes a forward/backward Trigonometric Transform of a specified type using the appropriate formula (see Transforms Implemented).
free_trig_transform	Cleans the memory used by a data structure needed for calling DFT interface[1].

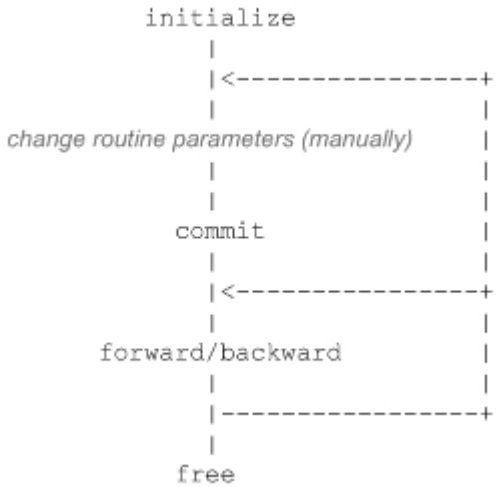
[1]TT routines call Intel MKL DFT interface for better performance.

To find once a transformed vector for a particular input vector, the Intel MKL TT interface routines are normally invoked in the order in which they are listed in Table 13-1.

NOTE. Though the order of invoking TT routines may be changed, it is highly recommended to follow the above order of routine calls.

The diagram in Figure 13-1 indicates the typical order in which TT interface routines can be invoked in a general case (prefixes and suffixes in routine names are omitted).

Figure 13-1 Typical Order of Invoking TT Interface Routines



A general scheme of using TT routines for double-precision computations is shown below. Similar scheme holds for single-precision computations with the only difference in the initial letter of routine names.

```
...
    d_init_trig_transform(&n, &tt_type, ipar, dpar, &ir);
/* Change parameters in ipar if necessary. */
/* Note that the result of the Transform will be in f ! If you want to
preserve   the data stored in f,
save them before this place in your code */
    d_commit_trig_transform(f, &handle, ipar, dpar, &ir);
    d_forward_trig_transform(f, &handle, ipar, dpar, &ir);
    d_backward_trig_transform(f, &handle, ipar, dpar, &ir);
    free_trig_transform(&handle, ipar, &ir);
/* here the user may clean the memory used by f, dpar, ipar */
...
```

You can find examples of Fortran-90 and C code that use TT interface routines to solve one-dimensional Helmholtz problem in the "Trigonometric Transform Code Examples" section in Appendix C.

Interface Description

All types in this documentation are standard C types: `INT`, `FLOAT`, and `DOUBLE`. Fortran-90 users can call the routines with `INTEGER`, `REAL`, and `DOUBLE PRECISION` Fortran types, respectively (see examples in the "Trigonometric Transform Code Examples" section in Appendix C).

Routine Options

All TT routines have parameters that are used for passing various options to the routines. These parameters are arrays *ipar*, *dpar* and *spar*. Values for these parameters should be specified very carefully (see Common Parameters). You can change these values during computations to meet your needs.

NOTE. You must provide correct and consistent parameters to the routines to avoid failure or wrong results.

User Data Arrays

TT routines take arrays of user data as input. For example, user arrays are passed to the routine `d_forward_trig_transform` to compute a forward Trigonometric Transform. To minimize storage requirements and improve the overall run-time efficiency, Intel MKL TT routines do not make copies of user input arrays.

NOTE. If you need a copy of your input data arrays, you should save them yourself.

TT Routines

The section gives detailed description of TT routines, their syntax, parameters and values they return. Double-precision and single-precision versions of the same routine are described together.

TT routines call Intel MKL DFT interface (described in section "DFT Functions" in chapter "Fast Fourier Transforms"), which enhances performance of the routines.

?_init_trig_transform

Initializes basic data structures of a Trigonometric Transform.

Syntax

```
void d_init_trig_transform (int *n, int *tt_type, int ipar[], double dpar[], int *stat);
```

```
void s_init_trig_transform (int *n, int *tt_type, int ipar[], float spar[], int *stat);
```

Input Parameters

n	`int*`. Contains the size of the problem, which should be an even positive integer. Note that data vector of the transform, which other TT routines will use, must have size *n*+1.
tt_type	`int*`. Contains the type of transform to compute, defined via a set of named constants. The following constants are available in the current implementation of TT interface: `MKL_SINE_TRANSFORM`, `MKL_COSINE_TRANSFORM` and `MKL_STAGGERED_COSINE_TRANSFORM`.

Output Parameters

ipar	`int` array of size 128. Contains integer data needed for Trigonometric Transform computations.
dpar	`double` array of size 3*n*/2+1. Contains double-precision data needed for Trigonometric Transform computations.
spar	`float` array of size 3*n*/2+1. Contains single-precision data needed for Trigonometric Transform computations.
stat	`int*`. Contains the routine completion status, which is also written to *ipar[6]*. The status should be 0 to proceed to other TT routines.

Description

The routine initializes basic data structures for Trigonometric Transforms of appropriate precision. After a call to `?_init_trig_transform`, all subsequently invoked TT routines use values of *ipar* and *dpar* (*spar*) array parameters returned by `?_init_trig_transform`. The routine

initializes the entire array *ipar*. In the *dpar* or *spar* array, ?_init_trig_transform initializes elements that do not depend upon the type of transform. For detailed description of arrays *ipar*, *dpar* and *spar*, refer to the Common Parameters section.You can skip calling the initialization routine in your code. For more information, see Caveat on Parameter Modifications.

Return Values

stat= 0	The routine successfully completed the task. In general, to proceed with computations, the routine should complete with this *stat* value.
stat= -99999	The routine failed to complete the task.

?_commit_trig_transform

Checks consistency and correctness of user's data as well as initializes certain data structures required to perform the Trigonometric Transform.

Syntax

```
void d_commit_trig_transform (double f[], DFTI_DESCRIPTOR_HANDLE *handle,
int ipar[], double dpar[], int *stat);
```

```
void s_commit_trig_transform (float f[], DFTI_DESCRIPTOR_HANDLE *handle, int
ipar[], float spar[], int *stat);
```

Input Parameters

f	double for d_commit_trig_transform,float for s_commit_trig_transform array of size n+1, where n is the size of the problem. Contains data vector to be transformed.
ipar	int array of size 128. Contains integer data needed for Trigonometric Transform computations.
dpar	double array of size 3n/2+1. Contains double-precision data needed for Trigonometric Transform computations. Most of the array elements are to be initialized by the routine.

spar	`float` array of size 3*n*/2+1. Contains single-precision data needed for Trigonometric Transform computations. Most of the array elements are to be initialized by the routine.

Output Parameters

handle	`DFTI_DESCRIPTOR_HANDLE*`. The data structure used by Intel MKL DFT interface (for details, refer to section "DFT Functions" in chapter "Fast Fourier Transforms").
ipar	Contains integer data needed for Trigonometric Transform computations. On output, *ipar[6]* is updated with the *stat* value.
dpar	Contains double-precision data needed for Trigonometric Transform computations. On output, the entire array is initialized.
spar	Contains single-precision data needed for Trigonometric Transform computations. On output, the entire array is initialized.
stat	`int*`. Contains the routine completion status, which is also written to *ipar[6]*.

Description

The routine `?_commit_trig_transform` checks consistency and correctness of the parameters to be passed to the transform routines `?_forward_trig_transform` and/or `?_backward_trig_transform`. The routine also initializes the following data structures: *handle*, *dpar* in case of `d_commit_trig_transform`, and *spar* in case of `s_commit_trig_transform`. The `?_commit_trig_transform` routine initializes only those elements of *dpar* or *spar* that depend upon the type of transform, defined in the `?_init_trig_transform` routine and passed to `?_commit_trig_transform` with the *ipar* array. The size of the problem *n*, which determines sizes of the array parameters, is also passed to the routine with the *ipar* array and defined in the previously called `?_init_trig_transform` routine. For detailed description of arrays *ipar*, *dpar* and *spar*, refer to the Common Parameters section. The routine performs only a basic check for correctness and consistency of the parameters. If you are going to modify parameters of TT routines, see the Caveat on Parameter Modifications section. Unlike `?_init_trig_transform`, you cannot skip calling this routine in your code.

Return Values

stat= 11	The routine produced some warnings and made some changes in the parameters to achieve their correctness and/or consistency. You may proceed with computations by assigning *ipar[6]*=0 if you are sure that the parameters are correct.
stat= 10	The routine made some changes in the parameters to achieve their correctness and/or consistency. You may proceed with computations by assigning *ipar[6]*=0 if you are sure that the parameters are correct.
stat= 1	The routine produced some warnings. You may proceed with computations by assigning *ipar[6]*=0 if you are sure that the parameters are correct.
stat= 0	The routine completed the task normally.
stat= -100	The routine stopped for any of the following reasons: • An error in the user's data was encountered. • Data in *ipar*, *dpar* or *spar* parameters became incorrect and/or inconsistent as a result of modifications.
stat= -1000	The routine stopped because of DFT interface error.
stat= -10000	The routine stopped as the initialization failed to complete or parameter *ipar[0]* was altered by mistake.

NOTE. Although positive values of *stat* usually indicate minor problems with the input data and Trigonometric Transform computations can be continued, it is highly recommended to investigate the problem first and achieve *stat*=0.

?_forward_trig_transform

Computes the forward Trigonometric Transform of type specified by a parameter.

Syntax

```
void d_forward_trig_transform (double f[], DFTI_DESCRIPTOR_HANDLE *handle,
int ipar[], double dpar[], int *stat);
```

```
void s_forward_trig_transform (float f[], DFTI_DESCRIPTOR_HANDLE *handle,
int ipar[], float spar[], int *stat);
```

Input Parameters

f	`double` for `d_forward_trig_transform`, `float` for `s_forward_trig_transform` array of size *n*+1, where *n* is the size of the problem. At input, contains data vector to be transformed.
handle	`DFTI_DESCRIPTOR_HANDLE*`. The data structure used by Intel MKL DFT interface (for details, refer to section "DFT Functions" in chapter "Fast Fourier Transforms").
ipar	`int` array of size 128. Contains integer data needed for Trigonometric Transform computations.
dpar	`double` array of size 3*n*/2+1. Contains double-precision data needed for Trigonometric Transform computations.
spar	`float` array of size 3*n*/2+1. Contains single-precision data needed for Trigonometric Transform computations.

Output Parameters

f	Contains the transformed vector on output.
ipar	Contains integer data needed for Trigonometric Transform computations. On output, *ipar[6]* is updated with the *stat* value.
stat	`int*`. Contains the routine completion status, which is also written to *ipar[6]*.

Description

The routine computes the forward Trigonometric Transform of type defined in the ?_init_trig_transform routine and passed to ?_forward_trig_transform with the *ipar* array. The size of the problem *n*, which determines sizes of the array parameters, is also passed to the routine with the *ipar* array and defined in the previously called ?_init_trig_transform routine. Other data that facilitates the computation is created by ?_commit_trig_transform and supplied in *dpar* or *spar*. For detailed description of arrays *ipar*, *dpar* and *spar*, refer to the Common Parameters section. The routine has a commit step, which calls the ?_commit_trig_transform routine. The transform is computed according to formulas given in the Transforms Implemented section. The routine replaces the input vector *f* with the transformed vector.

NOTE. If you need a copy of the data vector *f* to be transformed, you should make the copy before calling the ?_forward_trig_transform routine.

Return Values

stat= 0	The routine completed the task normally.
stat= -100	The routine stopped for any of the following reasons: • An error in the user's data was encountered. • Data in *ipar*, *dpar* or *spar* parameters became incorrect and/or inconsistent as a result of modifications.
stat= -1000	The routine stopped because of DFT interface error.
stat= -10000	The routine stopped as its commit step failed to complete or the parameter *ipar[0]* was altered by mistake.

?_backward_trig_transform

Computes the backward Trigonometric Transform of type specified by a parameter.

Syntax

```
void d_backward_trig_transform (double f[], DFTI_DESCRIPTOR_HANDLE *handle,
int ipar[], double dpar[], int *stat);
```

```
void s_backward_trig_transform (float f[], DFTI_DESCRIPTOR_HANDLE *handle,
int ipar[], float spar[], int *stat);
```

Input Parameters

f	`double` for `d_backward_trig_transform`, `float` for `s_backward_trig_transform` array of size *n*+1, where *n* is the size of the problem. At input, contains data vector to be transformed.
handle	`DFTI_DESCRIPTOR_HANDLE*`. The data structure used by Intel MKL DFT interface (for details, refer to section "DFT Functions" in chapter "Fast Fourier Transforms").
ipar	`int` array of size 128. Contains integer data needed for Trigonometric Transform computations.
dpar	`double` array of size 3*n*/2+1. Contains double-precision data needed for Trigonometric Transform computations.
spar	`float` array of size 3*n*/2+1. Contains single-precision data needed for Trigonometric Transform computations.

Output Parameters

f	Contains the transformed vector on output.
ipar	Contains integer data needed for Trigonometric Transform computations. On output, *ipar[6]* is updated with the *stat* value.
stat	`int*`. Contains the routine completion status, which is also written to *ipar[6]*.

Description

The routine computes the backward Trigonometric Transform of type defined in the ?_init_trig_transform routine and passed to ?_backward_trig_transform with the *ipar* array. The size of the problem *n*, which determines sizes of the array parameters, is also passed to the routine with the *ipar* array and defined in the previously called ?_init_trig_transform routine. Other data that facilitates the computation is created by ?_commit_trig_transform and supplied in *dpar* or *spar*. For detailed description of arrays *ipar*, *dpar* and *spar*, refer to the Common Parameters section. The routine has a commit step, which calls the ?_commit_trig_transform routine. The transform is computed according to formulas given in the Transforms Implemented section. The routine replaces the input vector *f* with the transformed vector.

NOTE. If you need a copy of the data vector *f* to be transformed, you should make the copy before calling the ?_backward_trig_transform routine.

Return Values

stat= 0	The routine completed the task normally.
stat= -100	The routine stopped for any of the following reasons: • An error in the user's data was encountered. • Data in *ipar*, *dpar* or *spar* parameters became incorrect and/or inconsistent as a result of modifications.
stat= -1000	The routine stopped because of DFT interface error.
stat= -10000	The routine stopped as its commit step failed to complete or the parameter *ipar[0]* was altered by mistake.

free_trig_transform

Cleans the memory allocated for the data structure used by DFT interface.

Syntax

```
void free_trig_transform (DFTI_DESCRIPTOR_HANDLE *handle, int ipar[], int *stat);
```

Input Parameters

ipar	`int` array of size 128. Contains integer data needed for Trigonometric Transform computations.
handle	`DFTI_DESCRIPTOR_HANDLE*`. The data structure used by Intel MKL DFT interface (for details, refer to section "DFT Functions" in chapter "Fast Fourier Transforms").

Output Parameters

handle	The data structure used by Intel MKL DFT interface. Memory allocated for the structure is released on output.
ipar	Contains integer data needed for Trigonometric Transform computations. On output, *ipar[6]* is updated with the *stat* value.
stat	`int*`. Contains the routine completion status, which is also written to *ipar[6]*.

Description

The routine cleans the memory used by the *handle* structure, needed for Intel MKL DFT functions. If you need to release memory allocated for other parameters, you should include the memory cleaning in your code.

Return Values

stat= 0	The routine completed the task normally.
stat= -1000	The routine stopped because of DFT interface error.
stat= -99999	The routine failed to complete the task.

Common Parameters

This section provides description of array parameters that hold TT routine options: *ipar*, *dpar* and *spar*.

NOTE. Initial values are assigned to the array parameters by the appropriate ?_init_trig_transform and ?_commit_trig_transform routines.

ipar — int array of size 128, holds integer data needed for Trigonometric Transform computations. Its elements are described in Table 13-2:

Table 13-2 Elements of the ipar Array

Index	Description
0	Contains the size of the problem to solve. The ?_init_trig_transform routine sets *ipar[0]*=*n* and all subsequently called TT routines use *ipar[0]* as the size of the transform. Current implementation of TT interface supports transforms of even size only.
1	Contains error messaging options: • *ipar[1]*=-1 indicates that all error messages will be printed to the file MKL_Trig_Transforms_log.txt in the folder from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device. • *ipar[1]*=0 indicates that no error messages will be printed. • *ipar[1]*=1 is the default value. It indicates that all error messages will be printed to the preconnected default output device (usually, screen). In case of errors, any TT routine will assign a non-zero value to *stat* regardless of the *ipar[1]* setting.
2	Contains warning messaging options: • *ipar[2]*=-1 indicates that all warning messages will be printed to the file MKL_Trig_Transforms_log.txt in the directory from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device.

Index	Description
	• *ipar[2]*=0 indicates that no warning messages will be printed. • *ipar[2]*=1 is the default value. It indicates that all warning messages will be printed to the preconnected default output device (usually, screen). In case of warnings, the *stat* parameter will acquire a non-zero value regardless of the *ipar[2]* setting.
3 through 4	Reserved for future use.
5	Contains the type of the transform. The ?_init_trig_transform routine sets *ipar[5]*=*tt_type* and all subsequently called TT routines use *ipar[5]* as the type of the transform.
6	Contains the *stat* value returned by the last completed TT routine. Used to check that the previous call to a TT routine completed with *stat*=0.
7	Informs the ?_commit_trig_transform routines whether to initialize data structures *dpar* (*spar*) and *handle*. *ipar[7]*=0 indicates that the routine should skip the initialization and only check correctness and consistency of the parameters. Otherwise, the routine initializes the data structures. The default value is 1. The possibility to check correctness and consistency of input data without initializing data structures *dpar*, *spar* and *handle* prevents from losing performance in a repeated use of the same transform for different data vectors.Note that you can benefit from the opportunity that *ipar[7]* gives only if you are sure to have supplied proper tolerance value in the *dpar* or *spar* array. Otherwise, avoid tuning this parameter.
8	Contains message style options for TT routines. If *ipar[8]*=0 then TT routines print all error and warning messages in Fortran-style notations. Otherwise, TT routines print the messages in C-style notations. The default value is 1. When selecting between these notations, you should mind that by default, numbering of elements in C arrays starts from 0 and in Fortran, it starts from 1. For example, if a part of a C-style message looks like "parameter ipar[0]=3 should be an even integer", then the corresponding Fortran-style message will be "parameter ipar(1)=3 should be an even integer". *ipar[8]* enables users to view messages in a more convenient style.

Index	Description
9	Specifies the number of OpenMP threads to run TT routines in the OpenMP environment of the Poisson Library. The default value is 1. It is highly recommended not to alter this value. See also Caveat on Parameter Modifications.
10 through 127	Reserved for future use.

NOTE. You may declare the *ipar* array in your code as `int ipar[10]`. However, for compatibility with later versions of Intel MKL TT interface, which may require more *ipar* values, it is highly recommended to declare *ipar* as `int ipar[128]`.

Arrays *dpar* and *spar* are similar to each other and differ only in the data precision:

dpar `double` array of size 3*n*/2+1, holds data needed for double-precision routines to perform TT computations. This array is initialized in the `d_init_trig_transform` and `d_commit_trig_transform` routines.

spar `float` array of size 3*n*/2+1, holds data needed for single-precision routines to perform TT computations. This array is initialized in the `s_init_trig_transform` and `s_commit_trig_transform` routines.

As *dpar* and *spar* have similar elements in respective positions, the elements are described together in Table 13-3:

Table 13-3 Elements of the dpar and spar Arrays

Index	Description
0	The element contains the first absolute tolerance used by the appropriate `?_commit_trig_transform` routine. For a staggered cosine or a sine transform, *f[n]* should be equal to 0.0 and for a sine transform, *f[0]* should be equal to 0.0. The `?_commit_trig_transform` routine checks if absolute values of these parameters are below *dpar[0]*n* or *spar[0]*n*, depending on the routine precision. You can suppress warnings resulting from tolerance checks by setting *dpar[0]* or *spar[0]* to a sufficiently large number.
1	The element is reserved for future use.

Index	Description
2 through 3*n*/2	The elements contain tabulated values of trigonometric functions. Contents of the elements depend upon the type of transform *tt_type*, set up in the ?_commit_trig_transform routine: • If *tt_type*=MKL_SINE_TRANSFORM, then the array contains *n*/2 elements with tabulated sine values in *n*/2 successive array elements starting from the third element (with index 2). The rest of the array is not used in this transform. • If *tt_type*=MKL_COSINE_TRANSFORM, then the array contains *n* elements with tabulated cosine values in *n* successive array elements starting from the third element (with index 2). The rest of the array is not used in this transform. • If *tt_type*=MKL_STAGGERED_COSINE_TRANSFORM, then the array contains 3*n*/2-2 elements with tabulated sine and cosine values in 3*n*/2-2 successive array elements starting from the third element (with index 2). The rest of the array is not used in this transform.

NOTE. You may define the array size depending upon the type of transform.

Caveat on Parameter Modifications

Flexibility of TT interface makes it possible to skip calling the ?_init_trig_transform routine and initialize the basic data structures explicitly in your code. You may also need to modify contents of *ipar*, *dpar* and *spar* arrays after initialization. When doing so, you should provide correct and consistent data in the arrays. Mistakenly altered arrays cause errors or wrong computation. You can perform basic check for correctness and consistency of parameters by calling the ?_commit_trig_transform routine but it does not ensure the correct result of a transform, it only reduces the chance of errors or wrong result.

NOTE. To supply correct and consistent parameters to TT routines, you should have considerable experience in using TT interface and good understanding of elements that the *ipar*, *spar* and *dpar* arrays contain and dependencies between values of these elements.

However, in rare occurrences, even advanced users may fail to compute a transform using TT routines after the parameter modifications.

WARNING. The only way that ensures proper computation of the Trigonometric Transforms is to follow a typical sequence of invoking the routines and not change the default set of parameters. So, avoid modifications of *ipar*, *dpar* and *spar* arrays unless a strong need arises.

Implementation Details

Several aspects of the Intel MKL TT interface are platform-specific and language-specific. To promote portability across platforms and ease of use across different languages, users are provided with Intel MKL TT language-specific header files to include in their code. Currently, the following of them are available:

- mkl_trig_transforms.h, to be used together with mkl_dfti.h, for C programs.
- mkl_trig_transforms.f90, to be used toghether with mkl_dfti.f90, for Fortran-90 programs.

NOTE. Use of the Intel MKL TT software without including one of the above header files is not supported.

C-specific Header File

The C-specific header file defines the following function prototypes:

```
void d_init_trig_transform(int *, int *, int *, double *, int *);

void d_commit_trig_transform(double *, DFTI_DESCRIPTOR_HANDLE *, int *,
double *, int *);


void d_forward_trig_transform(double *, DFTI_DESCRIPTOR_HANDLE *, int *, double *, int *);

void d_backward_trig_transform(double *, DFTI_DESCRIPTOR_HANDLE *, int *,
double *, int *);


void s_init_trig_transform(int *, int *, int *, float *, int *);

void s_commit_trig_transform(float *, DFTI_DESCRIPTOR_HANDLE *, int *, float
 *, int *);
```

```
void s_forward_trig_transform(float *, DFTI_DESCRIPTOR_HANDLE *, int *, float
 *, int *);

void s_backward_trig_transform(float *, DFTI_DESCRIPTOR_HANDLE *, int *,
float *, int *);


void free_trig_transform(DFTI_DESCRIPTOR_HANDLE *, int *, int *);
```

Fortran-Specific Header File

The Fortran-90-specific header file defines the following function prototypes:

```
SUBROUTINE D_INIT_TRIG_TRANSFORM(n, tt_type, ipar, dpar, stat)
    INTEGER, INTENT(IN) :: n, tt_type
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(8), INTENT(INOUT) :: dpar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE D_INIT_TRIG_TRANSFORM

SUBROUTINE D_COMMIT_TRIG_TRANSFORM(f, handle, ipar, dpar, stat)
    REAL(8), INTENT(INOUT) :: f(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: handle
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(8), INTENT(INOUT) :: dpar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE D_COMMIT_TRIG_TRANSFORM

SUBROUTINE D_FORWARD_TRIG_TRANSFORM(f, handle, ipar, dpar, stat)
    REAL(8), INTENT(INOUT) :: f(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: handle
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(8), INTENT(INOUT) :: dpar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE D_FORWARD_TRIG_TRANSFORM

SUBROUTINE D_BACKWARD_TRIG_TRANSFORM(f, handle, ipar, dpar, stat)
    REAL(8), INTENT(INOUT) :: f(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: handle
```

```
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(8), INTENT(INOUT) :: dpar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE D_BACKWARD_TRIG_TRANSFORM

SUBROUTINE S_INIT_TRIG_TRANSFORM(n, tt_type, ipar, spar, stat)
    INTEGER, INTENT(IN) :: n, tt_type
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(4), INTENT(INOUT) :: spar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE S_INIT_TRIG_TRANSFORM

SUBROUTINE S_COMMIT_TRIG_TRANSFORM(f, handle, ipar, spar, stat)
    REAL(4), INTENT(INOUT) :: f(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: handle
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(4), INTENT(INOUT) :: spar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE S_COMMIT_TRIG_TRANSFORM

SUBROUTINE S_FORWARD_TRIG_TRANSFORM(f, handle, ipar, spar, stat)
    REAL(4), INTENT(INOUT) :: f(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: handle
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(4), INTENT(INOUT) :: spar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE S_FORWARD_TRIG_TRANSFORM
```

```
SUBROUTINE S_BACKWARD_TRIG_TRANSFORM(f, handle, ipar, spar, stat)
    REAL(4), INTENT(INOUT) :: f(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: handle
    INTEGER, INTENT(INOUT) :: ipar(*)
    REAL(4), INTENT(INOUT) :: spar(*)
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE S_BACKWARD_TRIG_TRANSFORM

SUBROUTINE FREE_TRIG_TRANSFORM(handle, ipar, stat)
    INTEGER, INTENT(INOUT) :: ipar(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: handle
    INTEGER, INTENT(OUT) :: stat
END SUBROUTINE FREE_TRIG_TRANSFORM
```

Fortran-90 specifics of the TT routines usage are similar for all Intel MKL PDE support tools and described in the Calling PDE Support Routines from Fortran-90 section.

Poisson Library Routines

In addition to Real Discrete Trigonometric Transforms (TT) interface (refer to Trigonometric Transform Routines), Intel® MKL supports the Poisson Library interface referred to as PL interface. The interface implements a group of routines (PL routines) used to compute a solution of Laplace, Poisson, and Helmholtz problems of special kind using discrete Fourier transforms. Laplace and Poisson problems are special cases of a more general Helmholtz problem. The problems being solved are defined more exactly in the Poisson Library Implemented subsection. PL interface provides much flexibility of use: you can adjust routines to your particular needs at the cost of manual tuning routine parameters or just call routines with default parameter values. The interface can adjust style of error and warning messages to C or Fortran notations by setting up a dedicated parameter. This adds convenience to debugging, as users can read information in the way that is natural for their code. Intel MKL PL interface currently contains only routines that implement the following solvers:

- Fast Laplace, Poisson and Helmholtz solvers in a Cartesian coordinate system
- Fast Poisson and Helmholtz solvers in a spherical coordinate system.

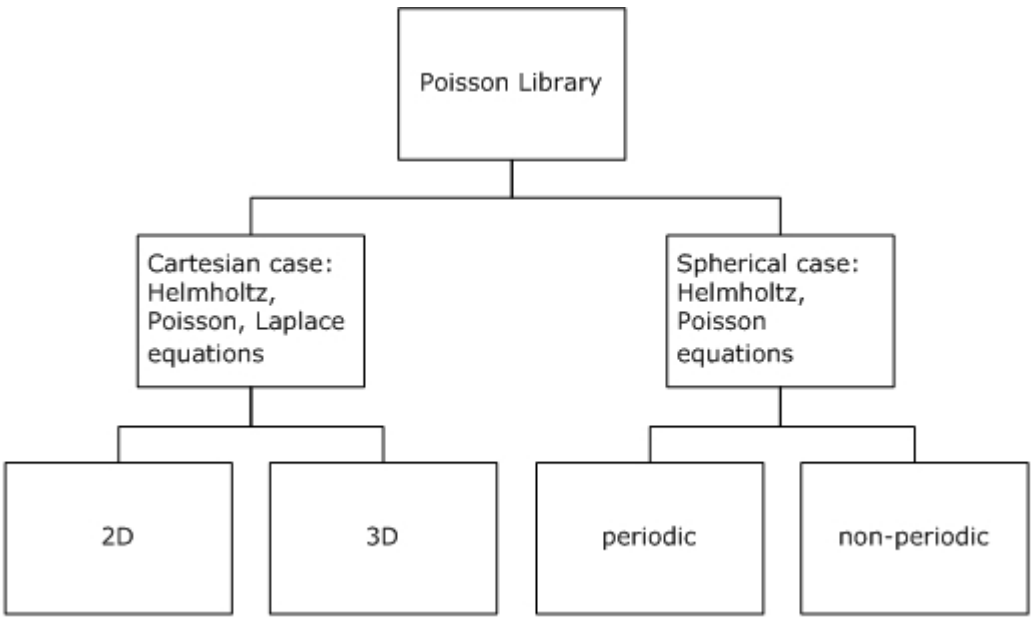
To describe Intel MKL PL interface, C convention will be used. Fortran usage specifics can be found in the Calling PDE Support Routines from Fortran-90 section.

NOTE. Fortran users should mind that respective array indices in Fortran increase by 1.

Poisson Library Implemented

PL routines enable approximate solving of certain two-dimensional and three-dimensional problems. Figure 13-2 shows general structure of the Poisson Library.

Figure 13-2 Structure of the Poisson Library



Sections below provide details of the problems that can be solved using Intel MKL PL.

Two-Dimensional Problems

Notational Conventions

PL interface description uses the following notation for boundaries of a rectangular domain $a_x < x < b_x$, $a_y < y < b_y$ on a Cartesian plane:

$bd\_a_x = \{x = a_x, a_y \le y \le b_y\}$, $bd\_b_x = \{x = b_x, a_y \le y \le b_y\}$

$bd\_a_y = \{a_x \le x \le b_x, y = a_y\}$, $bd\_b_y = \{a_x \le x \le b_x, y = b_y\}$.

The wildcard "+" may stand for any of the symbols a_x, b_x, a_y, b_y, so that bd_+ denotes any of the above boundaries.

PL interface description uses the following notation for boundaries of a rectangular domain $a_\phi < \phi < b_\phi$, $a_\theta < \theta < b_\theta$ on a sphere $0 \le \phi \le 2\pi$, $0 \le \theta \le \pi$:

$bd\_a_\phi = \{\phi = a_\phi, a_\theta \le \theta \le b_\theta\}$, $bd\_b_\phi = \{\phi = b_\phi, a_\theta \le \theta \le b_\theta\}$

$bd\_a_\theta = \{a_\phi \le \phi \le b_\phi, \theta = a_\theta\}$, $bd\_b_\theta = \{a_\phi \le \phi \le b_\phi, \theta = b_\theta\}$.

The wildcard "~" may stand for any of the symbols a_ϕ, b_ϕ, a_θ, b_θ, so that $bd\_\sim$ denotes any of the above boundaries.

Two-dimensional (2D) Helmholtz problem on a Cartesian plane

2D Helmholtz problem is to find an approximate solution of Helmholtz equation

$$-\frac{\partial^2 u}{\partial x^2}-\frac{\partial^2 u}{\partial y^2}+qu = f(x,y), q = const \ge 0$$

in a rectangle, that is, a rectangular domain $a_x < x < b_x$, $a_y < y < b_y$, with one of the following boundary conditions on each boundary bd_+:

- Dirichlet boundary condition

$$u(x,y) = G(x,y)$$

- Neumann boundary condition

$$\frac{\partial u}{\partial n}(x,y) = g(x,y)$$

where

n= -x on bd_a_x, n= x on bd_b_x,

n= -y on bd_a_y, n= y on bd_b_y.

Two-dimensional (2D) Poisson problem on a Cartesian plane

The Poisson problem is a special case of the Helmholtz problem, when q=0. 2D Poisson problem is to find an approximate solution of Poisson equation

$$-\frac{\partial^2 u}{\partial x^2}-\frac{\partial^2 u}{\partial y^2}=f(x,y)$$

in a rectangle $a_x < x < b_x$, $a_y < y < b_y$ with Dirichlet or Neumann boundary condition on each boundary bd_+. In case of a problem with Neumann boundary condition on the entire boundary, you can find the solution of the problem up to a constant only. In this case, Poisson Library will compute the solution that provides the minimal Euclidean norm of a residual.

Two-dimensional (2D) Laplace problem on a Cartesian plane

The Laplace problem is a special case of the Helmholtz problem, when q=0 and $f(x, y)$=0. 2D Laplace problem is to find an approximate solution of Laplace equation

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=0$$

in a rectangle $a_x < x < b_x$, $a_y < y < b_y$ with Dirichlet or Neumann boundary condition on each boundary bd_+.

Helmholtz problem on a sphere

Helmholtz problem on a sphere is to find an approximate solution of Helmholtz equation

$$-\Delta_s u + qu = f,\ q = const \geq 0,$$
$$\Delta_s = \frac{1}{\sin^2\theta}\frac{\partial^2}{\partial\varphi^2} + \frac{1}{\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial}{\partial\theta}\right)$$

in a spherical rectangle that is, a domain bounded by angles $a_\phi \leq \phi \leq b_\phi$, $a_\theta \leq \theta \leq b_\theta$, with boundary conditions for particular domains listed in Table 13-4.

Table 13-4 Details of Helmholtz Problem on a Sphere

Domain on a sphere	Boundary condition	Periodic/non-periodic case
Rectangular, that is, $b_\phi - a_\phi < 2\pi$ and $b_\theta - a_\theta < \pi$	Homogeneous Dirichlet boundary conditions on each boundary `bd_~`	non-periodic
Where $a_\phi = 0$, $b_\phi = 2\pi$, and $b_\theta - a_\theta < \pi$	Homogeneous Dirichlet boundary conditions on the boundaries `bd_`a_θ and `bd_`b_θ	periodic
Entire sphere, that is, $a_\phi = 0$, $b_\phi = 2\pi$, $a_\theta = 0$, and $b_\theta = \pi$	Boundary condition $\left(\sin\theta\frac{\partial u}{\partial\theta}\right)_{\substack{\theta\to 0\\ \theta\to\pi}} = 0$ at the poles.	periodic

Poisson problem on a sphere

The Poisson problem is a special case of the Helmholtz problem, when q=0. Poisson problem on a sphere is to find an approximate solution of Poisson equation

$$-\Delta_s u = f,\ \Delta_s = \frac{1}{\sin^2\theta}\frac{\partial^2}{\partial\varphi^2} + \frac{1}{\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial}{\partial\theta}\right)$$

in a spherical rectangle $a_\phi \le \phi \le b_\phi$, $a_\theta \le \theta \le b_\theta$ in cases listed in Table 13-4. The solution to the Poisson problem on the entire sphere can be found up to a constant only. In this case, Poisson Library will compute the solution that provides the minimal Euclidean norm of a residual.

Approximation of 2D problems

To find an approximate solution for any of the 2D problems, a uniform mesh is built in the rectangular domain:

$$\left\{x_i = a_x + ih_x, y_j = a_y + jh_y\right\},$$
$$i = 0,\ldots,n_x, j = 0,\ldots,n_y, h_x = \frac{b_x - a_x}{n_x}, h_y = \frac{b_y - a_y}{n_y}$$

in the Cartesian case and

$$\left\{\varphi_i = a_\varphi + ih_\varphi, \theta_j = a_\theta + jh_\theta\right\},$$
$$i = 0,\ldots,n_\varphi, j = 0,\ldots,n_\theta, h_\varphi = \frac{b_\varphi - a_\varphi}{n_\varphi}, h_\theta = \frac{b_\theta - a_\theta}{n_\theta}$$

in the spherical case.

Poisson Library uses standard five-point finite difference approximation on this mesh to compute the approximation to the solution. In the Cartesian case, the values of the approximate solution will be computed in the mesh points (x_i , y_j) provided that the user knows the values of the right-hand side `f(x, y)` in these points and the values of the appropriate boundary functions `G(x, y)` and/or `g(x,y)` in the mesh points laying on the boundary of the rectangular domain. In the spherical case, the values of the approximate solution will be computed in the mesh points (ϕ_i , θ_j) provided that the user knows the values of the right-hand side $f(\phi, \theta)$ in these points.

NOTE. The number of mesh intervals n_x in the x direction of a Cartesian mesh must be even. The number of mesh intervals n_ϕ in the ϕ direction of a spherical mesh must be even in the non-periodic case and divisible by four in the periodic case. Current implementation of the Poisson Library does not support meshes with the number of intervals that does not meet the above conditions.

Three-Dimensional Problems

Notational Conventions

PL interface description uses the following notation for boundaries of a parallelepiped domain $a_x < x < b_x$, $a_y < y < b_y$, $a_z < z < b_z$:

$bd\_a_x = \{x = a_x,\ a_y \le y \le b_y,\ a_z \le z \le b_z\}$, $bd\_b_x = \{x = b_x,\ a_y \le y \le b_y,\ a_z \le z \le b_z\}$

$bd\_a_y = \{a_x \le x \le b_x,\ y = a_y,\ a_z \le z \le b_z\}$, $bd\_b_y = \{a_x \le x \le b_x,\ y = b_y,\ a_z \le z \le b_z\}$

$bd\_a_z = \{a_x \le x \le b_x,\ a_y \le y \le b_y,\ z = a_z\}$, $bd\_b_x = \{a_x \le x \le b_x,\ a_y \le y \le b_y,\ z = b_z\}$.

The wildcard "+" may stand for any of the symbols a_x, b_x, a_y, b_y, a_z, b_z, so that bd_+ denotes any of the above boundaries.

Three-dimensional (3D) Helmholtz problem

3D Helmholtz problem is to find an approximate solution of Helmholtz equation

$$-\frac{\partial^2 u}{\partial x^2}-\frac{\partial^2 u}{\partial y^2}-\frac{\partial^2 u}{\partial z^2}+qu=f(x,y,z), q=const \ge 0$$

in a parallelepiped, that is, a parallelepiped domain $a_x < x < b_x$, $a_y < y < b_y$, $a_z < z < b_z$, with one of the following boundary conditions on each boundary bd_+:

- Dirichlet boundary condition

$$u(x,y,z)=G(x,y,z)$$

- Neumann boundary condition

$$\frac{\partial u}{\partial n}(x,y,z) = g(x,y,z)$$

where

n= -x on bd_a_x, n= x on bd_b_x,

n= -y on bd_a_y, n= y on bd_b_y,

n= -z on bd_a_z, n= z on bd_b_z.

Three-dimensional (3D) Poisson problem

The Poisson problem is a special case of the Helmholtz problem, when q=0. 3D Poisson problem is to find an approximate solution of Poisson equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - \frac{\partial^2 u}{\partial z^2} = f(x,y,z)$$

in a parallelepiped $a_x < x < b_x$, $a_y < y < b_y$, $a_z < z < b_z$ with Dirichlet or Neumann boundary condition on each boundary bd_+.

Three-dimensional (3D) Laplace problem

The Laplace problem is a special case of the Helmholtz problem, when q=0 and $f(x, y, z)$=0. 3D Laplace problem is to find an approximate solution of Laplace equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0$$

in a parallelepiped $a_x < x < b_x$, $a_y < y < b_y$, $a_z < z < b_z$ with Dirichlet or Neumann boundary condition on each boundary bd_+.

Approximation of 3D problems

To find an approximate solution for any of the 3D problems, a uniform mesh is built in the parallelepiped domain

$$\left\{x_i = a_x + ih_x, y_j = a_y + jh_y, z_k = a_z + kh_z\right\}$$

where

$$i = 0,...,n_x, j = 0,...,n_y, k = 0,...,n_z$$

$$h_x = \frac{b_x - a_x}{n_x}, h_y = \frac{b_y - a_y}{n_y}, h_z = \frac{b_z - a_z}{n_z}$$

Poisson Library uses standard seven-point finite difference approximation on this mesh to compute the approximation to the solution. The values of the approximate solution will be computed in the mesh points (x_i, y_j, z_k) provided that the user knows the values of the right-hand side `f(x, y, z)` in these points and the values of the appropriate boundary functions `G(x, y, z)` and/or `g(x, y, z)` in the mesh points laying on the boundary of the parallelepiped domain.

NOTE. The number of mesh intervals n_x and n_y in the x and y direction, respectively, must be even. Current implementation of Poisson Library does not support meshes with odd number of intervals.

Sequence of Invoking PL Routines

NOTE. Further description will always consider the solution process for Helmholtz problem, as Fast Poisson Solver and Fast Laplace Solver in Cartesian coordinates are special cases of Fast Helmholtz Solver (see Poisson Library Implemented).

Computation of a solution of Helmholtz problem using PL interface is conceptually divided into four steps each of which is performed via a dedicated routine. Table 13-5 lists names of the routines and briefly describes their purpose.

Most of PL routines have versions operating with single-precision and double-precision data. Names of such routines begin respectively with "s" and "d". The wildcard "?" stands for either of these symbols in routine names. The routines for Cartesian coordinate system have 2D and 3D versions. Their names end respectively in "2D" and "3D". The routines for spherical coordinate system have periodic and non-periodic versions. Their names end respectively in "p" and "np"

Table 13-5 PL Interface Routines

Routine	Description
?_init_Helmholtz_2D/?_init_Helmholtz_3D/ ?_init_sph_p/?_init_sph_np	Initializes basic data structures of Poisson Library for Fast Helmholtz Solver in 2D/3D/periodic/non-periodic case, respectively.
?_commit_Helmholtz_2D/?_commit_Helmholtz_3D/ ?_commit_sph_p/?_commit_sph_np	Checks consistency and correctness of user's data, creates and initializes data structures to be used by Intel MKL DFT interface[1] as well as other data structures needed for the solver.
?_Helmholtz_2D/?_Helmholtz_3D/ ?_sph_p/?_sph_np	Computes an approximate solution of 2D/3D/periodic/non-periodic Helmholtz problem (see Poisson Library Implemented) specified by parameters.
free_Helmholtz_2D/free_Helmholtz_3D/ free_sph_p/free_sph_np	Cleans the memory used by the data structures needed for calling Intel MKL DFT interface[1].

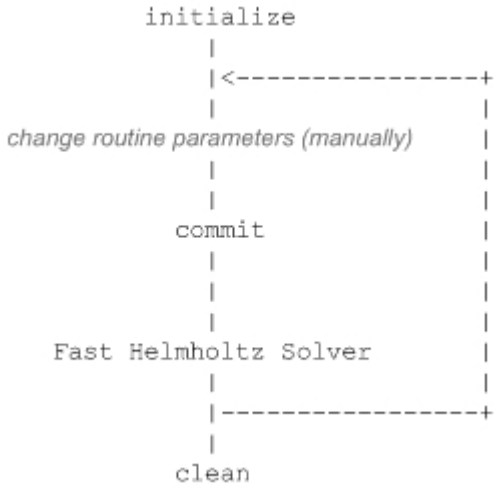
[1] PL routines call Intel MKL DFT interface for better performance.

To find once an approximate solution of Helmholtz problem, the Intel MKL PL interface routines are normally invoked in the order in which they are listed in Table 13-5.

NOTE. Though the order of invoking PL routines may be changed, it is highly recommended to follow the above order of routine calls.

The diagram in Figure 13-3 indicates the typical order in which PL routines can be invoked in a general case.

Figure 13-3 Typical Order of Invoking PL Routines



A general scheme of using PL routines for double-precision computations in a 3D Cartesian case is shown below. Similar scheme holds for single-precision computations with the only difference in the initial letter of routine names. The general scheme in a 2D Cartesian case differs form the one below in the set of routine parameters and the ending of routine names: "2D" instead of "3D".

```
...

d_init_Helmholtz_3D(&ax, &bx, &ay, &by, &az, &bz, &nx, &ny, &nz, BCtype,
ipar, dpar, &stat);

/* change parameters in ipar and/or dpar if necessary. */

/* note that the result of the Fast Helmholtz Solver will be in f! If you
want to keep the data stored in f,

save it before the function call below */

d_commit_Helmholtz_3D(f, bd_ax, bd_bx, bd_ay, bd_by, bd_az, bd_bz, &xhandle,
 &yhandle, ipar, dpar, &stat);

d_Helmholtz_3D(f, bd_ax, bd_bx, bd_ay, bd_by, bd_az, bd_bz, &xhandle,
&yhandle, ipar, dpar, &stat);

free_Helmholtz_3D (&xhandle, &yhandle, ipar, &stat);

/* here you may clean the memory used by f, dpar, ipar */

...
```

A general scheme of using PL routines for double-precision computations in a spherical periodic case is shown below. Similar scheme holds for single-precision computations with the only difference in the initial letter of routine names. The general scheme in a spherical non-periodic case differs from the one below in the set of routine parameters and the ending of routine names: "np" instead of "p".

```
...
d_init_sph_p(&ap,&bp,&at,&bt,&np,&nt,&q,ipar,dpar,&stat);
/* change parameters in ipar and/or dpar if necessary. */
/* note that the result of the Fast Helmholtz Solver will be in f! If you
want to
keep the data stored in f, save it before the function call below */
d_commit_sph_p(f,&handle_s,&handle_c,ipar,dpar,&stat);
d_sph_p(f,&handle_s,&handle_c,ipar,dpar,&stat);
free_sph_p(&handle_s,&handle_c,ipar,&stat);
/* here you may clean the memory used by f, dpar, ipar */
...
```

You can find examples of Fortran-90 and C code that use PL routines to solve Helmholtz problem (in both Cartesian and spherical cases) in the "Poisson Library Code Examples" section in Appendix C.

Interface Description

All types in this documentation are standard C types: `INT`, `FLOAT`, and `DOUBLE`. Fortran-90 users can call the routines with `INTEGER`, `REAL`, and `DOUBLE PRECISION` Fortran types, respectively (see examples in the "Poisson Library Code Examples" section in Appendix C).

Routine Options

All PL routines have parameters that are used for passing various options to the routines. These parameters are arrays *ipar*, *dpar* and *spar*. Values for these parameters should be specified very carefully (see Common Parameters). You can change these values during computations to meet your needs.

NOTE. You must provide correct and consistent parameters to the routines to avoid failure or wrong results.

User Data Arrays

PL routines take arrays of user data as input. For example, user arrays are passed to the routine `d_Helmholtz_3D` to compute an approximate solution to 3D Helmholtz problem. To minimize storage requirements and improve the overall run-time efficiency, Intel MKL PL routines do not make copies of user input arrays.

NOTE. If you need a copy of your input data arrays, you should save them yourself.

PL Routines for the Cartesian Solver

The section gives detailed description of Cartesian PL routines, their syntax, parameters and values they return. All flavors of the same routine, namely, double-precision and single-precision, 2D and 3D, are described together.

NOTE. Some of the routine parameters are used only in the 3D Fast Helmholtz Solver.

PL routines call Intel MKL DFT interface (described in section "DFT Functions" in chapter "Fast Fourier Transforms"), which enhances performance of the routines.

?_init_Helmholtz_2D/?_init_Helmholtz_3D

Initializes basic data structures of the Fast 2D/3D Helmholtz Solver.

Syntax

```
void d_init_Helmholtz_2D(double* ax, double* bx, double* ay, double* by,
int* nx, int* ny, char* BCtype, double* q, int* ipar, double* dpar, int*
stat);
```

```
void s_init_Helmholtz_2D(float* ax, float* bx, float* ay, float* by, int*
nx, int* ny, char* BCtype, float* q, int* ipar, float* spar, int* stat);
```

```
void d_init_Helmholtz_3D(double* ax, double* bx, double* ay, double* by,
double* az, double* bz, int* nx, int* ny, int* nz, char* BCtype, double* q,
int* ipar, double* dpar, int* stat);
```

```
void s_init_Helmholtz_3D(float* ax, float* bx, float* ay, float* by, float*
az, float* bz, int* nx, int* ny, int* nz, char* BCtype, float* q, int* ipar,
float* spar, int* stat);
```

Input Parameters

ax `double*` for `d_init_Helmholtz_2D/d_init_Helmholtz_3D`, `float*` for `s_init_Helmholtz_2D/s_init_Helmholtz_3D`. The coordinate of the leftmost boundary of the domain along x-axis.

bx `double*` for `d_init_Helmholtz_2D/d_init_Helmholtz_3D`, `float*` for `s_init_Helmholtz_2D/s_init_Helmholtz_3D`. The coordinate of the rightmost boundary of the domain along x-axis.

ay `double*` for `d_init_Helmholtz_2D/d_init_Helmholtz_3D`, `float*` for `s_init_Helmholtz_2D/s_init_Helmholtz_3D`. The coordinate of the leftmost boundary of the domain along y-axis.

by `double*` for `d_init_Helmholtz_2D/d_init_Helmholtz_3D`, `float*` for `s_init_Helmholtz_2D/s_init_Helmholtz_3D`. The coordinate of the rightmost boundary of the domain along y-axis.

az `double*` for `d_init_Helmholtz_2D/d_init_Helmholtz_3D`, `float*` for `s_init_Helmholtz_2D/s_init_Helmholtz_3D`. The coordinate of the leftmost boundary of the domain along z-axis. This parameter is needed only for the `?_init_Helmholtz_3D` routine.

bz `double*` for `d_init_Helmholtz_2D/d_init_Helmholtz_3D`, `float*` for `s_init_Helmholtz_2D/s_init_Helmholtz_3D`. The coordinate of the rightmost boundary of the domain along z-axis. This parameter is needed only for the `?_init_Helmholtz_3D` routine.

nx	`int*`. The number of mesh intervals along x-axis. Must be even.
ny	`int*`. The number of mesh intervals along y-axis. Must be even in the 3D case.
nz	`int*`. The number of mesh intervals along z-axis. This parameter is needed only for the `?_init_Helmholtz_3D` routine.
BCtype	`char*`. Contains the type of boundary conditions on each boundary. Must contain four characters for `?_init_Helmholtz_2D` and six characters for `?_init_Helmholtz_3D`. Each of the characters can be 'N' (Neumann boundary condition) or 'D' (Dirichlet boundary condition). Types of boundary conditions for the boundaries should be specified in the following order: bd_a_x, bd_b_x, bd_a_y, bd_b_y, bd_a_z, bd_b_z. Boundary condition types for the last two boundaries should be specified only in the 3D case.
q	`double*` for `d_init_Helmholtz_2D/d_init_Helmholtz_3D`, `float*` for `s_init_Helmholtz_2D/s_init_Helmholtz_3D`. The constant Helmholtz coefficient. Note that to solve Poisson or Laplace problem, you should set the value of *q* to 0.

Output Parameters

ipar	`int` array of size 128. Contains integer data to be used by Fast Helmholtz Solver (for details, refer to Common Parameters).
dpar	`double` array of size 5*nx/2+7 in the 2D case or 5*(nx+ny)/2+9 in the 3D case. Contains double-precision data to be used by Fast Helmholtz Solver (for details, refer to Common Parameters).
spar	`float` array of size 5*nx/2+7 in the 2D case or 5*(nx+ny)/2+9 in the 3D case. Contains single-precision data to be used by Fast Helmholtz Solver (for details, refer to Common Parameters).

stat	int*. Routine completion status, which is also written to *ipar[0]*. The status should be 0 to proceed to other PL routines.

Description

The routines ?_init_Helmholtz_2D/?_init_Helmholtz_3D are called to initialize basic data structures for Poisson Library computations of the appropriate precision. All routines invoked after a call to a ?_init_Helmholtz_2D/?_init_Helmholtz_3D routine use values of the *ipar*, *dpar* and *spar* array parameters returned by the routine. Detailed description of the array parameters can be found in Common Parameters.

WARNING. Data structures initialized and created by 2D/3D flavors of the routine cannot be used by 3D/2D flavors of any PL routines, respectively.

You can skip calling this routine in your code. However, see Caveat on Parameter Modifications before doing so.

Return Values

stat= 0	The routine successfully completed the task. In general, to proceed with computations, the routine should complete with this *stat* value.
stat= -99999	The routine failed to complete the task because of fatal error.

?_commit_Helmholtz_2D/?_commit_Helmholtz_3D

Checks consistency and correctness of user's data as well as initializes certain data structures required to solve 2D/3D Helmholtz problem.

Syntax

```
void d_commit_Helmholtz_2D(double* f, double* bd_ax, double* bd_bx, double* bd_ay, double* bd_by, DFTI_DESCIPTOR* xhandle, int* ipar, double* dpar, int* stat);
```

```
void s_commit_Helmholtz_2D (float* f, float* bd_ax, float* bd_bx, float*
bd_ay, float* bd_by, DFTI_DESCIPTOR* xhandle, int* ipar, float* spar, int*
stat);
```

```
void d_commit_Helmholtz_3D(double* f, double* bd_ax, double* bd_bx, double*
bd_ay, double* bd_by, double* bd_az, double* bd_bz, DFTI_DESCIPTOR* xhandle,
DFTI_DESCIPTOR* yhandle, int* ipar, double* dpar, int* stat);
```

```
void s_commit_Helmholtz_3D(float* f, float* bd_ax, float* bd_bx, float*
bd_ay, float* bd_by, float* bd_az, float* bd_bz, DFTI_DESCIPTOR* xhandle,
DFTI_DESCIPTOR* yhandle, int* ipar, float* spar, int* stat);
```

Input Parameters

f	`double*` for `d_commit_Helmholtz_2D/d_commit_Helmholtz_3D`, `float*` for `s_commit_Helmholtz_2D/s_commit_Helmholtz_3D`. Contains the right-hand side of the problem packed in a single vector. The size of the vector in the 2D case is (*nx*+1)*(*ny*+1). In this case, value of the right-hand side in the mesh point (*i, j*) is stored in *f*[*i*+*j**(*nx*+1)] . The size of the vector in the 3D case is (*nx*+1)*(*ny*+1)*(*nz*+1). In this case, value of the right-hand side in the mesh point (*i, j, k*) is stored in *f*[*i*+*j**(*nx*+1)+*k**(*nx*+1)*(*ny*+1)]. Note that to solve Laplace problem, you should set all the elements of the array *f* to 0. Note that the array *f* may be altered by the routine. Please save this vector in another memory location if you want to preserve it.
ipar	`int` array of size 128. Contains integer data to be used by Fast Helmholtz Solver (for details, refer to Common Parameters).
dpar	`double` array of size 5*nx/2+7 in the 2D case or 5*(*nx*+*ny*)/2+9 in the 3D case. Contains double-precision data to be used by Fast Helmholtz Solver (for details, refer to Common Parameters).

spar — `float` array of size 5*nx/2+7 in the 2D case or 5*(nx+ny)/2+9 in the 3D case. Contains single-precision data to be used by Fast Helmholtz Solver (for details, refer to Common Parameters).

bd_ax — `double*` for `d_commit_Helmholtz_2D/d_commit_Helmholtz_3D`, `float*` for `s_commit_Helmholtz_2D/s_commit_Helmholtz_3D`. Contains values of the boundary condition on the leftmost boundary of the domain along x-axis.
For `?_commit_Helmholtz_2D`, the size of the array is ny+1. In case of Dirichlet boundary condition (value of *BCtype*[0] is 'D'), it contains values of the function $G(ax, y_j)$, j=0, ..., ny. In case of Neumann boundary condition (value of *BCtype*[0] is 'N'), it contains values of the function $g(ax, y_j)$, j=0, ..., ny. The value corresponding to the index j is placed in *bd_ax*[j].
For `?_commit_Helmholtz_3D`, the size of the array is (ny+1)*(nz+1). In case of Dirichlet boundary condition (value of *BCtype*[0] is 'D'), it contains values of the function $G(ax, y_j, z_k)$, j=0, ..., ny, k=0, ..., nz. In case of Neumann boundary condition (value of *BCtype*[0] is 'N'), it contains the values of the function $g(ax, y_j, z_k)$, j=0, ..., ny, k=0, ..., nz. The values are packed in the array so that the value corresponding to indices (j, k) is placed in *bd_ax*[j+k*(ny+1)].

bd_bx — `double*` for `d_commit_Helmholtz_2D/d_commit_Helmholtz_3D`, `float*` for `s_commit_Helmholtz_2D/s_commit_Helmholtz_3D`. Contains values of the boundary condition on the rightmost boundary of the domain along x-axis.
For `?_commit_Helmholtz_2D`, the size of the array is ny+1. In case of Dirichlet boundary condition (value of *BCtype*[1] is 'D'), it contains values of the function $G(bx, y_j)$, j=0, ..., ny. In case of Neumann boundary condition (value of

	BCtype[1] is 'N'), it contains values of the function *g*(*bx*, y_j), *j*=0, ..., *ny*. The value corresponding to the index *j* is placed in *bd_bx*[*j*]. For ?_commit_Helmholtz_3D, the size of the array is (*ny*+1)*(*nz*+1). In case of Dirichlet boundary condition (value of *BCtype*[1] is 'D'), it contains values of the function *G*(*bx*, y_j, z_k), *j*=0, ..., *ny*, *k*=0, ..., *nz*. In case of Neumann boundary condition (value of *BCtype*[1] is 'N'), it contains the values of the function *g*(*bx*, y_j, z_k), *j*=0, ..., *ny*, *k*=0, ..., *nz*. The values are packed in the array so that the value corresponding to indices *(j, k)* is placed in *bd_bx*[*j*+*k**(*ny*+1)].
bd_ay	double* for d_commit_Helmholtz_2D/d_commit_Helmholtz_3D, float* for s_commit_Helmholtz_2D/s_commit_Helmholtz_3D. Contains values of the boundary condition on the leftmost boundary of the domain along *y*-axis. For ?_commit_Helmholtz_2D, the size of the array is *nx*+1. In case of Dirichlet boundary condition (value of *BCtype*[2] is 'D'), it contains values of the function *G*(x_i, *ay*), *i*=0, ..., *nx*. In case of Neumann boundary condition (value of *BCtype*[2] is 'N'), it contains values of the function *g*(x_i, *ay*), *i*=0, ..., *nx*. The value corresponding to the index *i* is placed in *bd_ay*[*i*]. For ?_commit_Helmholtz_3D, the size of the array is (*nx*+1)*(*nz*+1). In case of Dirichlet boundary condition (value of *BCtype*[2] is 'D'), it contains values of the function *G*(x_i,*ay*, z_k), *i*=0, ..., *nx*, *k*=0, ..., *nz*. In case of Neumann boundary condition (value of *BCtype*[2] is 'N'), it contains the values of the function *g*(x_i,*ay*, z_k), *i*=0, ..., *nx*, *k*=0, ..., *nz*. The values are packed in the array so that the value corresponding to indices *(i, k)* is placed in *bd_ay*[*i*+*k**(*nx*+1)].
bd_by	double* for d_commit_Helmholtz_2D/d_commit_Helmholtz_3D,

	`float*` for `s_commit_Helmholtz_2D/s_commit_Helmholtz_3D`. Contains values of the boundary condition on the rightmost boundary of the domain along y-axis. For `?_commit_Helmholtz_2D`, the size of the array is $nx+1$. In case of Dirichlet boundary condition (value of $BCtype[3]$ is 'D'), it contains values of the function $G(x_i, by)$, $i=0, ..., nx$. In case of Neumann boundary condition (value of $BCtype[3]$ is 'N'), it contains values of the function $g(x_i, by)$, $i=0, ..., nx$. The value corresponding to the index i is placed in $bd\_by[i]$. For `?_commit_Helmholtz_3D`, the size of the array is $(nx+1)*(nz+1)$. In case of Dirichlet boundary condition (value of $BCtype[3]$ is 'D'), it contains values of the function $G(x_i, by, z_k)$, $i=0, ..., nx$, $k=0, ..., nz$. In case of Neumann boundary condition (value of $BCtype[3]$ is 'N'), it contains the values of the function $g(x_i, by, z_k)$, $i=0, ..., nx$, $k=0, ..., nz$. The values are packed in the array so that the value corresponding to indices (i, k) is placed in $bd\_by[i+k*(nx+1)]$.
bd_az	`double*` for `d_commit_Helmholtz_2D/d_commit_Helmholtz_3D`, `float*` for `s_commit_Helmholtz_2D/s_commit_Helmholtz_3D`. This parameter is needed only for `?_commit_Helmholtz_3D`. Contains values of the boundary condition on the leftmost boundary of the domain along z-axis. The size of the array is $(nx+1)*(ny+1)$. In case of Dirichlet boundary condition (value of $BCtype[4]$ is 'D'), it contains values of the function $G(x_i, y_j, az)$, $i=0, ..., nx$, $j=0, ..., ny$. In case of Neumann boundary condition (value of $BCtype[4]$ is 'N'), it contains the values of the function $g(x_i, y_j, az)$, $i=0, ..., nx$, $j=0, ..., ny$. The values are packed in the array so that the value corresponding to indices (i, j) is placed in $bd\_az[i+j*(nx+1)]$.
bd_bz	`double*` for `d_commit_Helmholtz_2D/d_commit_Helmholtz_3D`,

	`float*` for `s_commit_Helmholtz_2D/s_commit_Helmholtz_3D`. This parameter is needed only for `?_commit_Helmholtz_3D`. Contains values of the boundary condition on the rightmost boundary of the domain along z-axis. The size of the array is $(nx+1)*(ny+1)$. In case of Dirichlet boundary condition (value of *BCtype*[5] is 'D'), it contains values of the function $G(x_i, y_j, bz)$, $i=0, ..., nx$, $j=0, ..., ny$. In case of Neumann boundary condition (value of *BCtype*[5] is 'N'), it contains the values of the function $g(x_i, y_j, bz)$, $i=0, ..., nx$, $j=0, ..., ny$. The values are packed in the array so that the value corresponding to indices *(i, j)* is placed in *bd_bz*[$i+j*(nx+1)$].

Output Parameters

f	Vector of the right-hand side of the problem. Possibly, altered on output.
ipar	Contains integer data to be used by Fast Helmholtz Solver. Modified on output as explained in Common Parameters.
dpar	Contains double-precision data to be used by Fast Helmholtz Solver. Modified on output as explained in Common Parameters.
spar	Contains single-precision data to be used by Fast Helmholtz Solver. Modified on output as explained in Common Parameters.
xhandle, yhandle	`DESCIPTOR_HANDLE*`. Data structures used by Intel MKL DFT interface (for details, refer to section "DFT Functions" in chapter "Fast Fourier Transforms"). *yhandle* is used only by `?_commit_Helmholtz_3D`.
stat	`int*`. Routine completion status, which is also written to *ipar[0]*. The status should be 0 to proceed to other PL routines.

Description

The routines `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` check consistency and correctness of the parameters to be passed to the solver routines `?_Helmholtz_2D/?_Helmholtz_3D`. They also initialize data structures *xhandle*, *yhandle* as

well as arrays *ipar* and *dpar/spar*, depending upon the routine precision. Refer to Common Parameters to find out which particular array elements the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routines initialize and what values are written there. The routines perform only a basic check for correctness and consistency. If you are going to modify parameters of PL routines, see the Caveat on Parameter Modifications section. Unlike ?_init_Helmholtz_2D/?_init_Helmholtz_3D, you cannot skip calling these routines in your code. Values of *ax*, *bx*, *ay*, *by*, *az*, *bz*, *nx*, *ny*, *nz*, and *BCtype* are passed to each of the routines with the *ipar* array and defined in a previous call to the appropriate ?_init_Helmholtz_2D/?_init_Helmholtz_3D routine.

Return Values

stat= 1	The routine completed without errors and produced some warnings.
stat= 0	The routine successfully completed the task.
stat= -100	The routine stopped as an error in the user's data was found or the data in the *dpar*, *spar* or *ipar* array was altered by mistake.
stat= -1000	The routine stopped because of Intel MKL DFT or TT interface error.
stat= -10000	The routine stopped as the initialization failed to complete or parameter *ipar*[0] was altered by mistake.
stat= -99999	The routine failed to complete the task because of fatal error.

?_Helmholtz_2D/?_Helmholtz_3D

Computes the solution of 2D/3D Helmholtz problem specified by the parameters.

