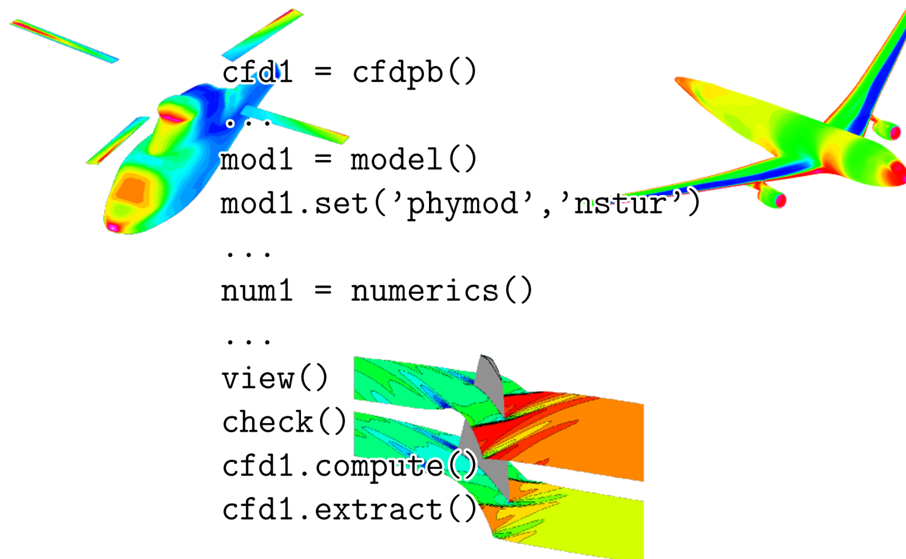
Additional Tools User's Manual



Quality	Author	For the reviewers	Approver
Function			
Name	M. Gazaix	M. Lazareff	L.Cambier
Visa			

Software management : ELSA SCM
Applicability date : immediate
Diffusion : see last page

HISTORY

version edition	DATE	CAUSE and/or NATURE of EVOLUTION
1.0	May 12, 2007	Creation

CONTENTS

Contents 3

1 Introduction 7
- 1.1 Available tools . . . 7

2 elsA_IO 8
- 2.1 Location . . . 8
  - 2.1.1 CVS . . . 8
- 2.2 Environment requirements . . . 8
- 2.3 Description . . . 9
- 2.4 `elsa_io` functions . . . 9
  - 2.4.1 Writing files . . . 9
    - 2.4.1.1 elsa_io.write . . . 9
  - 2.4.2 Reading files . . . 9
    - 2.4.2.1 elsa_io.read . . . 9
    - 2.4.2.2 elsa_io.readAll . . . 10
  - 2.4.3 Other functions . . . 10
    - 2.4.3.1 elsa_io.set_binv3d . . . 10
    - 2.4.3.2 elsa_io.byteorder_v3d . . . 10
    - 2.4.3.3 elsa_io.set_funit . . . 10
- 2.5 Examples . . . 11
  - 2.5.1 Available Python test scripts . . . 11
  - 2.5.2 Writing files . . . 11
  - 2.5.3 Reading files . . . 12

3 Merger 13
- 3.1 Location . . . 13
  - 3.1.1 CVS . . . 13
- 3.2 Environment requirements . . . 13
- 3.3 Description . . . 13
- 3.4 Extraction data definition . . . 14
- 3.5 Load balance step . . . 15
  - 3.5.1 Setting load balance script . . . 16
    - 3.5.1.1 Replace `compute()` with `load_balance(NPROC)` 16
    - 3.5.1.2 Extraction definition . . . 16
    - 3.5.1.3 Parallel script generation . . . 17
    - 3.5.1.4 Input file (Mesh, Flow, Distance, Boundary) Splitting 18
    - 3.5.1.5 Optionally, remove any `submit` . . . 18
  - 3.5.2 run *elsA* (in mono-processor) . . . 18