Syntax

```
void d_Helmholtz_2D(double* f, double* bd_ax, double* bd_bx, double* bd_ay,
double* bd_by, DFTI_DESCIPTOR* xhandle, int* ipar, double* dpar, int* stat);
```

```
void s_Helmholtz_2D (float* f, float* bd_ax, float* bd_bx, float* bd_ay,
float* bd_by, DFTI_DESCIPTOR* xhandle, int* ipar, float* spar, int* stat);
```

```
void d_Helmholtz_3D(double* f, double* bd_ax, double* bd_bx, double* bd_ay,
double* bd_by, double* bd_az, double* bd_bz, DFTI_DESCIPTOR* xhandle,
DFTI_DESCIPTOR* yhandle, int* ipar, double* dpar, int* stat);
```

```
void s_Helmholtz_3D(float* f, float* bd_ax, float* bd_bx, float* bd_ay,
float* bd_by, float* bd_az, float* bd_bz, DFTI_DESCIPTOR* xhandle,
DFTI_DESCIPTOR* yhandle, int* ipar, float* spar, int* stat);
```

Input Parameters

f	`double*` for `d_Helmholtz_2D/d_Helmholtz_3D`, `float*` for `s_Helmholtz_2D/s_Helmholtz_3D`. Contains the right-hand side of the problem packed in a single vector and modified by the appropriate `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routine. Note that an attempt to substitute the original right-hand side vector at this point will result in a wrong solution. The size of the vector in the 2D case is (*nx*+1)*(*ny*+1). In this case, value of the right-hand side in the mesh point (*i*, *j*) is stored in *f*[*i*+*j**(*nx*+1)] . The size of the vector in the 3D case is (*nx*+1)*(*ny*+1)*(*nz*+1). In this case, value of the right-hand side in the mesh point (*i*, *j*, *k*) is stored in *f*[*i*+*j**(*nx*+1)+*k**(*nx*+1)*(*ny*+1)].
xhandle, yhandle	`DESCIPTOR_HANDLE*`. Data structures used by Intel MKL DFT interface (for details, refer to section "DFT Functions" in chapter "Fast Fourier Transforms"). *yhandle* is used only by `?_Helmholtz_3D`.
ipar	`int` array of size 128. Contains integer data to be used by Fast Helmholtz Solver (for details, refer to Common Parameters).
dpar	`double` array of size 5**nx*/2+7 in the 2D case or 5*(*nx*+*ny*)/2+9 in the 3D case. Contains double-precision data to be used by Fast Helmholtz Solver (for details, refer to Common Parameters).
spar	float array of size 5**nx*/2+7 in the 2D case or 5*(*nx*+*ny*)/2+9 in the 3D case. Contains single-precision data to be used by Fast Helmholtz Solver (for details, refer to Common Parameters).

bd_ax — `double*` for `d_Helmholtz_2D/d_Helmholtz_3D`, `float*` for `s_Helmholtz_2D/s_Helmholtz_3D`. Contains values of the boundary condition on the leftmost boundary of the domain along *x*-axis.
For `?_Helmholtz_2D`, the size of the array is *ny*+1. In case of Dirichlet boundary condition (value of *BCtype*[0] is 'D'), it contains values of the function $G(ax, y_j)$, $j=0, ..., ny$. In case of Neumann boundary condition (value of *BCtype*[0] is 'N'), it contains values of the function $g(ax, y_j)$, $j=0, ..., ny$. The value corresponding to the index *j* is placed in *bd_ax*[*j*].
For `?_Helmholtz_3D`, the size of the array is (*ny*+1)*(*nz*+1). In case of Dirichlet boundary condition (value of *BCtype*[0] is 'D'), it contains values of the function $G(ax, y_j, z_k)$, $j=0, ..., ny$, $k=0, ..., nz$. In case of Neumann boundary condition (value of *BCtype*[0] is 'N'), it contains the values of the function $g(ax, y_j, z_k)$, $j=0, ..., ny$, $k=0, ..., nz$. The values are packed in the array so that the value corresponding to indices *(j, k)* is placed in *bd_ax*[*j*+*k**(*ny*+1)].

bd_bx — `double*` for `d_Helmholtz_2D/d_Helmholtz_3D`, `float*` for `s_Helmholtz_2D/s_Helmholtz_3D`. Contains values of the boundary condition on the rightmost boundary of the domain along *x*-axis.
For `?_Helmholtz_2D`, the size of the array is *ny*+1. In case of Dirichlet boundary condition (value of *BCtype*[1] is 'D'), it contains values of the function $G(bx, y_j)$, $j=0, ..., ny$. In case of Neumann boundary condition (value of *BCtype*[1] is 'N'), it contains values of the function $g(bx, y_j)$, $j=0, ..., ny$. The value corresponding to the index *j* is placed in *bd_bx*[*j*].
For `?_Helmholtz_3D`, the size of the array is (*ny*+1)*(*nz*+1). In case of Dirichlet boundary condition (value of *BCtype*[1] is 'D'), it contains values of the function $G(bx, y_j, z_k)$, $j=0, ..., ny$, $k=0, ..., nz$. In case of Neumann boundary condition (value of *BCtype*[1] is 'N'), it contains the values of the function $g(bx, y_j, z_k)$, $j=0, ..., ny$, $k=0$,

	..., *nz*. The values are packed in the array so that the value corresponding to indices *(j, k)* is placed in *bd_bx*[*j*+*k**(*ny*+1)].
bd_ay	double* for d_Helmholtz_2D/d_Helmholtz_3D, float* for s_Helmholtz_2D/s_Helmholtz_3D. Contains values of the boundary condition on the leftmost boundary of the domain along *y*-axis. For ?_Helmholtz_2D, the size of the array is *nx*+1. In case of Dirichlet boundary condition (value of *BCtype*[2] is 'D'), it contains values of the function $G(x_i, ay)$, *i*=0, ..., *nx*. In case of Neumann boundary condition (value of *BCtype*[2] is 'N'), it contains values of the function $g(x_i, ay)$, *i*=0, ..., *nx*. The value corresponding to the index *i* is placed in *bd_ay*[*i*]. For ?_Helmholtz_3D, the size of the array is (*nx*+1)*(*nz*+1). In case of Dirichlet boundary condition (value of *BCtype*[2] is 'D'), it contains values of the function $G(x_i, ay, z_k)$, *i*=0, ..., *nx*, *k*=0, ..., *nz*. In case of Neumann boundary condition (value of *BCtype*[2] is 'N'), it contains the values of the function $g(x_i, ay, z_k)$, *i*=0, ..., *nx*, *k*=0, ..., *nz*. The values are packed in the array so that the value corresponding to indices *(i, k)* is placed in *bd_ay*[*i*+*k**(*nx*+1)].
bd_by	double* for d_Helmholtz_2D/d_Helmholtz_3D, float* for s_Helmholtz_2D/s_Helmholtz_3D. Contains values of the boundary condition on the rightmost boundary of the domain along *y*-axis. For ?_Helmholtz_2D, the size of the array is *nx*+1. In case of Dirichlet boundary condition (value of *BCtype*[3] is 'D'), it contains values of the function $G(x_i, by)$, *i*=0, ..., *nx*. In case of Neumann boundary condition (value of *BCtype*[3] is 'N'), it contains values of the function $g(x_i, by)$, *i*=0, ..., *nx*. The value corresponding to the index *i* is placed in *bd_by*[*i*]. For ?_Helmholtz_3D, the size of the array is (*nx*+1)*(*nz*+1). In case of Dirichlet boundary condition (value of *BCtype*[3] is 'D'), it contains values of the function

$G(x_i, by, z_k)$, i=0, ..., nx, k=0, ..., nz. In case of Neumann boundary condition (value of *BCtype*[3] is 'N'), it contains the values of the function $g(x_i, by, z_k)$, i=0, ..., nx, k=0, ..., nz. The values are packed in the array so that the value corresponding to indices *(i, k)* is placed in *bd_by*[i+k*(nx+1)].

bd_az — `double*` for `d_Helmholtz_2D/d_Helmholtz_3D`, `float*` for `s_Helmholtz_2D/s_Helmholtz_3D`.
This parameter is needed only for `?_Helmholtz_3D`.
Contains values of the boundary condition on the leftmost boundary of the domain along z-axis.
The size of the array is (nx+1)*(ny+1). In case of Dirichlet boundary condition (value of *BCtype*[4] is 'D'), it contains values of the function $G(x_i, y_j, az)$, i=0, ..., nx, j=0, ..., ny. In case of Neumann boundary condition (value of *BCtype*[4] is 'N'), it contains the values of the function $g(x_i, y_j, az)$, i=0, ..., nx, j=0, ..., ny. The values are packed in the array so that the value corresponding to indices *(i, j)* is placed in *bd_az*[i+j*(nx+1)].

bd_bz — `double*` for `d_Helmholtz_2D/d_Helmholtz_3D`, `float*` for `s_Helmholtz_2D/s_Helmholtz_3D`.
This parameter is needed only for `?_Helmholtz_3D`.
Contains values of the boundary condition on the rightmost boundary of the domain along z-axis.
The size of the array is (nx+1)*(ny+1). In case of Dirichlet boundary condition (value of *BCtype*[5] is 'D'), it contains values of the function $G(x_i, y_j, bz)$, i=0, ..., nx, j=0, ..., ny. In case of Neumann boundary condition (value of *BCtype*[5] is 'N'), it contains the values of the function $g(x_i, y_j, bz)$, i=0, ..., nx, j=0, ..., ny. The values are packed in the array so that the value corresponding to indices *(i, j)* is placed in *bd_bz*[i+j*(nx+1)].

NOTE. To avoid wrong computation result, you should not change arrays *bd_ax*, *bd_bx*, *bd_ay*, *bd_by*, *bd_az*, *bd_bz* between a call to the `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routine and a subsequent call to the appropriate `?_Helmholtz_2D/?_Helmholtz_3D` routine.

Output Parameters

f	On output, contains the approximate solution to the problem packed the same way as the right-hand side of the problem was packed on input.
xhandle, yhandle	Data structures used by Intel MKL DFT interface.
ipar	Contains integer data to be used by Fast Helmholtz Solver. Modified on output as explained in Common Parameters.
dpar	Contains double-precision data to be used by Fast Helmholtz Solver. Modified on output as explained in Common Parameters.
spar	Contains single-precision data to be used by Fast Helmholtz Solver. Modified on output as explained in Common Parameters.
stat	`int*`. Routine completion status, which is also written to *ipar[0]*. The status should be 0 to proceed to other PL routines.

Description

The routines compute the approximate solution of Helmholtz problem defined in the previous calls to the corresponding initialization and commit routines. The solution is computed according to formulas given in the Poisson Library Implemented section. The *f* parameter, which initially holds the packed vector of the right-hand side of the problem, is replaced by the computed solution packed in the same way. Values of *ax*, *bx*, *ay*, *by*, *az*, *bz*, *nx*, *ny*, *nz*, and *BCtype* are passed to each of the routines with the *ipar* array and defined in the previous call to the appropriate `?_init_Helmholtz_2D/?_init_Helmholtz_3D` routine.

Return Values

stat= 1	The routine completed without errors and produced some warnings.
stat= 0	The routine successfully completed the task.
stat= -2	The routine stopped as division by zero occurred. It usually happens if the data in the *dpar* or *spar* array was altered by mistake.
stat= -3	The routine stopped as memory was insufficient to complete the computations.

stat= -100	The routine stopped as an error in the user's data was found or the data in the *dpar*, *spar* or *ipar* array was altered by mistake.
stat= -1000	The routine stopped because of Intel MKL DFT or TT interface error.
stat= -10000	The routine stopped as the initialization failed to complete or parameter *ipar*[0] was altered by mistake.
stat= -99999	The routine failed to complete the task because of fatal error.

free_Helmholtz_2D/free_Helmholtz_3D

Cleans the memory allocated for the data structures used by DFT interface.

Syntax

```
void free_Helmholtz_2D(DFTI_DESCIPTOR* xhandle, int* ipar, int* stat);
void free_Helmholtz_3D (DFTI_DESCIPTOR* xhandle, DFTI_DESCIPTOR* yhandle,
int* ipar, int* stat);
```

Input Parameters

xhandle, yhandle	DESCIPTOR_HANDLE*. Data structures used by Intel MKL DFT interface (for details, refer to section "DFT Functions" in chapter "Fast Fourier Transforms"). *yhandle* is used only by free_Helmholtz_3D.
ipar	int array of size 128. Contains integer data to be used by Fast Helmholtz Solver (for details, refer to Common Parameters).

Output Parameters

xhandle, yhandle	Data structures used by Intel MKL DFT interface. Memory allocated for the structures is released on output.
ipar	Contains integer data to be used by Fast Helmholtz Solver. Status of the routine call is written to *ipar[0]*.

stat	int*. Routine completion status, which is also written to *ipar[0]*.

Description

The routine cleans the memory used by the *xhandle* and *yhandle* structures, needed for calling Intel MKL DFT functions. If you need to release memory allocated for other parameters, you should include the memory cleaning in your code.

Return Values

stat= 0	The routine successfully completed the task.
stat= -1000	The routine stopped because of Intel MKL DFT or TT interface error.
stat= -99999	The routine failed to complete the task because of fatal error.

PL Routines for the Spherical Solver

The section gives detailed description of spherical PL routines, their syntax, parameters and values they return. All flavors of the same routine, namely, double-precision and single-precision, periodic (having names ending in "p") and non-periodic (having names ending in "np"), are described together.

These PL routines also call Intel MKL DFT interface (described in section "DFT Functions" in chapter "Fast Fourier Transforms"), which enhances performance of the routines.

?_init_sph_p/?_init_sph_np

Initializes basic data structures of the Fast periodic and non-periodic Helmholtz Solver on a sphere.

Syntax

```
void d_init_sph_p(double* ap, double* at, double* bp, double* bt, int* np,
int* nt, double* q, int* ipar, double* dpar, int* stat);
```

```
void s_init_sph_np(float* ap, float* at, float* bp, float* bt, int* np, int*
nt, float* q, int* ipar, float* spar, int* stat);
```

```
void d_init_sph_np(double* ap, double* at, double* bp, double* bt, int* np,
int* nt, double* q, int* ipar, double* dpar, int* stat);
```

```
void s_init_sph_np(float* ap, float* at, float* bp, float* bt, int* np, int*
nt, float* q, int* ipar, float* spar, int* stat);
```

Input Parameters

ap	`double*` for `d_init_sph_p/d_init_sph_np`, `float*` for `s_init_sph_p/s_init_sph_np`. The coordinate (angle) of the leftmost boundary of the domain along ϕ-axis.
bp	`double*` for `d_init_sph_p/d_init_sph_np`, `float*` for `s_init_sph_p/s_init_sph_np`. The coordinate (angle) of the rightmost boundary of the domain along ϕ-axis.
at	`double*` for `d_init_sph_p/d_init_sph_np`, `float*` for `s_init_sph_p/s_init_sph_np`. The coordinate (angle) of the leftmost boundary of the domain along θ-axis.
bt	`double*` for `d_init_sph_p/d_init_sph_np`, `float*` for `s_init_sph_p/s_init_sph_np`. The coordinate (angle) of the rightmost boundary of the domain along θ-axis.
np	`int*`. The number of mesh intervals along ϕ-axis. Must be even in the non-periodic case and divisible by 4 in the periodic case.
nt	`int*`. The number of mesh intervals along θ-axis.
q	`double*` for `d_init_sph_p/d_init_sph_np`, `float*` for `s_init_sph_p/s_init_sph_np`. The constant Helmholtz coefficient. Note that to solve Poisson problem, you should set the value of q to 0.

Output Parameters

ipar	`int` array of size 128. Contains integer data to be used by Fast Helmholtz Solver on a sphere (for details, refer to Common Parameters).

dpar	`double` array of size 5*np/2+nt+10. Contains double-precision data to be used by Fast Helmholtz Solver on a sphere (for details, refer to Common Parameters).
spar	`float` array of size 5*np/2+nt+10. Contains single-precision data to be used by Fast Helmholtz Solver on a sphere (for details, refer to Common Parameters).
stat	`int*`. Routine completion status, which is also written to *ipar[0]*. The status should be 0 to proceed to other PL routines.

Description

The routines `?_init_sph_p/?_init_sph_np` are called to initialize basic data structures for Poisson Library computations of the appropriate precision. All routines invoked after a call to a `?_init_Helmholtz_2D/?_init_Helmholtz_3D` routine use values of the *ipar*, *dpar* and *spar* array parameters returned by the routine. Detailed description of the array parameters can be found in Common Parameters.

WARNING. Data structures initialized and created by periodic/non-periodic flavors of the routine cannot be used by non-periodic/periodic flavors of any PL routines for Helmholtz Solver on a sphere, respectively.

You can skip calling this routine in your code. However, see Caveat on Parameter Modifications before doing so.

Return Values

stat= 0	The routine successfully completed the task. In general, to proceed with computations, the routine should complete with this *stat* value.
stat= -99999	The routine failed to complete the task because of fatal error.

?_commit_sph_p/?_commit_sph_np

Checks consistency and correctness of user's data as well as initializes certain data structures required to solve periodic/non-periodic Helmholtz problem on a sphere.

Syntax

```
void d_commit_sph_p(double* f, DFTI_DESCIPTOR* handle_s, DFTI_DESCIPTOR* handle_c, int* ipar, double* dpar, int* stat);
```

```
void s_commit_sph_p(float* f, DFTI_DESCIPTOR* handle_s, DFTI_DESCIPTOR* handle_c, int* ipar, float* spar, int* stat);
```

```
void d_commit_sph_np(double* f, DFTI_DESCIPTOR* handle, int* ipar, double* dpar, int* stat);
```

```
void s_commit_sph_np(float* f, DFTI_DESCIPTOR* handle, int* ipar, float* spar, int* stat);
```

Input Parameters

f	`double*` for `d_commit_sph_p/d_commit_sph_np`, `float*` for `s_commit_sph_p/s_commit_sph_np`. Contains the right-hand side of the problem packed in a single vector. The size of the vector is (*np*+1)*(*nt*+1) and value of the right-hand side in the mesh point (*i, j*) is stored in *f*[*i+j*(*np*+1)] . Note that the array *f* may be altered by the routine. Please save this vector in another memory location if you want to preserve it.
ipar	`int` array of size 128. Contains integer data to be used by Fast Helmholtz Solver on a sphere (for details, refer to Common Parameters).
dpar	`double` array of size 5**np*/2+*nt*+10. Contains double-precision data to be used by Fast Helmholtz Solver on a sphere (for details, refer to Common Parameters).
spar	`float` array of size 5**np*/2+*nt*+10. Contains single-precision data to be used by Fast Helmholtz Solver on a sphere (for details, refer to Common Parameters).

Output Parameters

f	Vector of the right-hand side of the problem. Possibly, altered on output.
ipar	Contains integer data to be used by Fast Helmholtz Solver on a sphere. Modified on output as explained in Common Parameters.
dpar	Contains double-precision data to be used by Fast Helmholtz Solver on a sphere. Modified on output as explained in Common Parameters.
spar	Contains single-precision data to be used by Fast Helmholtz Solver on a sphere. Modified on output as explained in Common Parameters.
handle_s, *handle_c*, *handle*	DESCIPTOR_HANDLE*. Data structures used by Intel MKL DFT interface (for details, refer to section "DFT Functions" in chapter "Fast Fourier Transforms"). *handle_s* and *handle_c* are used only in ?_commit_sph_p and *handle* is used only in ?_commit_sph_np.
stat	int*. Routine completion status, which is also written to *ipar[0]*. The status should be 0 to proceed to other PL routines.

Description

The routines ?_commit_sph_p/?_commit_sph_np check consistency and correctness of the parameters to be passed to the solver routines ?_sph_p/?_sph_np, respectively. They also initialize certain data structures. ?_commit_sph_p initializes structures *handle_s* and *handle_c* and ?_commit_sph_np initializes *handle*. The routines also initialize arrays *ipar* and *dpar/spar*, depending upon the routine precision. Refer to Common Parameters to find out which particular array elements the ?_commit_sph_p/?_commit_sph_np routines initialize and what values are written there. The routines perform only a basic check for correctness and consistency. If you are going to modify parameters of PL routines, see the Caveat on Parameter Modifications section. Unlike ?_init_sph_p/?_init_sph_np, you cannot skip calling these routines in your code. Values of *np* and *nt* are passed to each of the routines with the *ipar* array and defined in a previous call to the appropriate ?_init_sph_p/?_init_sph_np routine.

Return Values

stat= 1	The routine completed without errors and produced some warnings.
stat= 0	The routine successfully completed the task.
stat= -100	The routine stopped as an error in the user's data was found or the data in the *dpar*, *spar* or *ipar* array was altered by mistake.
stat= -1000	The routine stopped because of Intel MKL DFT or TT interface error.
stat= -10000	The routine stopped as the initialization failed to complete or parameter *ipar*[0] was altered by mistake.
stat= -99999	The routine failed to complete the task because of fatal error.

?_sph_p/?_sph_np

Computes the solution of a spherical Helmholtz problem specified by the parameters.

Syntax

```
void d_sph_p(double* f, DFTI_DESCIPTOR* handle_s, DFTI_DESCIPTOR* handle_c,
int* ipar, double* dpar, int* stat);
```

```
void s_sph_p(float* f, DFTI_DESCIPTOR* handle_s, DFTI_DESCIPTOR* handle_c,
int* ipar, float* spar, int* stat);
```

```
void d_sph_np(double* f, DFTI_DESCIPTOR* handle, int* ipar, double* dpar,
int* stat);
```

```
void s_sph_np(float* f, DFTI_DESCIPTOR* handle, int* ipar, float* spar, int*
stat);
```

Input Parameters

f	double* for d_sph_p/d_sph_np, float* for s_sph_p/s_sph_np.

	Contains the right-hand side of the problem packed in a single vector and modified by the appropriate ?_commit_sph_p/?_commit_sph_np routine. Note that an attempt to substitute the original right-hand side vector at this point will result in a wrong solution. The size of the vector is (*np*+1)*(*nt*+1) and value of the right-hand side in the mesh point (*i, j*) is stored in *f*[*i*+*j**(*np*+1)] .
handle_s, handle_c, handle	DESCIPTOR_HANDLE*. Data structures used by Intel MKL DFT interface (for details, refer to section "DFT Functions" in chapter "Fast Fourier Transforms"). *handle_s* and *handle_c* are used only in ?_sph_p and *handle* is used only in ?_sph_np.
ipar	int array of size 128. Contains integer data to be used by Fast Helmholtz Solver on a sphere (for details, refer to Common Parameters).
dpar	double array of size 5*np/2+nt+10. Contains double-precision data to be used by Fast Helmholtz Solver on a sphere (for details, refer to Common Parameters).
spar	float array of size 5*np/2+nt+10. Contains single-precision data to be used by Fast Helmholtz Solver on a sphere (for details, refer to Common Parameters).

Output Parameters

f	On output, contains the approximate solution to the problem packed the same way as the right-hand side of the problem was packed on input.
handle_s, handle_c, handle	Data structures used by Intel MKL DFT interface.
ipar	Contains integer data to be used by Fast Helmholtz Solver on a sphere. Modified on output as explained in Common Parameters.
dpar	Contains double-precision data to be used by Fast Helmholtz Solver on a sphere. Modified on output as explained in Common Parameters.

spar	Contains single-precision data to be used by Fast Helmholtz Solver on a sphere. Modified on output as explained in Common Parameters.
stat	`int*`. Routine completion status, which is also written to *ipar[0]*. The status should be 0 to proceed to other PL routines.

Description

The routines compute the approximate solution on a sphere of the Helmholtz problem defined in the previous calls to the corresponding initialization and commit routines. The solution is computed according to formulas given in the Poisson Library Implemented section. The *f* parameter, which initially holds the packed vector of the right-hand side of the problem, is replaced by the computed solution packed in the same way. Values of *np* and *nt* are passed to each of the routines with the *ipar* array and defined in the previous call to the appropriate `?_init_sph_p/?_init_sph_np` routine.

Return Values

stat= 1	The routine completed without errors and produced some warnings.
stat= 0	The routine successfully completed the task.
stat= -2	The routine stopped as division by zero occurred. It usually happens if the data in the *dpar* or *spar* array was altered by mistake.
stat= -3	The routine stopped as memory was insufficient to complete the computations.
stat= -100	The routine stopped as an error in the user's data was found or the data in the *dpar*, *spar* or *ipar* array was altered by mistake.
stat= -1000	The routine stopped because of Intel MKL DFT or TT interface error.
stat= -10000	The routine stopped as the initialization failed to complete or parameter *ipar*[0] was altered by mistake.
stat= -99999	The routine failed to complete the task because of fatal error.

free_sph_p/free_sph_np

Cleans the memory allocated for the data structures used by DFT interface.

Syntax

```
void free_sph_p(DFTI_DESCIPTOR* handle_s, DFTI_DESCIPTOR* handle_c, int* ipar, int* stat);
```

```
void free_sph_np(DFTI_DESCIPTOR* handle, int* ipar, int* stat);
```

Input Parameters

handle_s, handle_c, handle	DESCIPTOR_HANDLE*. Data structures used by Intel MKL DFT interface (for details, refer to section "DFT Functions" in chapter "Fast Fourier Transforms"). *handle_s* and *handle_c* are used only in free_sph_p and *handle* is used only in free_sph_np.
ipar	int array of size 128. Contains integer data to be used by Fast Helmholtz Solver on a sphere (for details, refer to Common Parameters).

Output Parameters

handle_s, handle_c, handle	Data structures used by Intel MKL DFT interface. Memory allocated for the structures is released on output.
ipar	Contains integer data to be used by Fast Helmholtz Solver on a sphere. Status of the routine call is written to *ipar[0]*.
stat	int*. Routine completion status, which is also written to *ipar[0]*.

Description

The routine cleans the memory used by the *handle_s*, *handle_c* or *handle* structures, needed for calling Intel MKL DFT functions. If you need to release memory allocated for other parameters, you should include the memory cleaning in your code.

Return Values

stat= 0	The routine successfully completed the task.

stat= -1000	The routine stopped because of Intel MKL DFT or TT interface error.
stat= -99999	The routine failed to complete the task because of fatal error.

Common Parameters

This section provides description of array parameters *ipar*, *dpar* and *spar*, which hold PL routine options in both Cartesian and spherical cases.

NOTE. Initial values are assigned to the array parameters by the appropriate ?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np and ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D/?_commit_sph_p/?_commit_sph_np routines.

ipar — `int` array of size 128, holds integer data needed for Fast Helmholtz Solver (both for Cartesian and spherical coordinate systems). Its elements are described in Table 13-6:

Table 13-6 Elements of the ipar Array

Index	Description
0	Contains status value of the last called PL routine. In general, it should be 0 to proceed with Fast Helmholtz Solver. The element has no predefined values. This element can also be used to inform the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D/?_commit_sph_p/?_commit_sph_np routines of how the Commit step of the computation (see Figure 13-3) should be carried out. Non-zero value of *ipar[0]* with decimal representation $\overline{abc} = 100a + 10b + c$ where each of *a*, *b*, and *c* is equal to 0 or 9, indicates that some parts of the Commit step should be omitted. If *c*=9, the routine omits checking of parameters and initialization of the data structures. If *b*=9, then in the Cartesian case, the routine omits the adjustment of the right-hand side vector *f* to the Neumann boundary condition (multiplication of boundary values by 0.5 as well as incorporation of the boundary function *g*) and/or Dirichlet boundary condition (setting boundary values to 0 as well as incorporation of the boundary function

Index	Description
	G). In this case, the routine also omits the adjustment of the right-hand side vector *f* to the particular boundary functions. For the Helmholtz solver on a sphere, the routine omits computation of the spherical weights for the *dpar*/*spar* array. If *a*=9, then the routine omits the normalization of the right-hand side vector *f*. In the 2D Cartesian case, it is the multiplication by hy^2, where *hy* is the mesh size in the *y* direction (for details, see Poisson Library Implemented). In the 3D (Cartesian) case, it is the multiplication by hz^2, where *hz* is the mesh size in the *z* direction. For Helmholtz solver on a sphere, it is the multiplication by h_θ^2, where h_θ is the mesh size in the θ direction (for details, see Poisson Library Implemented). Using *ipar[0]* you can adjust the routine to your needs and gain efficiency in solving multiple Helmholtz problems that differ only in the right-hand side. You must be cautious using this opportunity, as misunderstanding of the commit process may result in wrong results or program failure (see also Caveat on Parameter Modifications).
1	Contains error messaging options: • *ipar[1]*=-1 indicates that all error messages will be printed to the file MKL_Poisson_Library_log.txt in the folder from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device. • *ipar[1]*=0 indicates that no error messages will be printed. • *ipar[1]*=1 is the default value. It indicates that all error messages will be printed to the preconnected default output device (usually, screen). In case of errors, the *stat* parameter will acquire a non-zero value regardless of the *ipar[1]* setting.
2	Contains warning messaging options: • *ipar[2]*=-1 indicates that all warning messages will be printed to the file MKL_Poisson_Library_log.txt in the directory from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device. • *ipar[2]*=0 indicates that no warning messages will be printed. • *ipar[2]*=1 is the default value. It indicates that all warning messages will be printed to the preconnected default output device (usually, screen).

Index	Description
	In case of warnings, the *stat* parameter will acquire a non-zero value regardless of the *ipar[2]* setting.
3	Contains the number of the combination of boundary conditions. In the Cartesian case, it corresponds to the value that the *BCtype* parameter holds: • In the 2D case, 0 corresponds to 'DDDD' 1 corresponds to 'DDDN' … 15 corresponds to 'NNNN' • In the 3D case, 0 corresponds to 'DDDDDD' 1 corresponds to 'DDDDDN' ... 63 corresponds to 'NNNNNN'. Helmholtz solver on a sphere uses this parameter only in a periodic case. The *bp* and *bt* parameters of the ?_init_sph_p/?_init_sph_np routine, which initializes *ipar[3]*, determine its value: • 0 corresponds to the problem without poles. • 1 corresponds to the problem on the entire sphere.
Parameters 4 through 9 are used only in Cartesian case.	
4	Takes the value of 1 if *BCtype[0]*='N', 0 if *BCtype[0]*='D', and -1 otherwise.
5	Takes the value of 1 if *BCtype[1]*='N', 0 if *BCtype[1]*='D', and -1 otherwise.
6	Takes the value of 1 if *BCtype[2]*='N', 0 if *BCtype[2]*='D', and -1 otherwise.
7	Takes the value of 1 if *BCtype[3]*='N', 0 if *BCtype[3]*='D', and -1 otherwise.
8	Takes the value of 1 if *BCtype[4]*='N', 0 if *BCtype[4]*='D', and -1 otherwise. This parameter is used only in the 3D case.

Index	Description
9	Takes the value of 1 if *BCtype[5]*='N', 0 if *BCtype[5]*='D', and -1 otherwise. This parameter is used only in the 3D case.
10	Takes the value of *nx*, that is, the number of intervals along *x*-axis in the Cartesian case, and the value of *np*, that is, the number of intervals along ϕ-axis in the spherical case.
11	Takes the value of *ny*, that is, the number of intervals along *y*-axis in the Cartesian case, and the value of *nt*, that is, the number of intervals along θ-axis in the spherical case.
12	Takes the value of *nz*, the number of intervals along *z*-axis. This parameter is used only in the 3D (Cartesian) case.
13	Takes the value of 6, which specifies the internal partitioning of the *dpar/spar* array.
14	Takes the value of *ipar[13]*+*ipar[10]*+1, which specifies the internal partitioning of the *dpar/spar* array.

Subsequent values of *ipar* depend upon the dimension of the problem or upon whether the solver on a sphere is periodic.

	2D case	3D case	Periodic case	Non-periodic case
15	Unused	Takes the value of *ipar[14]*+1, which specifies the internal partitioning of the *dpar/spar* array.		
16	Unused	Takes the value of *ipar[14]*+ *ipar[11]*+1, which specifies the internal partitioning of the *dpar/spar* array.		
17	Takes the value of *ipar[14]*+1, which specifies the internal partitioning of the *dpar/spar* array.	Takes the value of *ipar[16]*+1, which specifies the internal partitioning of the *dpar/spar* array.		

Index	Description			
18	Takes the value of *ipar[14]*+3**ipar[10]* /2+1, which specifies the internal partitioning of the *dpar/spar* array.	Takes the value of *ipar[16]*+3* *ipar[10]*/2+1, which specifies the internal partitioning of the *dpar/spar* array.	Takes the value of *ipar[16]*+3 **ipar[10]*/4+1, which specifies the internal partitioning of the *dpar/spar* array.	Takes the value of *ipar[16]* +3**ipar[10]* /2+1, which specifies the internal partitioning of the *dpar/spar* array.
19	Unused	Takes the value of *ipar[18]*+1, which specifies the internal partitioning of the *dpar/spar* array.		Unused
20	Unused	Takes the value of *ipar[18]*+ 3**ipar[11]*/2+1, which specifies the internal partitioning of the dpar/spar array.	Takes the value of *ipar[18]*+3**ipar[10]* /4+1, which specifies the internal partitioning of the dpar/spar array.	Unused
Subsequent values of *ipar* are assigned regardless.				
21	Contains message style options. If *ipar[21]*=0, then PL routines print all error and warning messages in Fortran-style notations. If *ipar[21]*=1, then PL routines print the messages in C-style notations. The default value is 1.			
22	Contains the number of threads to be used for computations in a multithreaded environment. The default value is 1.			
23 through 39	Unused in the current implementation of the Poisson Library.			
40 through 59	Contain the first twenty elements of the *ipar* array of the first Trigonometric Transform that the Solver uses. (For details, see Common Parameters in the "Trigonometric Transform Routines" chapter.)			
60 through 79	Contain the first twenty elements of the *ipar* array of the second Trigonometric Transform that the 3D and periodic solvers use. (For details, see Common Parameters in the "Trigonometric Transform Routines" chapter.)			

NOTE. You may declare the *ipar* array in your code as `int ipar[80]`. However, for compatibility with later versions of Intel MKL Poisson Library, which may require more *ipar* values, it is highly recommended to declare *ipar* as `int ipar[128]`.

Arrays *dpar* and *spar* are similar to each other and differ only in the data precision:

dpar — Holds data needed for double-precision Fast Helmholtz Solver computations.

- For the Cartesian solver, `double` array of size 5*nx/2+7 in the 2D case or 5*(nx+ny)/2+9 in the 3D case; initialized in the `d_init_Helmholtz_2D/d_init_Helmholtz_3D` and `d_commit_Helmholtz_2D/d_commit_Helmholtz_3D` routines.
- For the spherical solver, `double` array of size 5*np/2+nt+10; initialized in the `d_init_sph_p/d_init_sph_np` and `d_commit_sph_p/d_commit_sph_np` routines.

spar — Holds data needed for single-precision Fast Helmholtz Solver computations.

- For the Cartesian solver, `float` array of size 5*nx/2+7 in the 2D case or 5*(nx+ny)/2+9 in the 3D case; initialized in the `s_init_Helmholtz_2D/s_init_Helmholtz_3D` and `s_commit_Helmholtz_2D/s_commit_Helmholtz_3D` routines.
- For the spherical solver, `float` array of size 5*np/2+nt+10; initialized in the `s_init_sph_p/s_init_sph_np` and `s_commit_sph_p/s_commit_sph_np` routines.

As *dpar* and *spar* have similar elements in respective positions, the elements are described together in Table 13-7:

Table 13-7 Elements of the dpar and spar Arrays

Index	Description
0	In the Cartesian case, contains the length of the interval along x-axis right after a call to the `?_init_Helmholtz_2D/?_init_Helmholtz_3D` routine or the mesh size h_x in the x direction (for details, see Poisson Library Implemented) after a call to the `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routine.

Index	Description
	In the spherical case, contains the length of the interval along ϕ-axis right after a call to the ?_init_sph_p/?_init_sph_np routine or the mesh size h_ϕ in the ϕ direction (for details, see Poisson Library Implemented) after a call to the ?_commit_sph_p/?_commit_sph_np routine.
1	In the Cartesian case, contains the length of the interval along y-axis right after a call to the ?_init_Helmholtz_2D/?_init_Helmholtz_3D routine or the mesh size h_y in the y direction (for details, see Poisson Library Implemented) after a call to the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routine. In the spherical case, contains the length of the interval along θ-axis right after a call to the ?_init_sph_p/?_init_sph_np routine or the mesh size h_θ in the θ direction (for details, see Poisson Library Implemented) after a call to the ?_commit_sph_p/?_commit_sph_np routine.
2	In the Cartesian case, contains the length of the interval along z-axis right after a call to the ?_init_Helmholtz_2D/?_init_Helmholtz_3D routine or the mesh size h_z in the z direction (for details, see Poisson Library Implemented) after a call to the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routine. In the Cartesian solver, this parameter is used only in the 3D case. In the spherical solver, contains the coordinate of the leftmost boundary along θ-axis after a call to the ?_init_sph_p/?_init_sph_np routine.
3	Contains the value of the coefficient q after a call to the ?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np routine.
4	Contains the tolerance parameter after a call to the ?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np routine. In the Cartesian case, this value is used only for the pure Neumann boundary conditions (*BCtype*="NNNN" in the 2D case; *BCtype*="NNNNNN" in the 3D case). This is a special case, as the right-hand side of the problem cannot be arbitrary if the coefficient q is zero. Poisson Library verifies that the classical solution exists (up to rounding errors) using this tolerance. In any case, Poisson Library computes the normal solution, that is, the solution

Index	Description
	that has the minimal Euclidean norm of residual. Nevertheless, the ?_Helmholtz_2D/?_Helmholtz_3D routine informs the user that the solution may not exist in a classical sense (up to rounding errors). In the spherical solver, the value is used for the special case of a periodic problem on the entire sphere. This special case is similar to the above described Cartesian case with pure Neumann boundary conditions. So, here Poisson Library computes the normal solution as well. The parameter is also used to detect if the problem is periodic up to rounding errors. The default value for this parameter is 1.0E-10 in case of double-precision computations or 1.0E-4 in case of single-precision computations. The user may increase the value of the tolerance, for instance, to avoid the warnings that may appear.
ipar[13]-1 through *ipar[14]*-1	In the Cartesian case, contain the spectrum of the 1D problem along x-axis after a call to the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routine. In the spherical case, contains the spectrum of the 1D problem along ϕ-axis after a call to the ?_commit_sph_p/?_commit_sph_np routine.
ipar[15]-1 through *ipar[16]*-1	In the Cartesian case, contain the spectrum of the 1D problem along y-axis after a call to the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routine. These elements are used only in the 3D case. In the spherical case, contains the spherical weights after a call to the ?_commit_sph_p/?_commit_sph_np routine.
ipar[17]-1 through *ipar[18]*-1	Take the values of the (staggered) sine/cosine in the mesh points: • along x-axis after a call to the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routine for a Cartesian solver • along ϕ-axis after a call to the ?_commit_sph_p/?_commit_sph_np routine for a spherical solver.
ipar[19]-1 through *ipar[20]*-1	Take the values of the (staggered) sine/cosine in the mesh points: • along y-axis after a call to the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routine for a Cartesian 3D solver • along ϕ-axis after a call to the ?_commit_sph_p routine for a spherical periodic solver.

Index	Description
	These elements are used neither in the 2D Cartesian case nor in the non-periodic spherical case.

NOTE. You may define the array size depending upon the type of the problem to solve.

Caveat on Parameter Modifications

Flexibility of PL interface makes it possible to skip calling the `?_init_Helmholtz_2D/?_init_Helmholtz_3D`/`?_init_sph_p/?_init_sph_np` routine and initialize the basic data structures explicitly in your code. You may also need to modify contents of *ipar*, *dpar* and *spar* arrays after initialization. When doing so, you should provide correct and consistent data in the arrays. Mistakenly altered arrays cause errors or wrong computation. You can perform basic check for correctness and consistency of parameters by calling the `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routine but it does not ensure the correct solution, it only reduces the chance of errors or wrong result.

NOTE. To supply correct and consistent parameters to PL routines, you should have considerable experience in using PL interface and good understanding of the solution process as well as elements that the *ipar*, *spar* and *dpar* arrays contain and dependencies between values of these elements.

However, in rare occurrences, even advanced users may fail in tuning parameters for the Fast Helmholtz Solver.

WARNING. The only way that ensures a proper solution of a Helmholtz problem is to follow a typical sequence of invoking the routines and not change the default set of parameters. So, avoid modifications of *ipar*, *dpar* and *spar* arrays unless a strong need arises.

Implementation Details

Several aspects of the Intel MKL PL interface are platform-specific and language-specific. To promote portability across platforms and ease of use across different languages, users are provided with Intel MKL PL language-specific header files to include in their code. Currently, the following of them are available:

- mkl_poisson.h, to be used together with mkl_dfti.h, for C programs.
- mkl_poisson.f90, to be used toghether with mkl_dfti.f90, for Fortran-90 programs.

NOTE. Use of the Intel MKL PL software without including one of the above header files is not supported.

The include files define function prototypes for appropriate languages.

C-specific Header File

The C-specific header file defines the following function prototypes for the Cartesian solver:

```
void d_init_Helmholtz_2D(double*, double*, double*, double*, int*, int*,
char*, double*, int*, double*, int*);

void d_commit_Helmholtz_2D(double*, double*, double*, double*, double*,
DFTI_DESCRIPTOR_HANDLE*, int*, double*, int*);


void d_Helmholtz_2D(double*, double*, double*, double*, double*,
DFTI_DESCRIPTOR_HANDLE*, int*, double*, int*);

void s_init_Helmholtz_2D(float*, float*, float*, float*, int*, int*, char*,
 float*, int*, float*, int*);

void s_commit_Helmholtz_2D(float*, float*, float*, float*, float*,
DFTI_DESCRIPTOR_HANDLE*, int*, float*, int*);
```

```
void s_Helmholtz_2D(float*, float*, float*, float*, float*,
DFTI_DESCRIPTOR_HANDLE*, int*, float*, int*);

void free_Helmholtz_2D(DFTI_DESCRIPTOR_HANDLE*, int*, int*);

void d_init_Helmholtz_3D(double*, double*, double*, double*, double*, double*,
 int*, int*, int*, char*, double*, int*, double*, int*);

void d_commit_Helmholtz_3D(double*, double*, double*, double*, double*,
double*, double*, DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, int*,
double*, int*);

void d_Helmholtz_3D(double*, double*, double*, double*, double*, double*,
double*, DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, int*, double*,
int*);

void s_init_Helmholtz_3D(float*, float*, float*, float*, float*, float*,
int*, int*, int*, char*, float*, int*, float*, int*);

void s_commit_Helmholtz_3D(float*, float*, float*, float*, float*, float*,
float*, DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, int*, float*,
int*);

void s_Helmholtz_3D(float*, float*, float*, float*, float*, float*, float*,
 DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, int*, float*, int*);

void free_Helmholtz_3D(DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*,
int*, int*);
```

The C-specific header file defines the following function prototypes for the spherical solver:

```
void d_init_sph_p(double*, double*, double*, double*, int*, int*, double*,
int*, double*, int*);

void d_commit_sph_p(double*, DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*,
 int*, double*, int*);

void d_sph_p(double*, DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, int*,
 double*, int*);

void s_init_sph_p(float*, float*, float*, float*, int*, int*, float*, int*,
 float*, int*);

void s_commit_sph_p(float*, DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*,
 int*, float*, int*);

void s_sph_p(float*, DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, int*,
 float*, int*);

void free_sph_p(DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, int*,
int*);

void d_init_sph_np(double*, double*, double*, double*, int*, int*, double*,
 int*, double*, int*);
```

```
void d_commit_sph_np(double*, DFTI_DESCRIPTOR_HANDLE*, int*, double*, int*);
void d_sph_np(double*, DFTI_DESCRIPTOR_HANDLE*, int*, double*, int*);
void s_init_sph_np(float*, float*, float*, float*, int*, int*, float*, int*,
 float*, int*);
void s_commit_sph_np(float*, DFTI_DESCRIPTOR_HANDLE*, int*, float*, int*);
void s_sph_np(float*, DFTI_DESCRIPTOR_HANDLE*, int*, float*, int*);
void free_sph_np(DFTI_DESCRIPTOR_HANDLE*, int*, int*);
```

Fortran-Specific Header File

The Fortran90-specific header file defines the following function prototypes for the Cartesian solver:

```
SUBROUTINE D_INIT_HELMHOLTZ_2D (AX, BX, AY, BY, NX, NY, BCTYPE, Q, IPAR,
DPAR, STAT)
   USE MKL_DFTI

   INTEGER NX, NY, STAT
   INTEGER IPAR(*)
   DOUBLE PRECISION AX, BX, AY, BY, Q
   DOUBLE PRECISION DPAR(*)
   CHARACTER(4) BCTYPE
END SUBROUTINE


SUBROUTINE D_COMMIT_HELMHOLTZ_2D (F, BD_AX, BD_BX, BD_AY, BD_BY, XHANDLE,
IPAR, DPAR, STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   DOUBLE PRECISION F(IPAR(11)+1,*)

   DOUBLE PRECISION DPAR(*)
```

```
   DOUBLE PRECISION BD_AX(*), BD_BX(*), BD_AY(*), BD_BY(*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE

SUBROUTINE D_HELMHOLTZ_2D (F, BD_AX, BD_BX, BD_AY, BD_BY, XHANDLE, IPAR,
DPAR, STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   DOUBLE PRECISION F(IPAR(11)+1,*)

   DOUBLE PRECISION DPAR(*)

   DOUBLE PRECISION BD_AX(*), BD_BX(*), BD_AY(*), BD_BY(*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE

SUBROUTINE S_INIT_HELMHOLTZ_2D (AX, BX, AY, BY, NX, NY, BCTYPE, Q, IPAR,
SPAR, STAT)
   USE MKL_DFTI

   INTEGER NX, NY, STAT
   INTEGER IPAR(*)
   REAL AX, BX, AY, BY, Q
   REAL SPAR(*)
   CHARACTER(4) BCTYPE
END SUBROUTINE
```

```
SUBROUTINE S_COMMIT_HELMHOLTZ_2D (F, BD_AX, BD_BX, BD_AY, BD_BY, XHANDLE,
IPAR, SPAR, STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   REAL F(IPAR(11)+1,*)

   REAL SPAR(*)

   REAL BD_AX(*), BD_BX(*), BD_AY(*), BD_BY(*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE


SUBROUTINE S_HELMHOLTZ_2D (F, BD_AX, BD_BX, BD_AY, BD_BY, XHANDLE, IPAR,
SPAR, STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   REAL F(IPAR(11)+1,*)

   REAL SPAR(*)

   REAL BD_AX(*), BD_BX(*), BD_AY(*), BD_BY(*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE


SUBROUTINE FREE_HELMHOLTZ_2D (XHANDLE, IPAR, STAT)
   USE MKL_DFTI
```

```
   INTEGER STAT
   INTEGER IPAR(*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE


SUBROUTINE D_INIT_HELMHOLTZ_3D (AX, BX, AY, BY, AZ, BZ, NX, NY, NZ, BCTYPE,
 Q, IPAR, DPAR, STAT)
   USE MKL_DFTI

   INTEGER NX, NY, NZ, STAT
   INTEGER IPAR(*)
   DOUBLE PRECISION AX, BX, AY, BY, AZ, BZ, Q
   DOUBLE PRECISION DPAR(*)

   CHARACTER(6) BCTYPE
END SUBROUTINE


SUBROUTINE D_COMMIT_HELMHOLTZ_3D (F, BD_AX, BD_BX, BD_AY, BD_BY, BD_AZ,
BD_BZ, XHANDLE, YHANDLE, IPAR, DPAR, STAT)
   USE MKL_DFTI
```

```
   INTEGER STAT

   INTEGER IPAR(*)

   DOUBLE PRECISION F(IPAR(11)+1,IPAR(12)+1,*)

   DOUBLE PRECISION DPAR(*)

   DOUBLE PRECISION BD_AX(IPAR(12)+1,*), BD_BX(IPAR(12)+1,*),
BD_AY(IPAR(11)+1,*)

   DOUBLE PRECISION BD_BY(IPAR(11)+1,*), BD_AZ(IPAR(11)+1,*),
BD_BZ(IPAR(11)+1,*)

   TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE

END SUBROUTINE


SUBROUTINE D_HELMHOLTZ_3D (F, BD_AX, BD_BX, BD_AY, BD_BY, BD_AZ, BD_BZ,
XHANDLE, YHANDLE, IPAR, DPAR, STAT)

   USE MKL_DFTI


   INTEGER STAT

   INTEGER IPAR(*)

   DOUBLE PRECISION F(IPAR(11)+1,IPAR(12)+1,*)

   DOUBLE PRECISION DPAR(*)

   DOUBLE PRECISION BD_AX(IPAR(12)+1,*), BD_BX(IPAR(12)+1,*),
BD_AY(IPAR(11)+1,*)

   DOUBLE PRECISION BD_BY(IPAR(11)+1,*), BD_AZ(IPAR(11)+1,*),
BD_BZ(IPAR(11)+1,*)

   TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE

END SUBROUTINE


SUBROUTINE S_INIT_HELMHOLTZ_3D (AX, BX, AY, BY, AZ, BZ, NX, NY, NZ, BCTYPE,
 Q, IPAR, SPAR, STAT)

   USE MKL_DFTI
```

```
   INTEGER NX, NY, NZ, STAT
   INTEGER IPAR(*)
   REAL AX, BX, AY, BY, AZ, BZ, Q
   REAL SPAR(*)

   CHARACTER(6) BCTYPE
END SUBROUTINE


SUBROUTINE S_COMMIT_HELMHOLTZ_3D (F, BD_AX, BD_BX, BD_AY, BD_BY, BD_AZ,
BD_BZ, XHANDLE, YHANDLE, IPAR, SPAR, STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   REAL F(IPAR(11)+1,IPAR(12)+1,*)
   REAL SPAR(*)
   REAL BD_AX(IPAR(12)+1,*), BD_BX(IPAR(12)+1,*), BD_AY(IPAR(11)+1,*)
   REAL BD_BY(IPAR(11)+1,*), BD_AZ(IPAR(11)+1,*), BD_BZ(IPAR(11)+1,*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE
END SUBROUTINE


SUBROUTINE S_HELMHOLTZ_3D (F, BD_AX, BD_BX, BD_AY, BD_BY, BD_AZ, BD_BZ,
XHANDLE, YHANDLE, IPAR, SPAR, STAT)
   USE MKL_DFTI
```

```
   INTEGER STAT
   INTEGER IPAR(*)
   REAL F(IPAR(11)+1,IPAR(12)+1,*)
   REAL SPAR(*)
   REAL BD_AX(IPAR(12)+1,*), BD_BX(IPAR(12)+1,*), BD_AY(IPAR(11)+1,*)
   REAL BD_BY(IPAR(11)+1,*), BD_AZ(IPAR(11)+1,*), BD_BZ(IPAR(11)+1,*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE
END SUBROUTINE

SUBROUTINE FREE_HELMHOLTZ_3D (XHANDLE, YHANDLE, IPAR, STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE
END SUBROUTINE
```

The Fortran90-specific header file defines the following function prototypes for the spherical solver:

```
SUBROUTINE D_INIT_SPH_P(AP,BP,AT,BT,NP,NT,Q,IPAR,DPAR,STAT)
   USE MKL_DFTI

   INTEGER NP, NT, STAT
   INTEGER IPAR(*)
   DOUBLE PRECISION AP,BP,AT,BT,Q
   DOUBLE PRECISION DPAR(*)

END SUBROUTINE
```

```
SUBROUTINE D_COMMIT_SPH_P(F,HANDLE_S,HANDLE_C,IPAR,DPAR,STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   DOUBLE PRECISION DPAR(*)

   DOUBLE PRECISION F(IPAR(11)+1,*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_C, HANDLE_S
END SUBROUTINE

SUBROUTINE D_SPH_P(F,HANDLE_S,HANDLE_C,IPAR,DPAR,STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   DOUBLE PRECISION DPAR(*)

   DOUBLE PRECISION F(IPAR(11)+1,*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_C, HANDLE_S
END SUBROUTINE

SUBROUTINE S_INIT_SPH_P(AP,BP,AT,BT,NP,NT,Q,IPAR,SPAR,STAT)
   USE MKL_DFTI

   INTEGER NP, NT, STAT
   INTEGER IPAR(*)
   REAL AP,BP,AT,BT,Q
   REAL SPAR(*)
```

```
END SUBROUTINE

SUBROUTINE S_COMMIT_SPH_P(F,HANDLE_S,HANDLE_C,IPAR,SPAR,STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   REAL SPAR(*)

   REAL F(IPAR(11)+1,*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_C, HANDLE_S
END SUBROUTINE

SUBROUTINE S_SPH_P(F,HANDLE_S,HANDLE_C,IPAR,SPAR,STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   REAL SPAR(*)

   REAL F(IPAR(11)+1,*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_C, HANDLE_S
END SUBROUTINE

SUBROUTINE FREE_SPH_P(HANDLE_S,HANDLE_C,IPAR,STAT)
   USE MKL_DFTI
```

```
   INTEGER STAT
   INTEGER IPAR(*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_S, HANDLE_C
END SUBROUTINE


SUBROUTINE D_INIT_SPH_NP(AP,BP,AT,BT,NP,NT,Q,IPAR,DPAR,STAT)
   USE MKL_DFTI

   INTEGER NP, NT, STAT
   INTEGER IPAR(*)
   DOUBLE PRECISION AP,BP,AT,BT,Q
   DOUBLE PRECISION DPAR(*)

END SUBROUTINE

SUBROUTINE D_COMMIT_SPH_NP(F,HANDLE,IPAR,DPAR,STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   DOUBLE PRECISION DPAR(*)

   DOUBLE PRECISION F(IPAR(11)+1,*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
END SUBROUTINE

SUBROUTINE D_SPH_NP(F,HANDLE,IPAR,DPAR,STAT)
   USE MKL_DFTI
```

```
      INTEGER STAT
      INTEGER IPAR(*)
      DOUBLE PRECISION DPAR(*)

      DOUBLE PRECISION F(IPAR(11)+1,*)
      TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
   END SUBROUTINE

   SUBROUTINE S_INIT_SPH_NP(AP,BP,AT,BT,NP,NT,Q,IPAR,SPAR,STAT)
      USE MKL_DFTI

      INTEGER NP, NT, STAT
      INTEGER IPAR(*)
      REAL AP,BP,AT,BT,Q
      REAL SPAR(*)

   END SUBROUTINE

   SUBROUTINE S_COMMIT_SPH_NP(F,HANDLE,IPAR,SPAR,STAT)
      USE MKL_DFTI

      INTEGER STAT
      INTEGER IPAR(*)
      REAL SPAR(*)

      REAL
   F(IPAR(11)+1,*)
      TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
   END SUBROUTINE
```

```
SUBROUTINE S_SPH_NP(F,HANDLE,IPAR,SPAR,STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   REAL SPAR(*)

   REAL F(IPAR(11)+1,*)

TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
END SUBROUTINE


SUBROUTINE FREE_SPH_NP(HANDLE,IPAR,STAT)
   USE MKL_DFTI

   INTEGER STAT
   INTEGER IPAR(*)
   TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
END SUBROUTINE
```

Fortran-90 specifics of the PL routines usage are similar for all Intel MKL PDE support tools and described in the Calling PDE Support Routines from Fortran-90 section.

Calling PDE Support Routines from Fortran-90

The calling interface for all Intel MKL TT and PL routines is designed to be easily used in C. However, any of the TT and PL routines can be invoked directly from Fortran-90 if users are familiar with the inter-language calling conventions of their platforms.

Neither TT nor PL interface can be invoked from Fortran-77 due to restrictions imposed by the use of Intel MKL DFT interface.

The inter-language calling conventions include, but are not limited to, the argument passing mechanisms for the language, the data type mappings from C to Fortran-90 and how C external names are decorated on the platform.

To promote portability and relieve a user of dealing with the calling conventions specifics, Fortran-90 header file mkl_trig_transforms.f90 for TT routines and mkl_poisson.f90 for PL routines, used together with mkl_dfti.f90, declare a set of macros and introduce type definitions intended to hide the inter-language calling conventions and provide an interface to the routines that looks natural in Fortran-90.

For example, consider a hypothetical library routine, `foo`, which takes a double-precision vector of length *n*. C users would access such a function as follows:

```
int n;

double *x;

…

foo(x, &n);
```

As noted above, to invoke `foo`, Fortran-90 users would need to know what Fortran-90 data types correspond to C types `int` and `double` (or `float` in case of single-precision), what argument-passing mechanism the C compiler uses and what, if any, name decoration is performed by the C compiler when generating the external symbol `foo`.However, with the Fortran-90 header files mkl_trig_transforms.f90 / mkl_poisson.f90 and mkl_dfti.f90 included, the invocation of `foo` within a Fortran-90 program will look as follows:

- For TT interface,

```
use mkl_dfti

use mkl_trig_transforms

INTEGER n

DOUBLE PRECISION, ALLOCATABLE :: x

…

CALL FOO(x,n)
```

- For PL interface,

```
use mkl_dfti

use mkl_poisson

INTEGER n

DOUBLE PRECISION, ALLOCATABLE :: x

…

CALL FOO(x,n)
```

Note that in the above example, the header files mkl_trig_transforms.f90 / mkl_poisson.f90 and mkl_dfti.f90 provide a definition for the subroutine `FOO`. To ease the use of PL or TT routines in Fortran-90, the general approach of providing Fortran-90 definitions of names is used throughout the libraries. Specifically, if a name from a PL or TT interface is documented as having the C-specific name `foo`, then the Fortran-90 header files provide an appropriate Fortran-90 language type definition `FOO`.

One of the key differences between Fortran-90 and C is the language argument-passing mechanism: C programs use pass-by-value semantics and Fortran-90 programs use pass-by-reference semantics. The Fortran-90 headers ensure proper treatment of this difference. In particular, in the above example, the header files mkl_trig_transforms.f90 / mkl_poisson.f90 and mkl_dfti.f90 hide the difference by defining a macro `FOO` that takes the address of the appropriate arguments.

14 Optimization Solvers Routines

Intel® Math Kernel Library provides tools for solving optimization problems. These tools are routines for solving nonlinear least squares problems through the Trust-Region (TR) algorithms.

Intel MKL Optimization solver routines that can be used for:

- solving nonlinear least-squares problems without constraints
- solving nonlinear least-squares problems with boundary constraints
- computing Jacobi matrix by central differences for solving nonlinear least-square problem.

The first two groups of routines are designed to find only local minimum point. However problems can have multiple local minima and trying different initial points is recommended for better solutions.

For more information on terms and key concepts required to understand the use of the Intel MKL nonlinear least-squares problem solver routines, see Appendix F, "Optimization Solvers Basics".

Routines described below are subdivided according to their purpose as follows:

Nonlinear Least-Squares Problem without Constraints

Nonlinear Least-Squares Problem with Linear (Boundary) Constraints

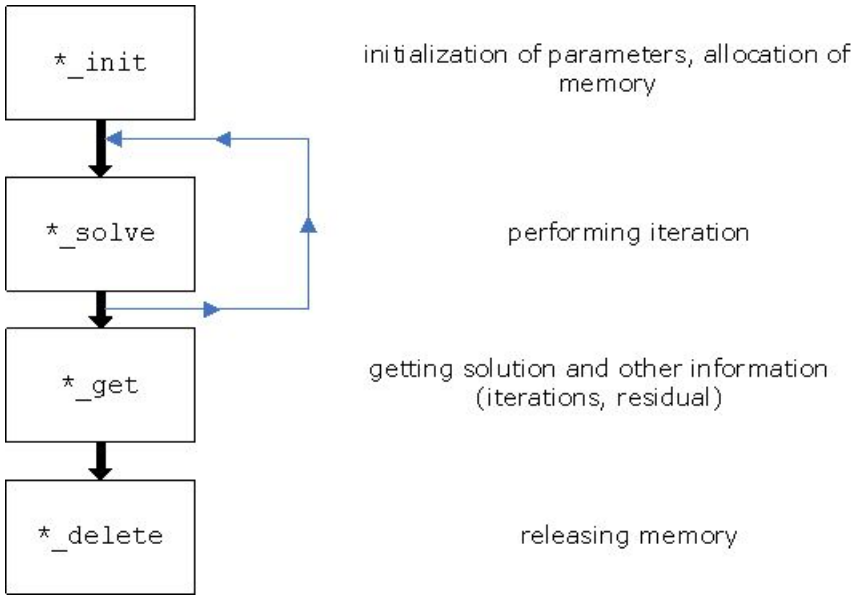
Jacobi Matrix Calculation Routines.

Organization and Implementation

The TR solvers have RCI-based interfaces. The RCI TR interface implements a group of user-callable routines that are used in the step-by-step solving process for nonlinear least square problem and follows the general RCI scheme described in [Dong95]. RCI means that user is supposed to perform certain operations for the solver (for example, to provide values of the objective function or Jacobi

matrix). When the solver needs the results of such operations, the user should pass them to the solver. This makes the solver independent of specific implementation of the operations. However, this approach requires some additional work by the user.

Figure 14-1 Typical order for invoking RCI solver routines



The Trust-Region solvers implement with OpenMP* support. For using multiprocessing mode, set the number of threads in environment variable `OMP_NUM_THREADS`.

Memory Allocation and Handles

To make the routines easy to use, the user is not required to allocate temporary working storage. Any required memory is allocated by the solver. To allow multiple users to access the solver simultaneously, the solver keeps track of the storage allocated for a particular application by using an opaque data object called a `handle`. Each of the Intel RCI TR solver routine either creates, uses or deletes a handle.

Handle declaration varies from language to language. This document declares it as to be of `_TRNSP_HANDLE_t` or `_TRNSPBC_HANDLE_t` type.

C and C++ programmers should declare a handle as:

```
#include "mkl_rci.h"

_TRNSP_HANDLE_t handle;
```

or

```
_TRNSPBC_HANDLE_t handle;
```

Fortran programmers using compilers that support eight byte integers should declare a handle as:

```
INCLUDE "mkl_rci.fi"

INTEGER*8 handle
```

In addition to the necessary definition for the correct declaration of a handle, the include file also defines the following:

- function prototypes for languages that support prototypes
- symbolic constants that are used for the error returns
- user options for the solver routines
- message severity.

Nonlinear Least-Squares Problem without Constraints

The nonlinear least squares problem without constraints can be described as follows:

$$\min_{x \in R^n} F(x) = \|y - f(x)\|_2^2, y \in R^m, x \in R^n, f: R^n \to R^m, m > n,$$

where $F(x) : R^n \to R$ is a twice differentiable function in R^n. Solving of nonlinear least squares problem is searching for the best approximation to vector y with model function $f_i(x)$, which has nonlinear dependence on variables x. The best approximation means that the sum of squares of residuals $y_i - f_i(x)$ is the lowest possible.

Table 14-1 RCI TR Routines

Routine Name	Operation
dtrnlsp_init	Initializes the solver.

Routine Name	Operation
dtrnlsp_solve	Solves a nonlinear least-squares problem on the basis of RCI using the Trust-Region algorithm.
dtrnlsp_get	Retrieves the number of iterations, stop criterion, initial residual, and final residual.
dtrnlsp_delete	Removes data.

dtrnlsp_init

Initializes the solver of nonlinear least square problem.

Syntax

Fortran:

```
res = dtrnlsp_init(handle, n, m, x, eps, iter1, iter2, rs)
```

C:

```
res = dtrnlsp_init(&handle, &n, &m, x, eps, &iter1, &iter2, &rs);
```

Description

The routine initializes the solver. After initialization all subsequent invocations of `dtrnlsp_solve` routine should use the values of the handle returned by `dtrnlsp_init`.

The *eps* array contains the stopping tests:

eps(1): Δ_k < *eps*(1)

eps(2): ||*F*(*x*)|| < *eps*(2)

eps(3): $||A(x)_{ij}||$ < *eps*(3)

eps(4): ||s|| < *eps*(4)

eps(5): ||*F*(*x*)||- ||*F*(*x*) - *A*(*x*)*s*|| < *eps*(5)

eps(6): trial step precision. If *eps*(6) = 0, then *eps*(6) = 1.d-10,

where *A* is a Jacobi matrix.

Input Parameters

n	INTEGER. Length of *X*.
m	INTEGER. Length of *F*(*x*).
x	DOUBLE PRECISION. Array of size *n*. Initial guess.
eps	DOUBLE PRECISION. Array of size 6; contains stopping tests. See the values in Description.
iter1	INTEGER. Specifies the maximum number of iterations.
iter2	INTEGER. Specifies the maximum number of iterations of trial-step calculation.
rs	DOUBLE PRECISION. Positive input variable used in determining the initial step bound. In most cases the factor should lie within the interval (0.1, 100.0). The generally recommended value is 100.

Output Parameters

handle	Data object of _TRNSP_HANDLE_t type for C/C++ programmers and INTEGER*8 for FORTRAN programmers.
res	INTEGER. Informs about the task completion. *res* = TR_SUCCESS means the routine completed the task normally. *res* = TR_INVALID_OPTION means an error in the input parameters. *res* = TR_OUT_OF_MEMORY means a memory error.

dtrnlsp_solve

Solves a nonlinear least-squares problem using Trust-Region algorithm.

Syntax

Fortran:

```
res = dtrnlsp_solve(handle, fvec, fjac, RCI_Request)
```

C:

```
res = dtrnlsp_solve(&handle, fvec, fjac, &RCI_Request);
```

Description

The dtrnlsp_solve routine based on Reverse Communication Interface (RCI) uses the Trust-Region algorithm to solve nonlinear least squares problems. The problem is stated as follows:

$$\min_{x \in R^n} \frac{1}{2} F(x)^T F(x) = \frac{1}{2} \sum_{i=1}^{m} f_i(x)^2,$$

where $m \geq n$, $F : R^n \rightarrow R^m$ and $f_i(x)$ is the i-th component function of $F(x)$. From a current point, the algorithm uses the trust region approach:

$$\min_{x \in R^n} \left\| F(x_c) + J(x_c)(x_n - x_c) \right\| \quad \text{subject to} \quad \left\| x_n - x_c \right\|_2 \leq \Delta_c$$

to get the new point x_n as solution to the following problem:

$$\min_{x \in R^n} \left\| J^T(x) J(x) + J^T F(x) s \right\|_2,$$

where s is the trial step, and $||s||_2 \leq \Delta_c$

then $x_n = x_c + s$.

The *RCI_Request* parameter informs about the task completion and may have the following values:

RCI_Request= 2 - calculate the Jacobian matrix and put the result into *fjac*.

RCI_Request= 1 - recalculate the function at vector X and put the result into *fvec*.

RCI_Request= 0 - successful completion of the task

RCI_Request= -1 - the algorithm has exceeded the maximal number of iterations.

RCI_Request= -2 - $\Delta_k <$ *eps*(1)

RCI_Request= -3 - ||*F*(*x*)|| < *eps*(2)

RCI_Request= -4 - $||A(x)_{ij}|| < eps(3)$

RCI_Request= -5 - $||s|| < eps(4)$

RCI_Request= -6 - $||F(x)|| - ||F(x) - A(x)s|| < eps(5)$,

where *A* is a Jacobi matrix.

Input Parameters

handle	Data object of _TRNSP_HANDLE_t type for C/C++ programmers and INTEGER*8 for FORTRAN programmers.
fvec	DOUBLE PRECISION. Array of size *m*. Contains the function values at X, where $fvec(i) = (y_i - f_i(x))$.
fjac	DOUBLE PRECISION. Array of size (*m*,*n*). Contains the Jacobi matrix of the function.

Output Parameters

fvec	DOUBLE PRECISION. Array of size *m*. Contains the updated function values at X
RCI_Request	INTEGER. Informs about the task completion. When equal to 0, the task is completed succefully. See Description for the other values of the parameter and their meaning.
res	INTEGER. Informs about the task completion. *res* = TR_SUCCESS means the routine completed the task normally.

dtrnlsp_get

Retrieves the number of iterations, stop criterion, initial residual, and final residual.

Syntax

Fortran:

```
res = dtrnlsp_get(handle, iter, st_cr, r1, r2)
```

C:

```
res = dtrnlsp_get(&handle, &iter, &st_cr, &r1, &r2);
```

Description

The routine retrieves the number of current iterations, stop criterion, initial residual, and final residual.

The *st_cr* parameter contains the stop criterion:

st_cr = 1 - the algorithm has exceeded the maximal number of iterations.

st_cr = 2 - Δ_k < *eps*(1)

st_cr = 3 - ||*F*(*x*)|| < *eps*(2)

st_cr = 4 - $||A(x)_{ij}||$ < *eps*(3)

st_cr = 5 - ||*s*|| < *eps*(4)

st_cr = 6 - ||*F*(*x*)||- ||*F*(*x*) - *A*(*x*)*s*|| < *eps*(5),

where *A* is a Jacobi matrix.

Input Parameters

handle	Data object of _TRNSP_HANDLE_t type for C/C++ programmers and INTEGER*8 for FORTRAN programmers.

Output Parameters

iter	INTEGER. Contains the current number of iterations.
st_cr	INTEGER. Contains the stop criterion. See Description for the parameter values and their meanings.
r1	DOUBLE PRECISION. Contains the initial residual, that is, the functional value (||*y* – *f*(*x*)||) of the initial *x* set by the user.
r2	DOUBLE PRECISION. Contains the final residual, that is, the functional value (||*y* – *f*(*x*)||) of the final *x* resulting from the algorithm operation.
res	INTEGER. Informs about task completion. *res* = TR_SUCCESS means the routine completed the task normally.

dtrnlsp_delete

Removes data object required by TR solver.

Syntax

Fortran:

```
res = dtrnlsp_delete(handle)
```

C:

```
res = dtrnlsp_delete(&handle);
```

Description

The routine removes a data object needed for the RCI TR solver.

Input Parameters

handle	Data object of `_TRNSP_HANDLE_t` type for C/C++ programmers and `INTEGER*8` for FORTRAN programmers.

Output Parameters

res	`INTEGER`. Informs about task completion. *res* = `TR_SUCCESS` means the routine completed the task normally.