3.6 Parallel run 19
3.6.1 Without block splitting 19
3.6.2 With block splitting 19
3.7 Merge of extracted data files 19
3.7.1 PYTHONPATH setting 19
3.7.2 merger.run() 20
3.7.3 Additional information 21
3.7.3.1 bin_v3d setting 21
3.7.3.2 Restart files 21

4 transPrepare **22**
4.1 Location 22
4.1.1 CVS 22
4.2 Environment requirements 22
4.3 Description 22
4.4 Principle of application 22
4.4.1 First step 22
4.4.2 Second step: transiMainPy 23
4.5 Good practice 26
4.6 Restrictions 26

5 adim_lib **28**
5.1 Location 28
5.1.1 CVS 28
5.2 Description 28
5.3 Principle of application 28
5.3.1 StateRef class 28
5.3.2 TurRef class 29
5.3.3 AdimRef class 29
5.4 Example 30

6 eplotx **31**
6.1 Location 31
6.1.1 CVS 31
6.2 Environment requirements 31
6.3 Description 31
6.4 Input 31
6.5 eplotx options 33

7 eplot **35**
7.1 Location 35
7.1.1 CVS 35
7.2 Environment requirements 35

7.3 Description 35
7.4 Example 35
7.5 eplot options 35

8 EpMonitor **37**
8.1 Location 37
8.1.1 CVS 37
8.2 Environment requirements 37
8.3 Description 37
8.4 Usage 37
8.5 Parallel computation 40

9 Polar computation : EpFlightPoint **41**
9.1 Location 41
9.1.1 CVS 41
9.2 Description 41
9.3 Usage 41
9.3.1 Updating *elsA* description object data 42
9.4 Example 42

10 Target-lift computation : target_lift **45**
10.1 Location 45
10.2 Environment requirements 45
10.3 Description 45
10.4 Principle of application 45

11 ParelsA **46**
11.1 Location 46
11.1.1 CVS 46
11.2 Description 46
11.3 `ParelsA` description 46
11.4 Use of environment variable `ELSA_MPI_MODULO_PROC` 50

12 elsA_digest **51**
12.1 Location 51
12.1.1 CVS 51
12.2 Environment requirements 51
12.3 Description 51
12.4 A first example 51
12.5 More complicated examples 52
12.5.1 Python `Split` module 52
12.5.2 `transPrepare` 52

13 transformMesh **53**

13.1 Location 53

13.1.1 CVS 53

13.2 Description 53

13.3 Example 53

14 Python module PySplit **55**

14.1 Location 55

14.1.1 CVS 55

14.2 Description 55

Index **56**

1. INTRODUCTION

This document describes several external [1] tools, or utilities, related to *elsA*. Two kinds of tools are available;

- supported tools: support identical to main *elsA* executable support;
- unsupported tools.

1.1 Available tools

- `elsa_io` (*p. 8*);
- `merger` (*p. 13*);
- `transPrepare` (*p. 22*).
- `eplotx` (*p. 31*);
- `eplot` (*p. 35*);
- `EpMonitor` (*p. 37*);
- `EpFlightPoint` (*p. 41*);
- `EpTargetLift` (*p. 45*);
- `ParelsA` (*p. 46*);
- `elsA_digest` (*p. 51*).
- `PySplit` (*p. 55*).
- transformMesh (*p. 53*).

In the following chapters, we give for each tool:

- its location;
- a brief description;
- some usage information;
- an example.