Examples of dtrnlsp Usage

Example 14-1. dtrnlsp Usage in Fortran

```
C** NONLINEAR LEAST SQUARE PROBLEM WITHOUT BOUNDARY CONSTRAINTS
     PROGRAM EXAMPLE_DTRNLSP_POWELL
        IMPLICIT NONE
C** HEADER-FILE WITH DEFINITIONS (CONSTANTS, EXTERNALS)
        INCLUDE 'mkl_rci.fi'
C** USER'S OBJECTIVE FUNCTION
        EXTERNAL            EXTENDET_POWELL
C** N - NUMBER OF FUNCTION VARIABLES
        INTEGER             N
        PARAMETER           (N = 40)
C** M - DIMENSION OF FUNCTION VALUE
        INTEGER             M
        PARAMETER           (M = 40)
C** SOLUTION VECTOR. CONTAINS VALUES X FOR F(X)
        DOUBLE PRECISION    X (N)
C** PRECISIONS FOR STOP-CRITERIA (SEE MANUAL FOR MORE DETAILS)
        DOUBLE PRECISION    ESP (6)
C** JACOBI CALCULATION PRECISION
        DOUBLE PRECISION    JAC_EPS
C** REVERSE COMMUNICATION INTERFACE PARAMETER
        INTEGRER            RCI_REQUEST
C** FUNCTION (F(X)) VALUE VECTOR
        DOUBLE PRECISION    FVEC (M)
C** JACOBI MATRIX
        DOUBLE PRECISION    FJAC (M, N)
```

```
C** NUMBER OF ITERATIONS
        INTEGRER            ITER
C** NUMBER OF STOP-CRITERION
        INTEGRER            ST_CR
C** CONTROLS OF RCI CYCLE
        INTEGRER            SUCCESSFUL
C** MAXIMUM NUMBER OF ITERATIONS
        INTEGRER            ITER1
C** MAXIMUM NUMBER OF ITERATIONS OF CALCULATION OF TRIAL-STEP
        INTEGRER            ITER2
C** INITIAL STEP BOUND
        DOUBLE PRECISION    RS
C** INITIAL AND FINAL RESIDUALS
        DOUBLE PRECISION    R1, R2
C** TR SOLVER HANDLE
        INTEGRER*8          HANDLE
C** CYCLE'S COUNTERS
        INTEGRER            I, J

C** SET PRECISIONS FOR STOP-CRITERIA
        EPS (1:6) = 1.D-5
C** SET MAXIMUM NUMBER OF ITERATIONS
        ITER1 = 1000
C** SET MAXIMUM NUMBER OF ITERATIONS OF CALCULATION OF TRIAL-STEP
        ITER2 = 100
C** SET INITIAL STEP BOUND
        RS = 100.D0
C** PRECISIONS FOR JACOBI CALCULATION
```

```
        JAC_EPS = 1.D-8
C** SET THE INITIAL GUESS
        DO I = 1, N/4
            X (4*I - 3) =  3.D0
            X (4*I - 2) = -1.D0
            X (4*I - 1) =  0.D0
            X (4*I)     =  1.D0
        ENDDO
C** SET INITIAL VALUES
        DO I = 1, M
            FVEC (I) = 0.D0
            DO J = 1, N
                FJAC (I, J) = 0.D0
            ENDDO
        ENDDO
C** INITIALIZE SOLVER (ALLOCATE MEMORY, SET INITIAL VALUES)
C**   HANDLE       IN/OUT: TR SOLVER HANDLE
C**   N            IN:     NUMBER OF FUNCTION VARIABLES
C**   M            IN:     DIMENSION OF FUNCTION VALUE
C**   X            IN:     SOLUTION VECTOR. CONTAINS VALUES X FOR F(X)
C**   EPS          IN:     PRECISIONS FOR STOP-CRITERIA
C**   ITER1        IN:     MAXIMUM NUMBER OF ITERATIONS
C**   ITER2        IN:     MAXIMUM NUMBER OF ITERATIONS OF CALCULATION OF
TRIAL-STEP
C**   RS           IN:     INITIAL STEP BOUND
        IF (DTRNLSP_INIT (HANDLE, N, M, X, EPS, ITER1, ITER2, RS)
     +   /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
            PRINT *, '| ERROR IN DTRNLSP_INIT'
```

```
C** AND STOP
            STOP
        ENDIF
C** SET INITIAL RCI CYCLE VARIABLES
        RCI_REQUEST = 0
        SUCCESSFUL = 0
C** RCI CYCLE
        DO WHILE (SUCCESSFUL == 0)
C** CALL TR SOLVER
C**   HANDLE            IN/OUT: TR SOLVER HANDLE
C**   FVEC              IN:     VECTOR
C**   FJAC              IN:     JACOBI MATRIX
C**   RCI_REQUEST       IN/OUT: RETURN NUMBER THAT DENOTES NEXT STEP FOR
PERFORMING
        IF (DTRNLSP_SOLVE (HANDLE, FVEC, FJAC, RCI_REQUEST)
     +   /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
            PRINT *, '| ERROR IN DTRNLSP_SOLVE'
C** AND STOP
            STOP
        ENDIF
C**   RCI_REQUEST       IN/OUT: RETURN NUMBER THAT DENOTES NEXT STEP FOR
PERFORMING
C** ACCORDING TO RCI_REQUEST VALUE WE DO NEXT STEP
            SELECT CASE (RCI_REQUEST)
            CASE (-1, -2, -3, -4, -5, -6)
C**   STOP RCI CYCLE
                  SUCCESSFUL = 1
            CASE (1)
```

```
C**   RECALCULATE FUNCTION VALUE
C**     M              IN:     DIMENSION OF FUNCTION VALUE
C**     N              IN:     NUMBER OF FUNCTION VARIABLES
C**     X              IN:     SOLUTION VECTOR
C**     FVEC           OUT:    FUNCTION VALUE F(X)
                  CALL EXTENDET_POWELL (M, N, X, FVEC)
            CASE (2)
C**   COMPUTE JACOBI MATRIX
C**     EXTENDET_POWELL IN:    EXTERNAL OBJECTIVE FUNCTION
C**     N                  IN:    NUMBER OF FUNCTION VARIABLES
C**     M                  IN:    DIMENSION OF FUNCTION VALUE
C**     FJAC               OUT:   JACOBI MATRIX
C**     X                  IN:    SOLUTION VECTOR
C**     JAC_EPS          IN:    JACOBI CALCULATION PRECISION
                   IF (DJACOBI (EXTENDET_POWELL, N, M, FJAC, X, JAC_EPS)
     +            /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
                  PRINT *, '| ERROR IN DTRNLSP_SOLVE'
C** AND STOP
                  STOP
              ENDIF
          ENDSELECT
      ENDDO
C** GET SOLUTION STATUSES
C**   HANDLE         IN:   TR SOLVER HANDLE
C**   ITER           OUT:  NUMBER OF ITERATIONS
C**   ST_CR          OUT:  NUMBER OF STOP CRITERION
C**   R1             OUT: INITIAL RESIDUALS
```

```
C**   R2            OUT: FINAL RESIDUALS
        IF (DTRNLSP_GET (HANDLE, ITER, ST_CR, R1_R2)
     +   /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
            PRINT *, '| ERROR IN DTRNLSP_GET'
C** AND STOP
            STOP
        ENDIF
C** FREE HANDLE MEMORY
        IF (DTRNLSP_DELETE (HANDLE) /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
            PRINT *, '| ERROR IN DTRNLSP_DELETE'
C** AND STOP
            STOP
        ENDIF

C** IF FINAL RESIDUAL IS LESS THAN REQUIRED PRECISION THEN PRINT PASS
        IF (R2 < 1.D-5) THEN
            PRINT *, '|            DTRNLSP POWELL.............PASS'!, R1, R2
C** ELSE PRINT FAILED
        ELSE
            PRINT *, '|            DTRNLSP POWELL.............FAILED'!, R1,
R2
        ENDIF

      END PROGRAM EXAMPLE_DTRNLSP_POWELL

C** ROUTINE FOR EXTENDET POWELL FUNCTION CALCULATION
C**   M         IN:     DIMENSION OF FUNCTION VALUE
```

```
C**   N            IN:     NUMBER OF FUNCTION VARIABLES
C**   X            IN:     VECTOR FOR FUNCTION CALCULATION
C**   F            OUT:    FUNCTION VALUE F(X)
      SUBROUTINE EXTENDET_POWELL (M, N, X, F)
        IMPLICIT NONE
        INTEGER M, N
        DOUBLE PRECISION X (*), F (*)
        INTEGER I

        DO I = 1, N/4
            F (4*I-3) = X(4*I - 3) + 10.D0 * X(4*I - 2)
            F (4*I-2) = 2.2360679774997896964091736687313D0*(X(4*I-1) -
X(4*I))
            F (4*I-1) = (X(4*I-2) - 2.D0*X(4*I-1))**2
            F (4*I)   = 3.1622776601683793319988935444327D0*(X(4*I-3) -
X(4*I))**2
        ENDDO

      ENDSUBROUTINE EXTENDET_POWELL
```

Example 14-2. dtrnlsp Usage in C

```
#include <stdio.h>
#include <malloc.h>
#include <math.h>

#include "mkl_rci.h"

/* nonlinear least square problem without boundary constraints */
int main ()
{
    /* user's objective function */
    extern void extendet_powell (int *, int *, double*, double*);
    /* n - number of function variables
       m - dimension of function value */
    int                 n = 4, m = 4;
    /* precisions for stop-criteria (see manual for more details) */
    double              eps[6];
    /* solution vector. contains values x for f(x) */
    double              *x;
    /* iter1 - maximum number of iterations
       iter2 - maximum number of iterations of calculation of trial-step */
    int                 iter1 = 1000, iter2 = 100;
    /* initial step bound */
    double              rs = 0.0;
    /* reverse communication interface parameter */
    int                 RCI_Request; // reverse communication interface
variable
    /* controls of rci cycle */
    int                 successful;
```

```
/* function (f(x)) value vector */
double              *fvec;
/* jacobi matrix */
double              *fjac;
/* number of iterations */
int                 iter;
/* number of stop-criterion */
int                 st_cr;
/* initial and final residuals */
double              r1, r2;
/* TR solver handle */
_TRNSP_HANDLE_t handle; // TR solver handle
/* cycle's counter */
int i;

/* memory allocation */
x = (double*) malloc (sizeof (double)*n);
fvec = (double*) malloc (sizeof (double)*m);
fjac = (double*) malloc (sizeof (double)*m*n);
/* set precisions for stop-criteria */
for (i = 0; i < 6; i++)
{
    eps [i] = 0.00001;
}
/* set the initial guess */
for (i = 0; i < n/4; i++)
{
    x [4*i]     =  3.0;
```

```
        x [4*i + 1] = -1.0;
        x [4*i + 2] =  0.0;
        x [4*i + 3] =  1.0;
    }
    /* set the initial values */
    for (i = 0; i < m; i++)
        fvec [i] = 0.0;
    for (i = 0; i < m*n; i++)
        fjac [i] = 0.0;
    /* initialize solver (allocate memory, set initial values)
        handle  in/out: TR solver handle
        n       in:     number of function variables
        m       in:     dimension of function value
        x       in:     solution vector. contains values x for f(x)
        eps     in:     precisions for stop-criteria
        iter1   in:     maximum number of iterations
        iter2   in:     maximum number of iterations of calculation of
trial-step
        rs      in:     initial step bound */
    if (dtrnlsp_init (&handle, &n, &m, x, eps, &iter1, &iter2, &rs) !=
TR_SUCCESS)
    {
       /* if function does not complete successfully then print error message
 */
        printf ("| error in dtrnlsp_init\n");
        /* and exit */
        return 0;
    }
    /* set initial rci cycle variables */
    RCI_Request = 0;
```

```
    successful = 0;
    /* rci cycle */
    while (successful == 0)
    {
        /* call tr solver
            handle      in/out: tr solver handle
            fvec        in:     vector
            fjac        in:     jacobi matrix
            RCI_request in/out: return number which denotes next step for
performing */
       if (dtrnlsp_solve (&handle, fvec, fjac, &RCI_Request) != TR_SUCCESS)
        {
            /* if function does not complete successfully then print error
message */
            printf ("| error in dtrnlsp_solve\n");
            /* and exit */
            return 0;
        }
        /* according to rci_request value we do next step */
        if (RCI_Request == -1 ||
            RCI_Request == -2 ||
            RCI_Request == -3 ||
            RCI_Request == -4 ||
            RCI_Request == -5 ||
            RCI_Request == -6)
            /* exit rci cycle */
            successful = 1;
        if (RCI_Request == 1)
        {
```

```
    /* recalculate function value
       m       in:     dimension of function value
       n       in:     number of function variables
       x       in:     solution vector
       fvec    out:    function value f(x) */
    extendet_powell (&m, &n, x, fvec);
}
if (RCI_Request == 2)
{
    /* compute jacobi matrix
        extendet_powell in:    external objective function
        n               in:    number of function variables
        m               in:    dimension of function value
        fjac            out:   jacobi matrix
```

```
                x               in:    solution vector
                jac_eps         in:    jacobi calculation precision */
            if (djacobi (extendet_powell, &n, &m, fjac, x, eps) != TR_SUCCESS)
              {
                /* if function does not complete successfully then print
error message */
                printf ("| error in djacobi\n");
                /* and exit */
                return 0;
              }
          }
      }
      /* get solution statuses
          handle              in:     TR solver handle
          iter                out:    number of iterations
          st_cr               out:    number of stop criterion
          r1                  out:    initial residuals
          r2                  out:    final residuals */
      if (dtrnlsp_get (&handle, &iter, &st_cr, &r1, &r2) != TR_SUCCESS)
      {
         /* if function does not complete successfully then print error message
 */
          printf ("| error in dtrnlsp_get\n");
          /* and exit */
          return 0;
      }
      /* free handle memory */
      if (dtrnlsp_delete (&handle) != TR_SUCCESS)
      {
```

```
        /* if function does not complete successfully then print error message
 */
         printf ("| error in dtrnlsp_delete\n");
         /* and exit */
         return 0;
    }
    /* free allocated memory */
    free (x);
    free (fvec);
    free (fjac);
    /* if final residual is less than required precision then print pass */
    if (r2 < 0.00001)
        printf ("|        dtrnlsp powell............PASS\n");
    /* else print failed */
    else
        printf ("|        dtrnlsp powell............FAILED\n");
    return 0;
}

/* nonlinear system equations without constraints */
/* routine for extendet powell function calculation
   m     in:     dimension of function value
   n     in:     number of function variables
   x     in:     vector for function calculation
   f     out:    function value f(x) */
void extendet_powell (int *m, int *n, double *x, double *f)
{
    int i;
```

```
    for (i = 0; i < (*n)/4; i++)
    {
        f [4*i] = x [4*i] + 10.0*x [4*i + 1];
       f [4*i + 1] = 2.2360679774997896964091736687313*(x [4*i + 2] - x [4*i
 + 3]);
        f [4*i + 2] = (x [4*i + 1] - 2.0*x [4*i + 2])*(x [4*i + 1] - 2.0*x
[4*i + 2]);
        f [4*i + 3] = 3.1622776601683793319988935444327*(x [4*i] - x [4*i +
3])*(x [4*i] - x [4*i + 3]);
    }
    return;
}
```

Nonlinear Least-Squares Problem with Linear (Bound) Constraints

The nonlinear least squares problem with linear bound constraints can be described and solved in the same way as the nonlinear least squares problem without constraints but it has the following constraints:

$$l_i \le x_i \le u_i, i = 1,\ldots,n, \quad l,u \in R^n.$$

Table 14-2 RCI TR Routines for Problem with Bound Constraints

Routine Name	Operation
dtrnlspbc_init	Initializes the solver.
dtrnlspbc_solve	Solves a nonlinear least-squares problem on the basis of RCI using the Trust-Region algorithm.
dtrnlspbc_get	Retrieves the number of iterations, stop criterion, initial residual, and final residual.
dtrnlspbc_delete	Removes data.

dtrnlspbc_init

Initializes the solver of nonlinear least square problem with linear (boundary) constraints.

Syntax

Fortran:

```
res = dtrnlspbc_init(handle, n, m, x, LW, UP, eps, iter1, iter2, rs)
```

C:

```
res = dtrnlspbc_init(&handle, &n, &m, x, LW, UP, eps, &iter1, &iter2, &rs);
```

Description

The routine initializes the solver. After initialization all subsequent invocations of `dtrnlspbc_solve` routine should use the values of the handle returned by `dtrnlspbc_init`.

The *eps* array contains the stopping tests:

eps(1): Δ_k < *eps*(1)

eps(2): ||*F*(*x*)|| < *eps*(2)

eps(3): $||A(x)_{ij}||$ < *eps*(3)

eps(4): ||s|| < *eps*(4)

eps(5): ||*F*(*x*)||- ||*F*(*x*) - *A*(*x*)*s*|| < *eps*(5)

eps(6): trial step precision. If *eps*(6) = 0, then *eps*(6) = 1.d-10,

where *A* is a Jacobi matrix.

Input Parameters

n	INTEGER. Length of *X*.
m	INTEGER. Length of *F*(*x*).
x	DOUBLE PRECISION. Array of size *n*. Initial guess.
LW	DOUBLE PRECISION. Array of size *n*. Contains low bounds for *x* ($lw_i \le x_i$).
UP	DOUBLE PRECISION. Array of size *n*. Contains upper bounds for *x* ($up_i \le x_i$).

eps	DOUBLE PRECISION. Array of size 6; contains stopping tests. See the values in Description.
iter1	INTEGER. Specifies the maximum number of iterations.
iter2	INTEGER. Specifies the maximum number of iterations of trial-step calculation.
rs	DOUBLE PRECISION. Positive input variable used in determining the initial step bound. In most cases the factor should lie within the interval (0.1, 100.0). The generally recommended value is 100.

Output Parameters

handle	Data object of _TRNSPBC_HANDLE_t type for C/C++ programmers and INTEGER*8 for FORTRAN programmers.
res	INTEGER. Informs about the task completion. *res* = TR_SUCCESS means the routine completed the task normally. *res* = TR_INVALID_OPTION means an error in the input parameters. *res* = TR_OUT_OF_MEMORY means a memory error.

dtrnlspbc_solve

Solves a nonlinear least-squares problem with linear (bound) constraints using Trust-Region algorithm.

Syntax

Fortran:

```
res = dtrnlspbc_solve(handle, fvec, fjac, RCI_Request)
```

C:

```
res = dtrnlspbc_solve(&handle, fvec, fjac, &RCI_Request);
```

Description

The dtrnlspbc_solve routine based on Reverse Communication Interface (RCI) uses the Trust-Region algorithm to solve nonlinear least squares problems with linear (bound) constraints. The problem is stated as follows:

$$\min_{x \in R^n} \frac{1}{2} F(x)^T F(x) = \frac{1}{2} \sum_{i=1}^{m} f_i(x)^2, \text{where } l_i \le x_i \le u_i \; i = 1, \ldots, n,$$

where $m \ge n$, $F : R^n \to R^m$ and $f_i(x)$ is the i-th component function of $F(x)$.

The *RCI_Request* parameter informs about the task completion and may have the following values:

RCI_Request= 2 - calculate the Jacobian matrix and put the result into *fjac*.

RCI_Request= 1 - recalculate the function at vector X and put the result into *fvec*.

RCI_Request= 0 - successful completion of the task

RCI_Request= -1 - the algorithm has exceeded the maximal number of iterations.

RCI_Request= -2 - Δ_k < *eps*(1)

RCI_Request= -3 - ||*F*(*x*)|| < *eps*(2)

RCI_Request= -4 - $||A(x)_{ij}||$ < *eps*(3)

RCI_Request= -5 - ||*s*|| < *eps*(4)

RCI_Request= -6 -||*F*(*x*)||- ||*F*(*x*) - *A*(*x*)*s*|| < *eps*(5),

where *A* is a Jacobi matrix.

Input Parameters

handle	Data object of _TRNSPBC_HANDLE_t type for C/C++ programmers and INTEGER*8 for FORTRAN programmers.
fvec	DOUBLE PRECISION. Array of size *m*. Contains the function values at X, where *fvec*(*i*) = (y_i - $f_i(x)$).
fjac	DOUBLE PRECISION. Array of size *m* by *n*. Contains the Jacobi matrix of the function.

Output Parameters

fvec	DOUBLE PRECISION. Array of size *m*. Contains the updated function values at X
RCI_Request	INTEGER. Informs about the task completion. When equal to 0, the task is completed succefully.

	See Description for the other values of the parameter and their meaning.
res	INTEGER. Informs about the task completion. *res* = TR_SUCCESS means the routine completed the task normally.

dtrnlspbc_get

Retrieves the number of iterations, stop criterion, initial residual, and final residual.

Syntax

Fortran:

```
res = dtrnlspbc_get(handle, iter, st_cr, r1, r2)
```

C:

```
res = dtrnlspbc_get(&handle, &iter, &st_cr, &r1, &r2);
```

Description

The routine retrieves the number of current iterations, stop criterion, initial residual, and final residual.

The *st_cr* parameter contains the stop criterion:

st_cr = 1 - the algorithm has exceeded the maximal number of iterations.

st_cr = 2 - $\Delta_k < eps(1)$

st_cr = 3 - $||F(x)|| < eps(2)$

st_cr = 4 - $||A(x)_{ij}|| < eps(3)$

st_cr = 5 - $||s|| < eps(4)$

st_cr = 6 - $||F(x)|| - ||F(x) - A(x)s|| < eps(5)$,

where *A* is a Jacobi matrix.

Input Parameters

handle	Data object of _TRNSPBC_HANDLE_t type for C/C++ programmers and INTEGER*8 for FORTRAN programmers.

Output Parameters

iter	INTEGER. Contains the current number of iterations.
st_cr	INTEGER. Contains the stop criterion. See Description for the parameter values and their meanings.
r1	DOUBLE PRECISION. Contains the initial residual, that is, the functional value (||*y* – *f*(*x*)||) of the initial *x* set by the user.
r2	DOUBLE PRECISION. Contains the final residual, that is, the functional value (||*y* – *f*(*x*)||) of the final *x* resulting from the algorithm operation.
res	INTEGER. Informs about task completion. *res* = TR_SUCCESS means the routine completed the task normally.

dtrnlspbc_delete

Removes data object required by TR solver.

Syntax

Fortran:

```
res = dtrnlspbc_delete(handle)
```

C:

```
res = dtrnlspbc_delete(&handle);
```

Description

The routine removes a data object needed for the RCI TR solver.

Input Parameters

handle	Data object of _TRNSPBC_HANDLE_t type for C/C++ programmers and INTEGER*8 for FORTRAN programmers.

Output Parameters

res	INTEGER. Informs about task completion. *res* = TR_SUCCESS means the routine completed the task normally.

Examples of dtrnlspbc Usage

Example 14-3. dtrnlspbc Usage in Fortran

```
C** NONLINEAR LEAST SQUARE PROBLEM WITH BOUNDARY CONSTRAINTS
     PROGRAM EXAMPLE_DTRNLSPBC_POWELL
        IMPLICIT NONE
C** HEADER-FILE WITH DEFINITIONS (CONSTANTS, EXTERNALS)
        INCLUDE 'mkl_rci.f'
C** USER'S OBJECTIVE FUNCTION
        EXTERNAL            EXTENDET_POWELL
C** N - NUMBER OF FUNCTION VARIABLES
        INTEGER             N
        PARAMETER           (N = 40)
C** M - DIMENSION OF FUNCTION VALUE
        INTEGER             M
        PARAMETER           (M = 40)
C** SOLUTION VECTOR. CONTAINS VALUES X FOR F(X)
        DOUBLE PRECISION    X (N)
C** PRECISIONS FOR STOP-CRITERIA (SEE MANUAL FOR MORE DETAILS)
        DOUBLE PRECISION    ESP (6)
C** JACOBI CALCULATION PRECISION
        DOUBLE PRECISION    JAC_EPS
C** LOWER AND UPPER BOUNDS
        DOUBLE PRECISION    LW (N), UP (N)
C** REVERSE COMMUNICATION INTERFACE PARAMETER
        INTEGRER            RCI_REQUEST
C** FUNCTION (F(X)) VALUE VECTOR
        DOUBLE PRECISION    FVEC (M)
```

```
C** JACOBI MATRIX
        DOUBLE PRECISION    FJAC (M, N)
C** NUMBER OF ITERATIONS
        INTEGRER            ITER
C** NUMBER OF STOP-CRITERION
        INTEGRER            ST_CR
C** CONTROLS OF RCI CYCLE
        INTEGRER            SUCCESSFUL
C** MAXIMUM NUMBER OF ITERATIONS
        INTEGRER            ITER1
C** MAXIMUM NUMBER OF ITERATIONS OF CALCULATION OF TRIAL-STEP
        INTEGRER            ITER2
C** INITIAL STEP BOUND
        DOUBLE PRECISION    RS
C** INITIAL AND FINAL RESIDUALS
        DOUBLE PRECISION    R1, R2
C** TR SOLVER HANDLE
        INTEGRER*8          HANDLE
C** CYCLE'S COUNTERS
        INTEGRER            I, J
C** SET PRECISIONS FOR STOP-CRITERIA
        EPS (1:6) = 1.D-5
C** SET MAXIMUM NUMBER OF ITERATIONS
        ITER1 = 1000
C** SET MAXIMUM NUMBER OF ITERATIONS OF CALCULATION OF TRIAL-STEP
        ITER2 = 100
C** SET INITIAL STEP BOUND
        RS = 100.D0
```

```
C** PRECISIONS FOR JACOBI CALCULATION
        JAC_EPS = 1.D-8
C** SET THE INITIAL GUESS
        DO I = 1, N/4
            X (4*I - 3) =    3.D0
            X (4*I - 2) = -1.D0
            X (4*I - 1) =    0.D0
            X (4*I)     =    1.D0
        ENDDO
C** SET LOWER AND UPPER BOUNDS
        DO I = 1, N/4
            LW(4*I-3) =      0.1D0

            LW(4*I-2) = -20.D0
            LW(4*I-1) =   -1.D0
            LW(4*I)   =   -1.D0

            UP(4*I-3) =     100.D0
```

```
            UP(4*I-2) =    20.D0
            UP(4*I-1) =    1.D0
            UP(4*I)   =   50.D0
        ENDDO
C** SET INITIAL VALUES
        DO I = 1, M
            FVEC (I) = 0.D0
            DO J = 1, N
                FJAC (I, J) = 0.D0
            ENDDO
        ENDDO
C** INITIALIZE SOLVER (ALLOCATE MEMORY, SET INITIAL VALUES)
C**   HANDLE       IN/OUT: TR SOLVER HANDLE
C**   N            IN:     NUMBER OF FUNCTION VARIABLES
C**   M            IN:     DIMENSION OF FUNCTION VALUE
C**   X            IN:     SOLUTION VECTOR. CONTAINS VALUES X FOR F(X)
C**   LW           IN:     LOWER BOUND
C**   UP           IN:     UPPER BOUND
C**   EPS          IN:     PRECISIONS FOR STOP-CRITERIA
C**   ITER1        IN:     MAXIMUM NUMBER OF ITERATIONS
C**   ITER2        IN:     MAXIMUM NUMBER OF ITERATIONS OF CALCULATION OF
TRIAL-STEP
C**   RS           IN:     INITIAL STEP BOUND
        IF (DTRNLSPBC_INIT (HANDLE, N, M, X, LW, UP, EPS, ITER1, ITER2
     +   , RS) /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
            PRINT *, '| ERROR IN DTRNLSPBC_INIT'
C** AND STOP
            STOP
```

```
        ENDIF
C** SET INITIAL RCI CYCLE VARIABLES
        RCI_REQUEST = 0
        SUCCESSFUL = 0
C** RCI CYCLE
        DO WHILE (SUCCESSFUL == 0)
C** CALL TR SOLVER
C**   HANDLE            IN/OUT: TR SOLVER HANDLE
C**   FVEC              IN:     VECTOR
C**   FJAC              IN:     JACOBI MATRIX
C**   RCI_REQUEST       IN/OUT: RETURN NUMBER THAT DENOTES NEXT STEP FOR
PERFORMING
        IF (DTRNLSPBC_SOLVE (HANDLE, FVEC, FJAC, RCI_REQUEST)
     +   /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
            PRINT *, '| ERROR IN DTRNLSPBC_SOLVE'
C** AND STOP
            STOP
        ENDIF
C**   RCI_REQUEST       IN/OUT: RETURN NUMBER THAT DENOTES NEXT STEP FOR
PERFORMING
C** ACCORDING TO RCI_REQUEST VALUE WE DO NEXT STEP
            SELECT CASE (RCI_REQUEST)
            CASE (-1, -2, -3, -4, -5, -6)
C**   STOP RCI CYCLE
                  SUCCESSFUL = 1
            CASE (1)
C**   RECALCULATE FUNCTION VALUE
C**     M               IN:     DIMENSION OF FUNCTION VALUE
```

```
C**     N              IN:     NUMBER OF FUNCTION VARIABLES
C**     X              IN:     SOLUTION VECTOR
C**     FVEC           OUT:    FUNCTION VALUE F(X)
                  CALL EXTENDET_POWELL (M, N, X, FVEC)
            CASE (2)
C**   COMPUTE JACOBI MATRIX
C**     EXTENDET_POWELL IN:    EXTERNAL OBJECTIVE FUNCTION
C**     N                  IN:    NUMBER OF FUNCTION VARIABLES
C**     M                  IN:    DIMENSION OF FUNCTION VALUE
C**     FJAC               OUT:   JACOBI MATRIX
C**     X                  IN:    SOLUTION VECTOR
C**     JAC_EPS          IN:    JACOBI CALCULATION PRECISION
                   IF (DJACOBI (EXTENDET_POWELL, N, M, FJAC, X, JAC_EPS)
     +           /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
                  PRINT *, '| ERROR IN DTRNLSPBC_SOLVE'
C** AND STOP
                  STOP
              ENDIF
         ENDSELECT
     ENDDO
C** GET SOLUTION STATUSES
C**   HANDLE        IN:   TR SOLVER HANDLE
C**   ITER          OUT:  NUMBER OF ITERATIONS
C**   ST_CR         OUT:  NUMBER OF STOP CRITERION
C**   R1            OUT: INITIAL RESIDUALS
C**   R2            OUT: FINAL RESIDUALS
        IF (DTRNLSPBC_GET (HANDLE, ITER, ST_CR, R1_R2)
```

```
     +   /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
            PRINT *, '| ERROR IN DTRNLSPBC_GET'
C** AND STOP
            STOP
        ENDIF
C** FREE HANDLE MEMORY
        IF (DTRNLSPBC_DELETE (HANDLE) /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
            PRINT *, '| ERROR IN DTRNLSPBC_DELETE'
C** AND STOP
            STOP
        ENDIF

C** IF FINAL RESIDUAL IS LESS THAN REQUIRED PRECISION THEN PRINT PASS
        IF (R2 < 1.D-1) THEN
            PRINT *, '|            DTRNLSPBC POWELL.............PASS'!, R1,
R2
C** ELSE PRINT FAILED
        ELSE
           PRINT *, '|           DTRNLSPBC POWELL.............FAILED'!, R1,
 R2
        ENDIF

      END PROGRAM EXAMPLE_DTRNLSPBC_POWELL

C** ROUTINE FOR EXTENDET POWELL FUNCTION CALCULATION
C**   M          IN:     DIMENSION OF FUNCTION VALUE
C**   N             IN:     NUMBER OF FUNCTION VARIABLES
```

```
C**   X            IN:     VECTOR FOR FUNCTION CALCULATION
C**   F            OUT:    FUNCTION VALUE F(X)
      SUBROUTINE EXTENDET_POWELL (M, N, X, F)
        IMPLICIT NONE
        INTEGER M, N
        DOUBLE PRECISION X (*), F (*)
        INTEGER I

        DO I = 1, N/4
            F (4*I-3) = X(4*I - 3) + 10.D0 * X(4*I - 2)
            F (4*I-2) = 2.2360679774997896964091736687313D0*(X(4*I-1) -
X(4*I))
            F (4*I-1) = (X(4*I-2) - 2.D0*X(4*I-1))**2
            F (4*I)   = 3.1622776601683793319988935444327D0*(X(4*I-3) -
X(4*I))**2
        ENDDO

      ENDSUBROUTINE EXTENDET_POWELL
```

Example 14-4. dtrnlspbc Usage in C

```
#include <stdio.h>
#include <malloc.h>
#include <math.h>

#include "mkl_rci.h"

/* nonlinear least square problem with boundary constraints */
int main ()
{
    /* user's objective function */
    extern void extendet_powell (int *, int *, double*, double*);
    /* n - number of function variables
       m - dimension of function value */
    int                 n = 4, m = 4;
    /* precisions for stop-criteria (see manual for more details) */
    double              eps[6];
    /* solution vector. contains values x for f(x) */
    double              *x;
    /* iter1 - maximum number of iterations
       iter2 - maximum number of iterations of calculation of trial-step */
    int                 iter1 = 1000, iter2 = 100;
    /* initial step bound */
    double              rs = 0.0;
    /* reverse communication interface parameter */
    int                 RCI_Request;
```

```
/* controls of rci cycle */
int                 successful;
/* function (f(x)) value vector */
double              *fvec;
/* jacobi matrix */
double              *fjac;
/* lower and upper bounds */
double              *LW, *UP;
/* number of iterations */
int                 iter;
/* number of stop-criterion */
int                 st_cr;
/* initial and final residuals */
double              r1, r2;
/* TR solver handle */
_TRNSPBC_HANDLE_t handle;
/* cycle's counter */
int i;

/* memory allocation */
x = (double*) malloc (sizeof (double)*n);
fvec = (double*) malloc (sizeof (double)*m);
fjac = (double*) malloc (sizeof (double)*m*n);
LW = (double*) malloc (sizeof (double)*n);
UP = (double*) malloc (sizeof (double)*n);
/* set precisions for stop-criteria */
for (i = 0; i < 6; i++)
{
```

```
    eps [i] = 0.00001;
}
/* set the initial guess */
for (i = 0; i < n/4; i++)
{
    x [4*i]     =   3.0;
    x [4*i + 1] = -1.0;
    x [4*i + 2] =   0.0;
    x [4*i + 3] =   1.0;
}
/* set the initial values */
for (i = 0; i < m; i++)
    fvec [i] = 0.0;
for (i = 0; i < m*n; i++)
    fjac [i] = 0.0;
/* set bounds */
for (i = 0; i < n/4; i++)
{
    LW [4*i]     =    0.1;
    LW [4*i + 1] = -20.0;
    LW [4*i + 2] =   -1.0;
    LW [4*i + 3] =   -1.0;
    UP [4*i]     =  100.0;
    UP [4*i + 1] =  20.0;
    UP [4*i + 2] =    1.0;
    UP [4*i + 3] =   50.0;
}
/* initialize solver (allocate memory, set initial values)
```

```
        handle  in/out: TR solver handle
        n       in:     number of function variables
        m       in:     dimension of function value
        x       in:     solution vector. contains values x for f(x)
        LW      in:     lower bound
        UP      in:     upper bound
        eps     in:     precisions for stop-criteria
        iter1   in:     maximum number of iterations
        iter2   in:     maximum number of iterations of calculation of
trial-step
        rs      in:     initial step bound */
    if (dtrnlspbc_init (&handle, &n, &m, x, LW, UP, eps, &iter1, &iter2,
&rs) != TR_SUCCESS)
    {
       /* if function does not complete successfully then print error message
 */
        printf ("| error in dtrnlspbc_init\n");
        /* and exit */
        return 0;
    }
    /* set initial rci cycle variables */
    RCI_Request = 0;
    successful = 0;
    /* rci cycle */
    while (successful == 0)
    {
        /* call tr solver
            handle      in/out: tr solver handle
            fvec        in:     vector
            fjac        in:     jacobi matrix
```

```
            RCI_request in/out: return number which denotes next step for
performing */
       if (dtrnlspbc_solve (&handle, fvec, fjac, &RCI_Request) != TR_SUCCESS)
        {
            /* if function does not complete successfully then print error
message */
            printf ("| error in dtrnlspbc_solve\n");
            /* and exit */
            return 0;
        }
        /* according to rci_request value we do next step */
        if (RCI_Request == -1 ||
            RCI_Request == -2 ||
            RCI_Request == -3 ||
            RCI_Request == -4 ||
            RCI_Request == -5 ||
            RCI_Request == -6)
            /* exit rci cycle */
            successful = 1;
        if (RCI_Request == 1)
        {
            /* recalculate function value
               m       in:     dimension of function value
               n       in:     number of function variables
               x       in:     solution vector
               fvec    out:    function value f(x) */
            extendet_powell (&m, &n, x, fvec);
        }
        if (RCI_Request == 2)
```

```
{
    /* compute jacobi matrix
        extendet_powell in:    external objective function
        n               in:    number of function variables
        m               in:    dimension of function value
        fjac            out:   jacobi matrix
```

```
                x               in:    solution vector
                jac_eps         in:    jacobi calculation precision */
            if (djacobi (extendet_powell, &n, &m, fjac, x, eps) != TR_SUCCESS)
              {
                /* if function does not complete successfully then print
error message */
                printf ("| error in djacobi\n");
                /* and exit */
                return 0;
              }
          }
      }
      /* get solution statuses
          handle              in:     TR solver handle
          iter                out:    number of iterations
          st_cr               out:    number of stop criterion
          r1                  out:    initial residuals
          r2                  out:    final residuals */
      if (dtrnlspbc_get (&handle, &iter, &st_cr, &r1, &r2) != TR_SUCCESS)
      {
         /* if function does not complete successfully then print error message
 */
          printf ("| error in dtrnlspbc_get\n");
          /* and exit */
          return 0;
      }
      /* free handle memory */
      if (dtrnlspbc_delete (&handle) != TR_SUCCESS)
      {
```

```
        /* if function does not complete successfully then print error message
 */
         printf ("| error in dtrnlspbc_delete\n");
         /* and exit */
         return 0;
    }
    /* free allocated memory */
    free (x);
    free (fvec);
    free (fjac);
    free (LW);
    free (UP);
    /* if final residual is less than required precision then print pass */
    if (r2 < 0.1)
        printf ("|        dtrnlspbc powell............PASS\n");
    /* else print failed */
    else
        printf ("|        dtrnlspbc powell............FAILED\n");
    return 0;
}

/* nonlinear system equations with constraints */
/* routine for extendet powell function calculation
   m     in:     dimension of function value
   n     in:     number of function variables
   x     in:     vector for function calculation
   f     out:    function value f(x) */
void extendet_powell (int *m, int *n, double *x, double *f)
{
```

```
    int i;

    for (i = 0; i < (*n)/4; i++)

    {

       f [4*i] = x [4*i] + 10.0*x [4*i + 1];

      f [4*i + 1] = 2.2360679774997896964091736687313*(x [4*i + 2] - x [4*i
 + 3]);

       f [4*i + 2] = (x [4*i + 1] - 2.0*x [4*i + 2])*(x [4*i + 1] - 2.0*x
[4*i + 2]);

       f [4*i + 3] = 3.1622776601683793319988935444327*(x [4*i] - x [4*i +
3])*(x [4*i] - x [4*i + 3]);

    }

    return;

}
```

Jacobi Matrix Calculation Routines

This section describes routines that compute Jacobi matrix by central differences. Jacobi matrix calculation is required while solving nonlinear least-square problem and systems of nonlinear equations (with or without linear bound constraints).Routines for calculation of Jacobi matrix have "Black-Box" interfaces, where users put the objective function via parameters. But in that case the user's objective function must have fixed interface.

Table 14-3 Jacobi Matrix Calculation Routines

Routine Name	Operation
djacobi_init	Initializes the solver.
djacobi_solve	Computes a Jacobi matrix of the function on the basis of RCI using central difference.
djacobi_delete	Removes data.
djacobi	Computes a Jacobi matrix of the `fcn` function using central difference.

djacobi_init

Initializes the solver of Jacobian calculations.

Syntax

Fortran:

```
res = djacobi_init(handle, n, m, x, a, esp)
```

C:

```
res = djacobi_init(&handle, &n, &m, x, a, &eps);
```

Description

The routine initializes the solver.

Input Parameters

n	INTEGER. Length of *X*.
m	INTEGER. Length of *F*.
x	DOUBLE PRECISION. Array of size *n*. Vector at which the function is evaluated.
eps	DOUBLE PRECISION. Precision of Jacobi matrix calculation.

Output Parameters

handle	Data object of _JACOBIMATRIX_HANDLE_t type for C/C++ programmers and INTEGER*8 for FORTRAN programmers.
a	DOUBLE PRECISION. Array of size (*m*,*n*). Contains the Jacobi matrix of the function.
res	INTEGER. Informs about the task completion. *res* = TR_SUCCESS means the routine completed the task normally. *res* = TR_INVALID_OPTION means an error in the input parameters. *res* = TR_OUT_OF_MEMORY means a memory error.

djacobi_solve

Computes Jacobi matrix of the function on the basis of RCI using central difference.

Syntax

Fortran:

```
res = djacobi_solve(handle, f1, f2, RCI_Request)
```

C:

```
res = djacobi_solve(&handle, f1, f2, &RCI_Request);
```

Description

The `djacobi_solve` routine computes a Jacobi matrix of the function on the basis of RCI using central difference.

Input Parameters

handle	Data object of `_JACOBIMATRIX_HANDLE_t` type for C/C++ programmers and `INTEGER*8` for FORTRAN programmers.

Output Parameters

f1	`DOUBLE PRECISION`. Array of size *m*. Contains the updated function values at `X` + ε.
f2	`DOUBLE PRECISION`. Array of size *m*. Contains the updated function values at `X` - ε.
RCI_Request	`INTEGER`. Informs about the task completion. When equal to 0, the task is completed succefully. *RCI_Request*= 1 means user should calculate the Jacobian matrix and put the result into *f1*. *RCI_Request*= 2 - means user should calculate the Jacobian matrix and put the result into *f2*.
res	`INTEGER`. Informs about the task completion. *res* = `TR_SUCCESS` means the routine completed the task normally. *res* = `TR_INVALID_OPTION` means an error in the input parameters.

djacobi_delete

Removes data object required by Jacobian calculation.

Syntax

Fortran:

```
res = djacobi_delete(handle)
```

C:

```
res = djacobi_delete(&handle);
```

Description

The routine removes a data object needed for the Jacobi matrix RCI solver.

Input Parameters

handle	Data object of `_JACOBIMATRIX_HANDLE_t` type for C/C++ programmers and `INTEGER*8` for FORTRAN programmers.

Output Parameters

res	`INTEGER`. Informs about task completion. *res* = `TR_SUCCESS` means the routine completed the task normally.

djacobi

Computes Jacobi matrix of user's objective function using cenral difference.

Syntax

Fortran:

```
res = djacobi(fcn, n, m, fjac, x, jac_eps)
```

C:

```
res = djacobi(fcn, &n, &m, fjac, x, &jac_eps);
```

Description

The routine computes a Jacobi matrix for function *fcn* using central difference. This routine has "Black-Box" interface, where user inputs the objective function via parameters. But in that case the user's objective function must have a fixed interface.

Input Parameters

fcn	User-supplied subroutine to evaluate the function that defines the least-squares problem. Call `fcn` (*m*, *n*, *x*, *f*), where *m* - `INTEGER`. Input parameter. Length of *f* *n* - `INTEGER`. Input parameter. Length of *x*. *x* - `DOUBLE PRECISION`. Input parameter. Array of size *n*. Vector at which the function is evaluated; *x* should not be changed by `fcn`. *f* - `DOUBLE PRECISION`. Output parameter. Array of size *m*; contains the function values at *x*. Declare `fcn` as `EXTERNAL` in the calling program.
n	`INTEGER`. Length of *X*.
m	`INTEGER`. Length of *F*.
x	`DOUBLE PRECISION`. Array of size *n*. Vector at which the function is evaluated.
eps	`DOUBLE PRECISION`. Precision of Jacobi matrix calculation.

Output Parameters

a	`DOUBLE PRECISION`. Array of size (*m*,*n*). Contains the Jacobi matrix of the function.
res	`INTEGER`. Informs about task completion. *res* = `TR_SUCCESS` means the routine completed the task normally. *res* = `TR_INVALID_OPTION` means an error in the input parameters. *res* = `TR_OUT_OF_MEMORY` means a memory error.

Examples of djacobi_solve Usage

Example 14-5. djacobi_solve Usage in Fortran

```
      PROGRAM JACOBI_MATRIX
         IMPLICIT NONE

         INCLUDE '../include/mkl_opt_tr.f'

         EXTERNAL            EXTENDET_POWELL
C** N - NUMBER OF FUNCTION VARIABLES
C** M - DIMENSION OF FUNCTION VALUE
         INTEGER N, M
         PARAMETER (N = 4, M = 4)
C** JACOBI MATRIX
C** SOLUTION VECTOR. CONTAINS VALUES X FOR F(X)
C** FUNCTION (F(X)) VALUE VECTOR
C** TEMPORARY ARRAYS F1 & F2 WHICH CONTAINS F1 = F(X+EPS) | F2 = F(X-EPS)
         DOUBLE PRECISION A (M,N), X(N), F1(N), F2(N)
         DOUBLE PRECISION EPS
C** PRECISIONS FOR JACOBI_MATRIX CALCULATION
         PARAMETER (EPS = 1.D-6)
C** JACOBI-MATRIX SOLVER HANDLE
         INTEGER*8 HANDLE
C** CONTROLS OF RCI CYCLE
         INTEGER SUCCESSFUL, RCI_REQUEST
C** SET THE X VALUES
         X = 10.d0
```

```
C** INITIALIZE SOLVER (ALLOCATE MEMORY, SET INITIAL VALUES)
        IF (DJACOBI_INIT (HANDLE, N, M, X, EPS) /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
            PRINT *, '#ERROR IN DJACOBI_INIT'
            STOP
        ENDIF
C** SET INITIAL RCI CYCLE VARIABLES
        RCI_REQUEST = 0
        SUCCESSFUL = 0
C** RCI CYCLE
        DO WHILE (SUCCESSFUL == 0)
C** CALL SOLVER
        IF (DJACOBI_SOLVE (HANDLE, F1, F2, RCI_REQUEST) /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
            PRINT *, '#ERROR IN DJACOBI_SOLVE'
            STOP
        ENDIF
        IF     RCI_REQUEST == 1) THEN
C**   CALCULATE FUNCTION VALUE F1 = F(X+EPS)
            CALL EXTENDET_POWELL (M, N, X, F1)
        ELSEIF (RCI_REQUEST == 2) THEN
C**   CALCULATE FUNCTION VALUE F1 = F(X-EPS)
            CALL EXTENDET_POWELL (M, N, X, F2)
        ELSEIF (RCI_REQUEST == 0) THEN
C**   EXIT RCI CYCLE
                SUCCESSFUL = 1
            ENDIF
        ENDDO
```

```
C** FREE HANDLE MEMORY
        IF (DJACOBI_DELETE (HANDLE) /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
            PRINT *, '#ERROR IN DJACOBI_DELETE'
            STOP
        ENDIF
      ENDPROGRAM JACOBI_MATRIX

C** ROUTINE FOR EXTENDET POWELL FUNCTION CALCULATION
C**   M         IN:     DIMENSION OF FUNCTION VALUE
C**   N            IN:     NUMBER OF FUNCTION VARIABLES
C**   X            IN:     VECTOR FOR FUNCTION CALCULATION
C**   F            OUT:    FUNCTION VALUE F(X)
      SUBROUTINE EXTENDET_POWELL (M, N, X, F)
        IMPLICIT NONE
        INTEGER M, N
        DOUBLE PRECISION X (*), F (*)
        INTEGER I

        DO I = 1, N/4
            F (4*I-3) = X(4*I - 3) + 10.D0 * X(4*I - 2)
            F (4*I-2) = DSQRT(5.D0) * (X(4*I-1) - X(4*I))
            F (4*I-1) = (X(4*I-2) - 2.D0*X(4*I-1))**2
            F (4*I)   = DSQRT(10.D0)*(X(4*I-3) - X(4*I))**2
        ENDDO

      ENDSUBROUTINE EXTENDET_POWELL
```

Example 14-6. djacobi_solve Usage in C

```
#include "../include/mkl_opt_tr.h"

#include <stdlib.h>
#include <stdio.h>

int main ()
{
    /* user's objective function */
    extern void extendet_powell (int*, int*, double*, double*);

/* n - number of function variables
   m - dimension of function value */
    int    n = 4, m = 4;
/* jacobi matrix */
   solution vector. contains values x for f(x)
   temporary arrays f1 & f2 that contain f1 = f(x+eps) | f2 = f(x-eps) */
    double *a, *x, *f1, *f2;
/* precisions for jacobi_matrix calculation */
    double eps = 0.000001;
/* jacobi-matrix solver handle */
    _JACOBIMATRIX_HANDLE_t handle;
/* controls of rci cycle */
    int successful, rci_request, i;

    a  = (double*) malloc (sizeof (double) * n*m);
    x  = (double*) malloc (sizeof (double) * n);
```

```
    f1 = (double*) malloc (sizeof (double) * n);
    f2 = (double*) malloc (sizeof (double) * n);
/* set the x values */
    for (i = 0; i < n; i++) x[i] = 10.0;
/* initialize solver (allocate mamory, set initial values) */
    if (djacobi_init (&handle, &n, &m, x, a,  &eps) != TR_SUCCESS){
/* if function does not complete successfully then print error message */
        printf ("\n#ERROR IN DJACOBI_INIT\n");
        return 0;
    }
/* set initial rci cycle variables */
    rci_request = 0;
    successful = 0;
/* rci cycle */
    while (successful == 0) {
/* call solver */
        if (djacobi_solve (&handle, f1, f2, &rci_request) != TR_SUCCESS){
/* if function does not complete successfully then print error message */
            printf ("\n#ERROR IN DJACOBI_SOLVE\n");
            return 0;
        }
        if (rci_request == 1)
/* calculate function value f1 = f(x+eps) */
            extendet_powell (&m, &n, x, f1);
        else if (rci_request == 2)
/* calculate function value f1 = f(x-eps) */
            extendet_powell (&m, &n, x, f2);
        else if (rci_request == 0)
```

```
/* exit rci cycle */
            successful = 1;
   }
/* free handle memory */
```

```
    if (djacobi_delete (&handle) != TR_SUCCESS) {
/* if function does not complete successfully then print error message */
        printf ("\n#ERROR IN DJACOBI_DELETE\n");
        return 0;
    }
    return 0;
}
/* routine for extendet powell function calculation
   m     in:     dimension of function value
   n     in:     number of function variables
   x     in:     vector for function calculation
   f     out:    function value f(x) */
void extendet_powell (int *m, int *n, double *x, double *f)
{
    int i;

    for (i = 0; i < (*n)/4; i++)
    {
       f [4*i] = x [4*i] + 10.0*x [4*i + 1];
       f [4*i + 1] = 2.2360679774*(x [4*i + 2] - x [4*i + 3]);
       f [4*i + 2] = (x [4*i + 1] - 2.0*x [4*i + 2])*(x [4*i + 1] - 2.0*x
[4*i + 2]);
       f [4*i + 3] = 3.1622776601*(x [4*i] - x [4*i + 3])*(x [4*i] - x [4*i
 + 3]);
    }
    return;
}
```

Examples of djacobi Usage

Example 14-7. djacobi Usage in Fortran

```
C**   COMPUTE JACOBI MATRIX
C**     EXTENDET_POWELL IN:    EXTERNAL OBJECTIVE FUNCTION
C**     N                  IN:    NUMBER OF FUNCTION VARIABLES
C**     M                  IN:    DIMENSION OF FUNCTION VALUE
C**     FJAC               OUT:   JACOBI MATRIX
C**     X                  IN:    SOLUTION VECTOR
C**     JAC_EPS          IN:    JACOBI CALCULATION PRECISION
                   IF (DJACOBI (EXTENDET_POWELL, N, M, FJAC, X, JAC_EPS)
     +            /= TR_SUCCESS) THEN
C** IF FUNCTION DOES NOT COMPLETE SUCCESSFULLY THEN PRINT ERROR MESSAGE
                  PRINT *, '| ERROR IN DJACOBI'
              ENDIF
.....................................................................

C** ROUTINE FOR EXTENDET POWELL FUNCTION CALCULATION
C**   M          IN:     DIMENSION OF FUNCTION VALUE
C**   N             IN:     NUMBER OF FUNCTION VARIABLES
C**   X             IN:     VECTOR FOR FUNCTION CALCULATION
C**   F             OUT:    FUNCTION VALUE F(X)
      SUBROUTINE EXTENDET_POWELL (M, N, X, F)
        IMPLICIT NONE
        INTEGER M, N
        DOUBLE PRECISION X (*), F (*)
        INTEGER I
```

```
      DO I = 1, N/4
          F (4*I-3) = X(4*I - 3) + 10.D0 * X(4*I - 2)
          F (4*I-2) = DSQRT(5.D0) * (X(4*I-1) - X(4*I))
          F (4*I-1) = (X(4*I-2) - 2.D0*X(4*I-1))**2
          F (4*I)   = DSQRT(10.D0)*(X(4*I-3) - X(4*I))**2
      ENDDO


    ENDSUBROUTINE EXTENDET_POWELL
```

Example 14-8. djacobi Usage in C

```
/* compute jacobi matrix
   extendet_powell in:    external objective function
   n               in:    number of function variables
   m               in:    dimension of function value
   fjac            out:   jacobi matrix
```

```
   x               in:    solution vector
   jac_eps         in:    jacobi calculation precision */

    if (djacobi (extendet_powell, &n, &m, fjac, x, &jac_eps) /= TR_SUCCESS){
/* if function does not complete successfully then print error message */
                printf ("\n#ERROR IN DJACOBI\n");
                return 0;
            }

/* ..................................................................*/

/* routine for extendet powell function calculation
   m     in:     dimension of function value
   n     in:     number of function variables
   x     in:     vector for function calculation
   f     out:    function value f(x) */
void extendet_powell (int *m, int *n, double *x, double *f)
{
    int i;

    for (i = 0; i < (*n)/4; i++)
    {
       f [4*i] = x [4*i] + 10.0*x [4*i + 1];
       f [4*i + 1] = 2.2360679774*(x [4*i + 2] - x [4*i + 3]);
       f [4*i + 2] = (x [4*i + 1] - 2.0*x [4*i + 2])*(x [4*i + 1] - 2.0*x
[4*i + 2]);
       f [4*i + 3] = 3.1622776601*(x [4*i] - x [4*i + 3])*(x [4*i] - x [4*i
 + 3]);
    }
```

```
        return;
    }
```

15 Support Functions

Intel® MKL support functions are used to:

- retrieve information about the current Intel MKL version
- handle errors
- test characters and character strings for equality
- measure user time for a process and elapsed CPU time
- set and measure CPU frequency
- free memory allocated by Intel MKL memory management software.

Functions described below are subdivided according to their purpose into the following groups:

Version Information Functions

Error Handling Functions

Equality Test Functions

Timing Functions

Memory Functions

Table 15-1 contains the list of support functions common for Intel MKL.

Table 15-1 Intel MKL Support Functions

Function Name	Operation
Version Information Functions	
MKLGetVersion	Returns information about the active library version.
MKLGetVersionString	Returns information about the library version string.
Error Handling Functions	
xerbla	Handles error conditions for BLAS, LAPACK, VML routines.
pxerbla	Handles error conditions for ScaLAPACK routines.
Equality Test Functions	
lsame	Tests two characters for equality regardless of case.

Function Name	Operation
lsamen	Tests two character strings for equality regardless of case.
Timing Functions	
second/dsecnd	Returns user time for a process.
getcpuclocks	Returns full precision elapsed CPU clocks.
getcpufrequency	Returns CPU frequency value in GHz.
setcpufrequency	Sets CPU frequency value in GHz.
Memory Functions	
MKL_FreeBuffers	Frees memory buffers.

Version Information Functions

Intel® MKL provides two methods for extracting information about the library version number. First, you may extract a version string using the MKLGetVersionString function . Or, alternatively, you can use the MKLGetVersion function to obtain an MKLVersion structure that contains the version information. A makefile is also provided to automatically build the examples and output summary files containing the version information for the current library.

MKLGetVersion

Returns information about the active library version.

Syntax

```
void MKLGetVersion(MKLVersion*pVersion)
```

Description

The MKLGetVersion function collects the information about the active version of Intel MKL software and returns this information in a structure of MKLVersion type by the *pVersion* address . MKLVersion structure type is defined in mkl_types.h file. The following fields of MKLVersion structure are available:

`MajorVersion`	is the major number of the current library version.
`MinorVersion`	is the minor number of the current library version.
`BuildNumber`	is the update number of the current library version.
`ProductStatus`	is the status of the current library version. Possible variants could be “Beta”, “Product”.
`Build`	is the string that contains the build date and the internal build number.
`Processor`	is the processor optimization that is targeted for the specific processor. It is not the definition of the processor installed in the system, rather the MKL library detection that is optimal for the processor installed in the system.

Output Parameters

`pVersion`	Pointer to the `MKLVersion` structure.

MKLGetVersion Usage

```
--------------------------------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include "mkl_blas.h"
#include "mkl_types.h"

int main(void)
  {
  MKLVersion pVersion;
  MKLGetVersion(pVersion);

printf("Major version:           %d\n",pVersion->MajorVersion);
printf("Minor version:           %d\n",pVersion->MinorVersion);
printf("Update number:            %d\n",pVersion->BuildNumber);
printf("Product status:          %s\n",pVersion->ProductStatus);
printf("Build:                   %s\n",pVersion->Build);
printf("Processor optimization:  %s\n",pVersion->Processor);
printf("=================================================================\n");
printf("\n");
return 0;
  }
```

Output:

Major Version	9
Minor Version	0
Build number	0
Product status	Product
Build	061909.09

Processor opimization	Intel® Xeon® Processor with Intel® Extended Memory 64 Technology

MKLGetVersionString

Gets the library version string.

Syntax

Fortran:

```
call MKLGetVersionString( buf )
```

C:

```
MKLGetVersionString( buf,  len );
```

Output Parameters

buf	Source string
len	Length of the source string

Description

The function `MKLGetVersionString` returns a string that contains the library version information.

See example below:

Examples

Fortran:

```
program getversionstring

character*198  buf

call mklgetversionstring(buf)
write(*,'(a)') buf

end
```

C:

```
#include <stdio.h>
#include "mkl_blas.h"

int main(void)
{
  int len=198;
  char buf[198];
  MKLGetVersionString(buf, len);
  printf("%s\n",buf);
  printf("\n");
  return 0;
}
```

Error Handling Functions

xerbla

Error handling routine called by BLAS, LAPACK, VML routines.

Syntax

`call xerbla( srname, info )`

Description

The routine `xerbla` is an error handler for the BLAS, LAPACK, and VML routines. It is called by a BLAS, LAPACK, or VML routine if an input parameter has an invalid value.

A message is printed and execution stops.

Installers may consider modifying the `stop` statement in order to call system-specific exception-handling facilities.

Input Parameters

srname	CHARACTER*6. The name of the routine which called xerbla.
info	INTEGER. The position of the invalid parameter in the parameter list of the calling routine.

pxerbla

Error handling routine called by ScaLAPACK routines.

Syntax

```
call pxerbla(ictxt, srname, info)
```

Description

This routine is an error handler for the ScaLAPACK routines. It is called by a ScaLAPACK routine if an input parameter has an invalid value. A message is printed. Program execution is not terminated. For the ScaLAPACK driver and computational routines, a RETURN statement is issued following the call to pxerbla.

Control returns to the higher-level calling routine, and it is left to the user to determine how the program should proceed. However, in the specialized low-level ScaLAPACK routines (auxiliary routines that are Level 2 equivalents of computational routines), the call to pxerbla() is immediately followed by a call to BLACS_ABORT() to terminate program execution since recovery from an error at this level in the computation is not possible.

It is always good practice to check for a nonzero value of *info* on return from a ScaLAPACK routine. Installers may consider modifying this routine in order to call system-specific exception-handling facilities.

Input Parameters

ictxt	(global) INTEGER The BLACS context handle, indicating the global context of the operation. The context itself is global.
srname	(global) CHARACTER*6 The name of the routine which called pxerbla.
info	(global) INTEGER.

	The position of the invalid parameter in the parameter list of the calling routine.

Equality Test Functions

lsame

Tests two characters for equality regardless of case.

Syntax

```
val = lsame(ca, cb)
```

Description

This logical function returns `.TRUE.` if *ca* is the same letter as *cb* regardless of case.

Input Parameters

ca, *cb*	`CHARACTER*1`. Specify the single characters to be compared.

Output Parameters

val	`LOGICAL`. Result of the comparison.

lsamen

Tests two character strings for equality regardless of case.

Syntax

```
val = lsamen(n, ca, cb)
```

Description

This logical function tests if the first *n* letters of the string *ca* are the same as the first *n* letters of *cb*, regardless of case. The function `lsamen` returns `.TRUE.` if *ca* and *cb* are equivalent except for case and `.FALSE.` otherwise. `lsamen` also returns `.FALSE.` if `len`(*ca*) or `len`(*cb*) is less than *n*.

Input Parameters

n	INTEGER. The number of characters in *ca* and *cb* to be compared.
ca, *cb*	CHARACTER*(*). Specify two character strings of length at least *n* to be compared. Only the first *n* characters of each string will be accessed.

Output Parameters

val	LOGICAL. Result of the comparison.

Timing Functions

second/dsecnd

Returns elapsed CPU time in seconds.

Syntax

```
val = second()
val = dsecnd()
```

Description

The functions second/dsecnd return the elapsed CPU time in seconds. These versions get the time from the elapsed CPU clocks divided by CPU frequency. The difference is that dsecnd returns the result with double presision.

The functions should be applied in pairs: the first time, before a routine to be measured, and the second time - after the measurement. The difference between the returned values is the time spent in the routine. The usage of second is discouraged for measuring short time intervals because the single precision format is not capable of holding sufficient timer precision.

Output Parameters

val	REAL for second DOUBLE PRECISION for dsecnd Elapsed CPU time in seconds.

getcpuclocks

Returns full precision elapsed CPU clocks.

Syntax

`getcpuclocks(clocks)`

Description

The `getcpuclocks` subroutine returns the elapsed CPU clocks.

This may be useful when timing short intervals with a high resolution. The `getcpuclocks` subroutine is also applied in pairs like `second/dsecnd`. Note that out-of-order code execution on IA-32 or on Intel® 64 architecture processors may disturb the exact elapsed CPU clocks value a little bit, which may be important while measuring extremely short time intervals.

Output Parameters

clocks	`INTEGER*8`. Elapsed CPU clocks.

getcpufrequency

Returns CPU frequency value in GHz.

Syntax

`val = getcpufrequency()`

Description

The function `getcpufrequency` returns the CPU frequency in GHz. This value is used by `second /dsecnd` functions while converting CPU clocks into seconds.

Obtaining a frequency may take some time for the first time `second /dsecnd/getcpufrequency` is called. To avoid it, call `setcpufrequency` before setting the exact CPU frequency if it is known in advance.

Output Parameters

val	`DOUBLE PRECISION`. CPU frequency value in GHz.

setcpufrequency

Sets CPU frequency value in GHz.

Syntax

```
setcpufrequency(freq)
```

Description

The `setcpufrequency` subroutine sets the CPU frequency in GHz, used then by `second`/`dsecnd` functions while converting CPU clocks into seconds. Setting the exact CPU frequency is useful to bypass obtaining a frequency by `getcpufrequency`.

Initially, CPU frequency value is unset. The CPU frequency value can be set only by `setcpufrequency` call, or during the first call of `getcpufrequency`, if `setcpufrequency` has not been called before. Calling `setcpufrequency` with a special parameter *`freq`* = -1.0 forces the CPU frequency value be unset.

Output Parameters

`freq`	`DOUBLE PRECISION`. CPU frequency value in GHz.

Memory Functions

This section describes the Intel MKL function that frees memory allocated by Intel® MKL Memory Manager. See the MKL User's Guide for details of MKL memory management.

MKL_FreeBuffers

Frees memory buffers.

Syntax

```
void MKL_FreeBuffers(void)
```

Description

The function `MKL_FreeBuffers` frees the memory allocated by the MKL Memory Manager. The Memory Manager allocates new buffers if no free buffers are currently available. Call `MKL_FreeBuffers()` to free all memory buffers and to avoid memory leaking on completion of work with Intel MKL functions, that is, after the last call of an MKL function from your application.

See MKL User's Guide for details.

MKL_FreeBuffers Usage with DFT Functions

```
-----------------------------------------------------------------------------------------
DFTI_DESCRIPTOR *hand1, *hand2;
void MKL_FreeBuffers(void);
. . . . . .
/* Using MKL DFT */
Status = DftiCreateDescriptor(&hand1, DFTI_SINGLE, DFTI_COMPLEX, dim, m1);
Status = DftiCommitDescriptor(hand1);
Status = DftiComputeForward(hand1, s_array1);
. . . . . .
Status = DftiCreateDescriptor(&hand2, DFTI_SINGLE, DFTI_COMPLEX, dim, m2);
Status = DftiCommitDescriptor(hand2);
. . . . . .
Status = DftiFreeDescriptor(&hand1);
/* Do not call MKL_FreeBuffers() here as the hand2 descriptor will be
destroyed!  */
. . . . . .
Status = DftiComputeBackward(hand2, s_array2));
Status = DftiFreeDescriptor(&hand2);
/* Here user finishes the MKL DFT usage */
/* Memory leak will be triggered by any memory control tool */
/* Use MKL_FreeBuffers() to avoid memory leaking */
MKL_FreeBuffers();
-----------------------------------------------------------------------------------------
```

If the memory space is sufficient, use `MKL_FreeBuffers` after the last call of the MKL functions. Otherwise, a drop in performance can occur due to reallocation of buffers for the subsequent MKL functions.

WARNING. For FFT calls, do not use `MKL_FreeBuffers` between `DftiCreateDescriptor(hand)` and `DftiFreeDescriptor(&hand)`.

BLACS Routines 16

This chapter describes the Intel® Math Kernel Library implementation of routines from the BLACS (Basic Linear Algebra Communication Subprograms) package that are used to support a linear algebra oriented message passing interface that may be implemented efficiently and uniformly across a large range of distributed memory platforms.

The BLACS routines make linear algebra applications both easier to program and more portable. For this purpose, they are used in Intel MKL as the communication layer of ScaLAPACK and Cluster DFT.

These routines perform distinct tasks that can be used for:

Initialization

Destruction

Information Purposes

Miscellaneous Tasks.

Initialization Routines

This section describes BLACS routines that deal with grid/context creation, and processing before the grid/context has been defined.

Table 16-1 BLACS Intialization Routines

Routine name	Operation performed
blacs_pinfo	Returns the number of processes available for use.
blacs_setup	Allocates virtual machine and spawns processes.
blacs_get	Gets values that BLACS use for internal defaults.
blacs_set	Sets values that BLACS use for internal defaults.
blacs_gridinit	Assigns available processes into BLACS process grid.
blacs_gridmap	Maps available processes into BLACS process grid.

blacs_pinfo

Returns the number of processes available for use.

Syntax

```
call blacs_pinfo( mypnum, nprocs )
```

Output Parameters

mypnum	INTEGER. An integer between 0 and (*nprocs* - 1) that uniquely identifies each process.
nprocs	INTEGER.The number of processes available for BLACS use.

Description

This routine is used when some initial system information is required before the BLACS are set up. On all platforms except PVM, *nprocs* is the actual number of processes available for use, that is, *nprows * npcols <= nprocs*. In PVM, the virtual machine may not have been set up before this call, and therefore no parallel machine exists. In this case, *nprocs* is returned as less than one. If a process has been spawned via the keyboard, it receives *mypnum* of 0, and all other processes get *mypnum* of -1. As a result, the user can distinguish between processes. Only after the virtual machine has been set up via a call to BLACS_SETUP, this routine returns the correct values for *mypnum* and *nprocs*.

blacs_setup

Allocates virtual machine and spawns processes.

Syntax

```
call blacs_setup( mypnum, nprocs )
```

Input Parameters

nprocs	INTEGER. On the process spawned from the keyboard rather than from pvmspawn, this parameter indicates the number of processes to create when building the virtual machine.

Output Parameters

mypnum	INTEGER. An integer between 0 and (*nprocs* - 1) that uniquely identifies each process.
nprocs	INTEGER. For all processes other than spawned from the keyboard, this parameter means the number of processes available for BLACS use.

Description

This routine only accomplishes meaningful work in the PVM BLACS. On all other platforms, it is functionally equivalent to blacs_pinfo. The BLACS assume a static system, that is, the given number of processes does not change. PVM supplies a dynamic system, allowing processes to be added to the system on the fly.

blacs_setup is used to allocate the virtual machine and spawn off processes. It reads in a file called blacs_setup.dat, in which the first line must be the name of your executable. The second line is optional, but if it exists, it should be a PVM spawn flag. Legal values at this time are 0 (PvmTaskDefault), 4 (PvmTaskDebug), 8 (PvmTaskTrace), and 12 (PvmTaskDebug + PvmTaskTrace). The primary reason for this line is to allow the user to easily turn on and off PVM debugging. Additional lines, if any, specify what machines should be added to the current configuration before spawning *nprocs*-1 processes to the machines in a round robin fashion.

nprocs is input on the process which has no PVM parent (that is, *mypnum*=0), and both parameters are output for all processes. So, on PVM systems, the call to blacs_pinfo informs you that the virtual machine has not been set up, and a call to blacs_setup then sets up the machine and returns the real values for *mypnum* and *nprocs*.

Note that if the file blacs_setup.dat does not exist, the BLACS prompt the user for the executable name, and processes are spawned to the current PVM configuration.

blacs_get

Gets values that BLACS use for internal defaults.