[1] external: not provided through the main *elsA* executable

2. ELSA_IO

2.1 Location

`elsa_io` is located in:
`$ELSADIST/Dist/lib/py/Tools` [1]

2.1.1 CVS

CVS location:
`/data/cvs/app`, CVS module name: `elsA_IO`

2.2 Environment requirements

`elsa_io` requires that Numerical Python is installed (either `numarray`, `Numerics`, or `numpy` (preferred)) [2]. In the following, for notation convenience, we assume `numpy` is used.
`elsa_io` is a wrapper: internally, it imports the C-extension module `_elsa_io` , where `_elsa_io.so` is a shared dynamic library. For multi-platform installation, we have to be careful, since different `_elsa_io.so` have to be installed in separate directories. At run time, to find the "correct" version, there are two options:

1. Option 1 (preferred): set environment variable `ELSADIST` and `ELSAPROD`; Internally, `elsa_io.py` will compute [3] the location of `_elsa_io.so` [4]; for example:

   ```
   export ELSADIST=/home/elsa/Public/v3.2
   export ELSAPROD=bull
   export PYTHONPATH=$ELSADIST/Dist/lib/py
   ```

   Then, at run time, `import Tools.elsa_io` will be successful.

2. Option 2: add explicitly `$ELSADIST/Dist/lib/pt/Tools/$ELSAPROD` to `PYTHONPATH`.

---

[1] On NEC platform, which does not provide shared (".so") module, `elsa_io` is statically linked, so `PYTHONPATH` does not have to be modified.

[2] On ONERA galibier, a Python installation with `numpy` is available:
`export PATH=/opt/python-2.4/bin:$PATH`

[3] by modifiyng `sys.path`

[4] currently: `$ELSADIST/Dist/lib/pt/Tools/$ELSAPROD`

2.3 Description

elsa_io, initiated by F. Cayré (SNECMA), allows to exchange data between *elsA* data files (Voir3D or Tecplot) and numpy arrays, thus allowing powerful data processing using built-in numpy functions. When reading a file, data is converted to numpy arrays; conversely, numpy arrays can be written to files.
elsa_io is currently used by other additional *elsA* tools (merger, transPrepare, py_split).
elsa_io is written in C and Fortran.

2.4 elsa_io functions

2.4.1 *Writing files*

2.4.1.1 *elsa_io.write*

```
Write aero variables or mesh coordinates into a file.

Input arguments:
       - variable names (e.g.: 'ro rou rov row roe')
       - file name
       - file format (See elsA User Manual)
       - a tuple of same-dimension-NumPy arrays

Output: None

Examples:
       elsa_io.write('x y z', 'blocHAm.mai', 'fmt_v3d', (x, y, z))
       elsa_io.write('ro, rou, rov, row, roe', 'blocHAm.ini', 'fmt_v3d',
                     (ro, rou, rov, row, roe))
```

2.4.2 *Reading files*

2.4.2.1 *elsa_io.read*

```
Read aero variables or mesh coordinates from a file.

Input arguments:
       - variable names (e.g.: 'ro rou rov row roe')
       - file name
       - file format (See elsA User Manual)

Output:
       - a tuple of NumPy arrays

Examples:
       (x, y, z) = elsa_io.read('x y z', 'blocHAm.mai', 'fmt_v3d')
       (ro, rou, rov, row, roe) = elsa_io.read('ro rou rov row roe',
                                               'blocHAm.ini', 'fmt_v3d')
```

2.4.2.2 *elsa_io.readAll*

```
Read any variables from a file.

Input arguments:
       - file name
       - file format

Output:
       - a tuple: 1st item = tuple of var names
                  2nd item = tuple of NumPy arrays

Example:
       (var, data) = elsa_io.readAll('blocHAm.mai', 'fmt_v3d')
```

2.4.3 *Other functions*

2.4.3.1 *elsa_io.set_binv3d*

The `set_binv3d` method can be useful to read / write binary files with different integer and /or real data size; it works exactly as `elsA.set_binv3d`.

```
Example:
       elsa_io.set_binv3d('i4', 'r8')
```

2.4.3.2 *elsa_io.byteorder_v3d*

The `byteorder_v3d` method returns the endianness of default bin_v3d files. Example:

```
deimos>python
iPython 2.4.4 (#9, Mar 27 2007, 14:32:49)
[GCC 3.3.3 (SuSE Linux)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> mport elsa_io
>>>
>>> elsa_io. byteorder_v3d()
'big'
>>> import sys
>>> sys.byteorder
'little'
```