Syntax

```
call blacs_get( icontxt, what, val )
```

Input Parameters

icontxt	INTEGER. On values of *what* that are tied to a particular context, this parameter is the integer handle indicating the context. Otherwise, ignored.
what	INTEGER. Indicates what BLACS internal(s) should be returned in *val*. Present options are: • *what* = 0 : Handle indicating default system context • *what* = 1 : The BLACS message ID range • *what* = 2 : The BLACS debug level the library was compiled with • *what* = 10 : Handle indicating the system context used to define the BLACS context whose handle is *icontxt* • *what* = 11 : Number of rings multiring topology is presently using • *what* = 12 : Number of branches general tree topology is presently using.

Output Parameters

val	INTEGER. The value of the BLACS internal.

Description

This routine gets the values that the BLACS are using for internal defaults. Some values are tied to a BLACS context, and some are more general. The most common use is in retrieving a default system context for input into blacs_gridinit or blacs_gridmap.

Some systems, such as MPI*, supply their own version of context. For those users who mix system code with BLACS code, a BLACS context should be formed in reference to a system context. Thus, the grid creation routines take a system context as input. If you wish to have strictly portable code, you may use blacs_get to retrieve a default system context that will include all available processes. This value is not tied to a BLACS context, so the parameter *icontxt* is unused.

blacs_get returns information on three quantities that are tied to an individual BLACS context, which is passed in as *icontxt*. The information that may be retrieved is:

- The handle of the system context upon which this BLACS context was defined

- The number of rings for `TOP = 'M'` (multiring broadcast)
- The number of branches for `TOP = 'T'` (general tree broadcast/general tree gather).

blacs_set

Sets values that BLACS use for internal defaults.

Syntax

```
call blacs_set( icontxt, what, val )
```

Input Parameters

icontxt — `INTEGER`. On values of *what* that are tied to a particular context, this parameter is the integer handle indicating the context. Otherwise, ignored.

what — `INTEGER`. Indicates what BLACS internal(s) should be set. Present values are:

- 1 = The BLACS message ID range
- 11 = Number of rings for multiring topology to use
- 12 = Number of branches for general tree topology to use.

val — `INTEGER`. Array of dimension (*).Indicates the value(s) the internals should be set to. The specific meanings depend on *what* values, as discussed in Description below.

Description

This routine sets the BLACS internal defaults depending on the *what* values:

what = 1 — Setting the BLACS message ID range.
If you wish to mix the BLACS with other message-passing packages, restrict the BLACS to a certain message ID range not to used by the non-BLACS routines. The message ID range must be set before the first call to `blacs_gridinit` or `blacs_gridmap`. Subsequent calls will have no effect. Because the message ID range is not tied to a particular context, the parameter *icontxt* is ignored, and *val* is defined as:
`VAL (input) INTEGER` array of dimension (2)

VAL(1) : The smallest message ID (also called message type or message tag) the BLACS should use.
VAL(2) : The largest message ID (also called message type or message tag) the BLACS should use.

what = 11 — Set number of rings for TOP = 'M' (multiring broadcast).This quantity is tied to a context, so *icontxt* is used, and *val* is defined as:
VAL (input) INTEGER array of dimension (1)
VAL(1) : The number of rings for multiring topology to use.

what = 12 — Set number of rings for TOP = 'T' (general tree broadcast/general tree gather). This quantity is tied to a context, so *icontxt* is used, and *val* is defined as:
VAL (input) INTEGER array of dimension (1)
VAL(1) : The number branches for general tree topology to use.

blacs_gridinit

Assigns available processes into BLACS process grid.

Syntax

```
call blacs_gridinit( icontxt, order, nprow, npcol )
```

Input Parameters

icontxt — INTEGER. Integer handle indicating the system context to be used in creating the BLACS context. Call blacs_get to obtain a default system context.

order — CHARACTER*1. Indicates how to map processes to BLACS grid. Options are:

- 'R' : Use row-major natural ordering
- 'C' : Use column-major natural ordering
- ELSE : Use row-major natural ordering

nprow — INTEGER. Indicates how many process rows the process grid should contain.

npcol — INTEGER. Indicates how many process columns the process grid should contain.

Output Parameters

icontxt	INTEGER. Integer handle to the created BLACS context.

Description

All BLACS codes must call this routine, or its sister routine `blacs_gridmap`. These routines take the available processes, and assign, or map, them into a BLACS process grid. In other words, they establish how the BLACS coordinate system maps into the native machine process numbering system. Each BLACS grid is contained in a context, so that it does not interfere with distributed operations that occur within other grids/contexts. These grid creation routines may be called repeatedly to define additional contexts/grids.

The creation of a grid requires input from all processes that are defined to be in this grid. Processes belonging to more than one grid have to agree on which grid formation will be serviced first, much like the globally blocking sum or broadcast.

These grid creation routines set up various internals for the BLACS, and one of them must be called before any calls are made to the non-initialization BLACS.

Note that these routines map already existing processes to a grid: the processes are not created dynamically. On most parallel machines, the processes are actual processors (hardware), and they are "created" when you run your executable. When using the PVM BLACS, if the virtual machine has not been set up yet, the routine `blacs_setup` should be used to create the virtual machine.

This routine creates a simple `nprow x npcol` process grid. This process grid uses the first `nprow * npcol` processes, and assigns them to the grid in a row- or column-major natural ordering. If these process-to-grid mappings are unacceptable, call `blacs_gridmap`.

blacs_gridmap

Maps available processes into BLACS process grid.

Syntax

```
call blacs_gridmap( icontxt, usermap, ldumap, nprow, npcol )
```

Input Parameters

icontxt	INTEGER. Integer handle indicating the system context to be used in creating the BLACS context. Call `blacs_get` to obtain a default system context.

usermap	INTEGER. Array, dimension (*ldumap*, *npcol*), indicating the process-to-grid mapping.
ldumap	INTEGER. Leading dimension of the 2D array *usermap*. *ldumap* ≥ *nprow*.
nprow	INTEGER. Indicates how many process rows the process grid should contain.
npcol	INTEGER. Indicates how many process columns the process grid should contain.

Output Parameters

icontxt	INTEGER. Integer handle to the created BLACS context.

Description

All BLACS codes must call this routine, or its sister routine `blacs_gridinit`. These routines take the available processes, and assign, or map, them into a BLACS process grid. In other words, they establish how the BLACS coordinate system maps into the native machine process numbering system. Each BLACS grid is contained in a context, so that it does not interfere with distributed operations that occur within other grids/contexts. These grid creation routines may be called repeatedly to define additional contexts/grids.

The creation of a grid requires input from all processes that are defined to be in this grid. Processes belonging to more than one grid have to agree on which grid formation will be serviced first, much like the globally blocking sum or broadcast.

These grid creation routines set up various internals for the BLACS, and one of them must be called before any calls are made to the non-initialization BLACS.

Note that these routines map already existing processes to a grid: the processes are not created dynamically. On most parallel machines, the processes are actual processors (hardware), and they are "created" when you run your executable. When using the PVM BLACS, if the virtual machine has not been set up yet, the routine `blacs_setup` should be used to create the virtual machine.

This routine allows the user to map processes to the process grid in an arbitrary manner. `usermap(i,j)` holds the process number of the process to be placed in `{i, j}` of the process grid. On most distributed systems, this process number is a machine defined number between 0 ... *nprow*-1. For PVM, these node numbers are the PVM TIDS (Task IDs). The `blacs_gridmap` routine is intended for an experienced user. The `blacs_gridinit` routine is much simpler. `blacs_gridinit` simply performs a `gridmap` where the first `nprow * npcol` processes are mapped into the current grid in a row-major natural ordering. If you are an an experienced

user, blacs_gridmap allows you to take advantage of your system's actual layout. That is, you can map nodes that are physically connected to be neighbors in the BLACS grid, etc. The blacs_gridmap routine also opens the way for multigridding: you can separate your nodes into arbitrary grids, join them together at some later date, and then re-split them into new grids. blacs_gridmap also provides the ability to make arbitrary grids or subgrids (for example, a "nearest neighbor" grid), which can greatly facilitate operations among processes that do not fall on a row or column of the main process grid.

Destruction Routines

This section describes BLACS routines that destroy grids, abort processes, and free resources.

Table 16-2 BLACS Destruction Routines

Routine name	Operation performed
blacs_freebuff	Frees BLACS buffer.
blacs_gridexit	Frees a BLACS context.
blacs_abort	Aborts all processes.
blacs_exit	Frees all BLACS contexts and releases all allocated memory.

blacs_freebuff

Frees BLACS buffer.

Syntax

```
call blacs_freebuff( icontxt, wait )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the BLACS context.
wait	INTEGER. Parameter indicating whether to wait for non-blocking operations or not. If equals 0, the operations should not be waited for; free only unused buffers. Otherwise, wait in order to free all buffers.

Description

This routine releases the BLACS buffer.

The BLACS have at least one internal buffer that is used for packing messages. The number of internal buffers depends on what platform you are running the BLACS on. On systems where memory is tight, keeping this buffer or buffers may become expensive. Call `freebuff` to release the buffer. However, the next call of a communication routine that requires packing reallocates the buffer.

The *wait* parameter determines whether the BLACS should wait for any non-blocking operations to be completed or not. If *wait* = 0, the BLACS free any buffers that can be freed without waiting. If *wait* is not 0, the BLACS free all internal buffers, even if non-blocking operations must be completed first.

blacs_gridexit

Frees a BLACS context.

Syntax

```
call blacs_gridexit( icontxt )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the BLACS context to be freed.

Description

This routine frees a BLACS context.

Release the resources when contexts are no longer needed. After freeing a context, the context no longer exists, and its handle may be re-used if new contexts are defined.

blacs_abort

Aborts all processes.

Syntax

```
call blacs_abort( icontxt, errornum )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the BLACS context to be aborted.
errornum	INTEGER. User-defined integer error number.

Description

This routine aborts all the BLACS processes, not only those confined to a particular context.

Use blacs_abort to abort all the processes in case of a serious error. Note that both parameters are input, but the routine uses them only in printing out the error message. The context handle passed in is not required to be a valid context handle.

blacs_exit

Frees all BLACS contexts and releases all allocated memory.

Syntax

```
call blacs_exit( continue )
```

Input Parameters

continue	INTEGER. Flag indicating whether message passing continues after the BLACS are done. If *continue* is non-zero, the user is assumed to continue using the machine after completing the BLACS. Otherwise, no message passing is assumed after calling this routine.

Description

This routine frees all BLACS contexts and releases all allocated memory.

This routine should be called when a process has finished all use of the BLACS. The *continue* parameter indicates whether the user will be using the underlying communication platform after the BLACS are finished. This information is most important for the PVM BLACS. If *continue* is set to 0, then pvm_exit is called; otherwise, it is not called. Setting *continue* not equal to 0 indicates that explicit PVM send/recvs will be called after the BLACS are done. Make sure your code calls pvm_exit. PVM users should either call blacs_exit or explicitly call pvm_exit to avoid PVM problems.

Informational Routines

This section describes BLACS routines that return information involving the process grid.

Table 16-3 BLACS Informational Routines

Routine name	Operation performed
blacs_gridinfo	Returns information on the current grid.
blacs_pnum	Returns the system process number of the process in the process grid.
blacs_pcoord	Returns the row and column coordinates in the process grid.

blacs_gridinfo

Returns information on the current grid.

Syntax

```
call blacs_gridinfo( icontxt, nprow, npcol, myprow, mypcol )
```

Input Parameters

icontxt INTEGER. Integer handle that indicates the context.

Output Parameters

nprow INTEGER. Number of process rows in the current process grid.

npcol INTEGER. Number of process columns in the current process grid.

myprow INTEGER. Row coordinate of the calling process in the process grid.

mypcol INTEGER. Column coordinate of the calling process in the process grid.

Description

This routine returns information on the current grid. If the context handle does not point at a valid context, all quantities are returned as -1.

blacs_pnum

Returns the system process number of the process in the process grid.

Syntax

```
call blacs_pnum( icontxt, prow, pcol )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the context.
prow	INTEGER. Row coordinate of the process the system process number of which is to be determined.
pcol	INTEGER. Column coordinate of the process the system process number of which is to be determined.

Description

This function returns the system process number of the process at {PROW, PCOL} in the process grid.

blacs_pcoord

Returns the row and column coordinates in the process grid.

Syntax

```
call blacs_pcoord( icontxt, pnum, prow, pcol )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the context.
pnum	INTEGER. Process number the coordinates of which are to be determined. This parameter stand for the process number of the underlying machine, that is, it is a tid for PVM.

Output Parameters

prow	INTEGER. Row coordinates of the *pnum* process in the BLACS grid.
pcol	INTEGER. Column coordinates of the *pnum* process in the BLACS grid.

Description

Given the system process number, this function returns the row and column coordinates in the BLACS process grid.

Miscellaneous Routines

This section describes blacs_barrier routine.

Table 16-4 BLACS Informational Routines

Routine name	Operation performed
blacs_barrier	Holds up execution of all processes within the indicated scope until they have all called the routine.

blacs_barrier

Holds up execution of all processes within the indicated scope.

Syntax

```
call blacs_barrier( icontxt, scope )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the context.
scope	CHARACTER*1. Parameter that indicates whether a process row (*scope*='R'), column ('C'), or entire grid ('A') will participate in the barrier.

Description

This routine holds up execution of all processes within the indicated scope until they have all called the routine.

Examples of BLACS Routines Usage

Example 16-1. BLACS Usage. Hello World

The following routine takes the available processes, forms them into a process grid, and then has each process check in with the process at {0,0} in the process grid.

```
      PROGRAM HELLO
*     -- BLACS example code --
*     Written by Clint Whaley 7/26/94
*     Performs a simple check-in type hello world
*     ..
*     .. External Functions ..
      INTEGER BLACS_PNUM
      EXTERNAL BLACS_PNUM
*     ..
*     .. Variable Declaration ..
      INTEGER CONTXT, IAM, NPROCS, NPROW, NPCOL, MYPROW, MYPCOL
      INTEGER ICALLER, I, J, HISROW, HISCOL
*
*     Determine my process number and the number of processes in
*     machine
*
      CALL BLACS_PINFO(IAM, NPROCS)
*
*     If in PVM, create virtual machine if it doesn't exist
*
      IF (NPROCS .LT. 1) THEN
         IF (IAM .EQ. 0) THEN
            WRITE(*, 1000)
```

```
            READ(*, 2000) NPROCS
         END IF
         CALL BLACS_SETUP(IAM, NPROCS)
      END IF
*
*     Set up process grid that is as close to square as possible
*
      NPROW = INT( SQRT( REAL(NPROCS) ) )
      NPCOL = NPROCS / NPROW
*
*     Get default system context, and define grid
*
      CALL BLACS_GET(0, 0, CONTXT)
      CALL BLACS_GRIDINIT(CONTXT, 'Row', NPROW, NPCOL)
      CALL BLACS_GRIDINFO(CONTXT, NPROW, NPCOL, MYPROW, MYPCOL)
*
*     If I'm not in grid, go to end of program
*
      IF ( (MYPROW.GE.NPROW) .OR. (MYPCOL.GE.NPCOL) ) GOTO 30
*
*     Get my process ID from my grid coordinates
*
      ICALLER = BLACS_PNUM(CONTXT, MYPROW, MYPCOL)
*
*     If I am process {0,0}, receive check-in messages from
*     all nodes
*
      IF ( (MYPROW.EQ.0) .AND. (MYPCOL.EQ.0) ) THEN
```

```
         WRITE(*,*) ' '
         DO 20 I = 0, NPROW-1
            DO 10 J = 0, NPCOL-1

               IF ( (I.NE.0) .OR. (J.NE.0) ) THEN
                  CALL IGERV2D(CONTXT, 1, 1, ICALLER, 1, I, J)
               END IF
*
*              Make sure ICALLER is where we think in process grid
*
              CALL BLACS_PCOORD(CONTXT, ICALLER, HISROW, HISCOL)
              IF ( (HISROW.NE.I) .OR. (HISCOL.NE.J) ) THEN
                 WRITE(*,*) 'Grid error!  Halting . . .'
                 STOP
              END IF
              WRITE(*, 3000) I, J, ICALLER

10          CONTINUE
20       CONTINUE
         WRITE(*,*) ' '
         WRITE(*,*) 'All processes checked in.  Run finished.'
*
*     All processes but {0,0} send process ID as a check-in
*
      ELSE
         CALL IGESD2D(CONTXT, 1, 1, ICALLER, 1, 0, 0)
```

```
      END IF

30    CONTINUE


      CALL BLACS_EXIT(0)
1000  FORMAT('How many processes in machine?')
2000  FORMAT(I)
3000  FORMAT('Process {',i2,',',i2,'} (node number =',I,
     $       ') has checked in.')

      STOP
      END
```

Example 16-2. BLACS Usage. PROCMAP

This routine maps processes to a grid using `blacs_gridmap`.

```
      SUBROUTINE PROCMAP(CONTEXT, MAPPING, BEGPROC, NPROW, NPCOL, IMAP)
*
*     -- BLACS example code --
*     Written by Clint Whaley 7/26/94
*     ..
*     .. Scalar Arguments ..
      INTEGER CONTEXT, MAPPING, BEGPROC, NPROW, NPCOL
*     ..
*     .. Array Arguments ..
      INTEGER IMAP(NPROW, *)
*     ..
*
*  Purpose
*  =======
*  PROCMAP maps NPROW*NPCOL processes starting from process BEGPROC to
*  the grid in a variety of ways depending on the parameter MAPPING.
*
*  Arguments
*  =========
*
*  CONTEXT      (output) INTEGER
*               This integer is used by the BLACS to indicate a context.
*               A context is a universe where messages exist and do not
*               interact with other context's messages.  The context
*               includes the definition of a grid, and each process's
```

```
*              coordinates in it.
*
*  MAPPING     (intput) INTEGER
*              Way to map processes to grid.  Choices are:
*              1 : row-major natural ordering
*              2 : column-major natural ordering
*
*  BEGPROC     (input) INTEGER
*              The process number (between 0 and NPROCS-1) to use as
*              {0,0}. From this process, processes will be assigned
*              to the grid as indicated by MAPPING.
*
*  NPROW       (input) INTEGER
*              The number of process rows the created grid
*              should have.
*
*  NPCOL       (input) INTEGER
*              The number of process columns the created grid
*              should have.
*
*  IMAP        (workspace) INTEGER array of dimension (NPROW, NPCOL)
*              Workspace, where the array which maps the
*              processes to the grid will be stored for the
*              call to GRIDMAP.
```

```
*
*     ============================================================
*
*     ..
*     .. External Functions ..
      INTEGER  BLACS_PNUM
      EXTERNAL BLACS_PNUM
*     ..
*     .. External Subroutines ..
      EXTERNAL BLACS_PINFO, BLACS_GRIDINIT, BLACS_GRIDMAP
*     ..
*     .. Local Scalars ..
      INTEGER TMPCONTXT, NPROCS, I, J, K
*     ..
*     .. Executable Statements ..
*
*     See how many processes there are in the system
*
      CALL BLACS_PINFO( I, NPROCS )
      IF (NPROCS-BEGPROC .LT. NPROW*NPCOL) THEN
         WRITE(*,*) 'Not enough processes for grid'
         STOP
      END IF
*
*     Temporarily map all processes into 1 x NPROCS grid

*
      CALL BLACS_GET( 0, 0, TMPCONTXT )
      CALL BLACS_GRIDINIT( TMPCONTXT, 'Row', 1, NPROCS )
```

```
      K = BEGPROC
*
*     If we want a row-major natural ordering
*
      IF (MAPPING .EQ. 1) THEN
         DO I = 1, NPROW
            DO J = 1, NPCOL
               IMAP(I, J) = BLACS_PNUM(TMPCONTXT, 0, K)
               K = K + 1W
            END DO
         END DO
*
*     If we want a column-major natural ordering
*
      ELSE IF (MAPPING .EQ. 2) THEN
         DO J = 1, NPCOL
            DO I = 1, NPROW
               IMAP(I, J) = BLACS_PNUM(TMPCONTXT, 0, K)
               K = K + 1
            END DO
         END DO
      ELSE
         WRITE(*,*) 'Uknown mapping.'
         STOP
      END IF
*
*     Free temporary context
```

```
*
      CALL BLACS_GRIDEXIT(TMPCONTXT)
*
*     Apply the new mapping to form desired context
*
      CALL BLACS_GET( 0, 0, CONTEXT )
      CALL BLACS_GRIDMAP( CONTEXT, IMAP, NPROW, NPROW, NPCOL )


      RETURN
      END
```

Linear Solvers Basics

Many applications in science and engineering require the solution of a system of linear equations. This problem is usually expressed mathematically by the matrix-vector equation, $Ax = b$, where A is an m-by-n matrix, x is the n element column vector and b is the m element column vector. The matrix A is usually referred to as the coefficient matrix, and the vectors x and b are referred to as the solution vector and the right-hand side, respectively.

Basic concepts related to solving linear systems with sparse matrices are described in section Sparse Linear Systems that follows.

If the coefficients in matrix A and right-hand sides in vector b are not defined exactly but rather belong to known intervals, the system is called an `interval linear system`. Some basic definitions and concepts used in solving interval linear systems are described in Interval Linear Systems section below.

Sparse Linear Systems

In many real-life applications, most of the elements in A are zero. Such a matrix is referred to as sparse. Conversely, matrices with very few zero elements are called dense. For sparse matrices, computing the solution to the equation $Ax = b$ can be made much more efficient with respect to both storage and computation time, if the sparsity of the matrix can be exploited. The more an algorithm can exploit the sparsity without sacrificing the correctness, the better the algorithm.

Generally speaking, computer software that finds solutions to systems of linear equations is called a solver. A solver designed to work specifically on sparse systems of equations is called a sparse solver. Solvers are usually classified into two groups - direct and iterative.

Iterative Solvers start with an initial approximation to a solution and attempt to estimate the difference between the approximation and the true result. Based on the difference, an iterative solver calculates a new approximation that is closer to the true result than the initial approximation. This process is repeated until the difference between the approximation and the true result is sufficiently small. The main drawback to iterative solvers is that the rate of convergence depends greatly on the values in the matrix A. Consequently, it is not possible to predict how long it will take for an iterative solver to produce a solution. In fact, for ill-conditioned matrices, the iterative process will not converge to a solution at all. However, for well-conditioned matrices it is possible for iterative solvers to converge to a solution very quickly. Consequently for the right applications, iterative solvers can be very efficient.

Direct Solvers, on the other hand, often factor the matrix A into the product of two triangular matrices and then perform a forward and backward triangular solve.

This approach makes the time required to solve a systems of linear equations relatively predictable, based on the size of the matrix. In fact, for sparse matrices, the solution time can be predicted based on the number of non-zero elements in the array *A*.

Matrix Fundamentals

A matrix is a rectangular array of either real or complex numbers. A matrix is denoted by a capital letter; its elements are denoted by the same lower case letter with row/column subscripts. Thus, the value of the element in row *i* and column *j* in matrix *A* is denoted by *a*(*i*,*j*). For example, a 3 by 4 matrix *A*, is written as follows:

$$A = \begin{bmatrix} a(1,1) & a(1,2) & a(1,3) & a(1,4) \\ a(2,1) & a(2,2) & a(2,3) & a(2,4) \\ a(3,1) & a(3,2) & a(3,3) & a(3,4) \end{bmatrix}$$

Note that with the above notation, we assume the standard Fortran programming language convention of starting array indices at 1 rather than the C programming language convention of starting them at 0.

A matrix in which all of the elements are real numbers is called a real matrix. A matrix that contains at least one complex number is called a complex matrix. A real or complex matrix *A* with the property that *a*(*i*,*j*) = *a*(*j*,*i*), is called a symmetric matrix. A complex matrix *A* with the property that *a*(*i*,*j*) = *conj*(*a*(*j*,*i*)), is called a Hermitian matrix. Note that programs that manipulate symmetric and Hermitian matrices need only store half of the matrix values, since the values of the non-stored elements can be quickly reconstructed from the stored values.

A matrix that has the same number of rows as it has columns is referred to as a square matrix. The elements in a square matrix that have same row index and column index are called the diagonal elements of the matrix, or simply the diagonal of the matrix.

The transpose of a matrix *A* is the matrix obtained by "flipping" the elements of the array about its diagonal. That is, we exchange the elements *a*(*i*,*j*) and *a*(*j*,*i*). For a complex matrix, if we both flip the elements about the diagonal and then take the complex conjugate of the element, the resulting matrix is called the Hermitian transpose or conjugate transpose of the original matrix. The transpose and Hermitian transpose of a matrix *A* are denoted by A^T and A^H respectively.

A column vector, or simply a vector, is a $n \times 1$ matrix, and a row vector is a $1 \times n$ matrix. A real or complex matrix A is said to be positive definite if the vector-matrix product $x^T Ax$ is greater than zero for all non-zero vectors x. A matrix that is not positive definite is referred to as indefinite.

An upper (or lower) triangular matrix, is a square matrix in which all elements below (or above) the diagonal are zero. A unit triangular matrix is an upper or lower triangular matrix with all 1's along the diagonal.

A matrix P is called a permutation matrix if, for any matrix A, the result of the matrix product PA is identical to A except for interchanging the rows of A. For a square matrix, it can be shown that if PA is a permutation of the rows of A, then AP^T is the same permutation of the columns of A. Additionally, it can be shown that the inverse of P is P^T.

In order to save space, a permutation matrix is usually stored as a linear array, called a permutation vector, rather than as an array. Specifically, if the permutation matrix maps the i-th row of a matrix to the j-th row, then the i-th element of the permutation vector is j.

A matrix with non-zero elements only on the diagonal is called a diagonal matrix. As is the case with a permutation matrix, it is usually stored as a vector of values, rather than as a matrix.

Direct Method

For solvers that use the direct method, the basic technique employed in finding the solution of the system $Ax = b$ is to first factor A into triangular matrices. That is, find a lower triangular matrix L and an upper triangular matrix U, such that $A = LU$. Having obtained such a factorization (usually referred to as an LU decomposition or LU factorization), the solution to the original problem can be rewritten as follows.

$$Ax = b$$
$$\Rightarrow \quad LUx = b$$
$$\Rightarrow \quad (Ux) = b$$

This leads to the following two-step process for finding the solution to the original system of equations:

1. Solve the systems of equations $Ly = b$.
2. Solve the system $Ux = y$.

Solving the systems $Ly = b$ and $Ux = y$ is referred to as a forward solve and a backward solve, respectively.

If a symmetric matrix A is also positive definite, it can be shown that A can be factored as LL^T where L is a lower triangular matrix. Similarly, a Hermitian matrix, A, that is positive definite can be factored as $A = LL^H$. For both symmetric and Hermitian matrices, a factorization of this form is called a Cholesky factorization.

In a Cholesky factorization, the matrix U in an LU decomposition is either L^T or L^H. Consequently, a solver can increase its efficiency by only storing L, and one-half of A, and not computing U. Therefore, users who can express their application as the solution of a system of positive definite equations will gain a significant performance improvement over using a general representation.

For matrices that are symmetric (or Hermitian) but not positive definite, there are still some significant efficiencies to be had. It can be shown that if A is symmetric but not positive definite, then A can be factored as $A = LDL^T$, where D is a diagonal matrix and L is a lower unit triangular matrix. Similarly, if A is Hermitian, it can be factored as $A = LDL^H$. In either case, we again only need to store L, D, and half of A and we need not compute U. However, the backward solve phases must be amended to solving $L^Tx = D^{-1}y$ rather than $L^Tx = y$.

Fill-In and Reordering of Sparse Matrices

Two important concepts associated with the solution of sparse systems of equations are fill-in and reordering. The following example illustrates these concepts.

Consider the system of linear equation $Ax = b$, where A is the symmetric positive definite sparse matrix defined by the following:

$$A = \begin{bmatrix} 9 & \frac{3}{2} & 6 & \frac{3}{4} & 3 \\ \frac{3}{2} & \frac{1}{2} & * & * & * \\ 6 & * & 12 & * & * \\ \frac{3}{4} & * & \frac{5}{8} & * & * \\ 3 & * & * & * & 16 \end{bmatrix} b \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

A star (*) is used to represent zeros and to emphasize the sparsity of A. The Cholesky factorization of A is: $A = LL^T$, where L is the following:

$$L = \begin{bmatrix} 3 & * & * & * & * \\ \frac{1}{2} & \frac{1}{2} & * & * & * \\ 2 & -2 & 2 & * & * \\ \frac{1}{4} & \frac{1}{-4} & \frac{1}{-2} & \frac{1}{2} & * \\ 1 & -1 & -2 & -3 & 1 \end{bmatrix}$$

Notice that even though the matrix A is relatively sparse, the lower triangular matrix L has no zeros below the diagonal. If we computed L and then used it for the forward and backward solve phase, we would do as much computation as if A had been dense.

The situation of L having non-zeros in places where A has zeros is referred to as fill-in. Computationally, it would be more efficient if a solver could exploit the non-zero structure of A in such a way as to reduce the fill-in when computing L. By doing this, the solver would only need to compute the non-zero entries in L. Toward this end, consider permuting the rows and columns of A. As described in Matrix Fundamentals section , the permutations of the rows of A can be represented as a permutation matrix, P. The result of permuting the rows is the product of P and A. Suppose, in the above example, we swap the first and fifth row of A, then swap the first and fifth columns of A, and call the resulting matrix B. Mathematically, we can express the process of permuting the rows and columns of A to get B as $B = PAP^T$. After permuting the rows and columns of A, we see that B is given by the following:

$$B = \begin{bmatrix} 16 & * & * & * & 3 \\ * & \frac{1}{2} & * & * & \frac{3}{2} \\ * & * & 12 & * & 6 \\ * & * & * & \frac{5}{8} & \frac{3}{4} \\ 3 & \frac{3}{2} & 6 & \frac{3}{4} & 9 \end{bmatrix}$$

Since B is obtained from A by simply switching rows and columns, the numbers of non-zero entries in A and B are the same. However, when we find the Cholesky factorization, $B = LL^T$, we see the following:

$$L = \begin{bmatrix} 4 & * & * & * & * \\ * & \frac{1}{\sqrt{2}} & * & * & * \\ * & * & 2(\sqrt{3}) & * & * \\ * & * & * & \frac{\sqrt{10}}{4} & * \\ \frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{\frac{3}{5}}}{4} \end{bmatrix}$$

The fill-in associated with B is much smaller than the fill-in associated with A. Consequently, the storage and computation time needed to factor B is much smaller than to factor A. Based on this, we see that an efficient sparse solver needs to find permutation P of the matrix A, which minimizes the fill-in for factoring $B = PAP^T$, and then use the factorization of B to solve the original system of equations.

Although the above example is based on a symmetric positive definite matrix and a Cholesky decomposition, the same approach works for a general LU decomposition. Specifically, let P be a permutation matrix, $B = PAP^T$ and suppose that B can be factored as $B = LU$. Then

$Ax = b$

$\Rightarrow \quad PA(P^{-1}P)x = Pb$

$\Rightarrow \quad PA(P^TP)x = Pb$

$\Rightarrow \quad (PAP^T)(Px) = Pb$

$\Rightarrow \quad B(Px) = Pb$

$\Rightarrow \quad LU(Px) = Pb$

It follows that if we obtain an LU factorization for B, we can solve the original system of equations by a three step process:

1. Solve $Ly = Pb$.

2. Solve $Uz = y$.

3. Set $x = P^Tz$.

If we apply this three-step process to the current example, we first need to perform the forward solve of the systems of equation $Ly = Pb$:

$$L = \begin{bmatrix} 4 & * & * & * & * \\ * & \frac{1}{\sqrt{2}} & * & * & * \\ * & * & 2(\sqrt{3}) & * & * \\ * & * & * & \frac{\sqrt{10}}{4} & * \\ \frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{\frac{3}{5}}}{4} \end{bmatrix} * \begin{bmatrix} y1 \\ y2 \\ y3 \\ y4 \\ y5 \end{bmatrix} = \begin{bmatrix} 5 \\ 2 \\ 3 \\ 4 \\ 1 \end{bmatrix}$$

This gives:

$$y^T = \frac{5}{4}, 2\sqrt{2}, \frac{\sqrt{3}}{2}, \frac{16}{\sqrt{10}}, \frac{-979\sqrt{\frac{3}{5}}}{12}.$$

The second step is to perform the backward solve, $Uz = y$. Or, in this case, since we are using a Cholesky factorization, $L^T z = y$.

$$\begin{bmatrix} 4 & * & * & * & \frac{3}{4} \\ * & \frac{1}{\sqrt{2}} & * & * & \frac{3}{\sqrt{2}} \\ * & * & 2(\sqrt{3}) & * & \sqrt{3} \\ * & * & * & \frac{\sqrt{10}}{4} & \frac{3}{\sqrt{10}} \\ \frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{\frac{3}{5}}}{4} \end{bmatrix} * \begin{bmatrix} z1 \\ z2 \\ z3 \\ z4 \\ z5 \end{bmatrix} = \begin{bmatrix} \frac{5}{4} \\ 2(\sqrt{2}) \\ \frac{\sqrt{3}}{2} \\ \frac{16}{\sqrt{10}} \\ \frac{-979\sqrt{\frac{3}{5}}}{12} \end{bmatrix}$$

This gives

$$z = \frac{123}{2}, 983, \frac{1961}{12}, 398, \frac{-979}{3}.$$

The third and final step is to set $x = P^T z$. This gives

$$X^T = \frac{-979}{3}, 983, \frac{1961}{12}, 398, \frac{123}{2}.$$

Sparse Matrix Storage Formats

As discussed above, it is more efficient to store only the non-zeros of a sparse matrix. This assumes that the sparsity is large, that is, the number of non-zero entries is a small percentage of the total number of entries. If there is only an occasional zero entry, the cost of exploiting the sparsity actually slows down the computation when compared to simply treating the matrix as dense, meaning that all the values, zero and non-zero, are used in the computation.

There are a number of common storage schemes used for sparse matrices, but most of the schemes employ the same basic technique. That is, compress all of the non-zero elements of the matrix into a linear array, and then provide some number of auxiliary arrays to describe the locations of the non-zeros in the original matrix.

Storage Formats for the PARDISO Solver

The compression of the non-zeros of a sparse matrix *A* into a linear array is done by walking down each column (column major format) or across each row (row major format) in order, and writing the non-zero elements to a linear array in the order that they appear in the walk.

When storing symmetric matrices, it is necessary to store only the upper triangular half of the matrix (upper triangular format) or the lower triangular half of the matrix (lower triangular format).

The Intel MKL direct sparse solver uses a row major upper triangular storage format. That is, the matrix is compressed row-by-row and for symmetric matrices only non-zeros in the upper triangular half of the matrix are stored.

The Intel MKL storage format accepted for the PARDISO software for sparse matrices consists of three arrays, which are called the *values*, *columns*, and *rowIndex* arrays. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix *A*.

values	A real or complex array that contains the non-zero entries of *A*. The non-zero values of *A* are mapped into the *values* array using the row major, upper triangular storage mapping described above.
columns	Element *i* of the integer array *columns* contains the number of the column in *A* that contained the value in *values*(*i*).
rowIndex	Element *j* of the integer array *rowIndex* gives the index into the *values* array that contains the first non-zero element in a row *j* of *A*.

The length of the *values* and *columns* arrays is equal to the number of non-zeros in *A*.

Since the *rowIndex* array gives the location of the first non-zero within a row, and the non-zeros are stored consecutively, then the number of non-zeros in the *i*-th row is equal to the difference of *rowIndex*(*i*) and *rowIndex*(*i*+1).

In order to have this relationship hold for the last row of *A*, an additional entry (dummy entry) is added to the end of *rowIndex* whose value is equal to the number of non-zeros in *A*, plus one. This makes the total length of the *rowIndex* array one larger than the number of rows of *A*.

NOTE. The Intel MKL sparse storage scheme uses the Fortran programming language convention of starting array indices at 1, rather than the C programming language convention of starting at 0.

With the above in mind, consider storing the symmetric matrix discussed in the example from the previous section.

$$A = \begin{bmatrix} 9 & \frac{3}{2} & 6 & \frac{3}{4} & 3 \\ * & \frac{1}{2} & * & * & * \\ * & * & \frac{1}{2} & * & * \\ * & * & * & \frac{5}{8} & * \\ * & * & * & * & 16 \end{bmatrix}$$

In this case, *A* has nine non-zero elements, so the lengths of the *values* and *columns* arrays will be nine. Also, since the matrix *A* has five rows, the *rowIndex* array is of length six. The actual values for each of the arrays for the example matrix are as follows:

Table A-1 Storage Arrays for a Symmetric Example Matrix

one base indexing									
values	=	(9	3/2	6	3/4	3	1/2	1/2	5/8	16)
columns	=	(1	2	3	4	5	2	3	4	5)
rowIndex	=	(1	6	7	8	9	10)			
zero base indexing									
values	=	(9	3/2	6	3/4	3	1/2	1/2	5/8	16)
columns	=	(0	1	2	3	4	1	2	3	4)
rowIndex	=	(0	5	6	7	8	9)			

For a non-symmetric or non-Hermitian array, all of the non-zeros need to be stored. Consider the non-symmetric matrix *B* defined by the following:

$$B = \begin{bmatrix} 1 & -1 & * & -3 & * \\ -2 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & * & * & -5 \end{bmatrix}$$

We see that *B* has 13 non-zeros, and we store *B* as follows:

Table A-2 Storage Arrays for a Non-Symmetric Example Matrix

one base indexing													
values	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
columns	=	(1	2	4	1	2	3	4	5	1	3	4	2	5)
rowIndex	=	(1	4	6	9	12	14)							
zero base indexing													
values	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
columns	=	(0	1	3	0	1	2	3	4	0	2	3	1	4)
rowIndex	=	(0	3	5	8	11	13)							

In the current version of Intel MKL, direct sparse solvers cannot solve non-symmetric systems of equations. However, it can solve symmetrically structured systems of equations. A symmetrically structured system of equations is one where the pattern of non-zeros is symmetric. That is, a matrix has a symmetric structure if *a*(*j*,*i*)is non-zero if and only if *a*(*j*, *i*) is non-zero. From the point of view of the solver software, a non-zero element of a matrix is anything that is stored in the *values* array. In that sense, we can turn any non-symmetric matrix into a symmetrically structured matrix by carefully adding zeros to the *values* array. For example, suppose we consider the matrix *B* to have the following set of non-zero entries:

$$B = \begin{bmatrix} 1 & -1 & * & -3 & * \\ -2 & 5 & * & * & 0 \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & 0 & * & -5 \end{bmatrix}$$

Now *B* can be considered to be symmetrically structured with 15 non-zero entries.We would represent the matrix as:

Table A-3 Storage Arrays for a Symmetrically Structured Example Matrix

values	=	(1	-1	-3	-2	5	0	4	6	4	-4	2	7	8	0	-5)
columns	=	(1	2	4	1	2	5	3	4	5	1	3	4	2	3	5)
rowIndex	=	(1	4	7	10	13	16)									

Storage Format Restrictions

The storage format for the sparse solver must conform to two important restrictions:

First, the non-zero values in a given row must be placed into the *values* array in the order in which they occur in the row (from left to right). Second, no diagonal element can be omitted from the *values* array for any symmetric or structurally symmetric matrix.

The second restriction implies that when dealing with symmetric or structurally symmetric matrices that have zeros on the diagonal, the zero diagonal elements must be explicitly represented in the *values* array.

Sparse Storage Formats for Sparse BLAS Levels 2-3

This section describes in detail the sparse data structures supported in the current version of the Intel MKL Sparse BLAS level 2 and 3.

CSR Format

The Intel MKL compressed sparse row (CSR) format for sparse matrices consists of four arrays, which are called the *values*, *columns*, *pointerB*, and *pointerE* arrays. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix *A*.

values	A real or complex array that contains the non-zero entries of *A*. The non-zero values of *A* are mapped into the *values* array using the row major storage mapping described above.
columns	Element *i* of the integer array *columns* contains the number of the column in *A* that contained the value in *values*(*i*).
pointerB	Element *j* of this integer array gives the index into the *values* array that contains the first non-zero element in a row *j* of *A*. Note that this index is equal to *pointerB*(*j*) – *pointerB*(1)+1 .
pointerE	An integer array contains row indices, such that *pointerE*(*j*)-*pointerB*(1) is the index into the *values* array that contains the last non-zero element in a row *j* of *A*.

The length of the *values* and *columns* arrays is equal to the number of non-zeros in *A*.The length of the *pointerB* and *pointerE* arrays is equal to the number of rows in *A*.

Previously defined matrix *B*

$$B = \begin{bmatrix} 1 & -1 & * & -3 & * \\ -2 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & * & * & -5 \end{bmatrix}$$

can be represented in the CSR format as:

Table A-4 Storage Arrays for an Example Matrix in CSR Format

one base indexing													
values	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)

columns	=	(1	2	4	1	2	3	4	5	1	3	4	2	5)
pointerB	=	(1	4	6	9	12)								
pointerE	=	(4	6	9	12	14)								
zero base indexing														
values	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
columns	=	(0	1	3	0	1	2	3	4	0	2	3	1	4)
pointerB	=	(0	3	5	8	11)								
pointerE	=	(3	5	8	11	13)								

This storage format is used in the NIST Sparse BLAS library [Rem05].

Note that the storage format accepted for the PARDISO software and described above (see Storage Formats for the PARDISO Solver), is a variation of the CSR format. The PARDISO format has a restriction - all non-zero elements are stored continuously, that is the set of non-zero elements in the row *J* goes just after the set of non-zero elements in the row *J-1*.

There is no such restrictions in the CSR format. This advantage can be useful, for example, if there is a need to operate with different submatrices of the matrix at the same time. In this case, it is enough to define the arrays *pointerB* and *pointerE* for each needed submatrix so that all these array are pointers to the one array *values*.

Comparing the array *rowIndex* from the Table A-2 with the arrays *pointerB* and *pointerE* from the Table A-4 it is easy to see that

pointerB(*i*) = *rowIndex*(*i*) for *i*=1, ..5;

pointerE(*i*) = *rowIndex*(*i*+1) for *i*=1, ..5.

This gives the possibility to call a routine that has *values*, *columns*, *pointerB* and *pointerE* as input parameters for a sparse matrix stored in the format accepted for PARDISO. For example, a routine with the interface:

```
Subroutine name_routine(.... ,  values, columns, pointerB, pointerE, ...)
```

can be called with arguments *values*, *columns*, *rowIndex* in the following way:

```
call name_routine(.... ,  values, columns, rowIndex, rowindex(2), ...).
```

NOTE. Intel MKL Sparse BLAS level 2 provide routines for both flavors of the CSR format.

CSC Format

The compressed sparse column format (CSC), often called `Harwell-Boeing sparse matrix format`, is similar to the CSR format, but the columns are used instead the rows. Or, in other words, CSC format is equal to the CSR format for the transposed matrix.

By analogy with the CSR format Intel MKL Sparse BLAS level 2 library provides routines for two variations of the CSC format.

Variation of this format accepted for the PARDISO software consists of three arrays, which are called the *values*, *rows*, and *colIndex* arrays. The following table describes these arrays:

values	A real or complex array that contains the non-zero entries of *A*. The non-zero values of *A* are mapped into the *values* array using the column major storage mapping described above.
rows	Element *i* of the integer array rows contains the number of the row in *A* that contained the value in *values*(*i*).
colIndex	Element *j* of the integer array *colIndex* gives the index into the *values* array that contains the first non-zero element in a column *j* of *A*.

The length of the *values* and *rows* arrays is equal to the number of non-zero elements in *A*.

For example, the sparse matrix *B*

$$B = \begin{bmatrix} 1 & -1 & * & -3 & * \\ -2 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & * & * & -5 \end{bmatrix}$$

can be represented in the CSC format for PARDISO as follows:

Table A-5 Storage Arrays for an Example Matrix in the Harwell-Boeing format

values	=	(1	-2	-4	-1	5	8	4	2	-3	6	7	4	-5)
rows	=	(1	2	4	1	2	3	4	5	1	3	4	2	5)
colIndex	=	(1	4	7	9	12	14)							

Coordinate Format

The coordinate format is the most flexible and simplest format for the sparse matrix representation. Only nonzero entries are provided, and the coordinates of each nonzero entry is given explicitly. Many commercial libraries support the matrix-vector multiplication for the sparse matrices in the coordinate format.

The Intel MKL coordinate format consists of three arrays, which are called the *values*, *rows*, and *column* arrays, and a parameter *nnz* which is number of non-zero entries in *A*. All three arrays have to be dimensioned as *nnz*. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix *A*.

values	A real or complex array that contains the non-zero entries of *A* given in any order.
rows	Element *i* of the integer array *rows* contains the number of the row in *A* that contained the value in *values*(*i*).
columns	Element *i* of the integer array *columns* contains the number of the column in *A* that contained the value in *values*(*i*).

For example, the sparse matrix *C*

$$C = \begin{bmatrix} 1 & -1 & -3 & 0 & 0 \\ -2 & 5 & 0 & 0 & 0 \\ 0 & 0 & 4 & 6 & 4 \\ -4 & 0 & 2 & 7 & 0 \\ 0 & 8 & 0 & 0 & -5 \end{bmatrix}$$

can be represented in the coordinate format as follows:

Table A-6 Storage Arrays for an Example Matrix in case of the coordinate format

values	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
rows	=	(1	1	1	2	2	3	3	3	4	4	4	5	5)
columns	=	(1	2	3	1	2	3	4	5	1	3	4	2	5)

Diagonal Storage Scheme

If the matrix *A* has a few diagonals, then this structure can be used to reduce the amount of information needed for the location of the non-zero elements. This storage scheme is particularly useful in many applications where the matrix arises from a finite element or finite difference discretization. The Intel MKL diagonal storage scheme consists of two arrays, which are called the *values* and *distance* arrays, and parameters *ndiag* which is the number of non-empty diagonals, and *lval* which is declared leading dimension in the calling (sub) program. The following table describes the arrays *values* and *distance*:

values	A real or complex two dimensional array is dimensioned as *lval* by *ndiag*. It contains the non-zero diagonals of *A*. The key point of the storage is that each element in *values* retains the row corresponding to the row in the original matrix. In order to do so diagonals in the lower triangular part of *A* are padded from the top, and those in the upper triangular part are padded from the bottom. Note that the value of *distance*(*i*) is the number of elements to be padded for diagonal *i*.
distance	An integer array is dimensioned as *ndiag*. Element *i* of the array integer *distance* contains the distance between *i*-diagonal and the main diagonal. The distance is positive if the diagonal is above the main diagonal, and negative if the diagonal is below the main diagonal. The main diagonal has a distance equal to zero.

The sparse matrix *C* given above can be stored in the diagonal storage scheme as follows:

$$distance = (-3 \quad -1 \quad 0 \quad 1 \quad 2)$$

$$values = \begin{bmatrix} * & * & 1 & -1 & -3 \\ * & -2 & 5 & 0 & 0 \\ * & 0 & 4 & 6 & 4 \\ -4 & 2 & 7 & 0 & * \\ 8 & 0 & -5 & * & * \end{bmatrix}$$

where the asterisks denote padded elements.

It is clear that the upper triangle or lower triangle can be stored if the matrix is symmetric, hermitian, or skew-symmetric.

The diagonals can be stored in any order if the sparse diagonal representation is used for Intel MKL sparse matrix-matrix or matrix-vector multiplication routines. However, all elements of the array *distance* must be sorted in increasing order if the sparse diagonal representation is used for Intel MKL sparse triangular solver routines.

Skyline Storage Scheme

The skyline storage scheme is important in the direct sparse solvers, and it is well suited for Cholesky or LU decomposition when no pivoting is required.

The skyline storage scheme accepted in the Intel MKL can store only triangular matrix or triangular part of the matrix. This variant consists of two arrays which are called *values* and *pointers* arrays. The following table describes these arrays:

values	A scalar array. It contains the set of elements from each row of *A* starting from the first non-zero elements to and uncluding the diagonal element if the matrix is lower triangular, and the set of elements from each column of *A* starting with the first non-zero element down to and including the diagonal element. Encountered zero elements are included in the sets.
pointers	An integer array is dimensioned as *m*+1, where *m* is the number of rows for lower triangle (columns for the upper triangle). *pointers*(*i*) – *pointers*(1)+1 points to the location in *values* of the first non-zero element in row (column) *i*. The value of *pointers*(*m*+1) is set to the value *nnz*+*pointers*(1), where *nnz* is the number of elements in the array *values*.

Note that Intel MKL Sparse BLAS does not support general matrices for the routines operating with the skyline storage format.

For example, the low triangle of the matrix *C* given above can be stored as follows:

values = (1 -2 5 4 -4 0 2 7 8 0 0 -5)

pointers = (1 2 4 5 9 13)

and the upper triangle of this matrix *C* can be stored as follows:

values = (1 -1 5 -3 0 4 6 7 4 0 -5)

pointers = (1 2 4 7 9 12)

This storage format is supported by the NIST Sparse BLAS library [Rem05].

BSR Format

The Intel MKL block compressed sparse row (BSR) format for sparse matrices consists of four arrays, which are called the *values*, *columns*, *pointerB*, and *pointerE* arrays. The following table describes these arrays.

values	A real array that contains the non-zero blocks of a sparse matrix storing them block by block in row-wise fashion. A non-zero block is the block that contains at least one non-zero element. All elements of non-zero blocks are stored, even if some of them is equal to zero. Elements of nonzero blocks are stored in column major order within each dense block in the case of the Fortran programming language convention, and in row major order in the case of the C programming language convention.
columns	Element *i* of the integer array *columns* contains the number of the column in the block matrix that contains the *i*-th non-zero block.
pointerB	Element *j* of this integer array gives the index into the *columns* array that contains the first non-zero block in a row *j* of the block matrix.
pointerE	Element *j* of this integer array gives the index into the *columns* array that contains the last non-zero block in a row *j* of the block matrix plus 1.

The length of the *values* array is equal to the number of all elements in the non-zero blocks, the length of the *columns* array is equal to the number of non-zero blocks. The length of the *pointerB* and *pointerE* arrays is equal to the number of rows in the block matrix.

For example, consider the sparse matrix *D*

$$D = \begin{bmatrix} 1 & 0 & 6 & 7 & * & * \\ 2 & 1 & 8 & 2 & * & * \\ * & * & 1 & 4 & * & * \\ * & * & 5 & 2 & * & * \\ * & * & 4 & 3 & 7 & 2 \\ * & * & 0 & 0 & 0 & 0 \end{bmatrix}$$

If the size of the block equals to 2, then the sparse matrix D can be represented as a 3x3 block matrix E with the following structure:

$$E = \begin{bmatrix} L & M & * \\ * & N & * \\ * & P & Q \end{bmatrix}$$

where

$$L = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}, M = \begin{bmatrix} 6 & 7 \\ 8 & 2 \end{bmatrix}, N = \begin{bmatrix} 1 & 4 \\ 5 & 1 \end{bmatrix}, P = \begin{bmatrix} 4 & 3 \\ 0 & 0 \end{bmatrix}, Q = \begin{bmatrix} 7 & 2 \\ 0 & 0 \end{bmatrix}$$

The matrix D can be represented in the BSR format as follows:

one-based indexing

values = (1 2 0 1 6 8 7 2 1 5 4 2 4 0 3 0 7 0 2 0)

columns = (1 2 2 2 3)

pointerB = (1 3 4)

pointerE = (3 4 6)

zero-based indexing

values = (1 2 0 1 6 8 7 2 1 5 4 2 4 0 3 0 7 0 2 0)

columns = (0 1 1 1 2)

pointerB = (0 2 3)

pointerE = (2 3 5)

This storage format is supported by the NIST Sparse BLAS library [Rem05].

The Intel MKL supports the PARDISO variation of the block compressed sparse row (BSR) format for sparse matrices. This format consists of three arrays, which are called the *values*, *columns*, and *rowIndex* arrays. The following table describes these arrays.

values	A real array that contains the non-zero blocks of a sparse matrix storing them block by block in row-wise fashion. A non-zero block is the block that contains at least one non-zero element. All elements of non-zero blocks are stored, even if some of them is equal to zero. Elements of nonzero blocks are stored in column major order within each block in the case of the Fortran programming language convention, and in row major order in the case of the C programming language convention.
columns	Element *i* of the integer array *columns* contains the number of the column in the block matrix that contains the *i*-th non-zero block.
rowIndex	Element *j* of this integer array gives the index in the *columns* array that contains the first non-zero block in a row *j* of the block matrix.

The length of the *values* array is equal to the number of all elements in the non-zero blocks, the length of the *columns* array is equal to the number of non-zero blocks.

Since the *rowIndex* array gives the location of the first non-zero block within a row, and the non-zero blocks are stored consecutively, then the number of non-zero blocks in the *i*-th row is equal to the difference of *rowIndex*(*i*) and *rowIndex*(*i*+1).

In order to have this relationship hold for the last row of the block matrix, an additional entry (dummy entry) is added to the end of *rowIndex* whose value is equal to the number of non-zeros blocks plus one. This makes the total length of the *rowIndex* array one larger than the number of rows of the block matrux.

The above matrix *D* can be represented in the BSR format (PARDISO) variation as follows:

one-based indexing

values = (1 2 0 1 6 8 7 2 1 5 4 2 4 0 3 0 7 0 2 0)

columns = (1 2 2 2 3)

rowIndex = (1 3 4 6)

zero-based indexing

values = (1 2 0 1 6 8 7 2 1 5 4 2 4 0 3 0 7 0 2 0)

columns = (0 1 1 1 2)

rowIndex = (0 2 3 5)

When storing symmetric matrices, it is necessary to store only the upper triangular half of the matrix (upper triangular format) or the lower triangular half of the matrix (lower triangular format).

For example, consider the symmetric sparse matrix *F*:

$$F = \begin{bmatrix} 1 & 0 & 6 & 7 & * & * \\ 2 & 1 & 8 & 2 & * & * \\ 6 & 8 & 1 & 4 & * & * \\ 7 & 2 & 5 & 2 & * & * \\ * & * & * & * & 7 & 2 \\ * & * & * & * & 0 & 0 \end{bmatrix}$$

If the size of the block equals to 2, then the sparse matrix F can be represented as a 3x3 block matrix G with the following structure:

$$G = \begin{bmatrix} L & M & * \\ M' & N & * \\ * & * & Q \end{bmatrix}$$

where

$$L = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}, M = \begin{bmatrix} 6 & 7 \\ 8 & 2 \end{bmatrix}, M' = \begin{bmatrix} 6 & 8 \\ 7 & 2 \end{bmatrix}, N = \begin{bmatrix} 1 & 4 \\ 5 & 2 \end{bmatrix}, Q = \begin{bmatrix} 7 & 2 \\ 0 & 0 \end{bmatrix}$$

The symmetric matrix F can be represented in the PARDISO variation of the BSR format (storing only upper triangular) as follows:

one-based indexing

values = (1 2 0 1 6 8 7 2 1 5 4 2 7 0 2 0)

columns = (1 2 2 3)

rowIndex = (1 3 4 5)

zero-based indexing

values = (1 2 0 1 6 8 7 2 1 5 4 2 4 0 3 0 7 0 2 0)

columns = (0 1 1 2)

rowIndex = (0 2 3 4)

Interval Linear Systems

Intervals

An interval is a compact connected subset of the real axis **R**. It is thus completely defined by two numbers, namely, its lower endpoint and upper endpoint (sometimes called left endpoint and right endpoint respectively), so that [*a*, *b*] denotes the interval {*x* ∈ **R**|*a* ≤ *x*≤*b*}. The set of all real intervals is denoted by **IR**. In mathematical notation, taking the lower and upper endpoints of an interval is usually denoted by inf[*a*, *b*] = *a*, sup[*a*, *b*] = *b*.

In the discussion below, intervals and interval objects are denoted by boldface letters, while underscores and overscores designate the lower and upper endpoints of the interval $\boldsymbol{x} = [\underline{\mathbf{x}}, \overline{\mathbf{x}}]$

Every interval is uniquely determined by its midpoint,

$$\operatorname{mid} \boldsymbol{a} = \frac{1}{2}(\overline{a} + \underline{a}),$$

and radius,

$$\operatorname{rad} \boldsymbol{a} = \frac{1}{2}(\overline{a} + \underline{a})$$

the latter being equivalent to the width wid $\boldsymbol{a} = \overline{\mathbf{a}} - \underline{\mathbf{a}}$. Intervals of the form [a, a] that have equal lower and upper endpoints, that is, intervals of zero width, are called degenerate or point or thin, and they coincide with usual real numbers so that it can be implied $\mathbf{R} \subset \mathbf{IR}$. On the contrary, the intervals with nonzero width are called thick intervals.

Since intervals are sets, set-theoretical relations and operations between them are applicable, for example, inclusion, intersection, and so on. In particular, a point $t \in \mathbf{R}$ is a member of the interval $\boldsymbol{a}$ (written as $t \in \boldsymbol{a}$) if $\underline{\mathbf{a}} \leq t \leq \overline{\mathbf{a}}$. Also, the inclusion is defined as $\boldsymbol{a} \subseteq \boldsymbol{b}$ if and only if $\underline{\mathbf{a}} \geq \underline{\mathbf{b}}$ and $\overline{\mathbf{a}} \leq \overline{\mathbf{b}}$

Intervals and interval objects (vectors, matrices, etc.) are a convenient tool to represent the so-called bounded uncertainty and ambiguity, when only the lower and upper bounds of the possible variation of some value are known. In this sense, intervals provide an alternative to probabilistic and fuzzy approaches for describing quantitative uncertainty.

Arithmetic operations, such as addition, subtraction, multiplication and division, can be extended to intervals according to the fundamental principle

$$\boldsymbol{a} * \boldsymbol{b} := \{a * b \mid a \in \boldsymbol{a}, b \in \boldsymbol{b}\}, * \in \{+, -, \cdot, /\}, \tag{1}$$

which makes it possible to define the so-called *classical interval arithmetic*. Note that the empty interval [∅] is often incorporated into the computer interval arithmetic structures.

Interval vectors and matrices

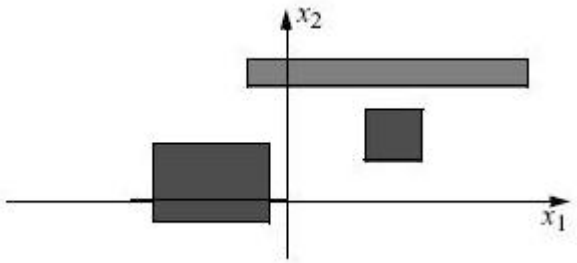
An interval vector is an ordered tuple of intervals placed vertically(column vector) or horizontally (row vector). So, if $\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_n$ are intervals, then

$$\boldsymbol{a} = \begin{pmatrix} \boldsymbol{a}_1 \\ \boldsymbol{a}_2 \\ \cdot \\ \cdot \\ \cdot \\ \boldsymbol{a}_n \end{pmatrix} \text{ is a column vector,}$$

and

$\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_n$ is a row vector.

The set of all interval n-vectors is denoted later in the text by $\mathbf{IR}^n$.



The interval vectors can be associated with their geometric images, namely rectangular boxes of the space $\mathbf{R}^n$, whose sides are parallel to the coordinate axes. For this reason, interval vectors are often called `boxes` for brevity.

An `interval matrix` is a rectangular table composed of the intervals:

$$\boldsymbol{A} := \begin{pmatrix} \boldsymbol{a}_1 \\ \boldsymbol{a}_2 \\ \cdot \\ \cdot \\ \cdot \\ \boldsymbol{a}_m \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix},$$

or $\boldsymbol{A} = (\boldsymbol{a}_{ij})$. Interval vectors can be identified with interval matrices either of the size $n \times 1$ (column vectors) or $1 \times n$ (row vectors). The set of all interval $m \times n$-matrices is denoted by $\mathbf{IR}^{m \times n}$. Arithmetic operations between interval vectors and matrices can be introduced on the basis of the relation that generalizes (1) (see [Alefeld83], [Neumaier90]).

An interval square matrix $\mathbf{A} \in \mathbf{IR}^{n \times n}$ is referred to as `regular` (nonsingular) if and only if all the point matrices $\boldsymbol{A} \in \boldsymbol{A}$ are regular (nonsingular), that is, have nonzero determinants.

Otherwise, the interval matrix $\boldsymbol{A} \in \mathbf{IR}^{n \times n}$ is called `singular`, which means that it contains at least one singular point matrix.

Generally, recognition of whether an interval matrix is regular or singular is an NP-hard problem, which implies that there may be no relatively simple (polynomially complex) algorithms that completely solve the problem in a reasonable time.

For practical needs, it is important to have a set of workable sufficient criteria for testing regularity of a wide range of interval matrices. Intel MKL provides routines that implement Ris-Beeck spectral criterion, Rump singular value criterion, as well as Rohn-Rex singular value criterion for testing regularity/singularity of interval matrices.

Sometimes, a related property (called `strong regularity`) needs to be checked for interval matrices. Strong regularity requires that the product of the interval matrix by its midpoint inverse is regular. The routine `?gerbr` enables to check the strong regularity judging by the value of its output parameter *sr*.

Interval Linear Systems

Solving systems of linear algebraic equations of the form

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2, \\ \vdots \qquad\quad \ddots \qquad\quad \vdots \\ a_{m1}x_1 + a_{n2}x_2 + \ldots + a_{mn}x_n = b_m, \end{cases} \tag{2}$$

(2)

or, concisely,

$Ax = b$

with an $m \times n$ matrix A and a right-hand side m-vector b, is one of the key problems in science and engineering. If a_{ij} and b_i are not defined exactly but rather belong to known intervals $\boldsymbol{a}_{ij}$ and $\boldsymbol{b}_i$ respectively, the system is called an `interval linear system` and can be written as

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2, \\ \quad\vdots \qquad\quad \ddots \qquad\quad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{mn}x_n = b_m, \end{cases} \tag{3}$$

(3)

with intervals $\boldsymbol{a}_{ij}$ and $\boldsymbol{b}_i$, or in a short form as

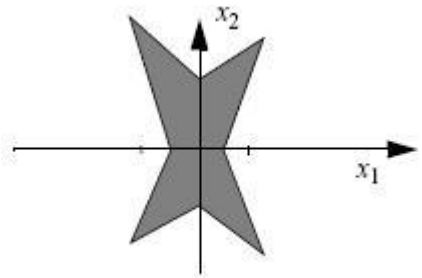
$$\boldsymbol{A}x = \boldsymbol{b} \tag{4}$$

with an interval matrix $\boldsymbol{A} = (\boldsymbol{a}_{ij})$ and interval right-hand side vector $\boldsymbol{b} = (\boldsymbol{b}_i)$. An interval linear system (3)–(4) is considered as a set of point linear systems of the same form $Ax = b$ with the parameters a_{ij} and b_i such that $a_{ij} \in \boldsymbol{a}_{ij}$ and $b_i \in \boldsymbol{b}_i$.

When a_{ij} and b_i are changing within intervals $\boldsymbol{a}_{ij}$ and $\boldsymbol{b}_i$, the solutions to the corresponding point systems $Ax = b$ with $A = (a_{ij})$ and $b = (b_i)$ form a set in the space $\mathbf{R}^n$, namely

$$\Xi(\boldsymbol{A}, \boldsymbol{b} := \{x \in \mathbf{R}^n \mid (\exists A \in \boldsymbol{A})(\exists b \in \boldsymbol{b})(Ax = b)\}. \tag{5}$$

The set (5), made up of solutions to all the point systems $Ax = b$ with $A \in \boldsymbol{A}$ and $b \in \boldsymbol{b}$, is called a `solution set` to the interval linear system (3)–(4). Usually, the solution set is a solid polyhedron in $\mathbf{R}^n$ for independent a_{ij} and b_i, $1 \le i, j \le n$, sometimes star-shaped as in the figure below.



NOTE. The above described set $\Xi(\boldsymbol{A}, \boldsymbol{B})$ is often called a `untied solution set`, since there exist a variety of other solution sets to interval systems of equations (see [Shary02]).

An exact description of the solution set is practically impossible for dimensions n larger than several tens, since its complexity grows exponentially with n. On the other hand, such an exact description is not really necessary in most cases. Usually, one needs to compute some `estimates`, in a prescribed sense, of the solution set. The most popular in practice is the following problem of `outer` (by supersets) interval estimation:

`For an interval system of linear equations` $\boldsymbol{Ax} = \boldsymbol{b}$

`find an interval enclosure of the solution set` $\Xi(\boldsymbol{A}, \boldsymbol{b})$. (6)

Frequently, a component-wise form of the problem (6) is considered:

`For an interval system of linear equations` $\boldsymbol{Ax} = \boldsymbol{b}$

`find estimates for` $\min\{x_v \mid x \in \Xi(\boldsymbol{A}, \boldsymbol{b})\}$ `from below,`

`for` $\max\{x_v \mid x \in \Xi(\boldsymbol{A}, \boldsymbol{b})\}$ `from above,` $v = 1, 2, \ldots, n$. (7)

In particular, Intel MKL `?gepps` routines operate with this type of the problem statement.

The problem (6)–(7) is one of the historically first and most popular in modern interval analysis. You can find an extensive bibliography on this problem, for example, in [Alefeld83], [Kearfott96] , [Neumaier90]).

Thus, solving an interval linear system is understood here as computing an outer interval estimate of the solution set to an interval linear system (3)–(4). The matrix A of the system is usually assumed to be square nonsingular.

Unlike classical computational linear algebra, solving interval linear systems proves to be very computationally hard in general. Computing the optimal (smallest) interval enclosures of the solution in (6), or, equivalently, computing exact estimates of the solution set in (7), is an NP-hard problem (see [Kreinovich97]), if there are no restrictions on the widths of the intervals in the system and/or the structure of nonzero elements in the matrix A. Moreover, the problem remains NP-hard even if we weaken the requirements on the solution and compute estimates of the solution sets that must be precise to within a predetermined absolute or relative accuracy.

From the practical standpoint, NP-hardness means that with a high probability a general problem cannot be solved in polynomial time with respect to problem size.

For this reason, numerical algorithms employed in Intel MKL for solving interval linear systems are divided into two classes depending on whether or not they provide a guaranteed accuracy of the result. "Fast" algorithms work fast and compute an enclosure of the solution set in a reasonable time, but without any accuracy assumptions. "Optimal", or "sharp" algorithms may take a lot of time to complete execution, but the results they obtain are less crude and may satisfy some accuracy requirements.

Intel MKL includes interval solver routines that implement algorithms of both types. For example, fast methods, such as interval Gauss method, interval Householder method, Hansen-Bliek-Rohn method, and Krawczyk iteration, are implemented in routines `?gegas`, `?gehss`, `?gehbs`, and `?gekws`, respectively. Parameter partitioning method (PPS-method) implemented in `?gepps` routine is an example of a sharp method. The routine `?trtrs` is subsumed under both categories due to a very special matrix structure.

Preconditioning

`Preconditioning` of interval linear system (4) is multiplying both the matrix $\boldsymbol{A}$ and the right-hand side vector $\boldsymbol{b}$ by a point matrix, with the intension to improve the properties of the system. So the system $\boldsymbol{A}x = b$ is substituted by the following system

$(\boldsymbol{CA})x = \boldsymbol{Cb}$

where C is some square point matrix. Preconditioning is widely used in classical computational linear algebra, and many interval solver algorithms (for example, interval Gauss method, interval Gauss-Seidel method and some others) also require a suitable preconditioning prior to their use.

One of the widely used preconditioning methods for the interval linear systems is preconditioning done by the inverse of the midpoint matrix, often called `midpoint-inverse preconditioning`. In Intel MKL, the midpoint inverse preconditioning is implemented in the routine `?gemip`.

Inverting interval matrices

Given an interval square matrix $\boldsymbol{A}$, an enclosure for the set of all inverse point matrices in $\boldsymbol{A}$ is called the inverse interval matrix $\boldsymbol{A}^{-1}$, that is,

$\boldsymbol{A}^{-1} \supseteq \{A^{-1} \mid A \in \boldsymbol{A}\}$.

In classical linear algebra, the solution to a system of linear algebraic equations $Ax = b$ with square nonsingular matrix A can be expressed as the product of the inverse A^{-1} by the right-hand side vector, or $x = A^{-1}b$.

In interval analysis, the similar product $\boldsymbol{A}^{-1}\boldsymbol{b}$ also produces an enclosure for the solution set $\Xi\boldsymbol{A},\boldsymbol{b}$ of the interval linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$. However, this method usually causes substantial overestimation and is not recommended. Using specialized procedures for outer estimation of the solution sets is preferable.

Nevertheless, computing tight enclosures for inverse interval matrices is essential in sensitivity-like analysis of equation systems and the like.

Computing the inverse interval matrix may be carried out as finding an enclosure for the solution set of the following interval matrix equation

$\boldsymbol{A}Y = I$, where I is the identity matrix,

by applying n times (for every column of the matrix Y) any method to solve the interval linear systems.

Note also that direct iterative procedures for finding the inverse interval matrix exist, such as Schulz method (see [Herzberger94]), which is included into Intel MKL as `?geszi` routine.

Routine and Function Arguments B

The major arguments in the BLAS routines are vector and matrix, whereas VML functions work on vector arguments only. The sections that follow discuss each of these arguments and provide examples.

Vector Arguments in BLAS

Vector arguments are passed in one-dimensional arrays. The array dimension (length) and vector increment are passed as integer variables. The length determines the number of elements in the vector. The increment (also called) determines the spacing between vector elements and the order of the elements in the array in which the vector is passed.

A vector of length *n* and increment *incx* is passed in a one-dimensional array *x* whose values are defined as

x(1), *x*(1+|*incx*|), ..., *x*(1+(*n*-1)* |*incx*|)

If *incx* is positive, then the elements in array *x* are stored in increasing order. If *incx* is negative, the elements in array *x* are stored in decreasing order with the first element defined as *x*(1+(*n*-1)* |*incx*|). If *incx* is zero, then all elements of the vector have the same value, *x*(1). The dimension of the one-dimensional array that stores the vector must always be at least

idimx = 1 + (*n*-1)* |*incx* |

Example B-1 One-dimensional Real Array

Let *x*(1:7) be the one-dimensional real array

x = (1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0).

If *incx* =2 and *n* = 3, then the vector argument with elements in order from first to last is (1.0, 5.0, 9.0).

If *incx* = -2 and *n* = 4, then the vector elements in order from first to last is (13.0, 9.0, 5.0, 1.0).

If *incx* = 0 and *n* = 4, then the vector elements in order from first to last is (1.0, 1.0, 1.0, 1.0).

One-dimensional substructures of a matrix, such as the rows, columns, and diagonals, can be passed as vector arguments with the starting address and increment specified. In Fortran, storing the *m*-by-*n* matrix is based on column-major ordering where the increment between elements in the same column is 1, the increment between elements in the same row is *m*, and the increment between elements on the same diagonal is *m* + 1.

Example B-2 Two-dimensional Real Matrix

Let *a* be the real 5 x 4 matrix declared as `REAL A (5,4)`.

To scale the third column of *a* by 2.0, use the BLAS routine `sscal` with the following calling sequence:

```
callsscal (5, 2.0, a(1,3), 1)
```

To scale the second row, use the statement:

```
callsscal (4, 2.0, a(2,1), 5)
```

To scale the main diagonal of A by 2.0, use the statement:

```
callsscal (5, 2.0, a(1,1), 6)
```

NOTE. The default vector argument is assumed to be 1.

Vector Arguments in VML

Vector arguments of VML mathematical functions are passed in one-dimensional arrays with unit vector increment. It means that a vector of length *n* is passed contiguously in an array *a* whose values are defined as `a[0], a[1], ..., a[`*n*`-1]` (for C- interface).

To accommodate for arrays with other increments, or more complicated indexing, VML contains auxiliary pack/unpack functions that gather the array elements into a contiguous vector and then scatter them after the computation is complete.

Generally, if the vector elements are stored in a one-dimensional array *a* as

```
a[m0], a[m1], ..., a[mn-1]
```

and need to be regrouped into an array *y* as

```
y[k0], y[k1], ..., y[kn-1],
```

VML pack/unpack functions can use one of the following indexing methods:

Positive Increment Indexing

```
kj = incy * j, mj = inca * j, j = 0 ,..., n-1
```

Constraint: `incy > 0` and `inca > 0`.

For example, setting incy = 1 specifies gathering array elements into a contiguous vector.

This method is similar to that used in BLAS, with the exception that negative and zero increments are not permitted.

Index Vector Indexing

`kj = iy[j], mj = ia[j], j = 0 ,..., n-1,`

where ia and iy are arrays of length *n* that contain index vectors for the input and output arrays *a* and *y*, respectively.

Mask Vector Indexing

Indices *kj*, *mj* are such that:

`my[kj] ≠ 0, ma[mj] ≠ 0 , j = 0,..., n-1,`

where ma and my are arrays that contain mask vectors for the input and output arrays *a* and *y*, respectively.

Matrix Arguments

Matrix arguments of the Intel® Math Kernel Library routines can be stored in either one- or two-dimensional arrays, using the following storage schemes:

- conventional full storage (in a two-dimensional array)
- packed storage for Hermitian, symmetric, or triangular matrices (in a one-dimensional array)
- band storage for band matrices (in a two-dimensional array).

Full storage is the following obvious scheme: a matrix A is stored in a two-dimensional array *a*, with the matrix element a_{ij} stored in the array element *a*(*i*,*j*).

If a matrix is triangular (upper or lower, as specified by the argument *uplo*), only the elements of the relevant triangle are stored; the remaining elements of the array need not be set.

Routines that handle symmetric or Hermitian matrices allow for either the upper or lower triangle of the matrix to be stored in the corresponding elements of the array:

if *uplo* ='U', a_{ij} is stored in *a*(*i*,*j*) for $i \le j$, other elements of *a* need not be set.

if *uplo* ='L', a_{ij} is stored in *a*(*i*,*j*) for $j \le i$, other elements of *a* need not be set.

Packed storage allows you to store symmetric, Hermitian, or triangular matrices more compactly: the relevant triangle (again, as specified by the argument *uplo*) is packed by columns in a one-dimensional array *ap*:

if *uplo* ='U', a_{ij} is stored in *ap*(*i*+*j*(*j*-1)/2) for $i \le j$

if *uplo* ='L', a_{ij} is stored in *ap*(*i*+(2**n*-*j*)*(*j*-1)/2) for $j \le i$.

In descriptions of LAPACK routines, arrays with packed matrices have names ending in *p*.

Band storage is as follows: an *m*-by-*n* band matrix with *kl* non-zero sub-diagonals and *ku* non-zero super-diagonals is stored compactly in a two-dimensional array *ab* with *kl*+*ku*+1 rows and *n* columns. Columns of the matrix are stored in the corresponding columns of the array, and diagonals of the matrix are stored in rows of the array. Thus,

a_{ij} is stored in *ab*(*ku*+1+*i*-*j*,*j*) for max(1,*j*-*ku*) ≤ *i* ≤ min(*n*,*j*+*kl*).

Use the band storage scheme only when *kl* and *ku* are much less than the matrix size *n*. Although the routines work correctly for all values of *kl* and *ku*, using the band storage is inefficient if your matrices are not really banded.

The band storage scheme is illustrated by the following example, when
m = *n* = 6, *kl* = 2, *ku* = 1

Array elements marked * are not used by the routines:

Banded matrix A

$$\begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 & 0 \\ a_{31} & a_{42} & a_{43} & a_{34} & 0 & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & 0 \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} \end{bmatrix}$$

Band storage of A

$$\begin{matrix} * & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \\ a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & * \\ a_{31} & a_{42} & a_{53} & a_{64} & * & * \end{matrix}$$

When a general band matrix is supplied for `LU factorization`, space must be allowed to store *kl* additional super-diagonals generated by fill-in as a result of row interchanges. This means that the matrix is stored according to the above scheme, but with *kl* + *ku* super-diagonals. Thus,

a_{ij} is stored in *ab*(*kl*+*ku*+1+*i*-*j*,*j*) for max(1,*j*-*ku*) ≤ *i* ≤ min(*n*,*j*+*kl*).

The band storage scheme for LU factorization is illustrated by the following example, when *m* = *n* = 6, *kl* = 2, *ku* = 1:

Banded matrix A

$$\begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 & 0 \\ a_{31} & a_{42} & a_{43} & a_{34} & 0 & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & 0 \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} \end{bmatrix}$$

Band storage of A

$$\begin{matrix} * & * & * & + & + & + \\ * & * & + & + & + & + \\ * & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \\ a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & * \\ a_{31} & a_{42} & a_{53} & a_{64} & * & * \end{matrix}$$

Array elements marked * are not used by the routines; elements marked + need not be set on entry, but are required by the LU factorization routines to store the results. The input array will be overwritten on exit by the details of the LU factorization as follows:

$$\begin{matrix} * & * & * & u_{14} & u_{25} & u_{36} \\ * & * & u_{13} & u_{24} & u_{35} & u_{46} \\ * & u_{12} & u_{23} & u_{34} & u_{45} & u_{56} \\ u_{11} & u_{22} & u_{33} & u_{44} & u_{55} & u_{66} \\ m_{21} & m_{32} & m_{43} & m_{54} & m_{65} & * \\ m_{31} & m_{42} & m_{53} & m_{64} & * & * \end{matrix}$$

where u_{ij} are the elements of the upper triangular matrix U, and m_{ij} are the multipliers used during factorization.

Triangular band matrices are stored in the same format, with either $kl = 0$ if upper triangular, or $ku = 0$ if lower triangular. For symmetric or Hermitian band matrices with k sub-diagonals or super-diagonals, you need to store only the upper or lower triangle, as specified by the argument *uplo*:

if *uplo* ='U', a_{ij} is stored in $ab(k+1+i-j,j)$ for $\max(1,j-k) \le i \le j$

if *uplo* ='L', a_{ij} is stored in $ab(1+i-j,j)$ for $j \le i \le \min(n,j+k)$.

In descriptions of LAPACK routines, arrays that hold matrices in band storage have names ending in *b*.

In Fortran, column-major ordering of storage is assumed. This means that elements of the same column occupy successive storage locations.

Three quantities are usually associated with a two-dimensional array argument: its leading dimension, which specifies the number of storage locations between elements in the same row, its number of rows, and its number of columns. For a matrix in full storage, the leading dimension of the array must be at least as large as the number of rows in the matrix.

A character transposition parameter is often passed to indicate whether the matrix argument is to be used in normal or transposed form or, for a complex matrix, if the conjugate transpose of the matrix is to be used.

The values of the transposition parameter for these three cases are the following:

'N' or *'n'*	normal (no conjugation, no transposition)
'T' or *'t'*	transpose
'C' or *'c'*	conjugate transpose.

Example B-3 Two-Dimensional Complex Array

Suppose *A (1:5, 1:4)* is the complex two-dimensional array presented by matrix

$$\begin{bmatrix} (1.1, 0.11) & (1.2, 0.12) & (1.3, 0.13) & (1.4, 0.14) \\ (2.1, 0.21) & (2.2, 0.22) & (2.3, 0.23) & (1.4, 0.24) \\ (3.1, 0.31) & (3.2, 0.32) & (3.3, 0.33) & (1.4, 0.34) \\ (4.1, 0.41) & (4.2, 0.42) & (4.3, 0.43) & (1.4, 0.44) \\ (5.1, 0.51) & (5.2, 0.52) & (5.3, 0.53) & (1.4, 0.54) \end{bmatrix}$$

Let *transa* be the transposition parameter, *m* be the number of rows, *n* be the number of columns, and *lda* be the leading dimension. Then if

transa = '*N*', *m* = 4, *n* = 2, and *lda* = 5, the matrix argument would be

$$\begin{bmatrix} (1.1, 0.11) & (1.2, 0.12) \\ (2.1, 0.21) & (2.2, 0.22) \\ (3.1, 0.31) & (3.2, 0.32) \\ (4.1, 0.41) & (4.2, 0.42) \end{bmatrix}$$

If *transa* = '*T*', *m* = 4, *n* = 2, and *lda* =5, the matrix argument would be

$$\begin{bmatrix} (1.1, 0.11) & (2.1, 0.21) & (3.1, 0.31) & (4.1, 0.41) \\ (1.2, 0.12) & (2.2, 0.22) & (3.2, 0.32) & (4.2, 0.42) \end{bmatrix}$$

If *transa* = '*C*', *m* = 4, *n* = 2, and *lda* =5, the matrix argument would be

$$\begin{bmatrix} (1.1, -0.11) & (2.1, -0.21) & (3.1, -0.31) & (4.1, -0.41) \\ (1.2, -0.12) & (2.2, -0.22) & (3.2, -0.32) & (4.2, -0.42) \end{bmatrix}$$

Note that care should be taken when using a leading dimension value which is different from the number of rows specified in the declaration of the two-dimensional array. For example, suppose the array *A* above is declared as `COMPLEX A (5,4)`.

Then if *transa* = 'N', *m* = 3, *n* = 4, and *lda* = 4, the matrix argument will be

$$\begin{bmatrix} (1.1,0.11) & (5.1,0.51) & (4.2,0.42) & (3.3,0.33) \\ (2.1,0.21) & (1.2,0.12) & (5.2,0.52) & (4.3,0.43) \\ (3.1,0.31) & (2.2,0.22) & (1.3,0.13) & (5.3,0.53) \end{bmatrix}$$

Code Examples

C

This appendix presents code examples of using some Intel MKL routines and functions. You can find here example code written in both Fortran and C.

Currently, the appendix includes the following sections:

- BLAS Code Examples
- PARDISO Code Examples
- Direct Sparse Solver Code Examples
- Iterative Sparse Solver Code Examples
- Fourier Transform Functions Code Examples
- Interval Linear Solvers Code Examples
- PDE Support Code Examples.

Please refer to respective chapters in the manual for detailed descriptions of function parameters and operation.

BLAS Code Examples

Example C-1. Using BLAS Level 1 Function

The following example illustrates a call to the BLAS Level 1 function `sdot`. This function performs a vector-vector operation of computing a scalar product of two single-precision real vectors *x* and *y*.

Parameters

n	Specifies the order of vectors *x* and *y*.
incx	Specifies the increment for the elements of *x*.
incy	Specifies the increment for the elements of *y*.

```
program dot_main
```

```
real x(10), y(10), sdot, res
integer n, incx, incy, i
external sdot
n = 5

incx = 2
incy = 1
do i = 1, 10
  x(i) = 2.0e0
  y(i) = 1.0e0
end do
res = sdot (n, x, incx, y, incy)
print*, `SDOT = `, res
end
```

As a result of this program execution, the following line is printed:

SDOT = 10.000

Example C-2. Using BLAS Level 1 Routine

The following example illustrates a call to the BLAS Level 1 routine `scopy`. This routine performs a vector-vector operation of copying a single-precision real vector *x* to a vector *y*.

Parameters

n	Specifies the order of vectors *x* and *y*.
incx	Specifies the increment for the elements of *x*.

Parameters

incy	Specifies the increment for the elements of *y*.

```
program copy_main
real x(10), y(10)
integer n, incx, incy, i
n = 3
incx = 3
incy = 1
do i = 1, 10
  x(i) = i
end do
call scopy (n, x, incx, y, incy)
print*, `Y = `, (y(i), i = 1, n)
end
```

As a result of this program execution, the following line is printed:

Y = 1.00000 4.00000 7.00000

Example C-3. Using BLAS Level 2 Routine

The following example illustrates a call to the BLAS Level 2 routine `sger`. This routine performs a matrix-vector operation

a := *alpha*x*y'* + *a*.

Parameters

alpha	Specifies a scalar *alpha*.
x	*m*-element vector.
y	*n*-element vector.
a	*m*-by-*n* matrix.

```
program ger_main
```

```
real a(5,3), x(10), y(10), alpha
integer m, n, incx, incy, i, j, lda
m = 2
 n = 3
lda = 5
incx = 2
incy = 1
alpha = 0.5
do i = 1, 10
  x(i) = 1.0
  y(i) = 1.0

end do
do i = 1, m
  do j = 1, n
    a(i,j) = j
  end do

end do
call sger (m, n, alpha, x, incx, y, incy, a, lda)
print*, `Matrix A: `
do i = 1, m
  print*, (a(i,j), j = 1, n)
end do
end
```

As a result of this program execution, matrix *a* is printed as follows:

Matrix A:

1.50000 2.50000 3.50000

1.50000 2.50000 3.50000

Example C-4. Using BLAS Level 3 Routine

The following example illustrates a call to the BLAS Level 3 routine `ssymm`. This routine performs a matrix-matrix operation

```
c :=  alpha*a*b' + beta*c.
```

Parameters

alpha	Specifies a scalar *alpha*.
beta	Specifies a scalar *beta*.
a	Symmetric matrix.
b	*m*-by-*n* matrix.

Parameters

c	*m*-by-*n* matrix.

```
program symm_main
real a(3,3), b(3,2), c(3,3), alpha, beta
integer m, n, lda, ldb, ldc, i, j
character uplo, side
uplo = 'u'
side = 'l'
m = 3
n = 2
lda = 3
ldb = 3
ldc = 3
alpha = 0.5
beta = 2.0
do i = 1, m
  do j = 1, m
    a(i,j) = 1.0
  end do
end do
do i = 1, m
  do j = 1, n
    c(i,j) = 1.0
    b(i,j) = 2.0
  end do
end do
call ssymm (side, uplo, m, n, alpha,
a, lda, b, ldb, beta, c, ldc)
```

```
print*, `Matrix C: `
do i = 1, m
  print*, (c(i,j), j = 1, n)
end do
end
```

As a result of this program execution, matrix `c` is printed as follows:

```
Matrix C:
5.00000 5.00000
5.00000 5.00000
5.00000 5.00000
```

Example C-5. Calling a Complex BLAS Level 1 Function from C

The following example illustrates a call from a C program to the complex BLAS Level 1 function `zdotc()`. This function computes the dot product of two double-precision complex vectors.

In this example, the complex dot product is returned in the structure `c`.

```
#define N 5
void main()
{
```

```
    int n, inca = 1, incb = 1, i;
    typedef struct{ double re; double im; } complex16;
    complex16 a[N], b[N], c;
    void zdotc();
    n = N;
    for( i = 0; i < n; i++ ){
      a[i].re = (double)i; a[i].im = (double)i * 2.0;
      b[i].re = (double)(n - i); b[i].im = (double)i * 2.0;
    }
    zdotc( &c, &n, a, &inca, b, &incb );
    printf( "The complex dot product is: ( %6.2f, %6.2f )\n", c.re, c.im );
}
```

NOTE. Instead of calling BLAS directly from C programs, you might wish to use the CBLAS interface; this is the supported way of calling BLAS from C. For more information about CBLAS, see Appendix D , which presents CBLAS, the C interface to the Basic Linear Algebra Subprograms (BLAS) implemented in Intel® MKL.

PARDISO Code Examples

This section presents code examples of using the PARDISO direct solver for computing solutions of linear systems with sparse matrices. For description of this solver, refer to Chapter 8 of the manual .

Examples for Sparse Symmetric Linear Systems

In this section two examples (Fortran, C) are provided to solve symmetric linear systems with PARDISO. To solve the systems of equations $Ax = b$, where

$$A = \begin{bmatrix} 7.0 & 0.0 & 1.0 & 0.0 & 0.0 & 2.0 & 7.0 & 0.0 \\ 0.0 & -4.0 & 8.0 & 0.0 & 2.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 8.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 5.0 \\ 0.0 & 0.0 & 0.0 & 7.0 & 0.0 & 0.0 & 9.0 & 0.0 \\ 0.0 & 2.0 & 0.0 & 0.0 & 5.0 & 1.0 & 5.0 & 0.0 \\ 2.0 & 0.0 & 0.0 & 0.0 & 1.0 & -1.0 & .0. & 5.0 \\ 7.0 & 0.0 & 0.0 & 9.0 & 5.0 & 0.0 & 11.0 & 0.0 \\ 0.0 & 0.0 & 5.0 & 0.0 & 0.0 & 5.0 & 0.0 & 5.0 \end{bmatrix} \text{ and } B = \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{bmatrix}$$

Example Results for Symmetric Systems

Upon successful execution of the solver, the result of the solution x is as follows

```
Reordering completed ...
Number of nonzeros in factors = 30
Number of factorization MFLOPS = 0
Factorization completed ...
Solve completed ...
The solution of the system is
x(1) = -0.0418602013
x(2) = -0.00341312416
x(3) = 0.117250377
x(4) = -0.11263958
x(5) = 0.0241722445
x(6) = -0.10763334
x(7) = 0.198719673
x(8) = 0.190382964
```

Example C-6. pardiso_sym.f for Symmetric Linear Systems

```
C------------------------------------------------------------------------
C Example program to show the use of the "PARDISO" routine
C for symmetric linear systems
C------------------------------------------------------------------------
C This program can be downloaded from the following site:
C http://www.computational.unibas.ch/cs/scicomp
C
C (C) Olaf Schenk, Department of Computer Science,
C University of Basel, Switzerland.
C Email: olaf.schenk@unibas.ch
C
C------------------------------------------------------------------------
      PROGRAM pardiso_sym
      IMPLICIT NONE
C.. Internal solver memory pointer for 64-bit architectures
C.. INTEGER*8 pt(64)
C.. Internal solver memory pointer for 32-bit architectures
C.. INTEGER*4 pt(64)
C.. This is OK in both cases
      INTEGER*8 pt(64)
C.. All other variables
      INTEGER maxfct, mnum, mtype, phase, n, nrhs, error, msglvl
      INTEGER iparm(64)
      INTEGER ia(9)
      INTEGER ja(18)
      REAL*8 a(18)
      REAL*8 b(8)
```

```
      REAL*8 x(8)
      INTEGER i, idum
      REAL*8 waltime1, waltime2, ddum
C.. Fill all arrays containing matrix data.
      DATA n /8/, nrhs /1/, maxfct /1/, mnum /1/
      DATA ia /1,5,8,10,12,15,17,18,19/
      DATA ja
     1 /1,  3,    6,7,
     2    2,3,  5,
     3      3,        8,
     4        4,    7,
     5          5,6,7,
     6            6,  8,
     7              7,
     8                8/
      DATA a
     1 /7.d0,     1.d0,         2.d0,7.d0,
     2       -4.d0,8.d0,    2.d0,
     3            1.d0,                   5.d0,
     4                 7.d0,    9.d0,
     5                      5.d0,1.d0,5.d0,
     6                          -1.d0,     5.d0,
     7                               11.d0,
     8                                    5.d0/
      integer omp_get_max_threads
      external omp_get_max_threads
C..
C.. Set up PARDISO control parameter
```

```
C..
      do i = 1, 64
         iparm(i) = 0
      end do
      iparm(1) = 1 ! no solver default
      iparm(2) = 2 ! fill-in reordering from METIS
      iparm(3) = omp_get_max_threads() !numbers of processors, value of
                                       !OMP_NUM_THREADS
      iparm(4) = 0 ! no iterative-direct algorithm
      iparm(5) = 0 ! no user fill-in reducing permutation
      iparm(6) = 0 ! =0 solution on the first n compoments of x
      iparm(7) = 16 ! default logical fortran unit number for output
      iparm(8) = 9 ! numbers of iterative refinement steps
      iparm(9) = 0 ! not in use
      iparm(10) = 13 ! perturbe the pivot elements with 1E-13
      iparm(11) = 1 ! use nonsymmetric permutation and scaling MPS
      iparm(12) = 0 ! not in use
      iparm(13) = 0 ! not in use
      iparm(14) = 0 ! Output: number of perturbed pivots
      iparm(15) = 0 ! not in use
      iparm(16) = 0 ! not in use
      iparm(17) = 0 ! not in use
      iparm(18) = -1 ! Output: number of nonzeros in the factor LU
      iparm(19) = -1 ! Output: Mflops for LU factorization
      iparm(20) = 0 ! Output: Numbers of CG Iterations
      error = 0 ! initialize error flag
      msglvl = 0 ! don't print statistical information
      mtype = -2 ! unsymmetric matrix symmetric, indefinite, no pivoting
C.. Initiliaze the internal solver memory pointer. This is only
```

```
C necessary for the FIRST call of the PARDISO solver.
      do i = 1, 64
         pt(i) = 0
      end do
C.. Reordering and Symbolic Factorization, This step also allocates
C all memory that is necessary for the factorization
      phase = 11 ! only reordering and symbolic factorization
      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
     1 idum, nrhs, iparm, msglvl, ddum, ddum, error)
      WRITE(*,*) 'Reordering completed ... '
      IF (error .NE. 0) THEN
         WRITE(*,*) 'The following ERROR was detected: ', error
         STOP
      END IF
      WRITE(*,*) 'Number of nonzeros in factors = ',iparm(18)
      WRITE(*,*) 'Number of factorization MFLOPS = ',iparm(19)
C.. Factorization.
      phase = 22 ! only factorization
      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
     1 idum, nrhs, iparm, msglvl, ddum, ddum, error)
      WRITE(*,*) 'Factorization completed ... '
      IF (error .NE. 0) THEN
         WRITE(*,*) 'The following ERROR was detected: ', error
         STOP
      ENDIF
C.. Back substitution and iterative refinement
      iparm(8) = 2 ! max numbers of iterative refinement steps
      phase = 33 ! only factorization
```

```
      do i = 1, n
         b(i) = 1.d0
      end do
      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
     1 idum, nrhs, iparm, msglvl, b, x, error)
      WRITE(*,*) 'Solve completed ... '
      WRITE(*,*) 'The solution of the system is '
      DO i = 1, n
         WRITE(*,*) ' x(',i,') = ', x(i)
      END DO
C.. Termination and release of memory
      phase = -1 ! release internal memory
      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, ddum, idum, idum,
     1 idum, nrhs, iparm, msglvl, ddum, ddum, error)
      END
```

Example C-7. pardiso_sym.c for Symmetric Linear Systems

```
/* -------------------------------------------------------------------*/
/* Example program to show the use of the "PARDISO" routine */
/* on symmetric linear systems*/
/* -------------------------------------------------------------------*/
/* This program can be downloaded from the following site: */
/* http://www.computational.unibas.ch/cs/scicomp*/
/* */
/* (C) Olaf Schenk, Department of Computer Science, */
/* University of Basel, Switzerland.*/
/* Email: olaf.schenk@unibas.ch*/
/* -------------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
extern int omp_get_max_threads();
/* PARDISO prototype. */
extern int PARDISO
   (void *, int *, int *, int *, int *, int *,
   double *, int *, int *, int*, int *, int *,
   int *, double *, double *, int*);

int main( void ) {
   /* Matrix data. */
   int n = 8;
   int ia[ 9] = { 1, 5, 8, 10, 12, 15, 17, 18, 19 };
   int ja[18] = { 1, 3, 6, 7,
      2, 3, 5,
```

```
    3, 8,
    4, 7,
    5, 6, 7,
    6, 8,
    7,
    8 };
double a[18] = { 7.0, 1.0, 2.0, 7.0,
    -4.0, 8.0, 2.0,
    1.0, 5.0,
    7.0, 9.0,
    5.0, 1.0, 5.0,
    -1.0, 5.0,
    11.0,
    5.0 };
int mtype = -2; /* Real symmetric matrix */
/* RHS and solution vectors.*/
double b[8], x[8];
int nrhs = 1; /* Number of right hand sides. */
/* Internal solver memory pointer pt, */
/* 32-bit: int pt[64]; 64-bit: long int pt[64] */
/* or void *pt[64] should be OK on both architectures */
void *pt[64];
/* Pardiso control parameters.*/
int iparm[64];
int maxfct, mnum, phase, error, msglvl;
/* Auxiliary variables. */
int i;
double ddum; /* Double dummy*/
```

```
   int idum; /* Integer dummy.*/
/* -------------------------------------------------------------------*/
/* .. Setup Pardiso control parameters.*/
/* -------------------------------------------------------------------*/
   for (i = 0; i < 64; i++) {
      iparm[i] = 0;
   }
   iparm[0] = 1; /* No solver default*/
   iparm[1] = 2; /* Fill-in reordering from METIS */
   /* Numbers of processors, value of OMP_NUM_THREADS */
   iparm[2] = omp_get_max_threads();
   iparm[3] = 0; /* No iterative-direct algorithm */
   iparm[4] = 0; /* No user fill-in reducing permutation */
   iparm[5] = 0; /* Write solution into x */
   iparm[6] = 16; /* Default logical fortran unit number for output */
   iparm[7] = 2; /* Max numbers of iterative refinement steps */
   iparm[8] = 0; /* Not in use*/
   iparm[9] = 13; /* Perturb the pivot elements with 1E-13 */
   iparm[10] = 1; /* Use nonsymmetric permutation and scaling MPS */
   iparm[11] = 0; /* Not in use*/
   iparm[12] = 0; /* Not in use*/
   iparm[13] = 0; /* Output: Number of perturbed pivots */
   iparm[14] = 0; /* Not in use*/
   iparm[15] = 0; /* Not in use*/
   iparm[16] = 0; /* Not in use*/
   iparm[17] = -1; /* Output: Number of nonzeros in the factor LU */
   iparm[18] = -1; /* Output: Mflops for LU factorization */
   iparm[19] = 0; /* Output: Numbers of CG Iterations */
```

```
   maxfct = 1; /* Maximum number of numerical factorizations. */
   mnum = 1; /* Which factorization to use. */
   msglvl = 0; /* Don't print statistical information in file */
   error = 0; /* Initialize error flag */
/* ------------------------------------------------------------------------*/
/* .. Initialize the internal solver memory pointer. This is only */
/* necessary for the FIRST call of the PARDISO solver. */
/* ------------------------------------------------------------------------*/
   for (i = 0; i < 64; i++) {
      pt[i] = 0;
   }
/* ------------------------------------------------------------------------*/
/* .. Reordering and Symbolic Factorization. This step also allocates */
/* all memory that is necessary for the factorization. */
/* ------------------------------------------------------------------------*/
   phase = 11;
   PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
      &n, a, ia, ja, &idum, &nrhs,
      iparm, &msglvl, &ddum, &ddum, &error);
   if (error != 0) {
      printf("\nERROR during symbolic factorization: %d", error);
      exit(1);
   }
   printf("\nReordering completed ... ");
   printf("\nNumber of nonzeros in factors = %d", iparm[17]);
   printf("\nNumber of factorization MFLOPS = %d", iparm[18]);
/* ------------------------------------------------------------------------*/
/* .. Numerical factorization.*/
```

```
/* ---------------------------------------------------------------------*/
   phase = 22;
   PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
      &n, a, ia, ja, &idum, &nrhs,
      iparm, &msglvl, &ddum, &ddum, &error);
   if (error != 0) {
      printf("\nERROR during numerical factorization: %d", error);
      exit(2);
   }
   printf("\nFactorization completed ... ");
/* ---------------------------------------------------------------------*/
/* .. Back substitution and iterative refinement. */
/* ---------------------------------------------------------------------*/
   phase = 33;
   iparm[7] = 2; /* Max numbers of iterative refinement steps. */
   /* Set right hand side to one.*/
   for (i = 0; i < n; i++) {
      b[i] = 1;
   }
   PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
      &n, a, ia, ja, &idum, &nrhs,
      iparm, &msglvl, b, x, &error);
   if (error != 0) {
      printf("\nERROR during solution: %d", error);
      exit(3);
   }
   printf("\nSolve completed ... ");
   printf("\nThe solution of the system is: ");
```

```
   for (i = 0; i < n; i++) {
      printf("\n x [%d] = % f", i, x[i] );
   }
   printf ("\n");
/* ---------------------------------------------------------------------*/
/* .. Termination and release of memory. */
/* ---------------------------------------------------------------------*/
   phase = -1; /* Release internal memory. */
   PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
      &n, &ddum, ia, ja, &idum, &nrhs,
      iparm, &msglvl, &ddum, &ddum, &error);
   return 0;
}
```

Examples for Sparse Unsymmetric Linear Systems

In this section two examples (Fortran, C) are provided to solve unsymmetric linear systems with PARDISO. To solve the systems of equations $Ax = b$, where

$$A = \begin{bmatrix} 1.0 & -1.0 & 0.0 & -3.0 & 0.0 \\ -2.0 & 5.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 4.0 & 6.0 & 4.0 \\ -4.0 & 0.0 & 2.0 & 7.0 & 0.0 \\ 0.0 & 8.0 & 0.0 & 0.0 & -5.0 \end{bmatrix} \text{ and } B = \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{bmatrix}$$

Example Results for Unsymmetric Systems

Upon successful execution of the solver, the result of the solution x is as follows

```
Reordering completed ...

 Number of nonzeros in factors = 21
 Number of factorization MFLOPS = 0
 Factorization completed ...

 Solve completed ...

 The solution of the system is

  x( 1) =  -0.522321429
  x( 2) =  -0.00892857143
  x( 3) =   1.22098214
  x( 4) =  -0.504464286
  x( 5) =  -0.214285714
```

Example C-8. pardiso_unsym.f for Unsymmetric Linear Systems

```
********************************************************************************
*   Copyright(C) 2004 Intel Corporation. All Rights Reserved.
*   The source code contained  or  described herein and all documents related
  to
*   the source code ("Material") are owned by Intel Corporation or its
suppliers
*   or licensors. Title to the  Material remains with  Intel Corporation or
 its
*   suppliers and licensors. The Material contains trade secrets and
proprietary
*   and  confidential information of  Intel or its suppliers and licensors.
 The
*   Material is  protected  by  worldwide  copyright  and trade secret laws
 and
*   treaty provisions. No part of the Material may be used, copied,
reproduced,
*   modified, published, uploaded, posted, transmitted, distributed or
disclosed
*   in any way without Intel's prior express written permission.
*   No license under any  patent, copyright, trade secret or other
intellectual
*   property right is granted to or conferred upon you by disclosure or
delivery
*   of the Materials, either expressly, by implication, inducement, estoppel
 or
*   otherwise. Any  license  under  such  intellectual property  rights must
 be
*   express and approved by Intel in writing.
*
********************************************************************************
*   Content : MKL DSS Fortran-77 example
*
********************************************************************************
```

```
C-----------------------------------------------------------------------
C Example program to show the use of the "PARDISO" routine
C for symmetric linear systems
C-----------------------------------------------------------------------
C This program can be downloaded from the following site:
C http://www.computational.unibas.ch/cs/scicomp
C
C (C) Olaf Schenk, Department of Computer Science,
C University of Basel, Switzerland.
C Email: olaf.schenk@unibas.ch
C
C-----------------------------------------------------------------------
      PROGRAM pardiso_unsym
      IMPLICIT NONE
C.. Internal solver memory pointer for 64-bit architectures
C.. INTEGER*8 pt(64)
C.. Internal solver memory pointer for 32-bit architectures
C.. INTEGER*4 pt(64)
C.. This is OK in both cases
      INTEGER*8 pt(64)
C.. All other variables
      INTEGER maxfct, mnum, mtype, phase, n, nrhs, error, msglvl
      INTEGER iparm(64)
      INTEGER ia(6)
      INTEGER ja(13)
      REAL*8 a(13)
      REAL*8 b(5)
      REAL*8 x(5)
```

```
      INTEGER i, idum
      REAL*8 waltime1, waltime2, ddum
C.. Fill all arrays containing matrix data.
      DATA n /5/, nrhs /1/, maxfct /1/, mnum /1/
      DATA ia /1,4,6,9,12,14/
      DATA ja
     1 /   1,    2,         4,
     2     1,    2,
     3                 3,   4,    5,
     4     1,          3,   4,
     5           2,               5/
      DATA a
     1 /1.d0,-1.d0,      -3.d0,
     2 -2.d0, 5.d0,
     3              4.d0, 6.d0, 4.d0,
     4 -4.d0,       2.d0, 7.d0,
     5        8.d0,           -5.d0/
      integer omp_get_max_threads
      external omp_get_max_threads
C..
C.. Set up PARDISO control parameter
C..
      do i = 1, 64
         iparm(i) = 0
      end do
      iparm(1) = 1 ! no solver default
      iparm(2) = 2 ! fill-in reordering from METIS
      iparm(3) = omp_get_max_threads() ! numbers of processors, value of
OMP_NUM_THREADS
```

```
      iparm(4) = 0 ! no iterative-direct algorithm
      iparm(5) = 0 ! no user fill-in reducing permutation
      iparm(6) = 0 ! =0 solution on the first n compoments of x
      iparm(7) = 0 ! not in use
      iparm(8) = 9 ! numbers of iterative refinement steps
      iparm(9) = 0 ! not in use
      iparm(10) = 13 ! perturbe the pivot elements with 1E-13
      iparm(11) = 1 ! use nonsymmetric permutation and scaling MPS
      iparm(12) = 0 ! not in use
      iparm(13) = 0 ! not in use
      iparm(14) = 0 ! Output: number of perturbed pivots
      iparm(15) = 0 ! not in use
      iparm(16) = 0 ! not in use
      iparm(17) = 0 ! not in use
      iparm(18) = -1 ! Output: number of nonzeros in the factor LU
      iparm(19) = -1 ! Output: Mflops for LU factorization
      iparm(20) = 0 ! Output: Numbers of CG Iterations
      error = 0 ! initialize error flag
      msglvl = 1 ! print statistical information
      mtype = 11 ! real unsymmetric
C.. Initiliaze the internal solver memory pointer. This is only
C necessary for the FIRST call of the PARDISO solver.
      do i = 1, 64
         pt(i) = 0
      end do
C.. Reordering and Symbolic Factorization, This step also allocates
C all memory that is necessary for the factorization
      phase = 11 ! only reordering and symbolic factorization
```

```
      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
     1 idum, nrhs, iparm, msglvl, ddum, ddum, error)
      WRITE(*,*) 'Reordering completed ... '
      IF (error .NE. 0) THEN
         WRITE(*,*) 'The following ERROR was detected: ', error
         STOP
      END IF
      WRITE(*,*) 'Number of nonzeros in factors = ',iparm(18)
      WRITE(*,*) 'Number of factorization MFLOPS = ',iparm(19)
C.. Factorization.
      phase = 22 ! only factorization
      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
     1 idum, nrhs, iparm, msglvl, ddum, ddum, error)
      WRITE(*,*) 'Factorization completed ... '
      IF (error .NE. 0) THEN
         WRITE(*,*) 'The following ERROR was detected: ', error
         STOP
      ENDIF
C.. Back substitution and iterative refinement
      iparm(8) = 2 ! max numbers of iterative refinement steps
      phase = 33 ! only factorization
      do i = 1, n
         b(i) = 1.d0
      end do
      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
     1 idum, nrhs, iparm, msglvl, b, x, error)
      WRITE(*,*) 'Solve completed ... '
      WRITE(*,*) 'The solution of the system is '
```

```
      DO i = 1, n
         WRITE(*,*) ' x(',i,') = ', x(i)
      END DO
C.. Termination and release of memory
      phase = -1 ! release internal memory
      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, ddum, idum, idum,
     1 idum, nrhs, iparm, msglvl, ddum, ddum, error)
      END
```

Example C-9. pardiso_unsym.c for Unsymmetric Linear Systems

```
/*
********************************************************************************
*   Copyright(C) 2004 Intel Corporation. All Rights Reserved.
*   The source code contained  or  described herein and all documents related
 to
*   the source code ("Material") are owned by Intel Corporation or its
suppliers
*   or licensors. Title to the  Material remains with  Intel Corporation or
 its
*   suppliers and licensors. The Material contains trade secrets and
proprietary
*   and  confidential information of  Intel or its suppliers and licensors.
 The
*   Material is  protected  by  worldwide  copyright  and trade secret laws
 and
*   treaty provisions. No part of the Material may be used, copied,
reproduced,
*   modified, published, uploaded, posted, transmitted, distributed or
disclosed
*   in any way without Intel's prior express written permission.
*   No license under any  patent, copyright, trade secret or other
intellectual
*   property right is granted to or conferred upon you by disclosure or
delivery
*   of the Materials, either expressly, by implication, inducement, estoppel
 or
*   otherwise. Any  license  under  such  intellectual property  rights must
 be
*   express and approved by Intel in writing.
*
********************************************************************************
*   Content : MKL DSS C example
*
```

```
*******************************************************************************
*/
/* ---------------------------------------------------------------------*/
/* Example program to show the use of the "PARDISO" routine */
/* on symmetric linear systems*/
/* ---------------------------------------------------------------------*/
/* This program can be downloaded from the following site: */
/* http://www.computational.unibas.ch/cs/scicomp*/
/* */
/* (C) Olaf Schenk, Department of Computer Science, */
/* University of Basel, Switzerland.*/
/* Email: olaf.schenk@unibas.ch*/
/* ---------------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
extern int omp_get_max_threads();
/* PARDISO prototype. */
#if defined(_WIN32) || defined(_WIN64)
#define pardiso_ PARDISO
#else
#define PARDISO pardiso_
#endif
extern int PARDISO
        (void *, int *, int *, int *, int *, int *,
        double *, int *, int*, int *, int *, int *,
        int *, double *, double*, int *);
```

```
int main( void ) {
        /* Matrix data. */
        int n = 5;
        int ia[ 6] = { 1, 4, 6, 9, 12, 14 };
        int ja[13] = { 1, 2, 4,
                1, 2,
                3, 4, 5,
                1, 3, 4,
                2, 5 };
        double a[18] = { 1.0, -1.0, -3.0,
                -2.0, 5.0,
                4.0, 6.0, 4.0,
                -4.0, 2.0, 7.0,
                8.0, -5.0 };
        int mtype = 11; /* Real unsymmetric matrix */
        /* RHS and solution vectors.*/
        double b[5], x[5];
        int nrhs = 1; /* Number of right hand sides. */
        /* Internal solver memory pointer pt, */
        /* 32-bit: int pt[64]; 64-bit: long int pt[64] */
        /* or void *pt[64] should be OK on both architectures */
        void *pt[64];
        /* Pardiso control parameters.*/
        int iparm[64];
        int maxfct, mnum, phase, error, msglvl;
        /* Auxiliary variables.*/
        int i;
        double ddum; /* Double dummy */
```

```
        int idum; /* Integer dummy. */
/* ------------------------------------------------------------------------*/
/* .. Setup Pardiso control parameters.*/
/* ------------------------------------------------------------------------*/
        for (i = 0; i < 64; i++) {
                iparm[i] = 0;
        }
        iparm[0] = 1; /* No solver default */
        iparm[1] = 2; /* Fill-in reordering from METIS */
        /* Numbers of processors, value of OMP_NUM_THREADS */
        iparm[2] = omp_get_max_threads();
        iparm[3] = 0; /* No iterative-direct algorithm */
        iparm[4] = 0; /* No user fill-in reducing permutation */
        iparm[5] = 0; /* Write solution into x */
        iparm[6] = 0; /* Not in use */
        iparm[7] = 2; /* Max numbers of iterative refinement steps */
        iparm[8] = 0; /* Not in use */
        iparm[9] = 13; /* Perturb the pivot elements with 1E-13 */
        iparm[10] = 1; /* Use nonsymmetric permutation and scaling MPS */
        iparm[11] = 0; /* Not in use */
        iparm[12] = 0; /* Not in use */
        iparm[13] = 0; /* Output: Number of perturbed pivots */
        iparm[14] = 0; /* Not in use */
        iparm[15] = 0; /* Not in use */
        iparm[16] = 0; /* Not in use */
        iparm[17] = -1; /* Output: Number of nonzeros in the factor LU */
        iparm[18] = -1; /* Output: Mflops for LU factorization */
        iparm[19] = 0; /* Output: Numbers of CG Iterations */
```

```
        maxfct = 1; /* Maximum number of numerical factorizations. */
        mnum = 1; /* Which factorization to use. */
        msglvl = 1; /* Print statistical information in file */
        error = 0; /* Initialize error flag */
/* ------------------------------------------------------------------------*/
/* .. Initialize the internal solver memory pointer. This is only */
/* necessary for the FIRST call of the PARDISO solver. */
/* ------------------------------------------------------------------------*/
        for (i = 0; i < 64; i++) {
                pt[i] = 0;
        }
/* ------------------------------------------------------------------------*/
/* .. Reordering and Symbolic Factorization. This step also allocates */
/* all memory that is necessary for the factorization. */
/* ------------------------------------------------------------------------*/
        phase = 11;
        PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
                &n, a, ia, ja, &idum, &nrhs,
                iparm, &msglvl, &ddum, &ddum, &error);
        if (error != 0) {
                printf("\nERROR during symbolic factorization: %d", error);
                exit(1);
        }
        printf("\nReordering completed ... ");
        printf("\nNumber of nonzeros in factors = %d", iparm[17]);
        printf("\nNumber of factorization MFLOPS = %d", iparm[18]);
/* ------------------------------------------------------------------------*/
/* .. Numerical factorization.*/
```

```
/* -------------------------------------------------------------------*/
        phase = 22;
        PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
                &n, a, ia, ja, &idum, &nrhs,
                iparm, &msglvl, &ddum, &ddum, &error);
        if (error != 0) {
               printf("\nERROR during numerical factorization: %d", error);
                exit(2);
        }
        printf("\nFactorization completed ... ");
/* -------------------------------------------------------------------*/
/* .. Back substitution and iterative refinement. */
/* -------------------------------------------------------------------*/
        phase = 33;
        iparm[7] = 2; /* Max numbers of iterative refinement steps. */
        /* Set right hand side to one. */
        for (i = 0; i < n; i++) {
                b[i] = 1;
        }
        PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
                &n, a, ia, ja, &idum, &nrhs,
                iparm, &msglvl, b, x, &error);
        if (error != 0) {
                printf("\nERROR during solution: %d", error);
                exit(3);
        }
        printf("\nSolve completed ... ");
        printf("\nThe solution of the system is: ");
```

```
        for (i = 0; i < n; i++) {
                printf("\n x [%d] = % f", i, x[i] );
        }
        printf ("\n");
/* ---------------------------------------------------------------------*/
/* .. Termination and release of memory. */
/* ---------------------------------------------------------------------*/
        phase = -1; /* Release internal memory. */
        PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
                &n, &ddum, ia, ja, &idum, &nrhs,
                iparm, &msglvl, &ddum, &ddum, &error);
        return 0;
}
```

Direct Sparse Solver Code Examples

This section contains example code in Fortran 77, Fortran 90 and C. For description of the sparse solver routines used in this code, refer to "Direct Sparse Solver (DSS) Interface Routines" in Chapter 8 of the manual . The example code solves the equations presented in Direct Method section of Appendix A - a symmetric positive definite system of equations $Ax = b$ with a sparse matrix, where

$$A = \begin{bmatrix} 9 & 1.5 & 6 & 0.75 & 3 \\ 1.5 & 0.5 & 0 & 0 & 0 \\ 6 & 0 & 12 & 0 & 0 \\ 0.75 & 0 & 0 & 0.625 & 0 \\ 3 & 0 & 0 & 0 & 16 \end{bmatrix} \text{ and } B = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

Example results for symmetric systems

Upon successful execution of the solver, the determinant and the result of the solution array are as follows

```
pow of determinant is      0.000
base of determinant is      2.250
Determinant is      2.250
Solution Array:   -326.333   983.000  163.417   398.000    61.500
```

Example C-10. Fortran 77 example to Solve Symmetric Positive Definite System

```
*******************************************************************************
*   Copyright(C) 2001-2004 Intel Corporation. All Rights Reserved.
*   The source code contained or described herein and all documents related
 to
*   the source code ("Material") are owned by Intel Corporation or its
suppliers
*   or licensors. Title to the Material remains with Intel Corporation or
 its
*   suppliers and licensors. The Material contains trade secrets and
proprietary
*   and confidential information of Intel or its suppliers and licensors.
 The
*   Material is protected by worldwide copyright and trade secret laws
 and
*   treaty provisions. No part of the Material may be used, copied,
reproduced,
*   modified, published, uploaded, posted, transmitted, distributed or
disclosed
*   in any way without Intel's prior express written permission.
*   No license under any patent, copyright, trade secret or other
intellectual
*   property right is granted to or conferred upon you by disclosure or
delivery
*   of the Materials, either expressly, by implication, inducement, estoppel
 or
*   otherwise. Any license under such intellectual property rights must
 be
*   express and approved by Intel in writing.
*
*******************************************************************************
*   Content : Intel MKL DSS Fortran-77 example
*
```

```
*******************************************************************************
C------------------------------------------------------------------------------
C Example program for solving symmetric positive definite system of
C equations.
C------------------------------------------------------------------------------
      PROGRAM solver_f77_test
      IMPLICIT NONE
      INCLUDE 'mkl_dss.f77'
C------------------------------------------------------------------------------
C Define the array and rhs vectors
C------------------------------------------------------------------------------
      INTEGER nRows, nCols, nNonZeros, i, nRhs
      PARAMETER (nRows = 5,
     1 nCols = 5,
     2 nNonZeros = 9,
     3 nRhs = 1)
      INTEGER rowIndex(nRows + 1), columns(nNonZeros)
      DOUBLE PRECISION values(nNonZeros), rhs(nRows)
      DATA rowIndex / 1, 6, 7, 8, 9, 10 /
      DATA columns / 1, 2, 3, 4, 5, 2, 3, 4, 5 /
      DATA values / 9, 1.5, 6, .75, 3, 0.5, 12, .625, 16 /
      DATA rhs / 1, 2, 3, 4, 5 /
C------------------------------------------------------------------------------
C Allocate storage for the solver handle and the solution vector
C------------------------------------------------------------------------------
      DOUBLE PRECISION solution(nRows)
      INTEGER*8 handle
      INTEGER error
```

```
      CHARACTER*15 statIn
      DOUBLE PRECISION statOut(5)
      INTEGER bufLen
      PARAMETER(bufLen = 20)
      INTEGER buff(bufLen)
C---------------------------------------------------------------------------
C Initialize the solver
C---------------------------------------------------------------------------
      error = dss_create(handle, MKL_DSS_DEFAULTS)
      IF (error .NE. MKL_DSS_SUCCESS ) GOTO 999
C---------------------------------------------------------------------------
C Define the non-zero structure of the matrix
C---------------------------------------------------------------------------
      error = dss_define_structure( handle, MKL_DSS_SYMMETRIC,
     & rowIndex, nRows, nCols, columns, nNonZeros )
      IF (error .NE. MKL_DSS_SUCCESS ) GOTO 999
C---------------------------------------------------------------------------
C Reorder the matrix
C---------------------------------------------------------------------------
      error = dss_reorder( handle, MKL_DSS_DEFAULTS, 0)
      IF (error .NE. MKL_DSS_SUCCESS ) GOTO 999
C---------------------------------------------------------------------------
C Factor the matrix
C---------------------------------------------------------------------------
      error = dss_factor_real( handle,
     & MKL_DSS_DEFAULTS, VALUES)
      IF (error .NE. MKL_DSS_SUCCESS ) GOTO 999
C---------------------------------------------------------------------------
```

```
C Get the solution vector
C---------------------------------------------------------------------------
      error = dss_solve_real( handle, MKL_DSS_DEFAULTS,
     & rhs, nRhs, solution)
      IF (error .NE. MKL_DSS_SUCCESS ) GOTO 999
C---------------------------------------------------------------------------
C Print Determinant of the matrix
C---------------------------------------------------------------------------
      statIn = 'determinant'
      call mkl_cvt_to_null_terminated_str(buff,bufLen,statIn)
      error = dss_statistics(handle, MKL_DSS_DEFAULTS,
     & buff,statOut)
      WRITE(*,"(' pow of determinant is ', 5(F10.3))") statOut(1)
      WRITE(*,"(' base of determinant is ', 5(F10.3))") statOut(2)
      WRITE(*,"(' Determinant is ', 5(F10.3))")(10**statOut(1))*
     & statOut(2)
C---------------------------------------------------------------------------
C Deallocate solver storage
C---------------------------------------------------------------------------
      error = dss_delete( handle, MKL_DSS_DEFAULTS )
      IF (error .NE. MKL_DSS_SUCCESS ) GOTO 999
C---------------------------------------------------------------------------
C Print solution vector
C---------------------------------------------------------------------------
      WRITE(*,900) (solution(i), i = 1, nCols)
  900 FORMAT(' Solution Array: ',5(F10.3))
      GOTO 1000
  999 WRITE(*,*) "Solver returned error code ", error
```

```
1000 END
```

Example C-11. C Example to Solve Symmetric Positive Definite System

```
/*
*******************************************************************************
*   Copyright(C) 2001-2004 Intel Corporation. All Rights Reserved.
*   The source code contained  or  described herein and all documents related
 to
*   the source code ("Material") are owned by Intel Corporation or its
suppliers
*   or licensors. Title to the  Material remains with  Intel Corporation or
 its
*   suppliers and licensors. The Material contains trade secrets and
proprietary
*   and  confidential information of  Intel or its suppliers and licensors.
 The
*   Material is  protected  by  worldwide  copyright  and trade secret laws
 and
*   treaty provisions. No part of the Material may be used, copied,
reproduced,
*   modified, published, uploaded, posted, transmitted, distributed or
disclosed
*   in any way without Intel's prior express written permission.
*   No license under any  patent, copyright, trade secret or other
intellectual
*   property right is granted to or conferred upon you by disclosure or
delivery
*   of the Materials, either expressly, by implication, inducement, estoppel
 or
*   otherwise. Any  license  under  such  intellectual property  rights must
 be
*   express and approved by Intel in writing.
*
*******************************************************************************
*   Content : Intel MKL DSS C example
*
```

```
*******************************************************************************/
/*

** Example program to solve symmetric positive definite system of equations.
*/
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include "mkl_dss.h"
/*
** Define the array and rhs vectors
*/
#define NROWS    5
#define NCOLS   5
#define NNONZEROS   9
#define NRHS   1
static const int nRows =     NROWS   ;
static const int nCols =     NCOLS   ;
static const int nNonZeros = NNONZEROS  ;
static const int nRhs =      NRHS   ;
static _INTEGER_t rowIndex[NROWS+1] = { 1, 6, 7, 8, 9, 10 };
static _INTEGER_t columns[NNONZEROS] = { 1, 2, 3, 4, 5, 2, 3, 4, 5 };
static _DOUBLE_PRECISION_t values[NNONZEROS] = { 9, 1.5, 6, .75, 3, 0.5, 12,
 .625, 16 };
static _DOUBLE_PRECISION_t rhs[NCOLS] = { 1, 2, 3, 4, 5 };


void main() {
   int i;
   /* Allocate storage for the solver handle and the right-hand side. */
```

```
   _DOUBLE_PRECISION_t solValues[NROWS];
   _MKL_DSS_HANDLE_t handle;
   _INTEGER_t error;
   _CHARACTER_STR_t statIn[] = "determinant";
   _DOUBLE_PRECISION_t statOut[5];
   int opt = MKL_DSS_DEFAULTS;
   int sym = MKL_DSS_SYMMETRIC;
   int type = MKL_DSS_POSITIVE_DEFINITE;
/* ---------------------*/
/* Initialize the solver */
/* ---------------------*/
   error = dss_create(handle, opt );
   if ( error != MKL_DSS_SUCCESS ) goto printError;
/* -------------------------------------------*/
/* Define the non-zero structure of the matrix */
/* -------------------------------------------*/
   error = dss_define_structure(
      handle, sym, rowIndex, nRows, nCols,
      columns, nNonZeros );
   if ( error != MKL_DSS_SUCCESS ) goto printError;
/* ------------------*/
/* Reorder the matrix */
/* ------------------*/
   error = dss_reorder(handle, opt, 0);
   if ( error != MKL_DSS_SUCCESS ) goto printError;
/* ------------------*/
/* Factor the matrix */
/* ------------------*/
```

```
   error = dss_factor_real(handle, type, values );
   if ( error != MKL_DSS_SUCCESS ) goto printError;
/* ------------------------*/
/* Get the solution vector */
/* ------------------------*/
   error = dss_solve_real(handle, opt, rhs, nRhs, solValues );
   if ( error != MKL_DSS_SUCCESS ) goto printError;
/* ------------------------*/
/* Get the determinant*/
/*-------------------------*/
   error = dss_statistics(handle, opt, statIn, statOut);
   if ( error != MKL_DSS_SUCCESS ) goto printError;
/*-------------------------*/
/* print determinant*/
/*-------------------------*/
   printf(" determinant power is %g \n", statOut[0]);
   printf(" determinant base is %g \n", statOut[1]);
   printf(" Determinant is %g \n", (pow(10.0,statOut[0]))*statOut[1]);
   free((void*) statIn);
/* -------------------------*/
/* Deallocate solver storage */
/* -------------------------*/
   error = dss_delete(handle, opt );
   if ( error != MKL_DSS_SUCCESS ) goto printError;
/* ----------------------*/
/* Print solution vector */
/* ----------------------*/
   printf(" Solution array: ");
```

```
   for(i = 0; i< nCols; i++)
      printf(" %g", solValues[i] );
   printf("\n");
   exit(0);
printError:
   printf("Solver returned error code %d\n", error);
   exit(1);
}
```

Example C-12. Fortran 90 Example to Solve Symmetric Positive Definite System

```
!*******************************************************************************
!   Copyright(C) 2001-2004 Intel Corporation. All Rights Reserved.
!   The source code contained or described herein and all documents related
 to
!   the source code ("Material") are owned by Intel Corporation or its
suppliers
!   or licensors. Title to the Material remains with Intel Corporation or
 its
!   suppliers and licensors. The Material contains trade secrets and
proprietary
!   and confidential information of Intel or its suppliers and licensors.
 The
!   Material is protected by worldwide copyright and trade secret laws
 and
!   treaty provisions. No part of the Material may be used, copied,
reproduced,
!   modified, published, uploaded, posted, transmitted, distributed or
disclosed
!   in any way without Intel's prior express written permission.
!   No license under any patent, copyright, trade secret or other
intellectual
!   property right is granted to or conferred upon you by disclosure or
delivery
!   of the Materials, either expressly, by implication, inducement, estoppel
 or
!   otherwise. Any license under such intellectual property rights must
 be
!   express and approved by Intel in writing.
!
!*******************************************************************************
!   Content : Intel MKL DSS Fortran-90 example
!
```

```
!*******************************************************************************
!-------------------------------------------------------------------------------
!
! Example program for solving a symmetric positive definite system of
! equations.
!
!-------------------------------------------------------------------------------
INCLUDE 'mkl_dss.f90' ! Include the standard DSS "header file."
PROGRAM solver_f90_test
use mkl_dss
IMPLICIT NONE
INTEGER, PARAMETER :: dp = KIND(1.0D0)
INTEGER :: error
INTEGER :: i
INTEGER, PARAMETER :: bufLen = 20
! Define the data arrays and the solution and rhs vectors.
INTEGER, ALLOCATABLE :: columns( : )
INTEGER :: nCols
INTEGER :: nNonZeros
INTEGER :: nRhs
INTEGER :: nRows
REAL(KIND=DP), ALLOCATABLE :: rhs( : )
INTEGER, ALLOCATABLE :: rowIndex( : )
REAL(KIND=DP), ALLOCATABLE :: solution( : )
REAL(KIND=DP), ALLOCATABLE :: values( : )
TYPE(MKL_DSS_HANDLE) :: handle ! Allocate storage for the solver handle.
REAL(KIND=DP),ALLOCATABLE::statOUt( : )
CHARACTER*15 statIn
```

```
INTEGER perm(1)
INTEGER buff(bufLen)
EXTERNAL MKL_CVT_TO_NULL_TERMINATED_STR
! Set the problem to be solved.
nRows = 5
nCols = 5
nNonZeros = 9
nRhs = 1
perm(1) = 0
ALLOCATE( rowIndex(nRows + 1 ) )
rowIndex = (/ 1, 6, 7, 8, 9, 10 /)
ALLOCATE( columns(nNonZeros ) )
columns = (/ 1, 2, 3, 4, 5, 2, 3, 4, 5 /)
ALLOCATE( values( nNonZeros ) )
values = (/ 9.0_DP, 1.5_DP, 6.0_DP, 0.75_DP, 3.0_DP, 0.5_DP, 12.0_DP, &
& 0.625_DP, 16.0_DP /)
ALLOCATE( rhs( nRows ) )
rhs = (/ 1.0_DP, 2.0_DP, 3.0_DP, 4.0_DP, 5.0_DP /)
! Initialize the solver.
error = dss_create( handle, MKL_DSS_DEFAULTS )
IF (error /= MKL_DSS_SUCCESS) GOTO 999
! Define the non-zero structure of the matrix.
error = dss_define_structure( handle, MKL_DSS_SYMMETRIC, rowIndex, nRows, &
& nCols, columns, nNonZeros )
IF (error /= MKL_DSS_SUCCESS) GOTO 999
! Reorder the matrix.
error = dss_reorder( handle, MKL_DSS_DEFAULTS, perm )
IF (error /= MKL_DSS_SUCCESS) GOTO 999
```

```
! Factor the matrix.
error = dss_factor_real( handle, MKL_DSS_DEFAULTS, values )
IF (error /= MKL_DSS_SUCCESS) GOTO 999
! Allocate the solution vector and solve the problem.
ALLOCATE( solution( nRows ) )
error = dss_solve_real(handle, MKL_DSS_DEFAULTS, rhs, nRhs, solution )
IF (error /= MKL_DSS_SUCCESS) GOTO 999
! Print Out the determinant of the matrix
ALLOCATE(statOut( 5 ) )
statIn = 'determinant'
call mkl_cvt_to_null_terminated_str(buff,bufLen,statIn);
error = dss_statistics(handle, MKL_DSS_DEFAULTS, buff, statOut )
IF (error /= MKL_DSS_SUCCESS) GOTO 999
WRITE(*,"('pow of determinant is '(5F10.3))") ( statOut(1) )
WRITE(*,"('base of determinant is '(5F10.3))") ( statOut(2) )
WRITE(*,"('Determinant is '(5F10.3))") ( (10**statOut(1))*statOut(2) )
! Deallocate solver storage and various local arrays.
error = dss_delete( handle, MKL_DSS_DEFAULTS )
IF (error /= MKL_DSS_SUCCESS ) GOTO 999
IF ( ALLOCATED( rowIndex) ) DEALLOCATE( rowIndex )
IF ( ALLOCATED( columns ) ) DEALLOCATE( columns )
IF ( ALLOCATED( values ) ) DEALLOCATE( values )
IF ( ALLOCATED( rhs ) ) DEALLOCATE( rhs )
IF ( ALLOCATED( statOut ) ) DEALLOCATE( statOut )
! Print the solution vector, deallocate it and exit
WRITE(*,"('Solution Array: '(5F10.3))") ( solution(i), i = 1, nCols )
IF ( ALLOCATED( solution ) ) DEALLOCATE( solution )
GOTO 1000
```

```
! Print an error message and exit
999 WRITE(*,*) "Solver returned error code ", error
1000 CONTINUE
END PROGRAM solver_f90_test
```

Iterative Sparse Solver Code Examples

This section contains example code in Fortran 77 and C. For description of the iterative sparse solver routines based on the reverse communication interface (RCI ISS) used in this code, refer to “Iterative Sparse Solvers based on Reverse Communication Interface (RCI ISS)” in Chapter 8 of the manual .

Example of Use RCI (Preconditioned) Conjugate Gradient Solver

Example results for symmetric positive definite systems. Upon successful execution of the solver, the result of the solution array is as follows:

```
The system is successfully solved

  The following solution obtained

     1.000     0.000     1.000     0.000
     1.000     0.000     1.000     0.000
  Expected solution

     1.000     0.000     1.000     0.000
     1.000     0.000     1.000     0.000
 Number of iterations: 8
```

Example C-13a. Fortran 77 Example to Solve Symmetric Positive Definite System

```
*******************************************************************************
*
* Copyright(C) 2001-2006 Intel Corporation. All Rights Reserved.
* The source code contained or described herein and all documents related
to
* the source code ("Material") are owned by Intel Corporation or its suppliers
* or licensors. Title to the Material remains with Intel Corporation or its
* suppliers and licensors. The Material contains trade secrets and proprietary
* and confidential information of Intel or its suppliers and licensors. The
* Material is protected by worldwide copyright and trade secret laws and
* treaty provisions. No part of the Material may be used, copied, reproduced,
* modified, published, uploaded, posted, transmitted, distributed or disclosed
* in any way without Intel's prior express written permission.
* No license under any patent, copyright, trade secret or other intellectual
* property right is granted to or conferred upon you by disclosure or delivery
* of the Materials, either expressly, by implication, inducement, estoppel
or
* otherwise. Any license under such intellectual property rights must be
* express and approved by Intel in writing.
*
*******************************************************************************
*
* Content : Intel MKL RCI (P)CG Fortran-77 example
*
*******************************************************************************
*
C-------------------------------------------------------------------------------
```

```
C Example program for solving symmetric positive definite system of
C equations.
C---------------------------------------------------------------------------
      PROGRAM rci_pcg_f77_test
      IMPLICIT NONE
C---------------------------------------------------------------------------
C Define arrays for the upper triangle of the coefficient matrix and rhs
vector
C Compressed sparse row storage is used for sparse representation
C---------------------------------------------------------------------------
      INTEGER N, RCI_request, itercount, i
      PARAMETER (N=8)
      DOUBLE PRECISION  rhs(N), solution(N)

     INTEGER ia(9)
      INTEGER ja(18)
      DOUBLE PRECISION a(18)
C.. Fill all arrays containing matrix data.
      DATA ia /1,5,8,10,12,15,17,18,19/
      DATA ja
     1 /1,  3, 6,7,
```

```
     2      2, 3,  5,
     3      3,          8,
     4        4,    7,
     5          5,6,7,
     6            6,  8,
     7              7,
     8                8/
      DATA a
     1 /7.D0,     1.D0,           2.D0, 7.D0,
     2      -4.D0,8.D0,     2.D0,
     3            1.D0,                      5.D0,
     4                 7.D0,      9.D0,
     5                      5.D0, 1.D0, 5.D0,
     6                           -1.D0,      5.D0,
     7                                11.D0,
     8                                      5.D0/
C-------------------------------------------------------------------------
C Allocate storage for the solver ?par and the initial solution vector
C-------------------------------------------------------------------------
      INTEGER length
      PARAMETER (length=128)
      DOUBLE PRECISION expected(N)
      DATA expected/1.D0, 0.D0, 1.D0, 0.D0, 1.D0, 0.D0, 1.D0, 0.D0/
      INTEGER ipar(length)
      DOUBLE PRECISION dpar(length),tmp(N,4)
C-------------------------------------------------------------------------
C Initialize the right hand side through matrix-vector product
C-------------------------------------------------------------------------
```

```
        CALL DCSRMV_SY('U', N, A, IA, JA, expected, rhs)
C-------------------------------------------------------------------------
C Initialize the initial guess
C-------------------------------------------------------------------------
        DO I=1, N
          solution(I)=1.D0
        ENDDO
```

```
C---------------------------------------------------------------------------
C Initialize the solver
C---------------------------------------------------------------------------
      CALL dcg_init(N,solution,rhs,RCI_request,ipar,dpar,tmp)
      IF (RCI_request .NE. 0 ) GOTO 999
C---------------------------------------------------------------------------
C Set the desired parameters:
C LOGICAL parameters:
C do residual stopping test
C do not request for the user defined stopping test
C do Preconditioned Conjugate Gradient iterations
C DOUBLE PRECISION parameters
C set the relative tolerance to 1.0D-5 instead of default value 1.0D-6
C---------------------------------------------------------------------------
      ipar(9)=1
      ipar(10)=0
      ipar(11)=1
      dpar(1)=1.D-5
C---------------------------------------------------------------------------
C Check the correctness and consistency of the newly set parameters
C---------------------------------------------------------------------------
      CALL dcg_check(N,solution,rhs,RCI_request,ipar,dpar,tmp)
      IF (RCI_request .NE. 0 ) GOTO 999
C---------------------------------------------------------------------------
C Compute the solution by RCI PCG solver
C Reverse Communications starts here
C---------------------------------------------------------------------------
1     CALL dcg(N,solution,rhs,RCI_request,ipar,dpar,tmp)
```

```
C--------------------------------------------------------------------------
C If RCI_request=0, then the solution was found with the required precision
C--------------------------------------------------------------------------
   IF (RCI_request .EQ. 0) THEN
          GOTO 700
C--------------------------------------------------------------------------
C If RCI_request=1, then compute the vector A*tmp(:,1)

C and put the result in vector tmp(:,2)
C--------------------------------------------------------------------------
     ELSE IF (RCI_request .EQ. 1) THEN
        CALL DCSRMV_SY('U', N, A, IA, JA, TMP, TMP(1, 2))
        GOTO 1
C--------------------------------------------------------------------------
C If RCI_request=3, then compute vector preconditioner matrix  on tmp(:,3)

C and put the result in vector tmp(:,4)
C--------------------------------------------------------------------------
      ELSE IF (RCI_request .EQ. 3) THEN
        CALL DCOPY(N, TMP(1,3),1, TMP(1, 4), 1)

        GOTO 1
      ELSE
C--------------------------------------------------------------------------
C If RCI_request=anything else, then dcg subroutine failed
```

```
C to compute the solution vector: solution(N)
C------------------------------------------------------------------------
              GOTO 999
      ENDIF
C------------------------------------------------------------------------
C Reverse Communication ends here
C Get the current iteration number
C------------------------------------------------------------------------
700   CALL dcg_get(N,solution,rhs,RCI_request,ipar,dpar,tmp,
     &             itercount)
C------------------------------------------------------------------------
C Print solution vector: solution(N) and number of iterations: itercount
C------------------------------------------------------------------------
      WRITE(*, *) ' The system is successfully solved '
      WRITE(*, *) ' The following solution obtained '
      WRITE(*,800)(solution(i),i =1,N)
      WRITE(*, *) ' Expected solution '
      WRITE(*,800)(expected(i),i =1,N)
800   FORMAT(4(F10.3))
      WRITE(*,900)(itercount)
900   FORMAT(' Number of iterations: ',1(I2))
      GOTO 1000
999   WRITE(*,*) 'Solver returned error code ', RCI_request
      STOP
1000  CONTINUE
      read *
      END
```

Fortran Example of Using RCI (Preconditioned) Flexible Generalized Minimal Residual Solver.