2.4.3.3 *elsa_io.set_funit*

The `set_funit` method can be useful to read / write binary files with specific unit numbers :

```
Set logical unit number for Fortran I/O

Input arguments:
       - funit_r: Fortran file in READ  mode (default: 10)
       - funit_w: Fortran file in WRITE mode (default: 11)

Examples:
       elsa_io.set_funit(13, 14)
```

2.5 Examples

2.5.1 Available Python test scripts

Several Python scripts using `elsa_io` are provided
(see `$ELSADIST/Dist/lib/py/Tools/Examples`):

- `test_elsa_io.py`: regression test illustrating most read / write operations.
- `conv_big2little.py`: converting `bin_v3d` files from big to little endian.
- `conv_little2big.py`: converting `bin_v3d` files from little to big endian.

2.5.2 Writing files

Let us give an example:

```
# import numarray as N
import numpy as N
import elsa_io

dimI = 6
dimJ = 4
dimK = 3

x   = N.zeros((dimI,dimJ,dimK), 'd')
y   = N.zeros((dimI,dimJ,dimK), 'd')
z   = N.zeros((dimI,dimJ,dimK), 'd')

ro  = N.zeros((dimI,dimJ,dimK), 'd')
rou = N.zeros((dimI,dimJ,dimK), 'd')
rov = N.zeros((dimI,dimJ,dimK), 'd')
row = N.zeros((dimI,dimJ,dimK), 'd')
roE = N.zeros((dimI,dimJ,dimK), 'd')

for     i in range(dimI):
  for   j in range(dimJ):
    for k in range(dimK):
      x  [i,j,k] = i
      y  [i,j,k] = j
```

```
        z  [i,j,k] = k

        ro [i,j,k] = 1.0
        rou[i,j,k] = 0.0
        rov[i,j,k] = 1.0
        row[i,j,k] = 0.0
        roE[i,j,k] = 0.4

elsa_io.write('x y z', 'xyz.fmt_v3d', 'fmt_v3d', (x, y, z))

elsa_io.write('ro rou rov row roe', 'var.fmt_tp', 'fmt_tp',
              (ro, rou, rov, row, roE))
```

2.5.3 Reading files

```
# import numarray as N
import numpy as N
import elsa_io

x,y,z = elsa_io.read('x y z', 'xyz.fmt_v3d',  'fmt_v3d')
```

In this example, `x`, `y`, and `z` are three-dimensional array objects, filled with data read in previously written file (*cf.* 2.5.2).

3. MERGER

3.1 Location

merger.py is located in :
$ELSADIST/Dist/lib/py/Tools/Merger

3.1.1 CVS

CVS location :
/data/cvs/app, CVS module name : Merger

3.2 Environment requirements

merger internally uses the Python module elsa_io (*p. 8*), which must be correctly installed. Note that elsa_io also requires that Numerical Python is installed (either numarray, Numerics, or numpy (preferred)).

3.3 Description

In parallel computations, it is sometimes necessary to split the blocks defining the original configuration, in order to obtain a good enough load balancing equilibrium. In such a case, extracted data files during the parallel computation correspond to the splitted topology. This means that post-processing may have to be modified, for example rewriting of Tecplot macros. To remove this problem, a Python module, merger, is available. merger, written by E. Raoult (SNECMA) and D.Memmi (ONERA), is able to gather (merge) data computed on the splitted configuration, reverting back to the original topology.
In the following, a description of the necessary steps to use the merger tool is given :

- section 3.4 describes how to define extraction data in a meaningful way for merger;
- for sake of completeness, a brief description of parallel *elsA* computations is also given in section 3.5 and 3.6 [1];
- merger use is described in section 3.7.

[1] for complete information, see *elsA* User's Reference Manual (MU-98057)

3.4 Extraction data definition

To use the merger capability, `extractor` objects must be known during the load balance step; the easiest way (and less error-prone) is to put all the `extractor` definition in a separate sub-script; then import this script during the load balance step, AND during the actual parallel computation. This script is also given as input to the merger utility, thus avoiding completely any translation or duplication errors.