Fortran example results for a non-symmetric indefinite system. Upon successful execution of the solver, the following result is printed (up to rounding errors that depend on the computer system used):

```
-------------------------------------------------

  The SIMPLEST example of usage of RCI FGMRES solver
```

```
     to solve a non-symmetric indefinite non-degenerate

     algebraic system of linear equations

    --------------------------------------------------

      Some info about the current run of RCI FGMRES method:

     As IPAR(8)=1, the automatic test for the maximal number of iterations
will be performed

As IPAR(9)=1, the automatic residual test will be performed

 +++

 As IPAR(10)=0, the user-defined stopping test will not be requested, thus,
 RCI_REQUEST will not take the value 2

 +++

 As IPAR(11)=0, the Preconditioned FGMRES iterations will not be performed,
 thus, RCI_REQUEST will not take the value 3

 +++

 As IPAR(12)=1, the automatic test for the norm of the next generated vector
 is not equal to zero up to rounding and computational errors will be
performed, thus, RCI_REQUEST will not take the value 4

 +++

 The system has been SUCCESSFULLY solved

The following solution has been obtained:

 COMPUTED_SOLUTION(1)=-0.100E+01

COMPUTED_SOLUTION(2)= 0.100E+01

COMPUTED_SOLUTION(3)= 0.305E-15

 COMPUTED_SOLUTION(4)= 0.100E+01

 COMPUTED_SOLUTION(5)=-0.100E+01
```

```
The expected solution is:
 EXPECTED_SOLUTION(1)=-0.100E+01
 EXPECTED_SOLUTION(2)= 0.100E+01
 EXPECTED_SOLUTION(3)= 0.000E+00
 EXPECTED_SOLUTION(4)= 0.100E+01
 EXPECTED_SOLUTION(5)=-0.100E+01
 +++
 Number of iterations:        5
 +++
```

Example C-13b. Fortran Example to Solve Non-Symmetric Indefinite System

```
C*******************************************************************************
C
C Copyright(C) 2005-2006 Intel Corporation. All Rights Reserved.
C The source code contained or described herein and all documents related
to
C the source code ("Material") are owned by Intel Corporation or its suppliers
C or licensors. Title to the Material remains with Intel Corporation or its
C suppliers and licensors. The Material contains trade secrets and proprietary
C and confidential information of Intel or its suppliers and licensors. The
C Material is protected by worldwide copyright and trade secret laws and
C treaty provisions. No part of the Material may be used, copied, reproduced,
C modified, published, uploaded, posted, transmitted, distributed or disclosed
C in any way without Intel's prior express written permission.
C No license under any patent, copyright, trade secret or other intellectual
C property right is granted to or conferred upon you by disclosure or delivery
C of the Materials, either expressly, by implication, inducement, estoppel
or
C otherwise. Any license under such intellectual property rights must be
C express and approved by Intel in writing.
C
C*******************************************************************************
C  Content:
C Intel MKL RCI (P)FGMRES ((Preconditioned) Flexible Generalized Minimal
C
          RESidual method) example
C*******************************************************************************
C
C----------------------------------------------------------------------------
```

```
C  Example program for solving non-symmetric indefinite system of equations
C  Simplest case: no preconditioning and no user-defined stopping tests
C-------------------------------------------------------------------------
      PROGRAM DFGMRES_NO_PRECON_F
      INCLUDE "mkl_rci.fi"
INTEGER N
 PARAMETER(N=5)
 INTEGER SIZE
 PARAMETER (SIZE=128)
C-------------------------------------------------------------------------
 C Define arrays for the upper triangle of the coefficient matrix
 C Compressed sparse row storage is used for sparse representation
---------------------------------------------------------------------------
     INTEGER IA(6)
     DATA IA /1,3,6,9,12,14/
     INTEGER JA(13)
     DATA JA    /  1,        3,
    1              1,   2,        4,
    2                2,   3,        5,
    3             3,   4,   5,
    4              4,   5  /
      DOUBLE PRECISION A(13)
      DATA A     / 1.0,     -1.0,
     1                     -1.0, 1.0,     -1.0,
     2                      1.0,-2.0,      1.0,
     3                          -1.0, 2.0,-1.0,
     4                            -1.0,-3.0 /
C-------------------------------------------------------------------------
```

```
C Allocate storage for the ?par parameters and the solution/rhs vectors
C-----------------------------------------------------------------------
```

```
      INTEGER IPAR(SIZE)
      DOUBLE PRECISION DPAR(SIZE), TMP(N*(2*N+1)+(N*(N+9))/2+1)
      DOUBLE PRECISION EXPECTED_SOLUTION(N)
      DATA EXPECTED_SOLUTION /-1.0,1.0,0.0,1.0,-1.0/
      DOUBLE PRECISION RHS(N)
      DOUBLE PRECISION COMPUTED_SOLUTION(N)
C---------------------------------------------------------------------------
C Some additional variables to use with the RCI (P)FGMRES solver
C---------------------------------------------------------------------------
      INTEGER ITERCOUNT
      INTEGER RCI_REQUEST, I
      PRINT *,'------------------------------------------------'
      PRINT *,'The SIMPLEST example of usage of RCI FGMRES solver'
      PRINT *,'to solve a non-symmetric indefinite non-degenerate'
      PRINT *,'       algebraic system of linear equations'
      PRINT *,'------------------------------------------------'
C---------------------------------------------------------------------------
C Initialize variables and the right hand side through matrix-vector product
C---------------------------------------------------------------------------
      CALL MKL_DCSRGEMV('N', N, A, IA, JA, EXPECTED_SOLUTION, RHS)
C---------------------------------------------------------------------------
C Initialize the initial guess
C---------------------------------------------------------------------------
      DO I=1,N
         COMPUTED_SOLUTION(I)=1.0
      ENDDO
C---------------------------------------------------------------------------
C Initialize the solver
```

```
C---------------------------------------------------------------------------
      CALL DFGMRES_INIT(N, COMPUTED_SOLUTION, RHS, RCI_REQUEST, IPAR,
     1 DPAR, TMP)
      IF (RCI_REQUEST.NE.0) GOTO 999
C---------------------------------------------------------------------------
C Set the desired parameters:
C LOGICAL parameters:
C do residual stopping test
C do not request for the user defined stopping test
C do the check of the norm of the next generated vector automatically
C DOUBLE PRECISION parameters
C set the relative tolerance to 1.0D-3 instead of default value 1.0D-6
C---------------------------------------------------------------------------

      IPAR(9)=1
      IPAR(10)=0
      IPAR(12)=1
      DPAR(1)=1.0D-3
C---------------------------------------------------------------------------
C Check the correctness and consistency of the newly set parameters
C---------------------------------------------------------------------------
      CALL DFGMRES_CHECK(N, COMPUTED_SOLUTION, RHS, RCI_REQUEST,

     1 IPAR, DPAR, TMP)
      IF (RCI_REQUEST.NE.0) GOTO 999
C---------------------------------------------------------------------------
C Print the info about the RCI FGMRES method
```

```
C--------------------------------------------------------------------------
      PRINT *, ''
      PRINT *,'Some info about the current run of RCI FGMRES method:'
      PRINT *, ''
      IF (IPAR(8).NE.0) THEN
         WRITE(*,'(A,I1,A)') 'As IPAR(8)=',IPAR(8),', the automatic
     1 test for the maximal number of iterations will be'
         PRINT *,'performed'
      ELSE

         WRITE(*,'(A,I1,A)') 'As IPAR(8)=',IPAR(8),', the automatic test
     1 for the maximal number of iterations will be'
         PRINT *,'skipped'
      ENDIF
      PRINT *,'+++'
      IF (IPAR(9).NE.0) THEN
         WRITE(*,'(A,I1,A)') 'As IPAR(9)=',IPAR(9),', the automatic
     1 residual test will be performed'
      ELSE

         WRITE(*,'(A,I1,A)') 'As IPAR(9)=',IPAR(9),', the automatic
     1 residual test will be skipped'
      ENDIF
      PRINT *,'+++'
      IF (IPAR(10).NE.0) THEN
         WRITE(*,'(A,I1,A)') 'As IPAR(10)=',IPAR(10),', the user-defined
     1 stopping test will be requested via'
         PRINT *,'RCI_REQUEST=2'
      ELSE
```

```
    WRITE(*,'(A,I1,A)') 'As IPAR(10)=',IPAR(10),', the user-defined
1 stopping test will not be requested, thus,'
    PRINT *,'RCI_REQUEST will not take the value 2'
 ENDIF
 PRINT *,'+++'
 IF (IPAR(11).NE.0) THEN
    WRITE(*,'(A,I1,A)') 'As IPAR(11)=',IPAR(11),', the
1 Preconditioned FGMRES iterations will be performed, thus,'
    PRINT *,'the preconditioner action will be requested via
1 RCI_REQUEST=3'
 ELSE

    WRITE(*,'(A,I1,A)') 'As IPAR(11)=',IPAR(11),', the
1 Preconditioned FGMRES iterations will not be performed,'
    PRINT *,'thus, RCI_REQUEST will not take the value 3'
 ENDIF
 PRINT *,'+++'
 IF (IPAR(12).NE.0) THEN
    WRITE(*,'(A,I1,A)')'As IPAR(12)=',IPAR(12),', the automatic
1 test for the norm of the next generated vector is'
    PRINT *,'not equal to zero up to rounding and computational
1 errors will be performed,'
    PRINT *,'thus, RCI_REQUEST will not take the value 4'
 ELSE
```

```
          WRITE(*,'(A,I1,A)')'As IPAR(12)=',IPAR(12),', the automatic
     1 test for the norm of the next generated vector is'
          PRINT *,'not equal to zero up to rounding and computational
     1 errors will be skipped,'
          PRINT *,'thus, the user-defined test will be requested via
     1 RCI_REQUEST=4'
       ENDIF
       PRINT *,'+++'
C---------------------------------------------------------------------------
C Compute the solution by RCI (P)FGMRES solver without preconditioning
C Reverse Communication starts here
C---------------------------------------------------------------------------
1     CALL DFGMRES(N, COMPUTED_SOLUTION, RHS, RCI_REQUEST, IPAR,
     1 DPAR, TMP)
C---------------------------------------------------------------------------
C If RCI_REQUEST=0, then the solution was found with the required precision
C---------------------------------------------------------------------------
      IF (RCI_REQUEST.EQ.0) GOTO 3
C---------------------------------------------------------------------------
C If RCI_REQUEST=1, then compute the vector A*TMP(IPAR(22))

C and put the result in vector TMP(IPAR(23))
C---------------------------------------------------------------------------
      IF (RCI_REQUEST.EQ.1) THEN
         CALL MKL_DCSRGEMV('N',N, A, IA, JA, TMP(IPAR(22)), TMP(IPAR(23)))
         GOTO 1
C---------------------------------------------------------------------------
C If RCI_REQUEST=anything else, then DFGMRES subroutine failed
```

```
C to compute the solution vector: COMPUTED_SOLUTION(N)
C------------------------------------------------------------------------
      ELSE
         GOTO 999
      ENDIF
C------------------------------------------------------------------------
C Reverse Communication ends here
C Get the current iteration number and the FGMRES solution (DO NOT FORGET
to
C call DFGMRES_GET routine as COMPUTED_SOLUTION is still containing

C the initial guess!)
C------------------------------------------------------------------------
3     CALL DFGMRES_GET(N, COMPUTED_SOLUTION, RHS, RCI_REQUEST, IPAR,
     1 DPAR, TMP, ITERCOUNT)
C------------------------------------------------------------------------
C Print solution vector: COMPUTED_SOLUTION(N) and
```

```
C the number of iterations: ITERCOUNT
C------------------------------------------------------------------------
      PRINT *, ''
      PRINT *,' The system has been SUCCESSFULLY solved'
      PRINT *, ''
      PRINT *,' The following solution has been obtained:'
      DO I=1,N
         WRITE(*,'(A18,I1,A2,E10.3)') 'COMPUTED_SOLUTION(',I,')=',
     1 COMPUTED_SOLUTION(I)
      ENDDO
      PRINT *, ''
      PRINT *,' The expected solution is:'
      DO I=1,N
         WRITE(*,'(A18,I1,A2,E10.3)') 'EXPECTED_SOLUTION(',I,')=',
     1 EXPECTED_SOLUTION(I)
      ENDDO
      PRINT *, ''
      PRINT *,' Number of iterations: ',ITERCOUNT
      GOTO 1000

999   PRINT *,'The solver has returned the ERROR code ', RCI_REQUEST
1000  CONTINUE
      END
```

C Example of Using RCI (Preconditioned) Flexible Generalized Minimal Residual Solver.

C example results for the same non-symmetric indefinite system as in the previous example. The results are the same up to the notational convention between C and Fortran. Please pay special attention to how it is recommended to handle the differences between the Fortran and C arrays. Specifically, in this example we adjust the addresses for input/result for user-defined

operations from IPAR(22) to *ipar*[21]-1, and from IPAR(23) to *ipar*[22]-1, respectively. Upon successful execution of the solver, the following result is printed (up to rounding errors that depend on the computer system used):

```
------------------------------------------------


  The SIMPLEST example of usage of RCI FGMRES solver


     to solve a non-symmetric indefinite non-degenerate

     algebraic system of linear equations

    ------------------------------------------------

      Some info about the current run of RCI FGMRES method:


As ipar[7]=1, the automatic test for the maximal number of iterations will
be performed

+++

As ipar[8]=1, the automatic residual test will be performed

+++

As ipar[9]=0, the user-defined stopping test will not be requested, thus,

RCI_request will not take the value 2

+++

As ipar[10]=0, the Preconditioned FGMRES iterations will not be performed,
thus, RCI_request will not take the value 3

+++

As ipar[11]=1, the automatic test for the norm of the next generated vector
 is not equal to zero up to rounding and computational errors will be
performed, thus, RCI_request will not take the value 4

+++

 The system has been SUCCESSFULLY solved


 The following solution has been obtained:
```

```
computed_solution[0]=-1.000000e+000
computed_solution[1]=1.000000e+000
computed_solution[2]=3.053113e-016
computed_solution[3]=1.000000e+000
computed_solution[4]=-1.000000e+000
 The expected solution is:

expected_solution[0]=-1.000000e+000
expected_solution[1]=1.000000e+000
expected_solution[2]=0.000000e+000
expected_solution[3]=1.000000e+000
expected_solution[4]=-1.000000e+000
 Number of iterations: 5
```

Example C-13c. C Example to Solve Non-Symmetric Indefinite System

```
/*******************************************************************************

/*                                INTEL CONFIDENTIAL

/*   Copyright(C) 2005-2006 Intel Corporation. All Rights Reserved.

/*  The source code contained or described herein and all documents related
 to

/*  the source code ("Material")are owned by Intel Corporation or its
suppliers

/*  or licensors. Title to the  Material remains with  Intel Corporation or
 its

/*  suppliers and licensors.The Material contains trade secrets and
proprietary

/*  and confidential  information of  Intel or its suppliers and licensors.
 The

/*  Material  is  protected  by  worldwide  copyright  and trade secret laws
 and

/*  treaty provisions. No part of the Material may be used, copied,
reproduced,

/*  modified,published, uploaded, posted, transmitted, distributed or
disclosed

/*  in any way without Intel's prior express written permission.

/*  No license under any  patent, copyright, trade secret or other
intellectual

/*  property right is granted to or conferred upon you by disclosure or
delivery

/*  of the Materials,  either expressly, by implication, inducement, estoppel
 or

/*  otherwise.  Any  license  under  such  intellectual property  rights
must be

/*  express and approved by Intel in writing.

/*

/*******************************************************************************

/*  Content:
```

```
/*  Intel MKL RCI (P)FGMRES ((Preconditioned) Flexible Generalized Minimal

/*                                                        RESidual method)
example
/*******************************************************************************/
/*-------------------------------------------------------------------------------
/*  Example program for solving non-symmetric indefinite system of equations
/*  Simplest case: no preconditioning and no user-defined stopping tests
/*-------------------------------------------------------------------------------*/
#include <stdio.h>
#include "mkl_blas.h"
#include "mkl_spblas.h"
#include "mkl_rci.h"
#define N 5
#define size 128
int main(void)
{
```

```
/*---------------------------------------------------------------------------
   /* Define arrays for the upper triangle of the coefficient matrix
   /* Compressed sparse row storage is used for sparse representation
/*---------------------------------------------------------------------------*/
   int ia[6]={1,3,6,9,12,14};
   int ja[13]={    1,        3,
                   1,   2,        4,
                        2,   3,        5,
                             3,   4,   5,
                                  4,   5  };
   double A[13]={ 1.0,     -1.0,
                 -1.0, 1.0,     -1.0,
                       1.0,-2.0,      1.0,
                           -1.0, 2.0,-1.0,
                                 -1.0,-3.0 };
/*---------------------------------------------------------------------------
   /* Allocate storage for the ?par parameters and the solution/rhs vectors
/*---------------------------------------------------------------------------*/
```

```
   int ipar[size];
   double dpar[size], tmp[N*(2*N+1)+(N*(N+9))/2+1];
   double expected_solution[N]={-1.0,1.0,0.0,1.0,-1.0};
   double rhs[N];
   double computed_solution[N];

/*---------------------------------------------------------------------------
   /* Some additional variables to use with the RCI (P)FGMRES solver

/*---------------------------------------------------------------------------*/
   int itercount;
   int RCI_request, i, ivar;
   double dvar;
   char cvar;
   printf("-------------------------------------------------\n");
   printf("The SIMPLEST example of usage of RCI FGMRES solver\n");
   printf("to solve a non-symmetric indefinite non-degenerate\n");
   printf("       algebraic system of linear equations\n");
   printf("-------------------------------------------------\n\n");

/*---------------------------------------------------------------------------
   /* Initialize variables and the right hand side through matrix-vector
product

/*---------------------------------------------------------------------------*/
   ivar=N;
cvar='N';
   mkl_dcsrgemv(&cvar, &ivar, A, ia, ja, expected_solution, rhs);

/*---------------------------------------------------------------------------
   if (RCI_request!=0) goto FAILED;
```

```
/*---------------------------------------------------------------------------
   /* Set the desired parameters:
   /* LOGICAL parameters:
   /* do residual stopping test
   /* do not request for the user defined stopping test
   /* do the check of the norm of the next generated vector automatically
   /* DOUBLE PRECISION parameters
   /* set the relative tolerance to 1.0D-3 instead of default value 1.0D-6
/*---------------------------------------------------------------------------*/
   ipar[8]=1;
   ipar[9]=0;
   ipar[11]=1;
   dpar[0]=1.0E-3;
/*---------------------------------------------------------------------------
   /* Check the correctness and consistency of the newly set parameters
/*---------------------------------------------------------------------------*/
dfgmres_check(&ivar, computed_solution, rhs, &RCI_request, ipar, dpar, tmp);
   if (RCI_request!=0) goto FAILED;
/*---------------------------------------------------------------------------
   /* Print the info about the RCI FGMRES method

/*---------------------------------------------------------------------------*/
   printf("Some info about the current run of RCI FGMRES method:\n\n");
   if (ipar[7])
```

```
   {
      printf("As ipar[7]=%d, the automatic test for the maximal number of
iterations will be\n", ipar[7]);
      printf("performed\n");
   }
   else

   {
      printf("As ipar[7]=%d, the automatic test for the maximal number of
iterations will be\n", ipar[7]);
      printf("skipped\n");
   }
   printf("+++\n");
   if (ipar[8])

   {
     printf("As ipar[8]=%d, the automatic residual test will be performed\n",
 ipar[8]);
   }
   else

   {
     printf("As ipar[8]=%d, the automatic residual test will be skipped\n",
 ipar[8]);
   }
   printf("+++\n");
   if (ipar[9])
```

```
   {
     printf("As ipar[9]=%d, the user-defined stopping test will be requested
 via\n", ipar[9]);
      printf("RCI_request=2\n");
   }
   else

   {
      printf("As ipar[9]=%d, the user-defined stopping test will not be
requested, thus,\n", ipar[9]);
      printf("RCI_request will not take the value 2\n");
   }
   printf("+++\n");
   if (ipar[10])

   {
      printf("As ipar[10]=%d, the Preconditioned FGMRES iterations will be
performed, thus,\n", ipar[10]);
      printf("the preconditioner action will be requested via
RCI_request=3\n");
   }
   else

   {
      printf("As ipar[10]=%d, the Preconditioned FGMRES iterations will not
 be performed,\n", ipar[10]);
      printf("thus, RCI_request will not take the value 3\n");
   }
   printf("+++\n");
   if (ipar[11])
```

```
   {
      printf("As ipar[11]=%d, the automatic test for the norm of the next
generated vector is\n", ipar[11]);
     printf("not equal to zero up to rounding and computational errors will
 be performed,\n");
      printf("thus, RCI_request will not take the value 4\n");
   }
   else
```

```
   {
      printf("As ipar[11]=%d, the automatic test for the norm of the next
generated vector is\n", ipar[11]);
     printf("not equal to zero up to rounding and computational errors will
 be skipped,\n");
      printf("thus, the user-defined test will be requested via
RCI_request=4\n");
   }
   printf("+++\n\n");

/*---------------------------------------------------------------------------

   /* Compute the solution by RCI (P)FGMRES solver without preconditioning
   /* Reverse Communication starts here

/*---------------------------------------------------------------------------*/
ONE:  dfgmres(&ivar, computed_solution, rhs, &RCI_request, ipar, dpar, tmp);

/*---------------------------------------------------------------------------

   /* If RCI_request=0, then the solution was found with the required
precision

/*---------------------------------------------------------------------------*/
   if (RCI_request==0) goto COMPLETE;

/*---------------------------------------------------------------------------

   /* If RCI_request=1, then compute the vector A*tmp[ipar[21]-1]

   /* and put the result in vector tmp[ipar[22]-1]

/*---------------------------------------------------------------------------

   /* NOTE that ipar[21] and ipar[22] contain FORTRAN style addresses,
   /* therefore, in C code it is required to subtract 1 from them to get
```

```
   /* C style addresses

/*---------------------------------------------------------------------------*/
   if (RCI_request==1)
   {
mkl_dcsrgemv(&cvar, &ivar, A, ia, ja, &tmp[ipar[21]-1], &tmp[ipar[22]-1]);
      goto ONE;
   }

/*---------------------------------------------------------------------------
   /* If RCI_request=anything else, then dfgmres subroutine failed

   /* to compute the solution vector: computed_solution[N]

/*---------------------------------------------------------------------------*/
   else
   {
      goto FAILED;
   }

/*---------------------------------------------------------------------------
   /* Reverse Communication ends here
   /* Get the current iteration number and the FGMRES solution (DO NOT FORGET
   /* to call dfgmres_get routine as computed_solution is still containing
```

```
   /* the initial guess!)

/*---------------------------------------------------------------------------*/

COMPLETE:   dfgmres_get(&ivar, computed_solution, rhs, &RCI_request, ipar,
dpar, tmp, &itercount);

   /*

/*---------------------------------------------------------------------------

   /* Print solution vector: computed_solution[N] and the number of
iterations: itercount

/*---------------------------------------------------------------------------*/

   printf(" The system has been SUCCESSFULLY solved \n");

   printf("\n The following solution has been obtained: \n");

   for (i=0;i<N;i++)
printf("computed_solution[%d]=%e\n",i,computed_solution[i]);

   printf("\n The expected solution is: \n");

   for (i=0;i<N;i++)
printf("expected_solution[%d]=%e\n",i,expected_solution[i]);

   printf("\n Number of iterations: %d",itercount);

   goto SUCCEDED;

FAILED: printf("The solver has returned the ERROR code %d", RCI_request);


SUCCEDED: return 0;

}
```

Fourier Transform Functions Code Examples

This section presents code examples of functions described in the "DFT Functions" and "Cluster DFT Functions" sections in the "Fourier Transform Functions" chapter. The examples are grouped in subsections

- Examples for DFT Functions, including Examples of Using Multi-Threading for DFT Computation
- Examples for Cluster DFT Functions.

DFT Code Examples

This section presents code examples of using the DFT interface functions described in "Fourier Transform Functions" chapter. Here are the examples of two one-dimensional computations. These examples use the default settings for all of the configuration parameters, which are specified in "Configuration Settings".

Example C-16 One-dimensional In-place DFT (Fortran Interface)

```
! Fortran example.
! 1D complex to complex, and real to conjugate even
Use MKL_DFTI
Complex :: X(32)
Real :: Y(34)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc1_Handle, My_Desc2_Handle
Integer :: Status
...put input data into X(1),...,X(32); Y(1),...,Y(32)

! Perform a complex to complex transform
Status = DftiCreateDescriptor( My_Desc1_Handle, DFTI_SINGLE,
   DFTI_COMPLEX, 1, 32 )
Status = DftiCommitDescriptor( My_Desc1_Handle )
Status = DftiComputeForward( My_Desc1_Handle, X )
Status = DftiFreeDescriptor(My_Desc1_Handle)

! result is given by {X(1),X(2),...,X(32)}
```

```
! Perform a real to complex conjugate even transform
Status = DftiCreateDescriptor(My_Desc2_Handle, DFTI_SINGLE,
   DFTI_REAL, 1, 32)
Status = DftiCommitDescriptor(My_Desc2_Handle)
Status = DftiComputeForward(My_Desc2_Handle, Y)
Status = DftiFreeDescriptor(My_Desc2_Handle)
! result is given in CCS format.
```

Example C-16a One-dimensional Out-of-place DFT (Fortran Interface)

```
! Fortran example.
! 1D complex to complex, and real to conjugate even
Use MKL_DFTI
Complex :: X_in(32)
Complex :: X_out(32)
Real :: Y_in(32)
Real :: Y_out(34)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc1_Handle, My_Desc2_Handle
Integer :: Status
...put input data into X_in(1),...,X_in(32); Y_in(1),...,Y_in(32)
! Perform a complex to complex transform
Status = DftiCreateDescriptor( My_Desc1_Handle, DFTI_SINGLE,
DFTI_COMPLEX, 1, 32 )
Status = DftiSetValue( My_Desc1_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE)
Status = DftiCommitDescriptor( My_Desc1_Handle )
Status = DftiComputeForward( My_Desc1_Handle, X_in, X_out )
Status = DftiFreeDescriptor(My_Desc1_Handle)
! result is given by {X_out(1),X_out(2),...,X_out(32)}
! Perform a real to complex conjugate even transform
Status = DftiCreateDescriptor(My_Desc2_Handle, DFTI_SINGLE,
DFTI_REAL, 1, 32)
Status = DftiSetValue( My_Desc2_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE)
Status = DftiCommitDescriptor(My_Desc2_Handle)
Status = DftiComputeForward(My_Desc2_Handle, Y_in, Y_out)
Status = DftiFreeDescriptor(My_Desc2_Handle)
! result is given by Y_out in CCS format.
```

Example C-17 One-dimensional In-place DFT (C Interface)

```
/* C example, float _Complex is defined in C9X */
#include "mkl_dfti.h"
float _Complex x[32];
float y[34];
dfti_descriptor *my_desc1_handle,*my_desc2_handle;
/* .... or alternatively
dfti_descriptor_handle my_desc1_handle, my_desc2_handle; */
long status;
...put input data into x[0],...,x[31]; y[0],...,y[31]
status = DftiCreateDescriptor( &my_desc1_handle, DFTI_SINGLE,
          DFTI_COMPLEX, 1, 32);
status = DftiCommitDescriptor( my_desc1_handle );
status = DftiComputeForward( my_desc1_handle, x);
status = DftiFreeDescriptor(&my_desc1_handle);
/* result is x[0], ..., x[31]*/
status = DftiCreateDescriptor( &my_desc2_handle, DFTI_SINGLE,
          DFTI_REAL, 1, 32);
status = DftiCommitDescriptor( my_desc2_handle);
status = DftiComputeForward( my_desc2_handle, y);
status = DftiFreeDescriptor(&my_desc2_handle);
/* result is given in CCS format*/
```

Example C-17a One-dimensional Out-of-place DFT (C Interface)

```
/* C example, float _Complex is defined in C9X */
#include "mkl_dfti.h"
float _Complex x_in[32];
float _Complex x_out[32];
float y_in[32];
float y_out[34];
dfti_descriptor *my_desc1_handle,*my_desc2_handle;
/* .... or alternatively
dfti_descriptor_handle my_desc1_handle, my_desc2_handle; */
long status;
...put input data into x_in[0],...,x_in[31]; y_in[0],...,y_in[31]
status = DftiCreateDescriptor( &my_desc1_handle, DFTI_SINGLE,

          DFTI_COMPLEX, 1, 32);
Status = DftiSetValue( My_Desc1_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE);
status = DftiCommitDescriptor( my_desc1_handle );
status = DftiComputeForward( my_desc1_handle, x_in, x_out);
status = DftiFreeDescriptor(&my_desc1_handle);
/* result is x_out[0], ..., x_out[31]*/
status = DftiCreateDescriptor( &my_desc2_handle, DFTI_SINGLE,

          DFTI_REAL, 1, 32);
Status = DftiSetValue( My_Desc2_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE);
status = DftiCommitDescriptor( my_desc2_handle);
status = DftiComputeForward( my_desc2_handle, y_in, y_out);
status = DftiFreeDescriptor(&my_desc2_handle);
/* result is given by y_out in CCS format*
```

The following is an example of two simple two-dimensional transforms. Notice that the data and result parameters in computation functions are all declared as assumed-size rank-1 array `DIMENSION(0:*)`. Therefore two-dimensional array must be transformed to one-dimensional array by `EQUIVALENCE` statement or other facilities of Fortran.

Example C-18 Two-dimensional DFT (Fortran Interface)

```
! Fortran example.

! 2D complex to complex, and real to conjugate even

Use MKL_DFTI

Complex ::  X_2D(32,100)

Real :: Y_2D(34, 102)

Complex ::  X(3200)

Real :: Y(3468)

Equivalence (X_2D, X)

Equivalence (Y_2D, Y)

type(DFTI_DESCRIPTOR), POINTER :: My_Desc1_Handle, My_Desc2_Handle

Integer :: Status, L(2)

...put input data into X_2D(j,k), Y_2D(j,k), 1<=j=32,1<=k<=100

...set L(1) = 32, L(2) = 100

...the transform is a 32-by-100

! Perform a complex to complex transform

Status = DftiCreateDescriptor( My_Desc1_Handle, DFTI_SINGLE,

          DFTI_COMPLEX, 2, L)

Status = DftiCommitDescriptor( My_Desc1_Handle)

Status = DftiComputeForward( My_Desc1_Handle, X)

Status = DftiFreeDescriptor(My_Desc1_Handle)

! result is given by X_2D(j,k), 1<=j<=32, 1<=k<=100
```

```
! Perform a real to complex conjugate even transform
Status = DftiCreateDescriptor( My_Desc2_Handle, DFTI_SINGLE,
          DFTI_REAL, 2, L)
Status = DftiCommitDescriptor( My_Desc2_Handle)
Status = DftiComputeForward( My_Desc2_Handle, Y)
Status = DftiFreeDescriptor(My_Desc2_Handle)
! result is given by the complex value z(j,k) 1<=j<=32; 1<=k<=100
! and is stored in CCS format
```

Example C-19 Two-dimensional DFT (C Interface)

```
/* C example */
#include "mkl_dfti.h"
float _Complex x[32][100];
float y[34][102];
dfti_descriptor_handle my_desc1_handle, my_desc2_handle;
/* or alternatively
dfti_descriptor *my_desc1_handle,*my_desc2_handle; */
long status, l[2];
...put input data into x[j][k] 0<=j<=31, 0<=k<=99
...put input data into y[j][k] 0<=j<=31, 0<=k<=99
l[0] = 32; l[1] = 100;
status = DftiCreateDescriptor( &my_desc1_handle, DFTI_SINGLE,
          DFTI_COMPLEX, 2, l);
status = DftiCommitDescriptor( my_desc1_handle);
status = DftiComputeForward( my_desc1_handle, x);
status = DftiFreeDescriptor(&my_desc1_handle);
/* result is the complex value x[j][k], 0<=j<=31, 0<=k<=99 */
status = DftiCreateDescriptor( &my_desc2_handle, DFTI_SINGLE,
          DFTI_REAL, 2, l);
status = DftiCommitDescriptor( my_desc2_handle);
status = DftiComputeForward( my_desc2_handle, y);
status = DftiFreeDescriptor(&my_desc2_handle);
/* result is the complex value z(j,k) 0<=j<=31; 0<=k<=99
/* and is stored in CCS format*/
```

The following examples demonstrate how you can change the default configuration settings by using the `DftiSetValue` function.

For instance, to preserve the input data after the DFT computation, the configuration of the DFTI_PLACEMENT should be changed to "not in place" from the default choice of "in place."

The code below illustrates how this can be done:

Example C-20 Changing Default Settings (Fortran)

```
! Fortran example

! 1D complex to complex, not in place

Use MKL_DFTI

Complex :: X_in(32), X_out(32)

type(DFTI_DESCRIPTOR), POINTER :: My_Desc_Handle

Integer :: Status

...put input data into X_in(j), 1<=j<=32

Status = DftiCreateDescriptor( My_Desc_Handle,
DFTI_SINGLE, DFTI_COMPLEX, 1, 32)

Status = DftiSetValue( My_Desc_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE)

Status = DftiCommitDescriptor( My_Desc_Handle)

Status = DftiComputeForward( My_Desc_Handle, X_in, X_out)

Status = DftiFreeDescriptor (My_Desc_Handle)

! result is X_out(1),X_out(2),...,X_out(32)
```

Example C-21 Changing Default Settings (C)

```
/* C example */
#include "mkl_dfti.h"
float  _Complex x_in[32], x_out[32];
DFTI_DESCRIPTOR_HANDLE my_desc_handle;
/* or alternatively
DFTI_DESCRIPTOR *my_desc_handle;*/
long status;
...put input data into x_in[j], 0 <= j < 32
status = DftiCreateDescriptor( &my_desc_handle, DFTI_SINGLE,
   DFTI_COMPLEX, 1, 32);
status = DftiSetValue( my_desc_handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE);
status = DftiCommitDescriptor( my_desc_handle);
status = DftiComputeForward( my_desc_handle, x_in, x_out);
status = DftiFreeDescriptor(&my_desc_handle);
/* result is x_out[0], x_out[1], ..., x_out[31] */
```

The Example C-22 below illustrates the use of the status checking functions described in Chapter 11 .

Example C-22 Using Status Checking Function

```
from C language:


DFTI_DESCRIPTOR_HANDLE desc;
long status, class_error, value;
char* error_message;
 . . . descriptor creation and other code
status = DftiGetValue( desc, DFTI_PRECISION, &value); //
```

```
//or any DFTI function

class_error = DftiErrorClass(status, DFTI_NO_ERROR);
if (! class_error) {
    printf ("DftiGetValue() fixes the wrong situation and

             returns the corresponding value n");
    error_message = DftiErrorMessage(status);
    printf("error_message = %s \n", error_message);
}
. . .
from Fortran:

type(DFTI_DESCRIPTOR), POINTER :: desc

integer value, status
character(DFTI_MAX_MESSAGE_LENGTH) error_message
logical class_error
 . . . descriptor creation and other code
status = DftiGetValue( desc, DFTI_PRECISION, value)
class_error = DftiErrorClass(status, DFTI_NO_ERROR)
if (.not. class_error) then
    print *, ' DftiGetValue() fixes the wrong situation and

             returns the corresponding value '
    error_message = DftiErrorMessage(status)
    print *, 'error_message = ', error_message
endif
```

Below is an example where a 20-by-40 two-dimensional DFT is computed explicitly using one-dimensional transforms. Notice that the data and result parameters in computation functions are all declared as assumed-size rank-1 array `DIMENSION(0:*)`. Therefore two-dimensional array must be transformed to one-dimensional array by `EQUIVALENCE` statement or other facilities of Fortran.

Example C-23 Computing 2D DFT by One-Dimensional Transforms

```
! Fortran
Complex ::  X_2D(20,40),
Complex ::  X(800)
Equivalence (X_2D, X)
INTEGER :: STRIDE(2)
type(DFTI_DESCRIPTOR), POINTER
:: Desc_Handle_Dim1
type(DFTI_DESCRIPTOR), POINTER
:: Desc_Handle_Dim2
...
Status = DftiCreateDescriptor(Desc_Handle_Dim1, DFTI_SINGLE,
                                  DFTI_COMPLEX, 1, 20 )
Status = DftiCreateDescriptor(Desc_Handle_Dim2, DFTI_SINGLE,
                                  DFTI_COMPLEX, 1, 40 )
! perform 40 one-dimensional transforms along 1st dimension
Status = DftiSetValue(
Desc_Handle_Dim1, DFTI_NUMBER_OF_TRANSFORMS, 40 )
Status = DftiSetValue( Desc_Handle_Dim1,
DFTI_INPUT_DISTANCE, 20 )
Status = DftiSetValue( Desc_Handle_Dim1,
DFTI_OUTPUT_DISTANCE, 20 )
Status = DftiCommitDescriptor( Desc_Handle_Dim1 )
Status = DftiComputeForward( Desc_Handle_Dim1, X )
! perform 20 one-dimensional transforms along 2nd dimension
Stride(1) = 0; Stride(2) = 20
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_NUMBER_OF_TRANSFORMS, 20 )
```

```
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_INPUT_DISTANCE, 1 )
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_OUTPUT_DISTANCE, 1 )
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_INPUT_STRIDES, Stride )
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_OUTPUT_STRIDES, Stride )
Status = DftiCommitDescriptor( Desc_Handle_Dim2 )
Status = DftiComputeForward( Desc_Handle_Dim2, X )
Status = DftiFreeDescriptor( Desc_Handle_Dim1 )
Status = DftiFreeDescriptor( Desc_Handle_Dim2 )
```

```
/* C */
float _Complex x[20][40];
long stride[2];
long status;
DFTI_DESCRIPTOR_HANDLE desc_handle_dim1;
DFTI_DESCRIPTOR_HANDLE desc_handle_dim2;
...
status = DftiCreateDescriptor( &desc_handle_dim1, DFTI_SINGLE,
 DFTI_COMPLEX, 1, 20 );
status = DftiCreateDescriptor( &desc_handle_dim2, DFTI_SINGLE,
 DFTI_COMPLEX, 1, 40 );

/* perform 40 one-dimensional transforms along 1st dimension */
/* note that the 1st dimension data are not unit-stride */
stride[0] = 0; stride[1] = 40;
status = DftiSetValue( desc_handle_dim1, DFTI_NUMBER_OF_TRANSFORMS, 40 );
status = DftiSetValue( desc_handle_dim1, DFTI_INPUT_DISTANCE, 1 );
status = DftiSetValue( desc_handle_dim1, DFTI_OUTPUT_DISTANCE, 1 );
status = DftiSetValue( desc_handle_dim1, DFTI_INPUT_STRIDES, stride );
status = DftiSetValue( desc_handle_dim1, DFTI_OUTPUT_STRIDES, stride );
status = DftiCommitDescriptor( desc_handle_dim1 );
status = DftiComputeForward( desc_handle_dim1, x );
/* perform 20 one-dimensional transforms along 2nd dimension */
/* note that the 2nd dimension is unit stride */
status = DftiSetValue( desc_handle_dim2, DFTI_NUMBER_OF_TRANSFORMS, 20 );
status = DftiSetValue( desc_handle_dim2,
DFTI_INPUT_DISTANCE, 40 );
status = DftiSetValue( desc_handle_dim2,
```

```
DFTI_OUTPUT_DISTANCE, 40 );
status = DftiCommitDescriptor( desc_handle_dim2 );
status = DftiComputeForward( desc_handle_dim2, x );
status = DftiFreeDescriptor( &Desc_Handle_Dim1 );
status = DftiFreeDescriptor( &Desc_Handle_Dim2 );
```

The following are examples of real multi-dimensional transforms with CCE format storage of conjugate-even complex matrix. Example C-24 is two-dimensional in-place transform and Example C-24a is two-dimensional out-of-place transform in Fortran interface. Example C-25 is three-dimensional out-of-place transform in C interface. Notice that the data and result parameters in computation functions are all declared as assumed-size rank-1 array `DIMENSION(0:*)`. Therefore two-dimensional array must be transformed to one-dimensional array by `EQUIVALENCE` statement or other facilities of Fortran.

Example C-24 Two-Dimensional REAL In-place DFT (Fortran Interface)

```
! Fortran example.
! 2D and real to conjugate even
Use MKL_DFTI
Real :: X_2D(34,100) ! 34  = (32/2 + 1)*2
Real :: X(3400)
Equivalence (X_2D, X)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc_Handle
Integer :: Status, L(2)
Integer :: strides_in(3)
Integer :: strides_out(3)
...put input data into X_2D(j,k), 1<=j=32,1<=k<=100
...set L(1) = 32, L(2) = 100
...set strides_in(1) = 0, strides_in(2) = 1, strides_in(3) = 34
...set strides_out(1) = 0, strides_out(2) = 1, strides_out(3) = 17
...the transform is a 32-by-100
! Perform a real to complex conjugate even transform
Status = DftiCreateDescriptor( My_Desc_Handle, DFTI_SINGLE,
DFTI_REAL, 2, L )
Status = DftiSetValue(My_Desc_Handle, DFTI_CONJUGATE_EVEN_STORAGE,
DFTI_COMPLEX_COMPLEX)
Status = DftiSetValue(My_Desc_Handle, DFTI_INPUT_STRIDES, strides_in)
Status = DftiSetValue(My_Desc_Handle, DFTI_OUTPUT_STRIDES, strides_out)
Status = DftiCommitDescriptor( My_Desc_Handle)
Status = DftiComputeForward( My_Desc_Handle, X )
Status = DftiFreeDescriptor(My_Desc_Handle)
! result is given by the complex value z(j,k) 1<=j<=17; 1<=k<=100 and
! is stored in real matrix X_2D in CCE format.
```

Example C-24a Two-Dimensional REAL Out-of-place DFT (Fortran Interface)

```
! Fortran example.
! 2D and real to conjugate even
Use MKL_DFTI
Real :: X_2D(32,100)
Complex :: Y_2D(17, 100) ! 17 = 32/2 + 1
```

```
Real :: X(3200)
Complex :: Y(1700)
Equivalence (X_2D, X)
Equivalence (Y_2D, Y)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc_Handle
Integer :: Status, L(2)
Integer :: strides_out(3)

...put input data into X_2D(j,k), 1<=j=32,1<=k<=100
...set L(1) = 32, L(2) = 100
...set strides_out(1) = 0, strides_out(2) = 1, strides_out(3) = 17

...the transform is a 32-by-100
! Perform a real to complex conjugate even transform
Status = DftiCreateDescriptor( My_Desc_Handle, DFTI_SINGLE,
DFTI_REAL, 2, L )
Status = DftiSetValue(My_Desc_Handle,
DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX)
Status = DftiSetValue( My_Desc_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE )
Status = DftiSetValue(My_Desc_Handle,
DFTI_OUTPUT_STRIDES, strides_out)

Status = DftiCommitDescriptor(My_Desc_Handle)
Status = DftiComputeForward(My_Desc_Handle, X, Y)
Status = DftiFreeDescriptor(My_Desc_Handle)
! result is given by the complex value z(j,k) 1<=j<=17; 1<=k<=100 and
! is stored in complex matrix Y_2D in CCE format.
```

Example C-25 Three-Dimensional REAL DFT (C Interface)

```
/* C example */
#include "mkl_dfti.h"
float x[32][100][19];
float _Complex y[32][100][10]; /* 10 = 19/2 + 1 */
DFTI_DESCRIPTOR_HANDLE my_desc_handle
/* or alternatively
DFTI_DESCRIPTOR *my_desc_handle*/
long status, l[3];
long strides_out[4];

...put input data into x[j][k][s] 0<=j<=31, 0<=k<=99, 0<=s<=18
```

```
l[0] = 32; l[1] = 100; l[2] =
19;
strides_out[0] = 0; strides_out[1] = 1000;
strides_out[2] = 10; strides_out[3] = 1;

status = DftiCreateDescriptor( &my_desc_handle, DFTI_SINGLE,
DFTI_REAL, 3, l );
Status = DftiSetValue(my_desc_handle,
DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX);
Status = DftiSetValue( my_desc_handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE );
Status = DftiSetValue(my_desc_handle,
DFTI_OUTPUT_STRIDES, strides_out);

status = DftiCommitDescriptor(my_desc_handle);
status = DftiComputeForward(my_desc_handle, x, y);
status = DftiFreeDescriptor(&my_desc_handle);
/* result is the complex value z(j,k,s) 0<=j<=31; 0<=k<=99, 0<=s<=9
and is stored in complex matrix y in CCE format. */
```

Examples of Using Multi-Threading for DFT Computation

The following example program shows how to employ internal threading in Intel MKL for DFT computation (see case 1 in "Number of user threads").

To specify the number of threads inside Intel MKL, use the following settings:

`set OMP_NUM_THREADS = 1` for one-threaded mode;

`set OMP_NUM_THREADS = 4` for multi-threaded mode.

Note that the configuration parameter `DFTI_NUMBER_OF_USER_THREADS` must be equal to its default value 1.

Example C-26 Using Intel MKL Internal Threading Mode

```
#include "mkl_dfti.h"

void main () {

float x[200][100];
DFTI_DESCRIPTOR_HANDLE my_desc1_handle;
long status, len[2];
//...put input data into x[j][k] 0<=j<=199, 0<=k<=99
len[0] = 200; len[1] = 100;
status = DftiCreateDescriptor( &my_desc1_handle, DFTI_SINGLE,DFTI_REAL, 2,
 len);
status = DftiCommitDescriptor(
my_desc1_handle);
status = DftiComputeForward(
my_desc1_handle, x);
status = DftiFreeDescriptor(&my_desc1_handle);
}
```

The following Example C-27 illustrates a parallel customer program with each descriptor instance used only in a single thread (see case 2 in "Number of user threads").

To specify the number of threads, use the following settings:

set `MKL_SERIAL = yes` (or `YES`) for single-threaded mode in Intel MKL (recommended);

`set OMP_NUM_THREADS = 4` for multi-threaded mode in customer program.

The configuration parameter `DFTI_NUMBER_OF_USER_THREADS` must be equal to its default value 1.

Note that in this example the program can be transformed to become single-threaded on the customer level but using parallel mode within Intel MKL. To achieve this, you need to set the parameter `DFTI_NUMBER_OF_TRANSFORMS = 4` and to set the corresponding parameter `DFTI_INPUT_DISTANCE = 5000`.

Example C-27 Using Parallel Mode with Multiple Descriptors

```
#include "mkl_dfti.h"
#include "omp.h"
void main ()
{
   float _Complex x[200][100];
   long len[2];
   //...put input data into x[j][k] 0<=j<=199, 0<=k<=99
   len[0] = 50; len[1] = 100;

   // each thread calculates real DFT for matrix (50*100)
   #pragma omp parallel
   {
      DFTI_DESCRIPTOR_HANDLE my_desc_handle;
      long myStatus;
      int myID = omp_get_thread_num ();

myStatus    =   DftiCreateDescriptor (my_desc_handle, DFTI_SINGLE,
DFTI_COMPLEX, 2, len);
      myStatus = DftiCommitDescriptor (my_desc_handle);
      myStatus = DftiComputeForward (my_desc_handle, &x[myID * len[0] *
len[1]]);
      myStatus = DftiFreeDescriptor (&my_desc_handle);
   }   /* End OpenMP parallel region */
}
```

The following Example C-28 illustrates a parallel customer program with a common descriptor used in several threads (see case 3 in "Number of user threads".

In this case the number of threads, as well as any other configuration parameter, must not be changed after DFT initialization by the `DftiCommitDescriptor()` function is done.

Example C-28 Using Parallel Mode with a Common Descriptor

```
// set number of threads inside Intel MKL:
//rem set MKL_SERIAL = YES   -   is not required

// since one-threaded mode for Intel MKL is forced automatically
```

```
// set OMP_NUM_THREADS = 4   -   multi-threaded mode for customer

#include "mkl_dfti.h"
#include "omp.h"
void main ()
{
   float _Complex x[200][100];
   long status;
   DFTI_DESCRIPTOR_HANDLE desc_handle;
   int nThread = omp_get_max_threads ();
   long len[2];
   //...put input data into x[j][k] 0<=j<=199, 0<=k<=99
   len[0] = 50; len[1] = 100;

   status =
DftiCreateDescriptor (desc_handle, DFTI_SINGLE, DFTI_COMPLEX, 2, len);
   status = DftiSetValue (desc_handle,
DFTI_NUMBER_OF_USER_THREADS, nThread);
   status = DftiCommitDescriptor (desc_handle);

   // each thread calculates real DFT for matrix (50*100)
   #pragma omp parallel num_threads(nThread)
   {
      long myStatus;
      int myID = omp_get_thread_num ();

     myStatus = DftiComputeForward (desc_handle, &x[myID * len[0] * len[1]);
   }   /* End OpenMP parallel region */
```

```
    status = DftiFreeDescriptor (&desc_handle);
}
```

Examples for Cluster DFT Functions

The C example below computes 2-dimensional out-of-place FFT using Cluster DFT:

Example 29 2D Out-of-place Cluster DFT Computation

```
DFTI_DESCRIPTOR_DM_HANDLE desc;
long len[2],v,i,j,n,s;
Complex *in,*out;
MPI_Init(...);
// Create descriptor for 2D FFT
len[0]=nx;
len[1]=ny;
DftiCreateDescriptorDM(MPI_COMM_WORLD,&desc,DFTI_DOUBLE,DFTI_COMPLEX,2,l
en);
// Ask necessary length of in and out arrays and allocate memory
DftiGetValueDM(desc,CDFT_LOCAL_SIZE,&v);
in=(Complex*)malloc(v*sizeof(Complex));
out=(Complex*)malloc(v*sizeof(Complex));
// Fill local array with initial data. Current process performs n rows,

// 0 row of in corresponds to s row of virtual global array
DftiGetValueDM(desc,CDFT_LOCAL_NX,&n);
DftiGetValueDM(desc,CDFT_LOCAL_X_START,&s);
// Virtual global array globalIN is defined by function f as
```

```
// globalIN[i*ny+j]=f(i,j)
for(i=0;i<n;i++)
   for(j=0;j<ny;j++) in[i*ny+j]=f(i+s,j);
// Set that we want out-of-place transform (default is DFTI_INPLACE)
DftiSetValueDM(desc,DFTI_PLACEMENT,DFTI_NOT_INPLACE);
// Commit descriptor, calculate FFT, free descriptor
DftiCommitDescriptorDM(desc);
DftiComputeForwardDM(desc,in,out);
// Virtual global array globalOUT is defined by function g as

// globalOUT[i*ny+j]=g(i,j)
// Now out contains result of FFT. out[i*ny+j]=g(i+s,j)
DftiFreeDescriptorDM(&desc);
free(in);
free(out);
MPI_Finalize();
```

The C example below illustrates one-dimensional in-place cluster DFT computations effected with a user-defined workspace:

Example 30 1D In-place Cluster DFT Computations

```
DFTI_DESCRIPTOR_DM_HANDLE desc;

long len,v,i,n_out,s_out;

Complex *in,*work;

MPI_Init(...);

// Create descriptor for 1D FFT

DftiCreateDescriptorDM(MPI_COMM_WORLD,&desc,DFTI_DOUBLE,DFTI_COMPLEX,1,l
en);

// Ask necessary length of array and workspace and allocate memory

DftiGetValueDM(desc,CDFT_LOCAL_SIZE,&v);

in=(Complex*)malloc(v*sizeof(Complex));

work=(Complex*)malloc(v*sizeof(Complex));

// Fill local array with initial data. Local array has n elements,

// 0 element of in corresponds to s element of virtual global array

DftiGetValueDM(desc,CDFT_LOCAL_NX,&n);

DftiGetValueDM(desc,CDFT_LOCAL_X_START,&s);

// Set work array as a workspace

DftiSetValueDM(desc,CDFT_WORKSPACE,work);

// Virtual global array globalIN is defined by function f as globalIN[i]=f(i)

for(i=0;i<n;i++) in[i]=f(i+s);

// Commit descriptor, calculate FFT, free descriptor

DftiCommitDescriptorDM(desc);

DftiComputeForwardDM(desc,in);

DftiGetValueDM(desc,CDFT_LOCAL_OUT_NX,&n_out);

DftiGetValueDM(desc,CDFT_LOCAL_OUT_X_START,&s_out);

// Virtual global array globalOUT is defined by function g as
globalOUT[i]=g(i)

// Now in contains result of FFT. Local array has n_out elements,
```

```
// 0 element of in corresponds to s_out element of virtual global array.

// in[i]==g(i+s_out)
DftiFreeDescriptorDM(&desc);
free(in);
free(work);
MPI_Finalize();
```

Interval Linear Solvers Code Examples

This section presents code examples of using the routines described in "Interval Linear Solvers". These routines are intended for computing enclosures and estimates of the solution sets to interval linear systems of equations as well as for checking properties of interval matrices and their inversion.

Example C-27. Interval Gauss-Seidel method

Given an interval system of linear algebraic equations, interval Gauss-Seidel method (implemented as `?gegss` routine) is often applied for enclosing a desired portion of the solution set that is bounded by a prescribed interval box.

Consider the following interval linear system of equations

$$\begin{pmatrix} [2,3] & [0,1] \\ [1,2] & [2,3] \end{pmatrix} x = \begin{pmatrix} [0,120] \\ [60,240] \end{pmatrix}$$

proposed first by E. Hansen (see Hansen92). Does its solution set intersect the interval box

$$\begin{pmatrix} [0,200] \\ [0,200] \end{pmatrix} ?$$

The following sample program answers the above question.

```
PROGRAM DIGEGSS_EXAMPLE
!
! Example program enclosing the solution set to a square interval
! linear system by interval Gauss-Seidel iterative method
!
!------------------------------------------------------------------------!
USE INTERVAL_ARITHMETIC
IMPLICIT NONE
!------------------------------------------------------------------------!
INTEGER, PARAMETER :: DIM = 2
INTEGER :: NRHS, LDA, LDB, NITS,
INFO, I, J
REAL(8) :: EPSILON
TYPE(D_INTERVAL) :: A(DIM,DIM), B(DIM,1),
ENCL(DIM,1)
CHARACTER(1) :: TRANS
!------------------------------------------------------------------------!
PRINT 300
!------------------------------------------------------------------------!
! !
! Initializing the input data - !
! !
TRANS = 'N'
NRHS = 2
A(1,1) = DINTERVAL(2.,3.); A(1,2) = DINTERVAL(0.,1.);
A(2,1) = DINTERVAL(0.,1.); A(2,2) = DINTERVAL(2.,3.);
LDA = 2
B(1,1) = DINTERVAL(0.,120.); B(2,1) = DINTERVAL(60.,240.);
```

```
LDB = 2
EPSILON = 1.D-6
NITS = 20
!------------------------------------------------------------------------!
!
! Assigning the bounding box for the solution set -
DO I = 1, DIM
ENCL(I,1) = DINTERVAL(0.,200.)
END DO
!------------------------------------------------------------------------!
CALL DIGEGSS(TRANS,
DIM, NRHS, A, LDA, B, LDB, ENCL, EPSILON, NITS, INFO )
!------------------------------------------------------------------------!
!
! Outputting the solution
IF( INFO /= 0 ) THEN
PRINT 400
ELSE
PRINT 600
DO I = 1, DIM
PRINT *, '[', B(I,1), ']'
END DO
END IF
!------------------------------------------------------------------------!
300 FORMAT (/,' **** SOLVING INTERVAL LINEAR SYSTEM **** ',/, &
' by interval Gauss-Seidel method ')
400 FORMAT (/,' The interval Gauss-Seidel method fails. ')
600 FORMAT (/,' Outer interval estimate of the solution set:',/)
```

```
!------------------------------------------------------------------------!
END PROGRAM DIGEGSS_EXAMPLE
```

Assigning double-precision intervals to the entries of the matrix `A` and right-hand side vector `B` is carried out by `DINTERVAL` function that turns two real numbers into the interval having these reals as endpoints. Running the above code produces the answer

```
**** SOLVING INTERVAL LINEAR SYSTEM****
by interval Gauss-Seidel method
Outer interval estimate of the solution set:
[ 0.000000000000000E+000 60.0000000000000 ]
[ 0.000000000000000E+000 120.000000000000 ]
```

One can make sure that the resulting box really encloses the required portion of the solution set after having a look at the corresponding graph from the paper Hansen92. Moreover, it is even the tightest possible enclosure.

Example C-28. Hansen-Bliek-Rohn procedure

The following Fortran-90 program illustrates the use of `digehbs` routine implementing "semiinterval" Hansen-Bliek-Rohn procedure for outer interval estimation of the solution sets to interval linear systems.

```
PROGRAM DIGEHBS_EXAMPLE
!
! Example program for enclosing the solution set to square interval
! interval system of equations by Hansen-Bliek-Rohn procedure
!
!------------------------------------------------------------------------!
USE INTERVAL_ARITHMETIC
IMPLICIT NONE
!------------------------------------------------------------------------!
INTEGER, PARAMETER :: DIM = 2
INTEGER :: LDA, LDB, INFO, I, J
TYPE(D_INTERVAL), ALLOCATABLE ::
A(:,:), B(:)
CHARACTER(1) :: TRANS
!------------------------------------------------------------------------!
PRINT 300
!------------------------------------------------------------------------!
!
! Initializing the input data -
!
TRANS = 'N'
ALLOCATE( A(DIM,DIM), B(DIM) )
A(1,1) = DINTERVAL(2.,4.); A(1,2) = DINTERVAL(-2.,1.)
A(2,1) = DINTERVAL(-1.,2.); A(2,2) = DINTERVAL(2.,2.)
LDA = 2
```

```
B(1) = DINTERVAL(0.,2.); B(2) = DINTERVAL(0.,2.)
LDB = 2
!------------------------------------------------------------------------!
CALL DIGEHBS( TRANS, DIM, A, LDA,
B, LDB, INFO )
!------------------------------------------------------------------------!
IF( INFO /= 0 ) THEN
PRINT 400
ELSE
PRINT 600
DO I = 1, DIM
PRINT *, I, ') [', B(I), ']'
END DO
END IF
!------------------------------------------------------------------------!
DEALLOCATE( A, B )
!------------------------------------------------------------------------!
300 FORMAT (/,' **** SOLVING INTERVAL LINEAR SYSTEM ****',/, &
' by Hansen-Bliek-Rohn procedure ',/)
400 FORMAT (/,' The matrix of the system is not an H-matrix,',/, &
' Hansen-Bliek-Rohn procedure fails. ',/)
600 FORMAT (/,' Enclosure of the solution set: ',/)
!------------------------------------------------------------------------!
END PROGRAM DIGEHBS_EXAMPLE
```

However, the output of the program looks like

```
**** SOLVING INTERVAL LINEAR SYSTEM****
by Hansen-Bliek-Rohn procedure
The matrix of the system is not an H-matrix,
Hansen-Bliek-Rohn procedure fails.
```

This result is because the program is applied to the interval linear system

$$\begin{pmatrix} [2, 4] & [-2, 1] \\ [-1, 2] & [2, 4] \end{pmatrix} x = \begin{pmatrix} [0, 2] \\ [0, 2] \end{pmatrix}$$

where the interval matrix is not an H-matrix (that is, it does not have diagonal dominance).

However, preconditioning by `digemip` routine helps to resolve the problem. The next modified program, which incorporates preliminary preconditioning of the interval linear system under solution, makes the matrix diagonally dominant and produces an acceptable answer to the problem.

```
PROGRAM DIGEMIP_DIGEHBS_EXAMPLE
!
! Example program for enclosing the solution set to square interval
! interval system of equations by Hansen-Bliek-Rohn procedure
!
!------------------------------------------------------------------------!
USE INTERVAL_ARITHMETIC
IMPLICIT NONE
!------------------------------------------------------------------------!
INTEGER, PARAMETER :: DIM = 2, NRHS = 1
INTEGER :: LDA, LDB, INFO, I, J
TYPE(D_INTERVAL), ALLOCATABLE ::
A(:,:), B(:)
CHARACTER(1) :: TRANS
!------------------------------------------------------------------------!
PRINT 300
!------------------------------------------------------------------------!
!
! Initializing the input data -
!
TRANS = 'N'
ALLOCATE( A(DIM,DIM), B(DIM) )
A(1,1) = DINTERVAL(2.,4.); A(1,2) = DINTERVAL(-2.,1.)
A(2,1) = DINTERVAL(-1.,2.); A(2,2) = DINTERVAL(2.,2.)
LDA = 2
```

```
B(1) = DINTERVAL(0.,2.); B(2) = DINTERVAL(0.,2.)
LDB = 2
!------------------------------------------------------------------------!
CALL DIGEMIP( DIM, NRHS, A, LDA,
B, LDB, INFO )
CALL DIGEHBS( TRANS, DIM, A, LDA,
B, LDB, INFO )
!------------------------------------------------------------------------!
IF( INFO /= 0 ) THEN
PRINT 400
ELSE
PRINT 600
DO I = 1, DIM
PRINT *, I, ') [', B(I), ']'
END DO
END IF
DEALLOCATE( A, B )
!------------------------------------------------------------------------!
300 FORMAT (/,' **** SOLVING INTERVAL LINEAR SYSTEM ****',/, &
' by Hansen-Bliek-Rohn procedure ',/)
400 FORMAT (/,' The matrix of the system is not an H-matrix,',/, &
' Hansen-Bliek-Rohn procedure fails. ',/)
600 FORMAT (/,' Enclosure of the solution set: ',/)
!------------------------------------------------------------------------!
END PROGRAM DIGEMIP_DIGEHBS_EXAMPLE
```

This time, the output of the program is

```
**** SOLVING INTERVAL LINEAR SYSTEM****
by Hansen-Bliek-Rohn procedure
Enclosure of the solution set:
1 ) [ -4.23529411764708 10.7058823529412 ]
2 ) [ -6.70588235294119 10.8235294117647 ]
```

(the last digits may change for various computer architectures).

Example C-29. Computing enclosure for inverse interval matrix

Given an interval 2 × 2 matrix

$$\begin{pmatrix} 3 & [0, 1] \\ [1, 2] & [2, 3] \end{pmatrix}$$

the following Fortran-90 code computes an enclosure of its inverse interval matrix:

```
PROGRAM SIGESZI_EXAMPLE
!
!Example program inverting an interval matrix by Sczulz iterative procedure
!
!------------------------------------------------------------------------!
USE INTERVAL_ARITHMETIC
IMPLICIT NONE
!------------------------------------------------------------------------!
INTEGER, PARAMETER :: DIM = 2, LDA = 2
INTEGER :: INFO, I, J
TYPE(S_INTERVAL), ALLOCATABLE ::
A(:,:)
!------------------------------------------------------------------------!
PRINT 300
!------------------------------------------------------------------------!
!
! Initislizing the input data -
!
ALLOCATE( A(LDA,DIM) )
A(1,1) = SINTERVAL(3.,3.); A(1,2) = SINTERVAL(0.,1.)
A(2,1) = SINTERVAL(1.,2.); A(2,2) = SINTERVAL(2.,3.)
!------------------------------------------------------------------------!
CALL SIGESZI ( DIM, A, LDA, INFO )
!------------------------------------------------------------------------!
PRINT 600
DO I = 1, DIM
PRINT *, ('[', A(I,J), ']', J = 1,
DIM )
```

```
END DO
DEALLOCATE( A )
!------------------------------------------------------------------------!
300 FORMAT (/,' **** INVERTING INTERVAL MATRIX ****',/, &
' by interval Schulz method ' )
400 FORMAT (/,' Schulz inversion procedure failed. ',/)
600 FORMAT (/,' Enclosure of the inverse matrix ',/)
!------------------------------------------------------------------------!
END PROGRAM SIGESZI_EXAMPLE
```

The output listing (with small variations depending on the architecture) looks as follows:

```
**** INVERTING INTERVAL MATRIX ****
by interval Schulz method
Enclosure of the inverse matrix
[ 0.2407409 0.5000001 ][ -0.2500000 0.1018518 ]
[ -0.5000000 5.5555239E-02 ][ 0.1388889 0.7500001 ]
```

At the same time, if we widen the (1,1) entry of the matrix to the interval [2, 3], the `sigeszi` procedure fails to compute a finite enclosure of the inverse to the new interval matrix

$$\begin{pmatrix} [2,3] & [0,1] \\ [1,2] & [2,3] \end{pmatrix}$$

Nevertheless, the interval linear system with such matrix can be successfully solved by specialized routines, for example, by interval Gauss method or interval Gauss-Seidel method (see Example C-27).

PDE Support Code Examples

This section presents code examples for routines described in the in the "Partial Differential Equations Support" chapter. The examples are grouped in subsections

- Trigonometric Transform Code Examples
- Poisson Library Code Examples.

Trigonometric Transforms Interface Code Examples

Code presented in this section computes solutions of three simple 1D Helmholtz problems with different boundary conditions: DD, NN and ND cases, where "D" denotes a Dirichlet boundary condition and "N" stands for a Neumann boundary condition. Example C-34 implements the computations in C and Example C-35 provides Fortran-90 code.

The algorithm of computing the solution uses Trigonometric Transform routines, described in chapter 13. In the DD case, the sine transform is computed, the NN case uses the cosine transform and the ND case corresponds to the staggered cosine transform.

Other details of the Helmholtz problems being solved are printed out along with the computed solutions.

Upon successful execution of Example C-34 the following text is printed out (Example C-35 generates similar output):

```
Example of use of MKL Trigonometric Transforms
 ********************************************

 This example gives the the solutions of the 1D differential problems
 with the equation -u"+u=f(x), 0<x<1,
```

```
 and with 3 types of boundary conditions:
 DD case: u(0)=u(1)=0,
 NN case: u'(0)=u'(1)=0,
 ND case: u'(0)=u(1)=0.
 ------------------------------------------------------------------------
 In general, the error should be of order O(1.0/n**2)
 For this example, the value of n is 8
 The approximation error should be of order 5.0e-002 if everything is OK
 ------------------------------------------------------------------------
 Note that n should be even to use Trigonometric Transforms !
 ------------------------------------------------------------------------
                     DOUBLE PRECISION COMPUTATIONS

 ========================================================================

The computed solution of DD problem is

u[0]= 0.000
u[1]= 0.153
u[2]= 0.524
u[3]= 0.895
u[4]= 1.049
u[5]= 0.895
u[6]= 0.524
u[7]= 0.153
u[8]= 0.000

Error=4.873e-002

The computed solution of NN problem is
```

```
u[0]=-0.026
u[1]= 0.128
u[2]= 0.500
u[3]= 0.872
u[4]= 1.026
u[5]= 0.872
u[6]= 0.500
u[7]= 0.128
u[8]=-0.026

Error=2.583e-002

The computed solution of ND problem is

u[0]=-0.009
u[1]= 0.145
u[2]= 0.517
u[3]= 0.890
u[4]= 1.045
u[5]= 0.892
u[6]= 0.522
u[7]= 0.152
u[8]= 0.000

Error=4.470e-002
```

C code for the computations is given below:

Example C-34 C Example to Solve a Set of 1D Helmholtz Problems

```
!******************************************************************************

!                                INTEL CONFIDENTIAL

!   Copyright(C) 2005 Intel Corporation. All Rights Reserved.

!   The source code contained  or  described herein and all documents related
 to

!   the source code ("Material") are owned by Intel Corporation or its
suppliers

!   or licensors.  Title to the  Material remains with  Intel Corporation
or its

!   suppliers and licensors. The Material contains trade secrets and
proprietary

!   and  confidential  information of  Intel or its suppliers and licensors.
 The

!   Material  is  protected  by  worldwide  copyright  and trade secret laws
 and

!   treaty  provisions. No part of the Material may be used, copied,
reproduced,

!   modified, published, uploaded, posted, transmitted, distributed or
disclosed

!   in any way without Intel's prior express written permission.

!   No license  under any  patent, copyright, trade secret or other
intellectual

!   property right is granted to or conferred upon you by disclosure or
delivery

!   of the Materials,  either expressly, by implication, inducement, estoppel
 or

!   otherwise.  Any  license  under  such  intellectual property  rights
must be

!   express and approved by Intel in writing.

!

!******************************************************************************

!  Content:

!  Double precision C test example for trigonometric transforms
```

```
!*****************************************************************************
!
! This example gives the solution of the 1D differential problems
! with the equation  -u"+u=f(x), 0<x<1, and with 3 types of boundary
conditions:

! u(0)=u(1)=0 (DD case), or u'(0)=u'(1)=0 (NN case), or u'(0)=u(1)=0 (ND
case)
*/  #include <stdio.h>
#include <malloc.h>
#include <math.h>
#include "mkl_dfti.h"
#include "mkl_trig_transforms.h"
int main(void)
{
   int n=8, i, k, tt_type;
   int ir, ipar[128];
   /* Note that the size of the transform n must be even !!! */
   double pi=3.14159265358979324, xi, c;
   double c1, c2, c3, c4, c5, c6;
   double *u, *f, *dpar, *lambda;
  DFTI_DESCRIPTOR_HANDLE handle = 0;  /* Printing the header for the example
 */
   printf("\n Example of use of MKL Trigonometric Transforms\n");
   printf(" ********************************************\n\n");
   printf(" This example gives the the solutions of the 1D differential
problems\n");
   printf(" with the equation -u\"+u=f(x), 0<x<1, \n");
   printf(" and with 3 types of boundary conditions:\n");
   printf(" DD case: u(0)=u(1)=0,\n");
```

```
   printf(" NN case: u'(0)=u'(1)=0,\n");
   printf(" ND case: u'(0)=u(1)=0.\n");
printf("
------------------------------------------------------------------------\n");
   printf(" In general, the error should be of order O(1.0/n**2)\n");
   printf(" For this example, the value of n is %1i\n", n);
  printf(" The approximation error should be of order 5.0e-002 if everything
 is OK\n");
   printf("
------------------------------------------------------------------------\n");
  printf(" Note that n should be even to use Trigonometric Transforms !\n");
   printf("
------------------------------------------------------------------------\n");
   printf("                        DOUBLE PRECISION COMPUTATIONS
        \n");

printf("========================================================================\n\n");
   u=(double*)malloc((n+1)*sizeof(double));
   f=(double*)malloc((n+1)*sizeof(double));
   dpar=(double*)malloc((3*n/2+1)*sizeof(double));
   lambda=(double*)malloc((n+1)*sizeof(double));
   for(i=0;i<=2;i++)
   {
      /* Varying the type of the transform */
      tt_type=i;
      /* Computing test solutions u(x) */
      for(k=0;k<=n;k++)
      {
         xi=1.0E0*k/n;
         u[k]=pow(sin(pi*xi),2.0E0);
      }
```

```
/* Computing the right-hand side f(x) */
for(k=0;k<=n;k++)
{
   f[k]=(4.0E0*(pi*pi)+1.0E0)*u[k]-2.0E0*(pi*pi);
}
/* Computing the right-hand side for the algebraic system */
for(k=0;k<=n;k++)
{
   f[k]=f[k]/(n*n);
}
```

```
      if (tt_type==0)
      {
         /* The Dirichlet boundary conditions */
         f[0]=0.0E0;
         f[n]=0.0E0;
      }
      if (tt_type==2)
      {
         /* The mixed Neumann-Dirichlet boundary conditions */
         f[n]=0.0E0;
      }
      /* Computing the eigenvalues for the three-point finite-difference
problem */
      if (tt_type==0||tt_type==1)
      {
         for(k=0;k<=n;k++)
         {
            lambda[k]=pow(2.0E0*sin(0.5E0*pi*k/n),2.0E0)+1.0E0/(n*n);
         }

      }

      if (tt_type==2)
      {
         for(k=0;k<=n;k++)
         {</
           lambda[k]=pow(2.0E0*sin(0.25E0*pi*(2*k+1)/n),2.0E0)+1.0E0/(n*n);
         }
      }
```

```
      /* Computing the solution of 1D problem using trigonometric transforms
       First we initialize the transform */
       d_init_trig_transform(&n,&tt_type,ipar,dpar,&ir);
       if (ir!=0) goto FAILURE;
      /* Then we commit the transform. Note that the data in f will be changed
 at this stage !
       If you want to keep them, save them in some other array before the
call to the routine */
       d_commit_trig_transform(f,&handle,ipar,dpar,&ir);
       if (ir!=0) goto FAILURE;
       /* Now we can apply trigonometric transform */
       d_forward_trig_transform(f,&handle,ipar,dpar,&ir);

       if (ir!=0) goto FAILURE;
```

```
      /* Scaling the solution by the eigenvalues */
      for(k=0;k<=n;k++)
      {
         f[k]=f[k]/lambda[k];
      }
      /* Now we can apply trigonometric transform once again as ONLY
input vector f has changed */
      d_backward_trig_transform(f,&handle,ipar,dpar,&ir);
      if (ir!=0) goto FAILURE;
      /* Cleaning the memory used by handle
      Now we can use handle for other kind of trigonometric transform */
      free_trig_transform(&handle,ipar,&ir);
      if (ir!=0) goto FAILURE;
      /* Performing the error analysis */
      c1=0.0E0;
      c2=0.0E0;
      c3=0.0E0;
      for(k=0;k<=n;k++)
      {
         /* Computing the absolute value of the exact solution */
         c4=fabs(u[k]);
         /* Computing the absolute value of the computed solution
         Note that the solution is now in place of the former right-hand
side ! */
         c5=fabs(f[k]);
         /* Computing the absolute error */
         c6=fabs(f[k]-u[k]);
         /* Computing the maximum among the above 3 values c4-c6 */
         if (c4>c1) c1=c4;
```

```
   if (c5>c2) c2=c5;
   if (c6>c3) c3=c6;
}
/* Printing the results */
if (tt_type==0)
{
   printf("The computed solution of DD problem is\n\n");
   for(k=0;k<=n;k++)
   {
      printf("u[%1i]=%6.3f\n",k,f[k]);
   }
   printf("\nError=%6.3e\n\n",c3/c1);
}
if (tt_type==1)
{
   printf("The computed solution of NN problem is\n\n");
   for(k=0;k<=n;k++)
   {
      printf("u[%1i]=%6.3f\n",k,f[k]);
   }
   printf("\nError=%6.3e\n\n",c3/c1);
}

if (tt_type==2)
```

```
      {
         printf("The computed solution of ND problem is\n\n");
         for(k=0;k<=n;k++)
         {
            printf("u[%1i]=%6.3f\n",k,f[k]);
         }
         printf("\nError=%6.3e\n\n",c3/c1);
      }

      /* End of the loop over the different kind of transforms and problems
 */
   }
   /* Jumping over failure message */
   goto SUCCESS;

   /* Failure message to print if something went wrong */
FAILURE: printf("Failed to compute the solution(s)...");
SUCCESS: return 0;
   /* End of the example code */
}
```

Fortran code for the computations is given below:

Example C-35 Fortran Example to Solve a Set of 1D Helmholtz Problems

```
!*******************************************************************************
!                                INTEL CONFIDENTIAL
!   Copyright(C) 2005 Intel Corporation. All Rights Reserved.
!   The source code contained  or  described herein and all documents related
 to
!   the source code ("Material") are owned by Intel Corporation or its
suppliers
!   or licensors.  Title to the  Material remains with  Intel Corporation
or its
!   suppliers and licensors. The Material contains trade secrets and
proprietary
!   and  confidential  information of  Intel or its suppliers and licensors.
 The
!   Material  is  protected  by  worldwide  copyright  and trade secret laws
 and
!   treaty  provisions. No part of the Material may be used, copied,
reproduced,
!   modified, published, uploaded, posted, transmitted, distributed or
disclosed
!   in any way without Intel's prior express written permission.
!   No license  under any  patent, copyright, trade secret or other
intellectual
!   property right is granted to or conferred upon you by disclosure or
delivery
!   of the Materials,  either expressly, by implication, inducement, estoppel
 or
!   otherwise.  Any  license  under  such  intellectual property  rights
must be
!   express and approved by Intel in writing.
!
!
!*******************************************************************************
!  Content:
```

```
!  Double precision Fortran90 test example for trigonometric transforms
!****************************************************************************
! This example gives the solution of the 1D differential problems
! with the equation  -u"+u=f(x), 0<x<1, and with 3 types of boundary
conditions:

! u(0)=u(1)=0 (DD case), or u'(0)=u'(1)=0 (NN case), or u'(0)=u(1)=0 (ND
case)

program d_tt_example_bvp
   use mkl_dfti
   use mkl_trig_transforms
   implicit none
```

```
      integer n, i, k,j, tt_type
      integer ir, ipar(128)
! Note that the size of the transform n must be even !!!
      parameter (n=8)
      double precision pi, xi
      double precision c1, c2, c3, c4, c5, c6
      double precision u(n+1), f(n+1), dpar(3*n/2+1), lambda(n+1)
      parameter (pi=3.14159265358979324D0)
      type(dfti_descriptor), pointer :: handle
! Printing the header for the example
      print *, ''
      print *, ' Example of use of MKL Trigonometric Transforms'
      print *, ' ***********************************************'
      print *, ''
      print *, ' This example gives the solution of the 1D differential
problems'
      print *, ' with the equation -u"+u=f(x), 0<x<1, '
      print *, ' and with 3 types of boundary conditions:'
      print *, ' DD case: u(0)=u(1)=0,'
      print *, ' NN case: u''(0)=u''(1)=0,'
      print *, ' ND case: u''(0)=u(1)=0.'
      print *, '
-------------------------------------------------------------------------'
      print *, ' In general, the error should be of order O(1.0/n**2)'
      print *, ' For this example, the value of n is', n
      print *, ' The approximation error should be of order 0.5E-01, if
everything is OK'
      print *, '
-------------------------------------------------------------------------'
      print *, ' Note that n should be even to use Trigonometric Transforms
 !'
```

```
      print *, '
-------------------------------------------------------------------------'
      print *, '                      DOUBLE PRECISION COMPUTATIONS
               '
print*,'=========================================================================
 '
      print *, ''

      do i=0,2
! Varying the type of the transform
         tt_type=i

! Computing test solution u(x)

         do k=1,n+1
            xi=1.0D0*(k-1)/n
            u(k)=dsin(pi*xi)**2
         end do
! Computing the right-hand side f(x)

         do k=1,n+1
              f(k)=(4.0D0*(pi**2)+1.0D0)*u(k)-2.0D0*(pi**2)
         end do
! Computing the right-hand side for the algebraic system

         do k=1,n+1
              f(k)=f(k)/(n**2)
         end do
```

```
            if (tt_type.eq.0) then
    ! The Dirichlet boundary conditions
               f(1)=0.0D0
               f(n+1)=0.0D0
            end if
            if (tt_type.eq.2) then
    ! The mixed Neumann-Dirichlet boundary conditions
               f(n+1)=0.0D0
            end if

    ! Computing the eigenvalues for the three-point finite-difference problem

            if (tt_type.eq.0.or.tt_type.eq.1) then
               do k=1,n+1
                  lambda(k)=(2.0D0*dsin(0.5D0*pi*(k-1)/n))**2+1.0D0/(n**2)
               end do

            end if

            if (tt_type.eq.2) then
               do k=1,n+1
                  lambda(k)=(2.0D0*dsin(0.25D0*pi*(2*k-1)/n))**2+1.0D0/(n**2)
               end do

            end if

    ! Computing the solution of 1D problem using trigonometric transforms
    ! First we initialize the transform
```

```
          CALL D_INIT_TRIG_TRANSFORM(n,tt_type,ipar,dpar,ir)
          if (ir.ne.0) goto 99
! Then we commit the transform. Note that the data in f will be changed at
this stage !
! If you want to keep them, save them in some other array before the call
to the routine
          CALL D_COMMIT_TRIG_TRANSFORM(f,handle,ipar,dpar,ir)
          if (ir.ne.0) goto 99
! Now we can apply trigonometric transform
          CALL D_FORWARD_TRIG_TRANSFORM(f,handle,ipar,dpar,ir)
          if (ir.ne.0) goto 99
! Scaling the solution by the eigenvalues

            do k=1,n+1

               f(k)=f(k)/lambda(k)
            end do

! Now we can apply trigonometric transform once again as ONLY input vector
f has changed
          CALL D_BACKWARD_TRIG_TRANSFORM(f,handle,ipar,dpar,ir)
          if (ir.ne.0) goto 99
! Cleaning the memory used by handle
! Now we can use handle for other KIND of trigonometric transform

          CALL FREE_TRIG_TRANSFORM(handle,ipar,ir)
          if (ir.ne.0) goto 99
```

```
! Performing the error analysis
         c1=0.0D0
         c2=0.0D0
         c3=0.0D0
         do k=1,n+1
! Computing the absolute value of the exact solution
            c4=dabs(u(k))
! Computing the absolute value of the computed solution
! Note that the solution is now in place of the former right-hand side !
            c5=dabs(f(k))
! Computing the absolute error
            c6=dabs(f(k)-u(k))
! Computing the maximum among the above 3 values c4-c6
            if (c4.gt.c1) c1=c4
            if (c5.gt.c2) c2=c5
            if (c6.gt.c3) c3=c6
         end do
! Printing the results
         if (tt_type.eq.0) then
            print *, 'The computed solution of DD problem is'
            print *, ''
            do k=1,n+1
               write(*,11) k,f(k)
            end do
            print *, ''
               write(*,12) c3/c1
            print *, ''
         end if
```

```
          if (tt_type.eq.1) then
             print *, 'The computed solution of NN problem is'
             print *, ''
             do k=1,n+1
                write(*,11) k,f(k)
             end do
             print *, ''
                write(*,12) c3/c1
             print *, ''
          end if

          if (tt_type.eq.2) then

             print *, 'The computed  solution of ND problem is'
             print *, ''
             do k=1,n+1
                write(*,11) k,f(k)
             end do
             print *, ''
                write(*,12) c3/c1
             print *, ''
          end if

! End of the loop over the different kind of transforms and problems
       end do
```

```
! Jumping over failure message
      go to 1
! Failure message to print if something went wrong
99    continue
      print *, 'Failed to compute the solution(s)...'
1     continue
! Print formats
11    format(1x,'u(',I1,')=',F6.3)
12    format(1x,'Relative error =',E10.3)
! End of the example code
end
```

Poisson Library Code Examples

Cartesian case

The code below computes an approximate solution of a 2D Poisson problem

$$-\frac{\partial^2 u}{\partial x^2}-\frac{\partial^2 u}{\partial y^2}=8\pi^2 \sin 2\pi x \cdot \sin 2\pi y$$

in the rectangle 0<x<1, 0 <y<1.

The following boundary conditions are imposed for the problem:

- Dirichlet boundary condition

  $$u(0,y)=u(1,y)=1$$

  for 0≤y≤1.
- Neumann boundary condition

$$\frac{\partial u}{\partial n}(x,0) = -2\pi \sin 2\pi x, \frac{\partial u}{\partial n}(x,1) = 2\pi \sin 2\pi x$$

for 0<x<1.

The exact solution is known to be

$$u(x,y) = \sin 2\pi x \cdot \sin 2\pi y + 1$$

Example C-36 implements the computations in C and Example C-37 provides Fortran-90 code. Mind that PL interface cannot be invoked from Fortran-77 due to restrictions imposed by the use of Intel MKL DFT interface.

The algorithm of computing the approximate solution of a Poisson Problem uses Poisson Library routines, described in chapter 13. Details of the Poisson problem being solved and the errors are printed out between the computed solution and the exact one.

Upon successful execution of Example C-36 the following text is printed out (Example C-37 produces similar output):

```
Example of use of MKL Poisson Library
 ********************************************
```

```
 This example gives the solution of 2D Poisson problem
 with the equation -u_xx-u_yy=f(x,y), 0<x<1, 0<y<1,
 f(x,y)=(8*pi*pi)*sin(2*pi*x)*sin(2*pi*y),
 and with the following boundary conditions:
  u(0,y)=u(1,y)=1 (Dirichlet boundary conditions),
 -u_y(x,0)=-2.0*pi*sin(2*pi*x) (Neumann boundary condition),
  u_y(x,1)= 2.0*pi*sin(2*pi*x) (Neumann boundary condition).
 ------------------------------------------------------------------------
 In general, the error should be of order O(1.0/nx^2+1.0/ny^2)
 For this example, the value of nx=ny is 6
 The approximation error should be of order 1.0e-01, if everything is OK
 ------------------------------------------------------------------------
 Note that nx should be even to use Poisson Library !
 ------------------------------------------------------------------------
                      DOUBLE PRECISION COMPUTATIONS

 ========================================================================

The number of mesh intervals in x-direction is nx=6
The number of mesh intervals in y-direction is ny=6
```

```
   In the mesh point (0.167,0.000) the error between the computed and the
true solution is equal to -7.505e-02

   In the mesh point (0.167,0.167) the error between the computed and the
true solution is equal to  4.432e-02

   In the mesh point (0.167,0.333) the error between the computed and the
true solution is equal to  6.309e-02

   In the mesh point (0.167,0.500) the error between the computed and the
true solution is equal to -5.551e-16

   In the mesh point (0.167,0.667) the error between the computed and the
true solution is equal to -6.309e-02

   In the mesh point (0.167,0.833) the error between the computed and the
true solution is equal to -4.432e-02

   In the mesh point (0.167,1.000) the error between the computed and the
true solution is equal to  7.505e-02


    Double precision 2D Poisson example has successfully PASSED

    through all steps of computation!
```

Note that actual figures in the error may slightly differ from the figures printed above depending on the architecture and operating system used to run the example.

C code for the problem is presented below:

Example C-36 C Example to Solve 2D Poisson Problem

```
/*******************************************************************************

/*                                INTEL CONFIDENTIAL

/*   Copyright(C) 2006 Intel Corporation. All Rights Reserved.

/*   The source code contained  or  described herein and all documents related
 to

/*   the source code ("Material") are owned by Intel Corporation or its
suppliers

/*   or licensors.  Title to the  Material remains with  Intel Corporation
or its

/*   suppliers and licensors. The Material contains trade secrets and
proprietary

/*   and  confidential  information of  Intel or its suppliers and licensors.
 The

/*   Material  is  protected  by  worldwide  copyright  and trade secret
laws and

/*   treaty  provisions. No part of the Material may be used, copied,
reproduced,

/*   modified, published, uploaded, posted, transmitted, distributed or
disclosed

/*   in any way without Intel");s prior express written permission.

/*   No license  under any  patent, copyright, trade secret or other
intellectual

/*   property right is granted to or conferred upon you by disclosure or
delivery

/*   of the Materials,  either expressly, by implication, inducement, estoppel
 or

/*   otherwise.  Any  license  under  such  intellectual property  rights
must be

/*   express and approved by Intel in writing.

/*

/*******************************************************************************

/*  Content:

/*  C double precision example of solving 2D Poisson problem in a
```

```
/*  rectangular domain using MKL Poisson Library
/*
/*****************************************************************************/
#include <stdio.h>
#include <malloc.h>
#include <math.h>
/* Include Poisson Library header files */
#include "mkl_dfti.h"
#include "mkl_poisson.h"
int main(void)
```

```
{
   /* Note that the size of the transform nx must be even !!! */
   int nx=6, ny=6;
   double pi=3.14159265358979324;
   int ix, iy, i, stat;
   int ipar[128];
   double ax, bx, ay, by, lx, ly, hx, hy, xi, yi, cx, cy;
   double *dpar, *f, *u, *bd_ax, *bd_bx, *bd_ay, *bd_by;
   double q;
   DFTI_DESCRIPTOR_HANDLE xhandle = 0;
   char *BCtype;
   /* Printing the header for the example */
   printf("\n Example of use of MKL Poisson Library\n");
   printf(" *******************************************\n\n");
   printf(" This example gives the solution of 2D Poisson problem\n");
   printf(" with the equation -u_xx-u_yy=f(x,y), 0<x<1, 0<y<1,\n");
   printf(" f(x,y)=(8*pi*pi)*sin(2*pi*x)*sin(2*pi*y),\n");
   printf(" and with the following boundary conditions:\n");
   printf("  u(0,y)=u(1,y)=1 (Dirichlet boundary conditions),\n");
  printf(" -u_y(x,0)=-2.0*pi*sin(2*pi*x) (Neumann boundary condition),\n");
  printf("  u_y(x,1)= 2.0*pi*sin(2*pi*x) (Neumann boundary condition).\n");
   printf("
-------------------------------------------------------------------------\n");
  printf(" In general, the error should be of order O(1.0/nx^2+1.0/ny^2)\n");
   printf(" For this example, the value of nx=ny is %d\n", nx);
  printf(" The approximation error should be of order 1.0e-01, if everything
 is OK\n");
   printf("
-------------------------------------------------------------------------\n");
   printf(" Note that nx should be even to use Poisson Library !\n");
```

```
   printf("
-------------------------------------------------------------------------\n");
   printf("                        DOUBLE PRECISION COMPUTATIONS
        \n");
   printf("
=========================================================================\n\n");
   dpar=(double*)malloc((5*nx/2+7)*sizeof(double));
   f=(double*)malloc((nx+1)*(ny+1)*sizeof(double));
   u=(double*)malloc((nx+1)*(ny+1)*sizeof(double));

   bd_ax=(double*)malloc((ny+1)*sizeof(double));
   bd_bx=(double*)malloc((ny+1)*sizeof(double));
   bd_ay=(double*)malloc((nx+1)*sizeof(double));
   bd_by=(double*)malloc((nx+1)*sizeof(double));
   /* Defining the rectangular domain 0<x<1, 0<y<1 for 2D Poisson Solver */
   ax=0.0E0;
   bx=1.0E0;
   ay=0.0E0;
   by=1.0E0;

/**************************************************************************
 *
   Setting the coefficient q to 0.

  Note that this is the way to use Helmholtz Solver to solve Poisson problem!

**************************************************************************
 /
```

```
   q=0.0E0;
   /* Computing the mesh size hx in x-direction */
   lx=bx-ax;
   hx=lx/nx;
   /* Computing the mesh size hy in y-direction */
   ly=by-ay;
   hy=ly/ny;
   /* Filling in the values of the TRUE solution
u(x,y)=sin(2*pi*x)*sin(2*pi*y)+1

   in the mesh points into the array u
  Filling in the right-hand side f(x,y)=(8*pi*pi+q)*sin(2*pi*x)*sin(2*pi*y)+q

   in the mesh points into the array f.

  We choose the right-hand side to correspond to the TRUE solution of Poisson
 equation.

   Here we are using the mesh sizes hx and hy computed before to compute
```

```
   the coordinates (xi,yi) of the mesh points */
   for(iy=0;iy<=ny;iy++)
   {
      for(ix=0;ix<=nx;ix++)
      {
         xi=hx*ix/lx;
         yi=hy*iy/ly;
         cx=sin(2*pi*xi);
         cy=sin(2*pi*yi);
         u[ix+iy*(nx+1)]=1.0E0*cx*cy;
         f[ix+iy*(nx+1)]=(8.0E0*pi*pi)*u[ix+iy*(nx+1)];
         u[ix+iy*(nx+1)]=u[ix+iy*(nx+1)]+1.0E0;
      }
   }
   /* Setting the type of the boundary conditions on each side of the
rectangular domain:
   On the boundary laying on the line x=0(=ax) Dirichlet boundary condition
 will be used
   On the boundary laying on the line x=1(=bx) Dirichlet boundary condition
 will be used
   On the boundary laying on the line y=0(=ay) Neumann boundary condition
will be used
   On the boundary laying on the line y=1(=by) Neumann boundary condition
will be used */
   BCtype = "DDNN";
   /* Setting the values of the boundary function G(x,y) that is equal to
the TRUE solution
```

```
   in the mesh points laying on Dirichlet boundaries */
   for(iy=0;iy<=ny;iy++)
   {
      bd_ax[iy]=1.0E0;

      bd_bx[iy]=1.0E0;
   }
   /* Setting the values of the boundary function g(x,y) that is equal to
the normal derivative

   of the TRUE solution in the mesh points laying on Neumann boundaries */
   for(ix=0;ix<=nx;ix++)
   {
      bd_ay[ix]=-2.0*pi*sin(2*pi*ix/nx);
      bd_by[ix]= 2.0*pi*sin(2*pi*ix/nx);
   }
   /* Initializing ipar array to make it free from garbage */
   for(i=0;i<128;i++)
   {
      ipar[i]=0;
   }

  /* Initializing simple data structures of Poisson Library for 2D Poisson
 Solver */
  d_init_Helmholtz_2D(&ax, &bx, &ay, &by, &nx, &ny, BCtype, &q, ipar, dpar,
&stat);
  if (stat!=0) goto FAILURE;
  /* Initializing complex data structures of Poisson Library for 2D Poisson
 Solver
  NOTE: Right-hand side f may be altered after the Commit step. If you want
 to keep it,
```

```
   you should save it in another memory location! */

  d_commit_Helmholtz_2D(f, bd_ax, bd_bx, bd_ay, bd_by, &xhandle, ipar, dpar,
 &stat);

   if (stat!=0) goto FAILURE;

   /* Computing the approximate solution of 2D Poisson problem

  NOTE: Boundary data stored in the arrays bd_ax, bd_bx, bd_ay, bd_by should
 not be changed


   between the Commit step and the subsequent call to the Solver routine/*


   Otherwise the results may be wrong. */

  d_Helmholtz_2D(f, bd_ax, bd_bx, bd_ay, bd_by, &xhandle, ipar, dpar, &stat);

   if (stat!=0) goto FAILURE;

   /* Cleaning the memory used by xhandle */

   free_Helmholtz_2D(&xhandle, ipar, &stat);

   if (stat!=0) goto FAILURE;

   /* Now we can use xhandle to solve another 2D Poisson problem*/

   /* Printing the results */

   printf("The number of mesh intervals in x-direction is nx=%d\n", nx);

   printf("The number of mesh intervals in y-direction is ny=%d\n\n",ny);

   /* Watching the error along the line x=hx */

   ix=1;

      for(iy=0;iy<=ny;iy++)

      {

         printf("In the mesh point (%5.3f,%5.3f) the error between the
computed and the true solution is equal to %10.3e\n", ix*hx, iy*hy,
f[ix+iy*(nx+1)]-u[ix+iy*(nx+1)]);


      }
```

```
      /* Success message to print if everything is OK */
      printf("\n Double precision 2D Poisson example has successfully
PASSED\n");
      printf(" through all steps of computation!\n");

      /* Jumping over failure message */
      goto SUCCESS;

      /* Failure message to print if something went wrong */
  FAILURE: printf("\nDouble precision 2D Poisson example FAILED to compute
 the solution...\n");

  SUCCESS: return 0;

      /* End of the example code */
  }
```

Fortran code for the computations is given below:

Example C-37 Fortran Example to Solve 2D Poisson Problem

```
!*****************************************************************************

!                                INTEL CONFIDENTIAL

!   Copyright(C) 2006 Intel Corporation. All Rights Reserved.

!   The source code contained  or  described herein and all documents related
 to

!   the source code ("Material") are owned by Intel Corporation or its
suppliers

!   or licensors.  Title to the  Material remains with  Intel Corporation
or its

!   suppliers and licensors. The Material contains trade secrets and
proprietary

!   and  confidential  information of  Intel or its suppliers and licensors.
 The

!   Material  is  protected  by  worldwide  copyright  and trade secret laws
 and

!   treaty  provisions. No part of the Material may be used, copied,
reproduced,

!   modified, published, uploaded, posted, transmitted, distributed or
disclosed

!   in any way without Intel's prior express written permission.

!   No license  under any  patent, copyright, trade secret or other
intellectual

!   property right is granted to or conferred upon you by disclosure or
delivery

!   of the Materials,  either expressly, by implication, inducement, estoppel
 or

!   otherwise.  Any  license  under  such  intellectual property  rights
must be

!   express and approved by Intel in writing.

!

!*****************************************************************************

!  Content:

!  Fortran-90 double precision example of solving 2D Poisson problem in a
```

```
!  rectangular domain using MKL Poisson Library
!
!*******************************************************************************
program Poisson_2D_double_precision
! Include modules defined by mkl_poisson.f90 and mkl_dfti.f90 header files
use mkl_poisson
use mkl_dfti
implicit none
integer nx,ny
! Note that the size of the transform nx must be even !!!
parameter(nx=6, ny=6)
double precision pi
parameter(pi=3.14159265358979324D0)
integer ix, iy, i, stat
integer ipar(128)
double precision ax, bx, ay, by, lx, ly, hx, hy, xi, yi, cx, cy
double precision dpar(5*nx/2+7)
! Note that proper packing of data in right-hand side array f is
```

```
! automatically provided by the following declaration of the arrays
double precision f(nx+1,ny+1), u(nx+1,ny+1)
double precision bd_ax(ny+1), bd_bx(ny+1), bd_ay(nx+1), bd_by(nx+1)
double precision q
type(DFTI_DESCRIPTOR), pointer :: xhandle
character(4) BCtype
! Printing the header for the example
      print *, ''
      print *, ' Example of use of MKL Poisson Library'
      print *, ' ******************************************'
      print *, ''
      print *, ' This example gives the solution of 2D Poisson problem'
      print *, ' with the equation -u_xx-u_yy=f(x,y), 0<x<1, 0<y<1,'
      print *, ' f(x,y)=(8*pi*pi)*sin(2*pi*x)*sin(2*pi*y),'
      print *, ' and with the following boundary conditions:'
      print *, '  u(0,y)=u(1,y)=1 (Dirichlet boundary conditions),'
     print *, ' -u_y(x,0)=-2.0*pi*sin(2*pi*x) (Neumann boundary condition),'
     print *, '  u_y(x,1)= 2.0*pi*sin(2*pi*x) (Neumann boundary condition).'
      print *, '
----------------------------------------------------------------------'
      print *, ' In general, the error should be of order
O(1.0/nx^2+1.0/ny^2)'
      print '(1x,a,I1)', ' For this example, the value of nx=ny is ', nx
      print *, ' The approximation error should be of order 0.1E+0, if
everything is OK'
      print *, '
----------------------------------------------------------------------'
      print *, ' Note that nx should be even to use Poisson Library !'
      print *, '
----------------------------------------------------------------------'
      print *, '                      DOUBLE PRECISION COMPUTATIONS
```

```
              '
      print *, '
==========================================================================='
      print *, ''

! Defining the rectangular domain 0<x<1, 0<y<1 for 2D Poisson Solver<
ax=0.0D0
bx=1.0D0
ay=0.0D0
by=1.0D0
!*****************************************************************************
! Setting the coefficient q to 0.

! Note that this is the way to use Helmholtz Solver to solve Poisson problem!
!*****************************************************************************

q=0.0D0
! Computing the mesh size hx in x-direction
lx=bx-ax
hx=lx/nx
! Computing the mesh size hy in y-direction
ly=by-ay

hy=ly/ny
! Filling in the values of the TRUE solution u(x,y)=sin(2*pi*x)*sin(2*pi*y)+1

! in the mesh points into the array u
! Filling in the right-hand side f(x,y)=(8*pi*pi+q)*sin(2*pi*x)*sin(2*pi*y)+q

! in the mesh points into the array f.
```

```
! We choose the right-hand side to correspond to the TRUE solution of Poisson
 equation.

! Here we are using the mesh sizes hx and hy computed before to compute

! the coordinates (xi,yi) of the mesh points
do iy=1,ny+1
   do ix=1,nx+1
      xi=hx*(ix-1)/lx
      yi=hy*(iy-1)/ly

      cx=dsin(2*pi*xi)
      cy=dsin(2*pi*yi)
      u(ix,iy)=1.0D0*cx*cy
      f(ix,iy)=(8.0D0*pi**2)*u(ix,iy)
      u(ix,iy)=u(ix,iy)+1.0D0
   enddo
enddo
! Setting the type of the boundary conditions on each side of the rectangular
 domain:
! On the boundary laying on the line x=0(=ax) Dirichlet boundary condition
will be used
! On the boundary laying on the line x=1(=bx) Dirichlet boundary condition
will be used
! On the boundary laying on the line y=0(=ay) Neumann boundary condition
will be used
! On the boundary laying on the line y=1(=by) Neumann boundary condition
will be used
BCtype = 'DDNN'
! Setting the values of the boundary function G(x,y) that is equal to the
TRUE solution
```

```
! in the mesh points laying on Dirichlet boundaries
do iy = 1,ny+1
   bd_ax(iy) = 1.0D0

   bd_bx(iy) = 1.0D0

enddo
! Setting the values of the boundary function g(x,y) that is equal to the
normal derivative

! of the TRUE solution in the mesh points laying on Neumann boundaries
do ix = 1,nx+1
   bd_ay(ix) = -2.0*pi*dsin(2*pi*(ix-1)/nx)
   bd_by(ix) =  2.0*pi*dsin(2*pi*(ix-1)/nx)
enddo
! Initializing ipar array to make it free from garbage
do i=1,128
   ipar(i)=0
enddo

! Initializing simple data structures of Poisson Library for 2D Poisson
Solver
   call d_init_Helmholtz_2D(ax, bx, ay, by, nx, ny, BCtype, q, ipar, dpar,
stat)
   if (stat.ne.0) goto 999
! Initializing complex data structures of Poisson Library for 2D Poisson
Solver
! NOTE: Right-hand side f may be altered after the Commit step. If you want
 to keep it,

! you should save it in another memory location!
   call d_commit_Helmholtz_2D(f, bd_ax, bd_bx, bd_ay, bd_by, xhandle, ipar,
 dpar, stat)
```

```
   if (stat.ne.0) goto 999

! Computing the approximate solution of 2D Poisson problem

! NOTE: Boundary data stored in the arrays bd_ax, bd_bx, bd_ay, bd_by should
 not be changed


! between the Commit step and the subsequent call to the Solver routine!


! Otherwise the results may be wrong.

   call d_Helmholtz_2D(f, bd_ax, bd_bx, bd_ay, bd_by, xhandle, ipar, dpar,
stat)

   if (stat.ne.0) goto 999

! Cleaning the memory used by xhandle

   call free_Helmholtz_2D(xhandle, ipar, stat)

   if (stat.ne.0) goto 999

! Now we can use xhandle to solve another 2D Poisson problem


! Printing the results

write(*,10) nx

   write(*,11) ny

   print *, ''

   ! Watching the error along the line x=hx

   ix=2

   do iy=1,ny+1

      write(*,12) (ix-1)*hx, (iy-1)*hy, f(ix,iy)-u(ix,iy)


   enddo

   print *, ''
```

```
   ! Success message to print if everything is OK
     print *, ' Double precision 2D Poisson example has successfully PASSED'
      print *, ' through all steps of computation!'
   ! Jumping over failure message
      go to 1
   ! Failure message to print if something went wrong
   999 print *, 'Double precision 2D Poisson example FAILED to compute the
solution...'

   1 continue

   10    format(1x,'The number of mesh intervals in x-direction is nx=',I1)
   11    format(1x,'The number of mesh intervals in y-direction is ny=',I1)
   12    format(1x,'In the mesh point (',F5.3,',',F5.3,') the error between
 the computed and the true solution is equal to ', E10.3)

   ! End of the example code
   end
```

Spherical case

The code below computes an approximate solution of a Helmholtz problem on the entire sphere:

$$-\Delta_s u + u = 3\cos\theta,\ \Delta_s = \frac{1}{\sin^2\theta}\frac{\partial^2}{\partial\varphi^2} + \frac{1}{\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial}{\partial\theta}\right)$$

with $a_\phi = 0$, $b_\phi = 2\pi$, $a_\theta = 0$, $b_\theta = \pi$, and a boundary condition

$$\left(\sin\theta\frac{\partial u}{\partial\theta}\right)_{\substack{\theta\to 0\\ \theta\to\pi}} = 0$$

at the poles (periodic case).

The exact solution is known to be $u(\phi,\theta) = \cos\theta$.

Example C-38 implements the computations in C and Example C-39 provides Fortran-90 code. Mind that PL interface cannot be invoked from Fortran-77 due to restrictions imposed by the use of Intel MKL DFT interface.

The algorithm of computing the approximate solution of a Poisson Problem uses Poisson Library routines, described in chapter 13. Details of the Poisson problem being solved and the errors are printed out between the computed solution and the exact one. Upon successful execution of Example C-38 the following text is printed out (Example C-39produces similar output):

```
 Example of use of MKL Poisson Library
 ******************************************
 This example gives the solution of Helmholtz problem on a whole sphere
 0<p<2*pi, 0<t<pi, with Helmholtz coefficient q=1 and right-hand side
 f(p,t)=3*cos(t)
 ------------------------------------------------------------------------
 In general, the error should be of order O(1.0/np^2+1.0/nt^2)
 For this example, the value of np=nt is 8
 The approximation error should be of order 1.5e-01, if everything is OK
 ------------------------------------------------------------------------
   Note that np should be divisible by 4 to solve the PERIODIC problem!
 ------------------------------------------------------------------------
                      DOUBLE PRECISION COMPUTATIONS
```

```
 ===========================================================================

NO3NO4The number of mesh intervals in phi-direction is np=8

The number of mesh intervals in theta-direction is nt=8

In the mesh point (0.785,0.000) the error between the computed and the true
 solution is equal to -1.402e-001

In the mesh point (0.785,0.393) the error between the computed and the true
 solution is equal to -3.097e-002

In the mesh point (0.785,0.785) the error between the computed and the true
 solution is equal to -6.036e-003

In the mesh point (0.785,1.178) the error between the computed and the true
 solution is equal to 2.964e-004

In the mesh point (0.785,1.571) the error between the computed and the true
 solution is equal to 4.979e-017

In the mesh point (0.785,1.963) the error between the computed and the true
 solution is equal to -2.964e-004

In the mesh point (0.785,2.356) the error between the computed and the true
 solution is equal to 6.036e-003

In the mesh point (0.785,2.749) the error between the computed and the true
 solution is equal to 3.097e-002

In the mesh point (0.785,3.142) the error between the computed and the true
 solution is equal to 1.402e-001

 Double precision Helmholtz example on a whole sphere has successfully PASSED

 through all steps of computation!
```

Note that actual figures in the error may slightly differ from the figures printed above dependingon the architecture and operating system used to run the example.

C code for the problem is presented below:

Example C-38 C Example to Solve Helmholtz Problem on a Sphere

```
/*******************************************************************************

/*                                INTEL CONFIDENTIAL

/*   Copyright(C) 2006 Intel Corporation. All Rights Reserved.

/*   The source code contained  or  described herein and all documents related
 to

/*   the source code ("Material") are owned by Intel Corporation or its
suppliers

/*   or licensors.  Title to the  Material remains with  Intel Corporation
or its

/*   suppliers and licensors. The Material contains trade secrets and
proprietary

/*   and  confidential  information of  Intel or its suppliers and licensors.
 The

/*   Material  is  protected  by  worldwide  copyright  and trade secret
laws and

/*   treaty  provisions. No part of the Material may be used, copied,
reproduced,

/*   modified, published, uploaded, posted, transmitted, distributed or
disclosed

/*   in any way without Intel's prior express written permission.

/*   No license  under any  patent, copyright, trade secret or other
intellectual

/*   property right is granted to or conferred upon you by disclosure or
delivery

/*   of the Materials,  either expressly, by implication, inducement, estoppel
 or

/*   otherwise.  Any  license  under  such  intellectual property  rights
must be

/*   express and approved by Intel in writing.

/*

/*******************************************************************************

/*  Content:

/*  C double precision example of solving Helmholtz problem on a whole sphere
```

```
/*  using MKL Poisson Library
/*
/*****************************************************************************/
#include <stdio.h>
#include <malloc.h>
#include <math.h>
/* Include Poisson Library header files */
#include "mkl_dfti.h"
#include "mkl_poisson.h"
int main(void)
{
 /* Note that the size of the transform np must be divisible by 4 !!! */
 int np=8, nt=8;
 double pi=3.14159265358979324;
 int ip, it, i, stat;
 int ipar[128];
 double ap, bp, at, bt, lp, lt, hp, ht, theta_i, ct;
 double *dpar, *f, *u;
 double q;
 DFTI_DESCRIPTOR_HANDLE handle_s = 0;
 DFTI_DESCRIPTOR_HANDLE handle_c = 0;
 /* Printing the header for the example */
 printf("\n Example of use of MKL Poisson Library\n");
 printf(" ******************************************\n\n");
 printf(" This example gives the solution of Helmholtz problem on a whole
sphere\n");
 printf(" 0<p<2*pi, 0<t<pi, with Helmholtz coefficient q=1 and right-hand
side\n");
 printf(" f(p,t)=3*cos(t)\n");
```

```
 printf("
-------------------------------------------------------------------------\n");

 printf(" In general, the error should be of order O(1.0/np^2+1.0/nt^2)\n");

 printf(" For this example, the value of np=nt is %d\n", np);

 printf(" The approximation error should be of order 1.5e-01, if everything
 is OK\n");

 printf("
-------------------------------------------------------------------------\n");

 printf("   Note that np should be divisible by 4 to solve the PERIODIC
problem!\n");

 printf("
-------------------------------------------------------------------------\n");

 printf("                        DOUBLE PRECISION COMPUTATIONS
      \n");

 printf("
=========================================================================\n\n");

 dpar=(double*)malloc((7*np/2+10)*sizeof(double));

 f=(double*)malloc((np+1)*(nt+1)*sizeof(double));

 u=(double*)malloc((np+1)*(nt+1)*sizeof(double));

 /* Defining the rectangular domain on a sphere 0<p<2*pi, 0<t<pi for Helmholtz
 Solver on a sphere */

 /* Poisson Library will automatically detect that this problem is on a whole
 sphere! */

 ap=0.0E0;

 bp=2*pi;

 at=0.0E0;

 bt=pi;

 /* Setting the coefficient q to 1.0E0 for Helmholtz problem */

 /* If you like to solve Poisson problem, please set q to 0.0E0 */

 q=1.0E0;

 /* Computing the mesh size hp in phi-direction */

 lp=bp-ap;
```

```
hp=lp/np;
/* Computing the mesh size ht in theta-direction */
lt=bt-at;
ht=lt/nt;
/* Filling in the values of the TRUE solution u(p,t)=cos(t)

in the mesh points into the array u
Filling in the right-hand side f(p,t)=(2+q)*cos(t)
in the mesh points into the array f.

We choose the right-hand side to correspond to the TRUE solution of Helmholtz
equation on a sphere.

Here we are using the mesh sizes hp and ht computed before to compute

the coordinates (phi_i,theta_i) of the mesh points */
for(it=0;it<=nt;it++)
{</codeblock><codeblock>  for(ip=0;ip<=np;ip++)
 {
  theta_i=ht*it;
  ct=cos(theta_i);
  u[ip+it*(np+1)]=ct;
  f[ip+it*(np+1)]=ct*(2.+q);
 }
}
/* Initializing ipar array to make it free from garbage */
for(i=0;i<128;i++)
{
 ipar[i]=0;
}
```

```
 /* Initializing simple data structures of Poisson Library for Helmholtz
Solver on a sphere */

 /* As we are looking for the solution on a whole sphere, this is a PERIDOC
 problem */

 /* Therefore, the routines ending with "_p" are used to find the solution
*/


   d_init_sph_p(&ap,&bp,&at,&bt,&np,&nt,&q,ipar,dpar,&stat);

 if (stat!=0) goto FAILURE;

 /* Initializing complex data structures of Poisson Library for Helmholtz
Solver on a sphere

 NOTE: Right-hand side f may be altered after the Commit step. If you want
to keep it,


 you should save it in another memory location! */

   d_commit_sph_p(f,&handle_s,&handle_c,ipar,dpar,&stat);

 if (stat!=0) goto FAILURE;

 /* Computing the approximate solution of Helmholtz problem on a whole sphere
 */

   d_sph_p(f,&handle_s,&handle_c,ipar,dpar,&stat);

 if (stat!=0) goto FAILURE;

 /* Cleaning the memory used by handle_s and handle_c */

 free_sph_p(&handle_s,&handle_c,ipar,&stat);

 if (stat!=0) goto FAILURE;

 /* Now we can use handle_s and handle_c to solve another Helmholtz problem
 */

 /* after a proper initialization */
```

```
 /* Printing the results */
 printf("The number of mesh intervals in phi-direction is np=%d\n", np);
 printf("The number of mesh intervals in theta-direction is nt=%d\n\n", nt);
 /* Watching the error along the line phi=hp */
 ip=1;
 for(it=0;it<=nt;it++)
 {
  printf("In the mesh point (%5.3f,%5.3f) the error between the computed and
 the true solution is equal to %10.3e\n", ip*hp, it*ht,
f[ip+it*(np+1)]-u[ip+it*(np+1)]);

 }
 /* Success message to print if everything is OK */
 printf("\n Double precision Helmholtz example on a whole sphere has
successfully PASSED\n");
 printf(" through all steps of computation!\n");
 /* Jumping over failure message */
 goto SUCCESS;
 /* Failure message to print if something went wrong */
FAILURE: printf("\nDouble precision Helmholtz example on a whole sphere has
 FAILED to compute the solution...\n");
 return -1;
SUCCESS: return 0;
 /* End of the example code */
}
```

Fortran-90 code for the problem is presented below:

Example C-39 Fortran Example to Solve Helmholtz Problem on a Sphere

```
!*******************************************************************************

!                                INTEL CONFIDENTIAL

!   Copyright(C) 2006 Intel Corporation. All Rights Reserved.

!   The source code contained  or  described herein and all documents related
 to

!   the source code ("Material") are owned by Intel Corporation or its
suppliers

!   or licensors.  Title to the  Material remains with  Intel Corporation
or its

!   suppliers and licensors. The Material contains trade secrets and
proprietary

!   and  confidential  information of  Intel or its suppliers and licensors.
 The

!   Material  is  protected  by  worldwide  copyright  and trade secret laws
 and

!   treaty  provisions. No part of the Material may be used, copied,
reproduced,

!   modified, published, uploaded, posted, transmitted, distributed or
disclosed

!   in any way without Intel's prior express written permission.

!   No license  under any  patent, copyright, trade secret or other
intellectual

!   property right is granted to or conferred upon you by disclosure or
delivery

!   of the Materials,  either expressly, by implication, inducement, estoppel
 or

!   otherwise.  Any  license  under  such  intellectual property  rights
must be

!   express and approved by Intel in writing.

!

!*******************************************************************************

!  Content:

!  Fortran-90 precision example of solving Helmholtz problem on a whole
```

```
sphere
!  using MKL Poisson Library
!
!*******************************************************************************
!

program d_sph_with_poles
! Include modules defined by mkl_poisson.f90 and mkl_dfti.f90 header files
use mkl_dfti
use mkl_poisson
implicit none

integer np,nt
parameter(np=8,nt=8)
double precision pi
parameter(pi=3.14159265358979324D0)
double precision ap,bp,hp,at,bt,ht,q,lp,lt,theta_i,ct
double precision  u(np+1,nt+1),f(np+1,nt+1)
type(DFTI_DESCRIPTOR), pointer :: handle_s, handle_c
integer stat
double precision dpar(7*np/2+10)
integer ip,it,i
integer ipar(128)
```

```
! Printing the header for the example

      print *, ''

      print *, ' Example of use of MKL Poisson Library'

      print *, ' *****************************************'

      print *, ''

      print *, ' This example gives the solution of Helmholtz problem on a
whole sphere'

      print *, ' 0<p<2*pi, 0<t<pi, with Helmholtz coefficient q=1 and
right-hand side'

    print *, ' f(p,t)=3*cos(t)'

      print *, '
-------------------------------------------------------------------------'

      print *, ' In general, the error should be of order
O(1.0/np^2+1.0/nt^2)'

      print '(1x,a,I1)', ' For this example, the value of np=nt is ', np

      print *, ' The approximation error should be of order 0.15E+0, if
everything is OK'

      print *, '
-------------------------------------------------------------------------'

     print *, ' Note that np should be divisible by 4 to solve the PERIODIC
 problem!'

      print *, '
-------------------------------------------------------------------------'

      print *, '                       DOUBLE PRECISION COMPUTATIONS
             '

      print *, '
=========================================================================='

      print *, ''
```

```
! Defining the rectangular domain on a sphere 0<p<2*pi, 0<t<pi for Helmholtz
 Solver on a sphere

! Poisson Library will automatically detect that this problem is on a whole
 sphere!

ap=0.0D0

bp=2*pi

at=0.0D0

bt=pi

! Setting the coefficient q to 1.0D0 for Helmholtz problem

! If you like to solve Poisson problem, please set q to 0.0D0

q=1.0D0

! Computing the mesh size hp in phi-direction

lp=bp-ap

hp=lp/np

! Computing the mesh size ht in theta-direction

lt=bt-at


ht=lt/nt

! Filling in the values of the TRUE solution u(p,t)=cos(t)


! in the mesh points into the array u

! Filling in the right-hand side f(p,t)=3*cos(t)


! in the mesh points into the array f.


! We choose the right-hand side to correspond to the TRUE solution of
Helmholtz equation.


! Here we are using the mesh sizes hp and ht computed before to compute
```

```
! the coordinates (phi_i,theta_i) of the mesh points
do it=1,nt+1
   do ip=1,np+1
      theta_i=ht*(it-1)

      ct=dcos(theta_i)
      u(ip,it)=ct
      f(ip,it)=(2.0D0+q)*ct
   enddo
enddo
! Initializing ipar array to make it free from garbage
do i=1,128
   ipar(i)=0
enddo

! Initializing simple data structures of Poisson Library for Helmholtz Solver
 on a sphere
! As we are looking for the solution on a whole sphere, this is a PERIDOC
problem

! Therefore, the routines ending with "_P" are used to find the solution

   call D_INIT_SPH_P(ap,bp,at,bt,np,nt,q,ipar,dpar,stat)
   if (stat.ne.0) goto 999
! Initializing complex data structures of Poisson Library for Helmholtz
Solver on a sphere
! NOTE: Right-hand side f may be altered after the Commit step. If you want
 to keep it,
```

```
! you should save it in another memory location!
   call D_COMMIT_SPH_P(f,handle_s,handle_c,ipar,dpar,stat)
   if (stat.ne.0) goto 999
! Computing the approximate solution of Helmholtz problem on a whole sphere
   call D_SPH_P(f,handle_s,handle_c,ipar,dpar,stat)
   if (stat.ne.0) goto 999
! Cleaning the memory used by handle_s and handle_c
   call FREE_SPH_P(HANDLE_S,HANDLE_C,IPAR,STAT)
   if (stat.ne.0) goto 999
! Now we can use handle_s and handle_c to solve another Helmholtz problem
after a proper initialization

! Printing the results
write(*,10) np
write(*,11) nt
print *, ''
! Watching the error along the line phi=hp
ip=1
do it=1,nt+1
   write(*,12) (ip-1)*hp, (it-1)*ht, f(ip,it)-u(ip,it)

enddo
print *, ''
```

```
! Success message to print if everything is OK

 print *, ' Double precision Helmholtz example on a whole sphere has
successfully PASSED'

 print *, ' through all steps of computation!'

! Jumping over failure message

   go to 1

! Failure message to print if something went wrong

999 print *, 'Double precision Helmholtz example on a whole sphere FAILED
to compute the solution...'

1 continue

10    format(1x,'The number of mesh intervals in x-direction is np=',I1)

11    format(1x,'The number of mesh intervals in y-direction is nt=',I1)

12    format(1x,'In the mesh point (',F5.3,',',F5.3,') the error between the
 computed and the true solution is equal to ', E10.3)

! End of the example code

end
```

CBLAS Interface to the BLAS

D

This appendix presents CBLAS, the C interface to the Basic Linear Algebra Subprograms (BLAS) implemented in Intel® MKL.

Similar to BLAS, the CBLAS interface includes the following levels of functions:

- "Level 1 CBLAS" (vector-vector operations)
- "Level 2 CBLAS" (matrix-vector operations)
- "Level 3 CBLAS" (matrix-matrix operations).
- "Sparse CBLAS" (operations on sparse vectors).

To obtain the C interface, the Fortran routine names are prefixed with `cblas_` (for example, `dasum` becomes `cblas_dasum`). Names of all CBLAS functions are in lowercase letters.

Complex functions `?dotc` and `?dotu` become CBLAS subroutines (void functions); they return the complex result via a void pointer, added as the last parameter. CBLAS names of these functions are suffixed with `_sub`. For example, the BLAS function `cdotc` corresponds to `cblas_cdotc_sub`.

In the descriptions of CBLAS interfaces, links provided for each function group lead to the descriptions of the respective Fortran-interface BLAS functions.

CBLAS Arguments

The arguments of CBLAS functions obey the following rules:

- Input arguments are declared with the `const` modifier.
- Non-complex scalar input arguments are passed by value.
- Complex scalar input arguments are passed as void pointers.
- Array arguments are passed by address.
- Output scalar arguments are passed by address.
- BLAS character arguments are replaced by the appropriate enumerated type.
- Level 2 and Level 3 routines acquire an additional parameter of type `CBLAS_ORDER` as their first argument. This parameter specifies whether two-dimensional arrays are row-major (`CblasRowMajor`) or column-major (`CblasColMajor`).

Enumerated Types

The CBLAS interface uses the following enumerated types:

```
enum CBLAS_ORDER {
   CblasRowMajor=101,  /* row-major arrays */
   CblasColMajor=102};   /* column-major arrays */
enum CBLAS_TRANSPOSE {
   CblasNoTrans=111,     /* trans='N' */
   CblasTrans=112,       /* trans='T' */
   CblasConjTrans=113};  /* trans='C' */
enum CBLAS_UPLO {
   CblasUpper=121,        /* uplo ='U' */
   CblasLower=122};       /* uplo ='L' */
enum CBLAS_DIAG {
   CblasNonUnit=131,      /* diag ='N' */
   CblasUnit=132};        /* diag ='U' */
enum CBLAS_SIDE {
   CblasLeft=141,         /* side ='L' */
   CblasRight=142};       /* side ='R' */
```

Level 1 CBLAS

This is an interface to "BLAS Level 1 Routines and Functions", which perform basic vector-vector operations.

?asum

```
float cblas_sasum(const int N, const float *X, const int incX);
double cblas_dasum(const int N, const double *X, const int incX);
float cblas_scasum(const int N, const void *X, const int incX);
double cblas_dzasum(const int N, const void *X, const int incX);
```

?axpy

```
void cblas_saxpy(const int N, const float alpha, const float *X, const int
incX, float *Y, const int incY);

void cblas_daxpy(const int N, const double alpha, const double *X, const int
 incX, double *Y, const int incY);

void cblas_caxpy(const int N, const void *alpha, const void *X, const int
incX, void *Y, const int incY);

void cblas_zaxpy(const int N, const void *alpha, const void *X, const int
incX, void *Y, const int incY);
```

?copy

```
void cblas_scopy(const int N, const float *X, const int incX, float *Y, const
 int incY);

void cblas_dcopy(const int N, const double *X, const int incX, double *Y,
const int incY);

void cblas_ccopy(const int N, const void *X, const int incX, void *Y, const
 int incY);

void cblas_zcopy(const int N, const void *X, const int incX, void *Y, const
 int incY);
```

?dot

```
float cblas_sdot(const int N, const float *X, const int incX, const float
*Y, const int incY);

double cblas_ddot(const int N, const double *X, const int incX, const double
 *Y, const int incY);
```

?sdot

```
float cblas_sdsdot(const int N, const float *SB, const float *SX, const int
 incX, const float *SY, const int incY);

double cblas_dsdot(const int N, const float *SX, const int incX, const float
 *SY, const int incY);
```

?dotc

```
void cblas_cdotc_sub(const int N, const void *X, const int incX, const void
 *Y, const int incY, void *dotc);

void cblas_zdotc_sub(const int N, const void *X, const int incX, const void
 *Y, const int incY, void *dotc);
```

?dotu

```
void cblas_cdotu_sub(const int N, const void *X, const int incX, const void
 *Y, const int incY, void *dotu);

void cblas_zdotu_sub(const int N, const void *X, const int incX, const void
 *Y, const int incY, void *dotu);
```

?nrm2

```
float cblas_snrm2(const int N, const float *X, const int incX);

double cblas_dnrm2(const int N, const double *X, const int incX);

float cblas_scnrm2(const int N, const void *X, const int incX);

double cblas_dznrm2(const int N, const void *X, const int incX);
```

?rot

```
void cblas_srot(const int N, float *X, const int incX, float *Y, const int
incY, const float c, const float s);

void cblas_drot(const int N, double *X, const int incX, double *Y,const int
 incY, const double c, const double s);
```

?rotg

```
void cblas_srotg(float *a, float *b, float *c, float *s);


void cblas_drotg(double *a, double *b, double *c, double *s);
```

?rotm

```
void cblas_srotm(const int N, float *X, const int incX, float *Y, const int
 incY, const float *P);

void cblas_drotm(const int N, double *X, const int incX, double *Y, const
int incY, const double *P);
```

?rotmg

```
void cblas_srotmg(float *d1, float *d2, float *b1, const float b2, float
*P);

void cblas_drotmg(double *d1, double *d2, double *b1, const double b2, double
 *P);
```

?scal

```
void cblas_sscal(const int N, const float alpha, float *X, const int incX);

void cblas_dscal(const int N, const double alpha, double *X, const int incX);

void cblas_cscal(const int N, const void *alpha, void *X, const int incX);

void cblas_zscal(const int N, const void *alpha, void *X, const int incX);

void cblas_csscal(const int N, const float alpha, void *X, const int incX);

void cblas_zdscal(const int N, const double alpha, void *X, const int incX);
```

?swap

```
void cblas_sswap(const int N, float *X, const int incX, float *Y, const int
 incY);

void cblas_dswap(const int N, double *X, const int incX, double *Y, const
int incY);
```

```
void cblas_cswap(const int N, void *X, const int incX, void *Y, const int
incY);

void cblas_zswap(const int N, void *X, const int incX, void *Y, const int
incY);
```

i?amax

```
CBLAS_INDEX cblas_isamax(const int N, const float *X, const int incX);

CBLAS_INDEX cblas_idamax(const int N, const double *X, const int incX);

CBLAS_INDEX cblas_icamax(const int N, const void *X, const int incX);

CBLAS_INDEX cblas_izamax(const int N, const void *X, const int incX);
```

i?amin

```
CBLAS_INDEX cblas_isamin(const int N, const float *X, const int incX);

CBLAS_INDEX cblas_idamin(const int N, const double *X, const int incX);

CBLAS_INDEX cblas_icamin(const int N, const void *X, const int incX);

CBLAS_INDEX cblas_izamin(const int N, const void *X, const int incX);
```

Level 2 CBLAS

This is an interface to “BLAS Level 2 Routines”, which perform basic matrix-vector operations. Each C routine in this group has an additional parameter of type `CBLAS_ORDER` (the first argument) that determines whether the two-dimensional arrays use column-major or row-major storage.

?gbmv

```
void cblas_sgbmv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE
TransA, const int M, const int N, const int KL, const int KU, const float
alpha, const float *A, const int lda, const float *X, const int incX, const
 float beta, float *Y, const int incY);


void cblas_dgbmv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE
TransA, const int M, const int N, const int KL, const int KU, const double
alpha, const double *A, const int lda, const double *X, const int incX, const
 double beta, double *Y, const int incY);


void cblas_cgbmv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE
TransA, const int M, const int N, const int KL, const int KU, const void
*alpha, const void *A, const int lda, const void *X, const int incX, const
void *beta, void *Y, const int incY);


void cblas_zgbmv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE
TransA, const int M, const int N, const int KL, const int KU, const void
*alpha, const void *A, const int lda, const void *X, const int incX, const
void *beta, void *Y, const int incY);
```

?gemv

```
void cblas_sgemv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE
TransA, const int M, const int N, const float alpha, const float *A, const
int lda, const float *X, const int incX, const float beta, float *Y, const
int incY);


void cblas_dgemv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE
TransA, const int M, const int N, const double alpha, const double *A, const
 int lda, const double *X, const int incX, const double beta, double *Y,
const int incY);


void cblas_cgemv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE
TransA, const int M, const int N, const void *alpha, const void *A, const
int lda, const void *X, const int incX, const void *beta, void *Y, const int
 incY);


void cblas_zgemv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE
TransA, const int M, const int N, const void *alpha, const void *A, const
int lda, const void *X, const int incX, const void *beta, void *Y, const int
 incY);
```

?ger

```
void cblas_sger(const enum CBLAS_ORDER order, const int M, const int N, const
 float alpha, const float *X, const int incX, const float *Y, const int incY,
 float *A, const int lda);


void cblas_dger(const enum CBLAS_ORDER order, const int M, const int N, const
 double alpha, const double *X, const int incX, const double *Y, const int
incY, double *A, const int lda);
```

?gerc

```
void cblas_cgerc(const enum CBLAS_ORDER order, const int M, const int N,
const void *alpha, const void *X, const int incX, const void *Y, const int
incY, void *A, const int lda);


void cblas_zgerc(const enum CBLAS_ORDER order, const int M, const int N,
const void *alpha, const void *X, const int incX, const void *Y, const int
incY, void *A, const int lda);
```

?geru

```
void cblas_cgeru(const enum CBLAS_ORDER order, const int M, const int N,
const void *alpha, const void *X, const int incX, const void *Y, const int
incY, void *A, const int lda);


void cblas_zgeru(const enum CBLAS_ORDER order, const int M, const int N,
const void *alpha, const void *X, const int incX, const void *Y, const int
incY, void *A, const int lda);
```

?hbmv

```
void cblas_chbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const int K, const void *alpha, const void *A, const int lda,
const void *X, const int incX, const void *beta, void *Y, const int incY);


void cblas_zhbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const int K, const void *alpha, const void *A, const int lda,
const void *X, const int incX, const void *beta, void *Y, const int incY);
```

?hemv

```
void cblas_chemv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const void *alpha, const void *A, const int lda, const void *X,
 const int incX, const void *beta, void *Y, const int incY);


void cblas_zhemv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const void *alpha, const void *A, const int lda, const void *X,
 const int incX, const void *beta, void *Y, const int incY);
```

?her

```
void cblas_cher(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const float alpha, const void *X, const int incX, void *A, const
 int lda);


void cblas_zher(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const double alpha, const void *X, const int incX, void *A,
const int lda);
```

?her2

```
void cblas_cher2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const void *alpha, const void *X, const int incX, const void
*Y, const int incY, void *A, const int lda);


void cblas_zher2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const void *alpha, const void *X, const int incX, const void
*Y, const int incY, void *A, const int lda);
```

?hpmv

```
void cblas_chpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const void *alpha, const void *Ap, const void *X, const int
incX, const void *beta, void *Y, const int incY);


void cblas_zhpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const void *alpha, const void *Ap, const void *X, const int
incX, const void *beta, void *Y, const int incY);
```

?hpr

```
void cblas_chpr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const float alpha, const void *X, const int incX, void *A);

void cblas_zhpr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const double alpha, const void *X, const int incX, void *A);
```

?hpr2

```
void cblas_chpr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const void *alpha, const void *X, const int incX, const void
*Y, const int incY, void *Ap);

void cblas_zhpr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const void *alpha, const void *X, const int incX, const void
*Y, const int incY, void *Ap);
```

?sbmv

```
void cblas_ssbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const int K, const float alpha, const float *A, const int lda,
 const float *X, const int incX, const float beta, float *Y, const int incY);

void cblas_dsbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const int K, const double alpha, const double *A, const int
lda, const double *X, const int incX, const double beta, double *Y, const
int incY);
```

?spmv

```
void cblas_sspmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const float alpha, const float *Ap, const float *X, const int
incX, const float beta, float *Y, const int incY);

void cblas_dspmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const double alpha, const double *Ap, const double *X, const
int incX, const double beta, double *Y, const int incY);
```

?spr

```
void cblas_sspr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const float alpha, const float *X, const int incX, float *Ap);


void cblas_dspr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const double alpha, const double *X, const int incX, double
*Ap);
```

?spr2

```
void cblas_sspr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const float alpha, const float *X, const int incX, const float
 *Y, const int incY, float *A);


void cblas_dspr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const double alpha, const double *X, const int incX, const
double *Y, const int incY, double *A);
```

?symv

```
void cblas_ssymv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const float alpha, const float *A, const int lda, const float
*X, const int incX, const float beta, float *Y, const int incY);


void cblas_dsymv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const double alpha, const double *A, const int lda, const double
 *X, const int incX, const double beta, double *Y, const int incY);
```

?syr

```
void cblas_ssyr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const float alpha, const float *X, const int incX, float *A,
const int lda);


void cblas_dsyr(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const double alpha, const double *X, const int incX, double *A,
 const int lda);
```

?syr2

```
void cblas_ssyr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const float alpha, const float *X, const int incX, const float
 *Y, const int incY, float *A, const int lda);


void cblas_dsyr2(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const int N, const double alpha, const double *X, const int incX, const
double *Y, const int incY, double *A, const int lda);
```

?tbmv

```
void cblas_stbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,
 const int K, const float *A, const int lda, float *X, const int incX);


void cblas_dtbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,
 const int K, const double *A, const int lda, double *X, const int incX);


void cblas_ctbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,
 const int K, const void *A, const int lda, void *X, const int incX);


void cblas_ztbmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,
 const int K, const void *A, const int lda, void *X, const int incX);
```

?tbsv

```
void cblas_stbsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,
 const int K, const float *A, const int lda, float *X, const int incX);


void cblas_dtbsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,
 const int K, const double *A, const int lda, double *X, const int incX);


void cblas_ctbsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,
 const int K, const void *A, const int lda, void *X, const int incX);
```

```
void cblas_ztbsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,
 const int K, const void *A, const int lda, void *X, const int incX);
```

?tpmv

```
void cblas_stpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,
 const float *Ap, float *X, const int incX);

void cblas_dtpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int N,
 const double *Ap, double *X, const int incX);

void cblas_ctpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
N,const void *Ap, void *X, const int incX);

void cblas_ztpmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
N,const void *Ap, void *X, const int incX);
```

?tpsv

```
void cblas_stpsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
N,const float *Ap, float *X, const int incX);

void cblas_dtpsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
N,const double *Ap, double *X, const int incX);

void cblas_ctpsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
N,const void *Ap, void *X, const int incX);

void cblas_ztpsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
N,const void *Ap, void *X, const int incX);
```

?trmv

```
void cblas_strmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
N,const float *A, const int lda, float *X, const int incX);

void cblas_dtrmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
N,const double *A, const int lda, double *X, const int incX);

void cblas_ctrmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
N,const void *A, const int lda, void *X, const int incX);

void cblas_ztrmv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
N,const void *A, const int lda, void *X, const int incX);
```

?trsv

```
void cblas_strsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
N,const float *A, const int lda, float *X, const int incX);

void cblas_dtrsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
N,const double *A, const int lda, double *X, const int incX);

void cblas_ctrsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
N,const void *A, const int lda, void *X, const int incX);

void cblas_ztrsv(const enum CBLAS_ORDER order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE TransA, const enum CBLAS_DIAG Diag, const int
N,const void *A, const int lda, void *X, const int incX);
```

Level 3 CBLAS

This is an interface to "BLAS Level 3 Routines", which perform basic matrix-matrix operations. Each C routine in this group has an additional parameter of type `CBLAS_ORDER` (the first argument) that determines whether the two-dimensional arrays use column-major or row-major storage.