Remarks :

1. Note that extraction objects (`extractor`, `extract` or `extract_group`) defined outside of this script will be ignored by the merger.
2. Some extracted data do not have to be merged, for example :
   - residuals on whole configuration,
   - lift coefficient,
   - flow rate.

   The merger will ignore them, so they can be put in the same extract definition sub-script.

Merging extracted data files is meaningful only for extraction specified in a "block splitting independent" way. Currently, this translates into the following options :

- whole configuration :

  ```
  ext = extractor(name='ext')
  ```

- `family` :

  ```
  ext = extractor(family_code, name='ext')
  ```

- `globwindow` (or `globborder`) :

  ```
  g = globwindow(name='g')
  g.attach('some_window); ...; g.attach('some_bnd')
  ext = extractor('g', name='ext')
  ```

For example :

```
# --------------------
# extractToBeMerged.py
# --------------------
from elsA_user import *
from EpConstant import *

# ----------------------------------------------------------------------
# Field data
# ----------------------------------------------------------------------
extract_M = extractor(name='extract_M')
extract_M.set('title', 'Mach')
extract_M.set('file',  'Wksp/mach.tp')
extract_M.set('format','fmt_tp')
extract_M.set('var',   'xyz mach')
extract_M.set('loc',   'node')

# ----------------------------------------------------------------------
# Boundary data
# Here, You must define Family_Wall (integer)
# For instance~:
Family_Wall = F_WALLADIA  # F_WALLADIA: defined in EpConstant.py
extract_Wall = extractor(Family_Wall, name='extract_Wal')
extract_Wall.set('var',       'xyz psta')
extract_Wall.set('file',      'Wksp/pressure_wall')
extract_Wall.set('format',    'fmt_tp')
extract_Wall.set('title',     'Some Title')
extract_Wall.set('loc',       'node')

# glob_win_wal defined elsewhere (main script or topology sub-script)

extract_Pwall_globwindow = extractor('glob_win_wal', name='extract_Pwall_globwindow')
extract_Pwall_globwindow.set('var',       'xyz psta')
extract_Pwall_globwindow.set('format',    'fmt_tp')
extract_Pwall_globwindow.set('file',      'Wksp/pressure_wall_globwindow.tp')
extract_Pwall_globwindow.set('loc',       'node')

# end script extractToBeMerged.py
```

3.5 Load balance step

Note that the original script can use any variant of *elsA* syntax (Python-user, Python-API, use of `boundary()` or `new_boundary()...`). However, if `boundary` constructors are used (instead of `new_boundary`), and if you plan to use family-defined `extractor` (which is a good idea), then the `<boundary>.family` attribute must be set explicitly :

```
some_boundary.set('family', 14)
```

3.5.1 *Setting load balance script*

In most cases, load balance script is built from the original "compute" script, with only minor modifications. These modifications are described below.

3.5.1.1 *Replace `compute()` with `load_balance(NPROC)`*

Replace :
`my_cfdpb.compute()`
with
`my_cfdpb.load_balance(NPROC)`
where `NPROC` is the number of processors being used during the actual computation.
Note that `<cfdpb>.extract()` has no effect during load balance step : it will be silently ignored, so it is not necessary to comment it out.

3.5.1.2 *Extraction definition*

Do not forget to import the extract sub-script discussed above (*cf.* 3.4); the following line must of course be put before invocation of `<cfdpb>.load_balance` :
`import extractToBeMerged`
If you plan to use `extractor` objects based on a `globwindow` (or equivalently `globborder` [2]), these `globwindow` objects must be defined, for example :

```
glob_win_wal = globwindow(name='glob_win_wal')
glob_win_wal.attach('b1N')
glob_win_wal.attach('b1F')
glob_win_wal.attach('b2N')
glob_win_wal.attach('b2F')
```

Inside `print_script_topology`, the correct definition of `globwindow` objects, taking into account window splitting is done automatically.