?gemm

```
void cblas_sgemm(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE
TransA, const enum CBLAS_TRANSPOSE TransB, const int M, const int N, const
int K, const float alpha, const float *A, const int lda, const float *B,
const int ldb, const float beta, float *C, const int ldc);


void cblas_dgemm(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE
TransA, const enum CBLAS_TRANSPOSE TransB, const int M, const int N, const
int K, const double alpha, const double *A, const int lda, const double *B,
 const int ldb, const double beta, double *C, const int ldc);


void cblas_cgemm(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE
TransA, const enum CBLAS_TRANSPOSE TransB, const int M, const int N, const
int K, const void *alpha, const void *A, const int lda, const void *B, const
 int ldb, const void *beta, void *C, const int ldc);


void cblas_zgemm(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE
TransA, const enum CBLAS_TRANSPOSE TransB, const int M, const int N, const
int K, const void *alpha, const void *A, const int lda, const void *B, const
 int ldb, const void *beta, void *C, const int ldc);
```

?hemm

```
void cblas_chemm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
const enum CBLAS_UPLO Uplo, const int M, const int N, const void *alpha,
const void *A, const int lda, const void *B, const int ldb, const void *beta,
 void *C, const int ldc);


void cblas_zhemm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
const enum CBLAS_UPLO Uplo, const int M, const int N, const void *alpha,
const void *A, const int lda, const void *B, const int ldb, const void *beta,
 void *C, const int ldc);
```

?herk

```
void cblas_cherk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const float
alpha, const void *A, const int lda, const float beta, void *C, const int
ldc);


void cblas_zherk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const double
alpha, const void *A, const int lda, const double beta, void *C, const int
ldc);
```

?her2k

```
void cblas_cher2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
 const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const void
*alpha, const void *A, const int lda, const void *B, const int ldb, const
float beta, void *C, const int ldc);


void cblas_zher2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
 const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const void
*alpha, const void *A, const int lda, const void *B, const int ldb, const
double beta, void *C, const int ldc);
```

?symm

```
void cblas_ssymm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
const enum CBLAS_UPLO Uplo, const int M, const int N, const float alpha,
const float *A, const int lda, const float *B, const int ldb, const float
beta, float *C, const int ldc);


void cblas_dsymm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
const enum CBLAS_UPLO Uplo, const int M, const int N, const double alpha,
const double *A, const int lda, const double *B, const int ldb, const double
 beta, double *C, const int ldc);


void cblas_csymm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
const enum CBLAS_UPLO Uplo, const int M, const int N, const void *alpha,
const void *A, const int lda, const void *B, const int ldb, const void *beta,
 void *C, const int ldc);
```

```
void cblas_zsymm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
const enum CBLAS_UPLO Uplo, const int M, const int N, const void *alpha,
const void *A, const int lda, const void *B, const int ldb, const void *beta,
 void *C, const int ldc);
```

?syrk

```
void cblas_ssyrk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const float
alpha, const float *A, const int lda, const float beta, float *C, const int
 ldc);
```

```
void cblas_dsyrk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const double
alpha, const double *A, const int lda, const double beta, double *C, const
int ldc);
```

```
void cblas_csyrk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const void *alpha,
 const void *A, const int lda, const void *beta, void *C, const int ldc);
```

```
void cblas_zsyrk(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const void *alpha,
 const void *A, const int lda, const void *beta, void *C, const int ldc);
```

?syr2k

```
void cblas_ssyr2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
 const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const float
alpha, const float *A, const int lda, const float *B, const int ldb, const
float beta, float *C, const int ldc);
```

```
void cblas_dsyr2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
 const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const double
alpha, const double *A, const int lda, const double *B, const int ldb, const
 double beta, double *C, const int ldc);
```

```
void cblas_csyr2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
 const enum CBLAS_TRANSP SE Trans, const int N, const int K, const void
*alpha,const void *A, const int lda, const void *B, const int ldb, const
void *beta, void *C, const int ldc);
```

```
void cblas_zsyr2k(const enum CBLAS_ORDER Order, const enum CBLAS_UPLO Uplo,
 const enum CBLAS_TRANSPOSE Trans, const int N, const int K, const void
*alpha, const void *A, const int lda, const void *B, const int ldb, const
void *beta, void *C, const int ldc);
```

?trmm

```
void cblas_strmm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int M, const int N, const float alpha, const float
*A, const int lda, float *B, const int ldb);

void cblas_dtrmm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int M, const int N, const double alpha, const double
 *A, const int lda, double *B, const int ldb);

void cblas_ctrmm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int M, const int N, const void *alpha, const void *A,
 const int lda, void *B, const int ldb);

void cblas_ztrmm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int M, const int N, const void *alpha, const void *A,
 const int lda, void *B, const int ldb);
```

?trsm

```
void cblas_strsm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int M, const int N, const float alpha, const float
*A, const int lda, float *B, const int ldb);

void cblas_dtrsm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int M, const int N, const double alpha, const double
 *A, const int lda, double *B, const int ldb);

void cblas_ctrsm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int M, const int N, const void *alpha, const void *A,
 const int lda, void *B, const int ldb);
```

```
void cblas_ztrsm(const enum CBLAS_ORDER Order, const enum CBLAS_SIDE Side,
const enum CBLAS_UPLO Uplo, const enum CBLAS_TRANSPOSE TransA, const enum
CBLAS_DIAG Diag, const int M, const int N, const void *alpha, const void *A,
 const int lda, void *B, const int ldb);
```

Sparse CBLAS

This is an interface to "Sparse BLAS Level 1 Routines and Functions", which perform a number of common vector operations on sparse vectors stored in compressed form.

Note that all index parameters, *indx*, are in C-type notation and vary in the range [0..*N*–1].

?axpyi

```
void cblas_saxpyi(const int N, const float alpha, const float *X, const int
 *indx, float *Y);

void cblas_daxpyi(const int N, const double alpha, const double *X, const
int *indx, double *Y);


void cblas_caxpyi(const int N, const void *alpha, const void *X, const int
*indx, void *Y);

void cblas_zaxpyi(const int N, const void *alpha, const void *X, const int
*indx, void *Y);
```

?doti

```
float cblas_sdoti(const int N, const float *X, const int *indx, const float
 *Y);

double cblas_ddoti(const int N, const double *X, const int *indx, const
double *Y);
```

?dotci

```
void  cblas_cdotci_sub(const int N, const void *X, const int *indx, const
void *Y, void *dotui);

void  cblas_zdotci_sub(const int N, const void *X, const int *indx, const
void *Y, void *dotui);
```

?dotui

```
void  cblas_cdotui_sub(const int N, const void *X, const int *indx, const
void *Y, void *dotui);

void  cblas_zdotui_sub(const int N, const void *X, const int *indx, const
void *Y, void *dotui);
```

?gthr

```
void cblas_sgthr(const int N, const float *Y, float *X, const int *indx);

void cblas_dgthr(const int N, const double *Y, double *X, const int *indx);

void cblas_cgthr(const int N, const void *Y, void *X, const int *indx);

void cblas_zgthr(const int N, const void *Y, void *X, const int *indx);
```

?gthrz

```
void cblas_sgthrz(const int N, float *Y, float *X, const int *indx);

void cblas_dgthrz(const int N, double *Y, double *X, const int *indx);

void cblas_cgthrz(const int N, void *Y, void *X, const int *indx);

void cblas_zgthrz(const int N, void *Y, void *X, const int *indx);
```

?roti

```
void cblas_sroti(const int N, float *X, const int *indx, float *Y, const
float c, const float s);

void cblas_droti(const int N, double *X, const int *indx, double *Y, const
double c, const double s);
```

?sctr

```
void cblas_ssctr(const int N, const float *X, const int *indx, float *Y);

void cblas_dsctr(const int N, const double *X, const int *indx, double *Y);

void cblas_csctr(const int N, const void *X, const int *indx, void *Y);

void cblas_zsctr(const int N, const void *X, const int *indx, void *Y);
```

Specific Features of Fortran-95 Interfaces for LAPACK Routines

Intel® MKL implements Fortran-95 interface for LAPACK package, further referred to as MKL LAPACK-95, to provide full capacity of MKL Fortran-77 LAPACK routines. This is the principal difference of Intel MKL from the Netlib Fortran-95 implementation for LAPACK.

A new feature of MKL LAPACK-95 by comparison with Intel MKL LAPACK-77 implementation is presenting a package of source interfaces along with wrappers that make the implementation compiler-independent. As a result, the MKL LAPACK package can be used in all programming environments intended for Fortran-95.

Depending on the degree and type of difference from Netlib implementation, the MKL LAPACK-95 interfaces fall into several groups that require different transformations (see "MKL Fortran-95 Interfaces for LAPACK Routines vs. Netlib Implementation"). The groups are given in full with the calling sequences of the routines and appropriate differences from Netlib analogs.

The following conventions are used:

```
<interface> ::=  <name of interface> '(' <arguments list>')'

<arguments list> ::=  <first argument> {<argument>}*

<first argument> ::=  < identifier >

<argument> ::=  <required argument>|<optional argument>

<required argument> ::=  ',' <identifier>

<optional argument> ::=  '[,' <identifier> ']'

<name of interface> ::=  <identifier>
```

where defined notions are separated from definitions by `::=`, notion names are marked by angle brackets, terminals are given in quotes, and `{…}*` denotes repetition zero, one, or more times.

<first argument> and each *<required argument>* should be present in all calls of denoted interface, *<optional argument>* may be omitted. Comments to interface definitions are provided where necessary. Comment lines begin with character `!`.Two interfaces with one name are presented when two variants of subroutine calls (separated by types of arguments) exist.

Interfaces Identical to Netlib

```
GETRI(A, IPIV [,INFO])
GEEQU(A,R,C[,ROWCND][,COLCND][,AMAX][,INFO])
GESV(A,B[,IPIV][,INFO])
GESVX(A,B,X[,AF][,IPIV][,FACT][,TRANS][,EQUED][,R][,C][,FERR][,BERR]
[,RCOND][,RPVGRW][,INFO])
GBSV(A,B[,KL][,IPIV][,INFO])
GTSV(DL,D,DU,B[,INFO])
GTSVX(DL,D,DU,B,X[,DLF][,DF][,DUF][,DU2][,IPIV][,FACT][,TRANS][,FERR]
[,BERR][,RCOND][,INFO])
POSV(A,B[,UPLO][,INFO])
POSVX(A,B,X[,UPLO][,AF][,FACT][,EQUED][,S][,FERR][,BERR][,RCOND][,INFO])
PPSV(A,B[,UPLO][,INFO])
PPSVX(A,B,X[,UPLO][,AF][,FACT][,EQUED][,S][,FERR][,BERR][,RCOND][,INFO])
PBSV(A,B[,UPLO][,INFO])
PBSVX(A,B,X[,UPLO][,AF][,FACT][,EQUED][,S][,FERR][,BERR][,RCOND][,INFO])
PTSV(D,E,B[,INFO])
PTSVX(D,E,B,X[,DF][,EF][,FACT][,FERR][,BERR][,RCOND][,INFO])
SYSV(A,B[,UPLO][,IPIV][,INFO])
SYSVX(A,B,X[,UPLO][,AF][,IPIV][,FACT][,FERR][,BERR][,RCOND][,INFO])
HESVX(A,B,X[,UPLO][,AF][,IPIV][,FACT][,FERR][,BERR][,RCOND][,INFO])
SYTRD(A,TAU[,UPLO][,INFO])
ORGTR(A,TAU[,UPLO][,INFO])
HETRD(A,TAU[,UPLO][,INFO])
UNGTR(A,TAU[,UPLO][,INFO])
SYGST(A,B[,ITYPE][,UPLO][,INFO])
HEGST(A,B[,ITYPE][,UPLO][,INFO])
```

```
GELS(A,B[,TRANS][,INFO])
GELSY(A,B[,RANK][,JPVT][,RCOND][,INFO])
GELSS(A,B[,RANK][,S][,RCOND][,INFO])
GELSD(A,B[,RANK][,S][,RCOND][,INFO])
GGLSE(A,B,C,D,X[,INFO])
GGGLM(A,B,D,X,Y[,INFO])
SYEV(A,W[,JOBZ][,UPLO][,INFO])
HEEV(A,W[,JOBZ][,UPLO][,INFO])
SYEVD(A,W[,JOBZ][,UPLO][,INFO])
SPEV(A,W[,UPLO][,Z][,INFO])
HPEV(A,W[,UPLO][,Z][,INFO])
SPEVD(A,W[,UPLO][,Z][,INFO])
HPEVD(A,W[,UPLO][,Z][,INFO])
SPEVX(A,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO])
HPEVX(A,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO])
SBEV(A,W[,UPLO][,Z][,INFO])
HBEV(A,W[,UPLO][,Z][,INFO])
SBEVD(A,W[,UPLO][,Z][,INFO])
HBEVD(A,W[,UPLO][,Z][,INFO])
SBEVX(A,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,Q][,ABSTOL]
[,INFO])
HBEVX(A,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,Q][,ABSTOL]
[,INFO])
HPGV(A,B,W[,ITYPE][,UPLO][,Z][,INFO])
STEV(D,E[,Z][,INFO])
STEVD(D,E[,Z][,INFO])
STEVX(D,E,W[,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO])
STEVR(D,E,W[,Z][,VL][,VU][,IL][,IU][,M][,ISUPPZ][,ABSTOL][,INFO])
```

```
GEES(A,WR,WI[,VS][,SELECT][,SDIM][,INFO])
GEES(A,W[,VS][,SELECT][,SDIM][,INFO])
GEESX(A,WR,WI[,VS][,SELECT][,SDIM][,RCONDE][,RCONDV][,INFO])
GEESX(A,W[,VS][,SELECT][,SDIM][,RCONDE][,RCONDV][,INFO])
GEEV(A,WR,WI[,VL][,VR][,INFO])
GEEV(A,W[,VL][,VR][,INFO])
GEEVX(A,WR,WI[,VL][,VR][,BALANC][,ILO][,IHI][,SCALE][,ABNRM][,RCONDE][,RCONDV][,INFO])
GEEVX(A,W[,VL][,VR][,BALANC][,ILO][,IHI][,SCALE][,ABNRM][,RCONDE]
[,RCONDV][,INFO])
GESVD(A,S[,U][,VT][,WW][,JOB][,INFO])
GGSVD(A,B,ALPHA,BETA[,K][,L][,U][,V][,Q][,IWORK][,INFO])
SYGV(A,B,W[,ITYPE][,JOBZ][,UPLO][,INFO])
HEGV(A,B,W[,ITYPE][,JOBZ][,UPLO][,INFO])
SYGVD(A,B,W[,ITYPE][,JOBZ][,UPLO][,INFO])
HEGVD(A,B,W[,ITYPE][,JOBZ][,UPLO][,INFO])
SPGV(A,B,W[,ITYPE][,UPLO][,Z][,INFO])
SBGV(A,B,W[,UPLO][,Z][,INFO])
HBGV(A,B,W[,UPLO][,Z][,INFO])
GGES(A,B,ALPHAR,ALPHAI,BETA[,VSL][,VSR][,SELECT][,SDIM][,INFO])
GGES(A,B,ALPHA,BETA[,VSL][,VSR][,SELECT][,SDIM][,INFO])
GGESX(A,B,ALPHAR,ALPHAI,BETA[,VSL][,VSR][,SELECT][,SDIM][,RCONDE][,RCONDV][,INFO])
GGESX(A,B,ALPHA,BETA[,VSL][,VSR][,SELECT][,SDIM][,RCONDE][,RCONDV][,INFO])
GGEV(A,B,ALPHAR,ALPHAI,BETA[,VL][,VR][,INFO])
GGEV(A,B,ALPHA,BETA[,VL][,VR][,INFO])
GGEVX(A,B,ALPHAR,ALPHAI,BETA[,VL][,VR][,BALANC][,ILO][,IHI][,LSCALE][,RSCALE][,ABNRM]
[,BBNRM][,RCONDE][,RCONDV][,INFO])
GGEVX(A,B,ALPHA,BETA[,VL][,VR][,BALANC][,ILO][,IHI][,LSCALE][,RSCALE][,ABNRM]
```

```
[,BBNRM][,RCONDE][,RCONDV][,INFO])
GERFS(A,AF,IPIV,B,X[,TRANS][,FERR][,BERR][,INFO])
```

Interfaces with Replaced Argument Names

Argument names in the routines of this group are replaced as follows:

Netlib Argument Name	MKL Argument Name
AP	A
AB	A
AFP	AF
BP	B
BB	B

```
SPSV(A,B[,UPLO][,IPIV][,INFO])
!   netlib: (AP,B,UPLO,IPIV,INFO)
SPSVX(A,B,X[,UPLO][,AF][,IPIV][,FACT][,FERR][,BERR][,RCOND][,INFO])
!   netlib: (A,B,X,UPLO,AFP,IPIV,FACT,FERR,BERR,RCOND,INFO)
HPSVX(A,B,X[,UPLO][,AF][,IPIV][,FACT][,FERR][,BERR][,RCOND][,INFO])
!   netlib: (A,B,X,UPLO,AFP,IPIV,FACT,FERR,BERR,RCOND,INFO)
HEEVD(A,W[,JOB][,UPLO][,INFO])
!   netlib: (A,W,JOBZ,UPLO,INFO)
SPGVD(A,B,W[,ITYPE][,UPLO][,Z][,INFO])
!   netlib: (AP,BP,W,ITYPE,UPLO,Z,INFO)
HPGVD(A,B,W[,ITYPE][,UPLO][,Z][,INFO])
!   netlib: (AP,BP,W,ITYPE,UPLO,Z,INFO)
SPGVX(A,B,W[,ITYPE][,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO])
!   netlib: (AP,BP,W,ITYPE,UPLO,Z,VL,VU,IL,IU,M,IFAIL,ABSTOL,INFO)
HPGVX(A,B,W[,ITYPE][,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO])
!   netlib: (AP,BP,W,ITYPE,UPLO,Z,VL,VU,IL,IU,M,IFAIL,ABSTOL,INFO)
SBGVD(A,B,W[,UPLO][,Z][,INFO])
```

```
!   netlib: (AB,BB,W,UPLO,Z,INFO)
HBGVD(A,B,W[,UPLO][,Z][,INFO])
!   netlib: (AB,BB,W,UPLO,Z,INFO)
SBGVX(A,B,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,Q][,ABSTOL][,INFO])
!   netlib: (AB,BB,W,UPLO,Z,VL,VU,IL,IU,M,IFAIL,Q,ABSTOL,INFO)
HBGVX(A,B,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,Q][,ABSTOL][,INFO])
!   netlib: (AB,BB,W,UPLO,Z,VL,VU,IL,IU,M,IFAIL,Q,ABSTOL,INFO)
GBSVX(A,B,X[,KL][,AF][,IPIV][,FACT][,TRANS][,EQUED][,R][,C][,FERR]
[,BERR][,RCOND][,RPVGRW][,INFO])
!   netlib: !(A,B,X,KL,AFB,IPIV,FACT,TRANS,EQUED,R,C,FERR,
BERR,RCOND,RPVGRW,INFO)
```

Modified Netlib Interfaces

```
SYEVX(A,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO])
!   Interface netlib95 exists, parameters:
!   netlib: (A,W,JOBZ,UPLO,VL,VU,IL,IU,M,IFAIL,ABSTOL,INFO)
!   Different order for parameter UPLO, netlib: 4, mkl: 3
!   Absent mkl parameter: JOBZ
!   Extra mkl parameter: Z
HEEVX(A,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO])
!   Interface netlib95 exists, parameters:
!   netlib: (A,W,JOBZ,UPLO,VL,VU,IL,IU,M,IFAIL,ABSTOL,INFO)
!   Different order for parameter UPLO, netlib: 4, mkl: 3
!   Absent mkl parameter: JOBZ
!   Extra mkl parameter: Z
SYEVR(A,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,ISUPPZ][,ABSTOL][,INFO])
!   Interface netlib95 exists, parameters:
!   netlib: (A,W,JOBZ,UPLO,VL,VU,IL,IU,M,ISUPPZ,ABSTOL,INFO)
!   Different order for parameter UPLO, netlib: 4, mkl: 3
!   Absent mkl parameter: JOBZ
!   Extra mkl parameter: Z
HEEVR(A,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,ISUPPZ][,ABSTOL][,INFO])
!   Interface netlib95 exists, parameters:
!   netlib: (A,W,JOBZ,UPLO,VL,VU,IL,IU,M,ISUPPZ,ABSTOL,INFO)
!   Different order for parameter UPLO, netlib: 4, mkl: 3
!   Absent mkl parameter: JOBZ
!   Extra mkl parameter: Z
GESDD(A,S[,U][,VT][,JOBZ][,INFO])
!   Interface netlib95 exists, parameters:
```

```
!   netlib: (A,S,U,VT,WW,JOB,INFO)
!   Different number for parameter, netlib: 7, mkl: 6
!   Absent mkl parameter: WW
!   Absent mkl parameter: JOB
!   Different order for parameter INFO, netlib: 7, mkl: 6
!   Extra mkl parameter: JOBZ
SYGVX(A,B,W[,ITYPE][,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO])
!   Interface netlib95 exists, parameters:
!   netlib: (A,B,W,ITYPE,JOBZ,UPLO,VL,VU,IL,IU,M,IFAIL,ABSTOL,INFO)
!   Different order for parameter UPLO, netlib: 6, mkl: 5
!   Absent mkl parameter: JOBZ
!   Extra mkl parameter: Z
HEGVX(A,B,W[,ITYPE][,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO])
!   Interface netlib95 exists, parameters:
!   netlib: (A,B,W,ITYPE,JOBZ,UPLO,VL,VU,IL,IU,M,IFAIL,ABSTOL,INFO)
!   Different order for parameter UPLO, netlib: 6, mkl: 5
!   Absent mkl parameter: JOBZ
!   Extra mkl parameter: Z
GETRS(A,IPIV,B[,TRANS][,INFO])
!   Interface netlib95 exists:
!   Different intents for parameter A, netlib: INOUT, mkl: IN
```

Interfaces Absent From Netlib

```
GTTRF(DL,D,DU,DU2[,IPIV][,INFO])
PPTRF(A[,UPLO][,INFO])
PBTRF(A[,UPLO][,INFO])
PTTRF(D,E[,INFO])
SYTRF(A[,UPLO][,IPIV][,INFO])
HETRF(A[,UPLO][,IPIV][,INFO])
SPTRF(A[,UPLO][,IPIV][,INFO])
HPTRF(A[,UPLO][,IPIV][,INFO])
GBTRS(A,B,IPIV[,KL][,TRANS][,INFO])
GTTRS(DL,D,DU,DU2,B,IPIV[,TRANS][,INFO])
POTRS(A,B[,UPLO][,INFO])
PPTRS(A,B[,UPLO][,INFO])
PBTRS(A,B[,UPLO][,INFO])
PTTRS(D,E,B[,INFO])
```

```
PTTRS(D,E,B[,UPLO][,INFO])
SYTRS(A,B,IPIV[,UPLO][,INFO])
HETRS(A,B,IPIV[,UPLO][,INFO])
SPTRS(A,B,IPIV[,UPLO][,INFO])
HPTRS(A,B,IPIV[,UPLO][,INFO])
TRTRS(A,B[,UPLO][,TRANS][,DIAG][,INFO])
TPTRS(A,B[,UPLO][,TRANS][,DIAG][,INFO])
TBTRS(A,B[,UPLO][,TRANS][,DIAG][,INFO])
GECON(A,ANORM,RCOND[,NORM][,INFO])
GBCON(A,IPIV,ANORM,RCOND[,KL][,NORM][,INFO])
GTCON(DL,D,DU,DU2,IPIV,ANORM,RCOND[,NORM][,INFO])
POCON(A,ANORM,RCOND[,UPLO][,INFO])
PPCON(A,ANORM,RCOND[,UPLO][,INFO])
PBCON(A,ANORM,RCOND[,UPLO][,INFO])
PTCON(D,E,ANORM,RCOND[,INFO])
SYCON(A,IPIV,ANORM,RCOND[,UPLO][,INFO])
HECON(A,IPIV,ANORM,RCOND[,UPLO][,INFO])
SPCON(A,IPIV,ANORM,RCOND[,UPLO][,INFO])
HPCON(A,IPIV,ANORM,RCOND[,UPLO][,INFO])
TRCON(A,RCOND[,UPLO][,DIAG][,NORM][,INFO])
TPCON(A,RCOND[,UPLO][,DIAG][,NORM][,INFO])
TBCON(A,RCOND[,UPLO][,DIAG][,NORM][,INFO])
GBRFS(A,AF,IPIV,B,X[,KL][,TRANS][,FERR][,BERR][,INFO])
GTRFS(DL,D,DU,DLF,DF,DUF,DU2,IPIV,B,X[,TRANS][,FERR][,BERR][,INFO])
PORFS(A,AF,B,X[,UPLO][,FERR][,BERR][,INFO])
PPRFS(A,AF,B,X[,UPLO][,FERR][,BERR][,INFO])
PBRFS(A,AF,B,X[,UPLO][,FERR][,BERR][,INFO])
PTRFS(D,DF,E,EF,B,X[,FERR][,BERR][,INFO])
```

```
PTRFS(D,DF,E,EF,B,X[,UPLO][,FERR][,BERR][,INFO])
SYRFS(A,AF,IPIV,B,X[,UPLO][,FERR][,BERR][,INFO])
HERFS(A,AF,IPIV,B,X[,UPLO][,FERR][,BERR][,INFO])
SPRFS(A,AF,IPIV,B,X[,UPLO][,FERR][,BERR][,INFO])
HPRFS(A,AF,IPIV,B,X[,UPLO][,FERR][,BERR][,INFO])
TRRFS(A,B,X[,UPLO][,TRANS][,DIAG][,FERR][,BERR][,INFO])
TPRFS(A,B,X[,UPLO][,TRANS][,DIAG][,FERR][,BERR][,INFO])
TBRFS(A,B,X[,UPLO][,TRANS][,DIAG][,FERR][,BERR][,INFO])
POTRI(A[,UPLO][,INFO])
PPTRI(A[,UPLO][,INFO])
SYTRI(A,IPIV[,UPLO][,INFO])
HETRI(A,IPIV[,UPLO][,INFO])
SPTRI(A,IPIV[,UPLO][,INFO])
HPTRI(A,IPIV[,UPLO][,INFO])
TRTRI(A[,UPLO][,DIAG][,INFO])
TPTRI(A[,UPLO][,DIAG][,INFO])
GBEQU(A,R,C[,KL][,ROWCND][,COLCND][,AMAX][,INFO])
POEQU(A,S[,SCOND][,AMAX][,INFO])
PPEQU(A,S[,SCOND][,AMAX][,UPLO][,INFO])
PBEQU(A,S[,SCOND][,AMAX][,UPLO][,INFO])
HESV(A,B[,UPLO][,IPIV][,INFO])
HPSV(A,B[,UPLO][,IPIV][,INFO])
GEQRF(A[,TAU][,INFO])
GEQPF(A,JPVT[,TAU][,INFO])
GEQP3(A,JPVT[,TAU][,INFO])
ORGQR(A,TAU[,INFO])
ORMQR(A,TAU,C[,SIDE][,TRANS][,INFO])
UNGQR(A,TAU[,INFO])
```

```
UNMQR(A,TAU,C[,SIDE][,TRANS][,INFO])
GELQF(A[,TAU][,INFO])
ORGLQ(A,TAU[,INFO])
ORMLQ(A,TAU,C[,SIDE][,TRANS][,INFO])
UNGLQ(A,TAU[,INFO])
UNMLQ(A,TAU,C[,SIDE][,TRANS][,INFO])
GEQLF(A[,TAU][,INFO])
ORGQL(A,TAU[,INFO])
UNGQL(A,TAU[,INFO])
ORMQL(A,TAU,C[,SIDE][,TRANS][,INFO])
UNMQL(A,TAU,C[,SIDE][,TRANS][,INFO])
GERQF(A[,TAU][,INFO])
ORGRQ(A,TAU[,INFO])
UNGRQ(A,TAU[,INFO])
ORMRQ(A,TAU,C[,SIDE][,TRANS][,INFO])
UNMRQ(A,TAU,C[,SIDE][,TRANS][,INFO])
TZRZF(A[,TAU][,INFO])
ORMRZ(A,TAU,C,L[,SIDE][,TRANS][,INFO])
UNMRZ(A,TAU,C,L[,SIDE][,TRANS][,INFO])
GGQRF(A,B[,TAUA][,TAUB][,INFO])
GGRQF(A,B[,TAUA][,TAUB][,INFO])
GEBRD(A[,D][,E][,TAUQ][,TAUP][,INFO])
GBBRD(A[,C][,D][,E][,Q][,PT][,KL][,M][,INFO])
ORGBR(A,TAU[,VECT][,INFO])
ORMBR(A,TAU,C[,VECT][,SIDE][,TRANS][,INFO])
ORMTR(A,TAU,C[,SIDE][,UPLO][,TRANS][,INFO])
UNGBR(A,TAU[,VECT][,INFO])
UNMBR(A,TAU,C[,VECT][,SIDE][,TRANS][,INFO])
```

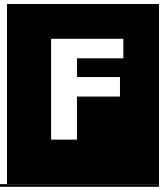
```
BDSQR(D,E[,VT][,U][,C][,UPLO][,INFO])
BDSDC(D,E[,U][,VT][,Q][,IQ][,UPLO][,INFO])
UNMTR(A,TAU,C[,SIDE][,UPLO][,TRANS][,INFO])
SPTRD(A,TAU[,UPLO][,INFO])
OPGTR(A,TAU,Q[,UPLO][,INFO])
OPMTR(A,TAU,C[,SIDE][,UPLO][,TRANS][,INFO])
HPTRD(A,TAU[,UPLO][,INFO])
UPGTR(A,TAU,Q[,UPLO][,INFO])
UPMTR(A,TAU,C[,SIDE][,UPLO][,TRANS][,INFO])
SBTRD(A[,Q][,VECT][,UPLO][,INFO])
HBTRD(A[,Q][,VECT][,UPLO][,INFO])
STERF(D,E[,INFO])
STEQR(D,E[,Z][,COMPZ][,INFO])
STEDC(D,E[,Z][,COMPZ][,INFO])
STEGR(D,E,W[,Z][,VL][,VU][,IL][,IU][,M][,ISUPPZ][,ABSTOL][,INFO])
PTEQR(D,E[,Z][,COMPZ][,INFO])
STEBZ(D,E,M,NSPLIT,W,IBLOCK,ISPLIT[,ORDER][,VL][,VU][,IL][,IU][,ABSTOL][,INFO])
STEIN(D,E,W,IBLOCK,ISPLIT,Z[,IFAILV][,INFO])
DISNA(D,SEP[,JOB][,MINMN][,INFO])
SPGST(A,B[,ITYPE][,UPLO][,INFO])
HPGST(A,B[,ITYPE][,UPLO][,INFO])
SBGST(A,B[,X][,UPLO][,INFO])
HBGST(A,B[,X][,UPLO][,INFO])
PBSTF(B[,UPLO][,INFO])
GEHRD(A[,TAU][,ILO][,IHI][,INFO])
ORGHR(A,TAU[,ILO][,IHI][,INFO])
ORMHR(A,TAU,C[,ILO][,IHI][,SIDE][,TRANS][,INFO])
UNGHR(A,TAU[,ILO][,IHI][,INFO])
```

```
UNMHR(A,TAU,C[,ILO][,IHI][,SIDE][,TRANS][,INFO])
GEBAL(A[,SCALE][,ILO][,IHI][,JOB][,INFO])
GEBAK(V,SCALE[,ILO][,IHI][,JOB][,SIDE][,INFO])
HSEQR(H,WR,WI[,ILO][,IHI][,Z][,JOB][,COMPZ][,INFO])
HSEQR(H,W[,ILO][,IHI][,Z][,JOB][,COMPZ][,INFO])
HSEIN(H,WR,WI,SELECT[,VL][,VR][,IFAILL][,IFAILR][,INITV][,EIGSRC][,M][,INFO])
HSEIN(H,W,SELECT[,VL][,VR][,IFAILL][,IFAILR][,INITV][,EIGSRC][,M][,INFO])
TREVC(T[,HOWMNY][,SELECT][,VL][,VR][,M][,INFO])
TRSNA(T[,S][,SEP][,VL][,VR][,SELECT][,M][,INFO])
TREXC(T,IFST,ILST[,Q][,INFO])
TRSEN(T,SELECT[,WR][,WI][,M][,S][,SEP][,Q][,INFO])
TRSEN(T,SELECT[,W][,M][,S][,SEP][,Q][,INFO])
TRSYL(A,B,C,SCALE[,TRANA][,TRANB][,ISGN][,INFO])
GGHRD(A,B[,ILO][,IHI][,Q][,Z][,COMPQ][,COMPZ][,INFO])
GGBAL(A,B[,ILO][,IHI][,LSCALE][,RSCALE][,JOB][,INFO])
GGBAK(V[,ILO][,IHI][,LSCALE][,RSCALE][,JOB][,INFO])
HGEQZ(H,T[,ILO][,IHI][,ALPHAR][,ALPHAI][,BETA][,Q][,Z][,JOB][,COMPQ][,COMPZ][,INFO])
HGEQZ(H,T[,ILO][,IHI][,ALPHA][,BETA][,Q][,Z][,JOB][,COMPQ][,COMPZ][,INFO])
TGEVC(S,P[,HOWMNY][,SELECT][,VL][,VR][,M][,INFO])
TGEXC(A,B[,IFST][,ILST][,Z][,Q][,INFO])
TGSEN(A,B,SELECT[,ALPHAR][,ALPHAI][,BETA][,IJOB][,Q][,Z][,PL][,PR][,DIF][,M][,INFO])
TGSEN(A,B,SELECT[,ALPHA][,BETA][,IJOB][,Q][,Z][,PL][,PR][,DIF][,M][,INFO])
TGSYL(A,B,C,D,E,F[,IJOB][,TRANS][,SCALE][,DIF][,INFO])
TGSNA(A,B[,S][,DIF][,VL][,VR][,SELECT][,M][,INFO])
GGSVP(A,B,TOLA,TOLB[,K][,L][,U][,V][,Q][,INFO])
TGSJA(A,B,TOLA,TOLB,K,L[,U][,V][,Q][,JOBU][,JOBV][,JOBQ][,ALPHA][,BETA][,NCYCLE][,INFO])
```

Interfaces of New Functionality

```
GETRF(A[,IPIV][,INFO])
!   Interface netlib95 exists, parameters:
!   netlib: (A,IPIV,RCOND,NORM,INFO)
!   Different number for parameter, netlib: 5, mkl: 3
!   Different order for parameter INFO, netlib: 5, mkl: 3
!   Absent mkl parameter: NORM
!   Absent mkl parameter: RCOND
GBTRF(A[,KL][,M][,IPIV][,INFO])
!   Interface netlib95 exists, parameters:
!   netlib: (A,K,M,IPIV,RCOND,NORM,INFO)
!   Different number for parameter, netlib: 7, mkl: 5
!   Different order for parameter INFO, netlib: 7, mkl: 5
!   Absent mkl parameter: NORM
!   Replace parameter name: netlib: K: mkl: KL
!   Absent mkl parameter: RCOND
POTRF(A[,UPLO][,INFO])
!   Interface netlib95 exists, parameters:
!   netlib: (A,UPLO,RCOND,NORM,INFO)
!   Different number for parameter, netlib: 5, mkl: 3
!   Different order for parameter INFO, netlib: 5, mkl: 3
!   Absent mkl parameter: NORM
!   Absent mkl parameter: RCOND
```

Optimization Solvers Basics



Classical optimization methods are linear methods that are based on calculation of direction and searching for the best point in the direction of the chosen gradient. Direction is calculated on each iteration. Examples of these methods are Conjugate Gradients method and Quickest Descent Method. Method of searching for direction usually requires calculation of subtask that approximates the objective function in neighborhood of the current point.

Optimization problems are generally made up of three basic components:

- An `objective function` that needs to be minimized or maximized. For instance, in a manufacturing process, you might want to maximize the profit or minimize the cost. In fitting experimental data to a user-defined model, the total deviation of observed data can be minimized from predictions based on the model.
- A set of `unknowns` or `variables` that affect the value of the objective function. In the manufacturing problem, the variables might include the amounts of different resources used or the time spent on each activity. In fitting-the-data problem, the unknowns are the parameters that define the model.
- A set of `constraints` that allow the unknowns to take on certain values but exclude others. For the manufacturing problem, it does not make sense to spend a negative amount of time on any activity, so all the "time" variables are constrained to be non-negative.

So the `optimization problem` can be formulated as follows: Find values of the variables that minimize or maximize the objective function while satisfying the constraints.

Intel® MKL provides math optimization tools, namely the Trust-Region (TR) solvers routines for solving nonlinear least squares problem with or without linear (bound) constraints.

Nonlinear Least Square Problem

Nonlinear least squares problem can be described as follows:

$$\min_{x \in R^n} F(x) = \|y - f(x)\|_2^2, y \in R^m, x \in R^n, f : R^n \to R^m, m > n,$$

where $F(x) : R^n \to R$ is a twice differentiable function in R^n. Solving of nonlinear least squares problem is searching for the best approximation to vector y with the model function that has nonlinear dependence on variables $\tilde{N}\square$. Best approximation means that the sum of squares of residuals is the lowest possible.

Let us designate

$$F(x) = \sum_{i=1}^{m} r_i(x)^2, \quad r_i(x) = y_i - f_i(x),$$

so

$$F(x) = \frac{1}{2} R(x)^T \ R(x),$$

where $R(x)=\{r_i(x)\}$, i = 1, ... m.

Trust Region Algorithm

The Trust Region (TR) algorithms are relatively new iterative algorithms for solving nonlinear optimization problems. They are widely used in power engineering, finance, applied mathematics, physics, computer science, economics, sociology, biology, medicine, mechanical engineering, chemistry, and other areas. TR methods have global convergence and local super convergence, which differenciates them from line search methods and Newton methods. TR methods have better convergence when compared with widely-used Newton-type methods.

The main idea behind TR algorithm is calculating a trial step and checking if the next values of x_+ belong to the `trust region` [Conn00]. Calculation of the trial step is strongly associated with the approximation model.

If the nonlinear least squares problem is defined as

$$\min_{x \in R^n} F(x) = \|y - f(x)\|_2^2, y \in R^m, x \in R^n, f : R^n \to R^m, m > n,$$

then the trial step is solution of the following subproblem:

$$\min_{d \in R^n} \left\| F(x_k) + J(x_k)^T d \right\|_2^2, \text{ where } \|d\|_2^2 < \Delta_k \text{ and } J(x_k) \text{ is Jacobi matrix.}$$

This operation is approximation of the objective function in neighborhood of the current point x.

Let us consider a TR algorithm with **`boundary (linear) constraints`** :

$$\min_{l \le x \le u} f(x) = \min_{l \le x \le u} \frac{1}{2} \|F(x)\|^2.$$

The first-order necessary conditions for the vector to be minimized with this constraint are stated as

$$D(x)^{-2} \nabla f(x) = D(x)^{-2} F'^T(x) F(x) = 0,$$

where D is the diagonal scaling matrix

$$D(x) = diag\left(\left|v_1^{-\frac{1}{2}}(x)\right|, \left|v_2^{-\frac{1}{2}}(x)\right|, \ldots, \left|v_n^{-\frac{1}{2}}(x)\right|\right)$$

with

$$v_i(x) = \begin{cases} x_i - u_i & \text{if } \nabla f(x)_i < 0 \text{ and } u_i < \infty \\ x_i - l_i & \text{if } \nabla f(x)_i > 0 \text{ and } l_i < \infty \\ \min(x_i - u_i, u_i - x_i) & \text{if } \nabla f(x)_i = 0 \text{ and } u_i < \infty \text{ or } l_i > -\infty \\ -1 & \text{if } \nabla f(x)_i < 0 \text{ and } u_i = \infty \\ 1 & \text{if } \nabla f(x)_i < 0 \text{ and } u_i = \infty \\ 1 & \text{if } \nabla f(x)_i = 0 \text{ and } u_i = -l_i = \infty \end{cases}$$

for i = 1, ..., n.

Assume $\Omega = \{x \in R^n: \ l \le x \le u\} \subset R^n$, and write $\mathrm{int}(\Omega)$ for the strict nonempty interior of Ω. At each iteration, the basic structure of the method involves solution of an elliptical trust-region subproblem and computation of a search step to update the current iteratation.

Assume $x_i \in \text{int}(\Omega)$ and the trust region size $\Delta_k > 0$. Consider the elliptical trust-region subproblem given by

$$\min_p \left\{ F(x_k) + J(x_k)^T p : \|D_k p\| \le \Delta_k \right\}.$$

Let $p_{tr}(\Delta_k)$ and $p_c(\Delta_k)$ be a solution and a Cauchy point of the above trust-region subproblem respectively [Conn00]. The search step (trial-step) used in the algorithm is defined by a linear combination of two vectors $p_{tr}(\Delta_k)$ and $p_c(\Delta_k)$, that is,

$$p(\Delta_k) = t\overline{p}_c(\Delta_k) + (1-t)\overline{p}_{tr}(\Delta_k).$$

A good agreement between the model function m_k and the objective function f is ensured for the following standard condition:

$$\rho_f(p(\Delta_k)) = \frac{f(x_k) - f(x_k + p(\Delta_k))}{m_k(0) - m_k(p_k(\Delta_k))} \ge \beta_2$$

for the given constant $\beta_2 \in (0,1)$. If this condition is not met, reject $p(\Delta_k)$ and the trust-region size Δ_k is adjusted to be successive reduction so that $p(\Delta_k)$ satisfies the accuracy requirement.

Bibliography

For more information about the BLAS, Sparse BLAS, LAPACK, ScaLAPACK, Sparse Solver, Interval Solver, VML, VSL, and DFT functionality, refer to the following publications:

- **BLAS Level 1**

  C. Lawson, R. Hanson, D. Kincaid, and F. Krough. Basic Linear Algebra Subprograms for Fortran Usage, ACM Transactions on Mathematical Software, Vol.5, No.3 (September 1979) 308-325.

- **BLAS Level 2**

  J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms, ACM Transactions on Mathematical Software, Vol.14, No.1 (March 1988) 1-32.

- **BLAS Level 3**

  J. Dongarra, J. DuCroz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms, ACM Transactions on Mathematical Software (December 1989).

- **Sparse BLAS**

  D. Dodson, R. Grimes, and J. Lewis. Sparse Extensions to the FORTRAN Basic Linear Algebra Subprograms, ACM Transactions on Math Software, Vol.17, No.2 (June 1991).

  D. Dodson, R. Grimes, and J. Lewis. Algorithm 692: Model Implementation and Test Package for the Sparse Basic Linear Algebra Subprograms, ACM Transactions on Mathematical Software, Vol.17, No.2 (June 1991).

  [Duff86] I.S.Duff, A.M.Erisman, and J.K.Reid. Direct Methods for Sparse Matrices. Clarendon Press, Oxford, UK, 1986.

  [CXML01] Compaq Extended Math Library. Reference Guide, Oct.2001.

  [Rem05] K.Remington. A NIST FORTRAN Sparse Blas User's Guide. (available on http://math.nist.gov/~KRemington/fspblas/)

  [Saad94] Y.Saad. SPARSKIT: A Basic Tool-kit for Sparse Matrix Computation. Version 2, 1994.(http://www.cs.umn.edu/~saad)

  [Saad96] Y.Saad. Iterative Methods for Linear Systems. PWS Publishing, Boston, 1996.

- **LAPACK**

  [LUG] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK Users' Guide, Third Edition, Society for Industrial and Applied Mathematics (SIAM), 1999.

  [Golub96] G. Golub and C. Van Loan. Matrix Computations, Johns Hopkins University Press, Baltimore, third edition,1996.

  [Bischof92] http://citeseer.ist.psu.edu/bischof92framework.html

[Marques06] O.Marques, E.J.Riedy, and Ch.Voemel. Benefits of IEEE-754 Features in Modern Symmetric Tridiagonal Eigensolvers, SIAM Journal on Scientific Computing, Vol.28, No.5, 2006. (Tech report version in LAPACK Working Note 172 with the same title.)

[Kahan66] W. Kahan. Accurate Eigenvalues of a Symmetric Tridiagonal Matrix, Report CS41, Computer Science Dept., Stanford University, July 21, 1966.

[Dhillon04] I. Dhillon, B. Parlett. Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices, Linear Algebra and its Applications, 387(1), pp. 1-28, August 2004.

[Dhillon04-02] I. Dhillon, B. Parlett. Orthogonal Eigenvectors and * Relative Gaps, SIAM Journal on Matrix Analysis and Applications, Vol. 25, 2004. (Also LAPACK Working Note 154.)

[Dhillon97] I. Dhillon. A new O(n^2) algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem, Computer Science Division Technical Report No. UCB/CSD-97-971, UC Berkeley, May 1997.

- **ScaLAPACK**

[SLUG] L. Blackford, J. Choi, A.Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K.Stanley, D. Walker, and R. Whaley. ScaLAPACK Users' Guide, Society for Industrial and Applied Mathematics (SIAM), 1997.

- **Sparse Solver**

[Duff99] I. S. Duff and J. Koster. The Design and Use of Algorithms for Permuting Large Entries to the Diagonal of Sparse Matrices. SIAM J. Matrix Analysis and Applications, 20(4):889-901, 1999.

[Dong95] J. Dongarra, V.Eijkhout, A.Kalhan. Reverse Communication Interface for Linear Algebra Templates for Iterative Methods. UT-CS-95-291, May 1995. http://www.netlib.org/lapack/lawnspdf/lawn99.pdf

[Karypis98] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM Journal on Scientific Computing, 20(1):359-392, 1998.

[Li99] X.S. Li and J.W. Demmel. A Scalable Sparse Direct Solver Using Static Pivoting. In Proceeding of the 9th SIAM conference on Parallel Processing for Scientific Computing, San Antonio, Texas, March 22-34,1999.

[Liu85] J.W.H. Liu. Modification of the Minimum-Degree algorithm by multiple elimination. ACM Transactions on Mathematical Software, 11(2):141-153, 1985.

[Menon98] R. Menon L. Dagnum. OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Computational Science & Engineering, 1:46-55, 1998. http://www.openmp.org.

[Saad03] Y. Saad. Iterative Methods for Sparse Linear Systems. 2nd edition, SIAM, Philadelphia, PA, 2003.

[Schenk00] O. Schenk. Scalable Parallel Sparse LU Factorization Methods on Shared Memory Multiprocessors. PhD thesis, ETH Zurich, 2000.

[Schenk00-2] O. Schenk, K. Gartner, and W. Fichtner. Efficient Sparse LU Factorization with Left-right Looking Strategy on Shared Memory Multiprocessors. BIT, 40(1):158-176, 2000.

[Schenk01] O. Schenk and K. Gartner. Sparse Factorization with Two-Level Scheduling in PARDISO. In Proceeding of the 10th SIAM conference on Parallel Processing for Scientific Computing, Portsmouth, Virginia, March 12-14, 2001.

[Schenk02] O. Schenk and K. Gartner. Two-level scheduling in PARDISO: Improved Scalability on Shared Memory Multiprocessing Systems. Parallel Computing, 28:187-197, 2002.

[Schenk03] O. Schenk and K. Gartner. Solving Unsymmetric Sparse Systems of Linear Equations with PARDISO. Journal of Future Generation Computer Systems, 20(3):475-487, 2004.

[Schenk04] O. Schenk and K. Gartner. On Fast Factorization Pivoting Methods for Sparse Symmetric Indefinite Systems. Technical Report, Department of Computer Science, University of Basel, 2004, submitted.

[Sonn89] P. Sonneveld. CGS, a Fast Lanczos-Type Solver for Nonsymmetric Linear Systems. SIAM Journal on Scientific and Statistical Computing, 10:36-52, 1989.

[Young71] D.M.Young. Iterative Solution of Large Linear Systems. New York, Academic Press, Inc., 1971.

- **VSL**

[VSL Notes] Document included with Intel® MKL product (file name vslnotes.pdf).

[Bratley87] Bratley P., Fox B.L., and Schrage L.E. A Guide to Simulation. 2nd edition. Springer-Verlag, New York, 1987.

[Bratley88] Bratley P. and Fox B.L. Implementing Sobol`s Quasirandom Sequence Generator, ACM Transactions on Mathematical Software, Vol. 14, No. 1, Pages 88-100, March 1988.

[Bratley92] Bratley P., Fox B.L., and Niederreiter H. Implementation and Tests of Low-Discrepancy Sequences, ACM Transactions on Modeling and Computer Simulation, Vol. 2, No. 3, Pages 195-213, July 1992.

[Coddington94] Coddington, P. D. Analysis of Random Number Generators Using Monte Carlo Simulation. Int. J. Mod. Phys. C-5, 547, 1994.

[Gentle98] Gentle, James E. Random Number Generation and Monte Carlo Methods, Springer-Verlag New York, Inc., 1998.

[L'Ecuyer94] L'Ecuyer, Pierre. Uniform Random Number Generation. Annals of Operations Research, 53, 77-120, 1994.

[L'Ecuyer99] L'Ecuyer, Pierre. Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure. Mathematics of Computation, 68, 225, 249-260, 1999.

[L'Ecuyer99a] L'Ecuyer, Pierre. Good Parameter Sets for Combined Multiple Recursive Random Number Generators. Operations Research, 47, 1, 159-164, 1999.

[L'Ecuyer01] L'Ecuyer, Pierre. Software for Uniform Random Number Generation: Distinguishing the Good and the Bad. Proceedings of the 2001 Winter Simulation Conference, IEEE Press, 95-105, Dec. 2001.

[Kirkpatrick81] Kirkpatrick, S., and Stoll, E. A Very Fast Shift-Register Sequence Random Number Generator. Journal of Computational Physics, V. 40. 517-526, 1981.

[Knuth81] Knuth, Donald E. The Art of Computer Programming, Volume 2, Seminumerical Algorithms. 2nd edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.

[Matsumoto98] Matsumoto, M., and Nishimura, T. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator, ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1, Pages 3-30, January 1998.

[Matsumoto00] Matsumoto, M., and Nishimura, T. Dynamic Creation of Pseudorandom Number Generators, 56-69, in: Monte Carlo and Quasi-Monte Carlo Methods 1998, Ed. Niederreiter, H. and Spanier, J., Springer 2000, http://www.math.sci.hiroshima-u.ac.jp/%7Em-mat/MT/DC/dc.html.

[NAG] NAG Numerical Libraries. http://www.nag.co.uk/numeric/numerical_libraries.asp

[Sobol76] Sobol, I.M., and Levitan, Yu.L. The production of points uniformly distributed in a multidimensional cube. Preprint 40, Institute of Applied Mathematics, USSR Academy of Sciences, 1976 (In Russian).

- **DFT**

[1] E. Oran Brigham, The Fast Fourier Transform and Its Applications, Prentice Hall, New Jersey, 1988.

[2] Athanasios Papoulis, The Fourier Integral and its Applications, 2nd edition, McGraw-Hill, New York, 1984.

[3] Ping Tak Peter Tang, DFTI, a New API for DFT: Motivation, Design, and Rationale, July 2002.

[4] Charles Van Loan, Computational Frameworks for the Fast Fourier Transform, SIAM, Philadelphia, 1992

- **VML**

J.M.Muller. Elementary functions: algorithms and implementation, Birkhauser Boston, 1997.

IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-1985.

- **Interval Solver**

[Alefeld83] G. Alefeld and J. Herzberger, Introduction to Interval Computations. - Academic Press, New York, 1983.

[Bentbib02] A.H.Bentbib, Solving the full rank interval least squares problem // Applied Numerical Mathematics. - 2002. - Vol. 41. - P. 283-294.

[Bliek92] Ch. Bliek, Computer methods for design automation, Ph.D. Thesis. - Dept. of Ocean Engineering, Massachusetts Institute of Technology, 1992.

[Hammer95] R. Hammer, M. Hocks, U. Kulisch, D. Ratz, C++ Toolbox for Verified Computing I. Basic Numerical Problems. - Berlin-Heidelberg: Springer, 1995.

[Hansen92] E. Hansen, Bounding the solution of interval linear equations // SIAM Journal on Numerical Analysis. - 1992. - Vol. 29, No. 5. - P. 1493-1503.

[Herzberger94] J. Herzberger, Iterative methods for the inclusion of the inverse of a matrix // Topics in Validated Computations, J. Herzberger, ed. - Amsterdam: Elsevier, 1994. -

[Jansson91] Ch.Jansson, Interval linear systems with symmetric matrices, skew-symmetric matrices, and dependencies in the right hand side // Computing. - 1991. - Vol. 46. - P. 265 - 274.

[Kearfott96] R.B. Kearfott, Rigorous Global Search: Continuous Problems. - Dordrecht, Kluwer, 1996.

[Kearfott] R.B. Kearfott, M.T. Nakao, A. Neumaier, S.M. Rump, S.P. Shary, P. van Hentenryck, Standardized notation in interval analysis. - An electronic version of the paper is accessible at http://www.mat.univie.ac.at/~neum/software/int/

[Kreinovich97] V. Kreinovich, A. Lakeyev, J. Rohn, P. Kahl, Computational Complexity and Feasibility of Data Processing and Interval Computations. - Kluwer, Dordrecht, 1997.

[Neumaier90] A. Neumaier, Interval Methods for Systems of Equations. - Cambridge, Cambridge University Press, 1990.

[Neumaier99] A.Neumaier, A simple derivation of Hansen-Bliek-Rohn-Ning-Kearfott enclosure for linear interval equations // Reliable Computing. - 1999. - Vol. 5, No. 2. - P. 131-136.

[Ning97] S. Ning, R.B. Kearfott, A comparison of some methods for solving linear interval equations // SIAM Journal on Numerical Analysis. - 1997. - Vol. 34, No. 4. - P. 1289-1305.

[Rex99] G.Rex, J.Rohn, Sufficient conditions for regularity and singularity of interval matrices // SIAM Journal on Numerical Analysis. - 1999. - Vol. 20. - P. 437-445.

[Rohn93] J. Rohn, Cheap and tight bounds: the recent result by E. Hansen can be made more efficient // Interval Computations. - 1993. - No. 4. - P. 13-21.

[Rump83] S. M.Rump, Solving algebraic problems with high accuracy // A New Approach to Scientific Computation; Kulisch U. W. and Miranker W. L., eds. - New York: Academic Press, 1983. - P. 51-120.

[Rump84] S. M.Rump, Solution of linear and nonlinear algebraic problems with sharp guaranteed bounds // Computing Supplement. - 1984. - Vol. 5. - P. 147-168.

[Rump80] S. M.Rump, Kaucher E. Small bounds for the solution of systems of linear equations // Computing Supplement. - 1980. - Vol. 2. - P. 157-164.

[Rump] S. Rump, INTLAB -- INTerval LABoratory. - 21 p. - An electronic version of the paper is accessible at http://www.ti3.tu-harburg.de/rump/intlab/.

[Shary92] S.P. Shary, A new class of algorithms for optimal solution of interval linear systems // Interval Computations. - 1992, No. 2(4). - P. 18-29.

[Shary95] S.P. Shary, On optimal solution of interval linear equations // SIAM Journal on Numerical Analysis. - 1995. - Vol. 32, No. 2. - P. 610-630.

[Shary02] S. P. Shary, A new technique in systems analysis under interval uncertainty and ambiguity // Reliable Computing. - 2002. - Vol. 8. - 321-419.

[Shary] S.P. Shary, A new class of methods for optimal enclosing solution sets to interval linear systems // Journal of Computational Mathematics. - to be published.

- **Optimization Solvers**

[Conn00] A. R. Conn, N. I.M. Gould, P. L. Toint.Trust-region Methods.SIAM Society for Industrial & Applied Mathematics, Englewood Cliffs, New Jersey, MPS-SIAM Series on Optimization edition, 2000.

[Dong95] J. Dongara, V. Eijkhout, A. Kalhan.Reverse communication interface for linear algebra templates for iterative methods.1995.

For a reference implementation of BLAS, sparse BLAS, LAPACK, and ScaLAPACK packages (without platform-specific optimizations) visit www.netlib.org

Glossary

A^H	Denotes the conjugate of a general matrix A. See also conjugate matrix.
A^T	Denotes the transpose of a general matrix A. See also transpose.
band matrix	A general m-by-n matrix A such that $a_{ij} = 0$ for $\|i - j\| > l$, where $1 < l < \min(m, n)$. For example, any tridiagonal matrix is a band matrix.
band storage	A special storage scheme for band matrices. A matrix is stored in a two-dimensional array: columns of the matrix are stored in the corresponding columns of the array, and `diagonals` of the matrix are stored in rows of the array.
BLAS	Abbreviation for Basic Linear Algebra Subprograms. These subprograms implement vector, matrix-vector, and matrix-matrix operations.
BRNG	Abbreviation for Basic Random Number Generator. Basic random number generators are pseudorandom number generators imitating i.i.d. random number sequences of uniform distribution. Distributions other than uniform are generated by applying different transformation techniques to the sequences of random numbers of uniform distribution.
BRNG registration	Standardized mechanism that allows a user to include a user-designed BRNG into the VSL and use it along with the predefined VSL basic generators.
Bunch-Kaufman factorization	Representation of a real symmetric or complex Hermitian matrix A in the form $A = PUDU^HP^T$ (or $A = PLDL^HP^T$) where P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D.
`c`	When found as the first letter of routine names, `c` indicates the usage of single-precision complex data type.
CBLAS	C interface to the BLAS. See BLAS.
CDF	Cumulative Distribution Function. The function that determines probability distribution for univariate or multivariate random variable X. For univariate distribution the cumulative distribution function is the function of real argument x, which for every x takes a value equal to probability of the event A: $X \le x$. For multivariate distribution the cumulative distribution function is the function of a real vector $x = (x_1, x_2, \ldots, x_n)$, which, for every x, takes a value equal to probability of the event $A = (X_1 \le x_1 \ \& \ X_2 \le x_2, \ \& \ \ldots, \ \& \ X_n \le x_n)$.

Cholesky factorization	Representation of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A in the form $A = U^{H}U$ or $A = LL^{H}$, where L is a lower triangular matrix and U is an upper triangular matrix.
condition number	The number $\kappa(A)$ defined for a given square matrix A as follows: $\kappa(A) = \|\|A\|\| \ \|\|A^{-1}\|\|$.
conjugate matrix	The matrix A^{H} defined for a given general matrix A as follows: $(A^{H})_{ij} = (a_{ji})^{*}$.
conjugate number	The conjugate of a complex number $z = a + bi$ is $z^{*} = a - bi$.
d	When found as the first letter of routine names, d indicates the usage of double-precision real data type.
DFTs	Abbreviation for Discrete Fourier Transforms. See Chapter 11 of this book.
dot product	The number denoted $x \cdot y$ and defined for given vectors x and y as follows: $x \cdot y = \Sigma_{i} \ x_{i}y_{i}$. Here x_{i} and y_{i} stand for the i-th elements of x and y, respectively.
double precision	A floating-point data type. On Intel® processors, this data type allows you to store real numbers x such that $2.23*10^{-308} < \| x \| < 1.79*10^{308}$. For this data type, the machine precision ε is approximately 10^{-15}, which means that double-precision numbers usually contain no more than 15 significant decimal digits.For more information, refer to Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture.
eigenvalue	See eigenvalue problem.
eigenvalue problem	A problem of finding non-zero vectors x and numbers λ (for a given square matrix A) such that $Ax = \lambda x$. Here the numbers λ are called the eigenvalues of the matrix A and the vectors x are called the eigenvectors of the matrix A.
eigenvector	See eigenvalue problem.
elementary reflector(Householder matrix)	Matrix of a general form $H = I - \tau vv^{T}$, where v is a column vector and τ is a scalar. In LAPACK elementary reflectors are used, for example, to represent the matrix Q in the QR factorization (the matrix Q is represented as a product of elementary reflectors).
factorization	Representation of a matrix as a product of matrices. See also Bunch-Kaufman factorization, Cholesky factorization, LU factorization, LQ factorization, QR factorization, Schur factorization.
FFTs	Abbreviation for Fast Fourier Transforms. See Chapter 11 of this book.

full storage — A storage scheme allowing you to store matrices of any kind. A matrix A is stored in a two-dimensional array a, with the matrix element a_{ij} stored in the array element $a(i,j)$.

Hermitian matrix — A square matrix A that is equal to its conjugate matrix A^H. The conjugate A^H is defined as follows: $(A^H)_{ij} = (a_{ji})^*$.

I — See identity matrix.

identity matrix — A square matrix I whose diagonal elements are 1, and off-diagonal elements are 0. For any matrix A, $AI = A$ and $IA = A$.

i.i.d. — Independent Identically Distributed.

in-place — Qualifier of an operation. A function that performs its operation in-place takes its input from an array and returns its output to the same array.

Intel MKL — Abbreviation for Intel® Math Kernel Library.

inverse matrix — The matrix denoted as A^{-1} and defined for a given square matrix A as follows: $AA^{-1} = A^{-1}A = I$. A^{-1} does not exist for singular matrices A.

LQ factorization — Representation of an m-by-n matrix A as $A = LQ$ or $A = (L\ 0)Q$. Here Q is an n-by-n orthogonal (unitary) matrix. For $m \le n$, L is an m-by-m lower triangular matrix with real diagonal elements; for $m > n$,

$$L = \begin{bmatrix} L_1 \\ L_2 \end{bmatrix}$$

where L_1 is an n-by-n lower triangular matrix, and L_2 is a rectangular matrix.

LU factorization — Representation of a general m-by-n matrix A as $A = PLU$, where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$).

machine precision — The number ε determining the precision of the machine representation of real numbers. For Intel® architecture, the machine precision is approximately 10^{-7} for single-precision data, and approximately 10^{-15} for double-precision data. The precision also determines the number of significant decimal digits in the machine representation of real numbers. See also double precision and single precision.

MPI — Message Passing Interface. This standard defines the user interface and functionality for a wide range of message-passing capabilities in parallel computing.

MPICH	A freely available, portable implementation of MPI standard for message-passing libraries.
orthogonal matrix	A real square matrix A whose transpose and inverse are equal, that is, $A^T = A^{-1}$, and therefore $AA^T = A^TA = I$. All eigenvalues of an orthogonal matrix have the absolute value 1.
packed storage	A storage scheme allowing you to store symmetric, Hermitian, or triangular matrices more compactly. The upper or lower triangle of a matrix is packed by columns in a one-dimensional array.
PDF	Probability Density Function. The function that determines probability distribution for univariate or multivariate continuous random variable X. The probability density function $f(x)$ is closely related with the cumulative distribution function $F(x)$. For univariate distribution the relation is $$F(x) = \int_{-\infty}^{x} f(t)dt .$$ For multivariate distribution the relation is $$F(x_1,x_2, \dots ,x_n) = \int_{-\infty}^{x_1}\int_{-\infty}^{x_2}\dots\int_{-\infty}^{x_n} f(t_1,t_2, \dots ,t_n)dt_1dt_2 \dots dt_n .$$
positive-definitematrix	A square matrix A such that $Ax \cdot x > 0$ for any non-zero vector x. Here · denotes the dot product.
pseudorandom number generator	A completely deterministic algorithm that imitates truly random sequences.
QR factorization	Representation of an m-by-n matrix A as $A = QR$, where Q is an m-by-m orthogonal (unitary) matrix, and R is n-by-n upper triangular with real diagonal elements (if $m \geq n$) or trapezoidal (if $m < n$) matrix.
random stream	An abstract source of independent identically distributed random numbers of uniform distribution. In this manual a random stream points to a structure that uniquely defines a random number sequence generated by a basic generator associated with a given random stream.

RNG	Abbreviation for Random Number Generator. In this manual the term "random number generators" stands for pseudorandom number generators, that is, generators based on completely deterministic algorithms imitating truly random sequences.
s	When found as the first letter of routine names, s indicates the usage of single-precision real data type.
ScaLAPACK	Stands for Scalable Linear Algebra PACKage.
Schur factorization	Representation of a square matrix A in the form $A = ZTZ^H$. Here T is an upper quasi-triangular matrix (for complex A, triangular matrix) called the Schur form of A; the matrix Z is orthogonal (for complex A, unitary). Columns of Z are called Schur vectors.
single precision	A floating-point data type. On Intel® processors, this data type allows you to store real numbers x such that $1.18*10^{-38} < \lvert x \rvert < 3.40*10^{38}$. For this data type, the machine precision (ε) is approximately 10^{-7}, which means that single-precision numbers usually contain no more than 7 significant decimal digits. For more information, refer to Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture.
singular matrix	A matrix whose determinant is zero. If A is a singular matrix, the inverse A^{-1} does not exist, and the system of equations $Ax = b$ does not have a unique solution (that is, there exist no solutions or an infinite number of solutions).
singular value	The numbers defined for a given general matrix A as the eigenvalues of the matrix AA^H. See also SVD.
SMP	Abbreviation for Symmetric MultiProcessing. The MKL offers performance gains through parallelism provided by the SMP feature.
sparse BLAS	Routines performing basic vector operations on sparse vectors. Sparse BLAS routines take advantage of vectors' sparsity: they allow you to store only non-zero elements of vectors. See BLAS.
sparse vectors	Vectors in which most of the components are zeros.
storage scheme	The way of storing matrices. See full storage, packed storage, and band storage.
SVD	Abbreviation for Singular Value Decomposition. See also Singular value decomposition section in Chapter 5.
symmetric matrix	A square matrix A such that $a_{ij} = a_{ji}$.
transpose	The transpose of a given matrix A is a matrix A^T such that $(A^T)_{ij} = a_{ji}$ (rows of A become columns of A^T, and columns of A become rows of A^T).

trapezoidal matrix	A matrix A such that $A = (A_1 A_2)$, where A_1 is an upper triangular matrix, A_2 is a rectangular matrix.
triangular matrix	A matrix A is called an upper (lower) triangular matrix if all its subdiagonal elements (superdiagonal elements) are zeros. Thus, for an upper triangular matrix $a_{ij} = 0$ when $i > j$; for a lower triangular matrix $a_{ij} = 0$ when $i < j$.
tridiagonal matrix	A matrix whose non-zero elements are in three diagonals only: the leading diagonal, the first subdiagonal, and the first super-diagonal.
unitary matrix	A complex square matrix A whose conjugate and inverse are equal, that is, that is, $A^H = A^{-1}$, and therefore $AA^H = A^H A = I$. All eigenvalues of a unitary matrix have the absolute value 1.
VML	Abbreviation for Vector Mathematical Library. See Chapter 9 of this book.
VSL	Abbreviation for Vector Statistical Library. See Chapter 10 of this book.
z	When found as the first letter of routine names, z indicates the usage of double-precision complex data type.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z