Remarks :

1. Definition of `globwindow` cannot be done inside extraction sub-script (to avoid duplicated `globwindow` object definition : one in topology script, one in extract script).

2. Note that in some cases one can reuse pre-existing `globborder` objects (defined for `jtype='`*`nomatch`*`'` or `jtype='`*`nomatch_linem`*`'`).

[2] class `globborder` inherits from class `globwindow`

3.5.1.3 *Parallel script generation*

After `load_balance`, insert code to control parallel script production; two "strategies" are available :

1. If no splitting occurs, the only required output script is the script giving the block allocation to individual processors. So at least the following line must be added :

   ```
   my_cfdpb.print_script_block2proc('example_block2proc_5')  # or any other name
   ```

   If block splitting occurs, the new topology must be written, by calling method `print_script_topology`. The topology written by this method uses construction methods (`new_boundary`, `new_join`, `new_join_nearmatch`, `new_join_nomatch`) ; the `new_boundary` syntax requires definition of `bndphys` objects. Note that `bndphys` objects are independent of block splitting. It is often convenient to gather `bndphys` object definitions in a separate script. This script can be easily written "by hand" since in most cases, a small number of `bndphys` objects have to be defined. It is also possible to use the method `print_script_bndphys_user`. So a typical example would be :

   ```
   my_cfdpb.print_script_topology('example_topo_5')
   my_cfdpb.print_script_bndphys_user('example_bndphys')
   ```

   Remark : Separation of physical versus topological information is not always trivial, so, to avoid potential conversion errors, users are advised to switch to `bndphys` / `new_boundary` boundary definition style, without bothering to use the "old" boundary definition style.

   It is often convenient (see below) to refer to the "driver" (main) Python script :

   ```
   my_cfdpb.print_script_cfd_user('example_cfd')
   ```

2. Instead of calling individual methods (`print_script_*()`) to produce each desired sub-script, one can call a single method, `print_script()`, which will produce all the splitted scripts. The only drawback of this approach is that unnecessary scripts may be generated [3] [4]. A typical example may be :

[3] For example, in most cases, `script_extract.py` (default name) is not really useful, its only purpose is to give a template, to be user-customized.

[4] Note also that for historical reasons, both Python-API and Python-User scripts are generated.

```
nproc = '_5'
# The following 7 'set' are optional;
# if not present, default values are used
my_cfdpb.set('script_cfd',            'example_cfd')
my_cfdpb.set('script_extract',        'extractToBeMerged')
my_cfdpb.set('script_bndphys',        'example_bndphys')
my_cfdpb.set('script_extract_split', 'extractToBeMerged'  + nproc)
my_cfdpb.set('script_topology',       'example_topo'       + nproc)
my_cfdpb.set('script_block',          'example_block'      + nproc)
my_cfdpb.set('script_block2proc',     'example_block2proc' + nproc)
my_cfdpb.print_script()
```

3.5.1.4 *Input file (Mesh, Flow, Distance, Boundary) Splitting*

If block splitting occurs, it is necessary to insert at least the following line :

```
my_cfdpb.split_init_file()
```

All the necessary files will be splitted (mesh, flow, distance and boundary files).
`split_init_file` expects that specific directories (for Mesh, Flow... files) already exist, and will cowardly refuse to overwrite any existing files. So a convenient way to avoid problems may be :

```
my_cfdpb.set('split_mesh_dir',       'Mesh_5')
my_cfdpb.set('split_flow_dir',       'Flow_5')
my_cfdpb.set('split_dist_dir',       'Dist_5')
my_cfdpb.set('split_bnd_dir',        'Bnd_5')
import os
os.system('rm -rf Mesh_5 Flow_5 Dist_5 Bnd_5; mkdir Mesh_5 Flow_5 Dist_5 Bnd_5')
my_cfdpb.split_init_file()
```

3.5.1.5 *Optionally, remove any* `submit`

In some cases, removing any `submit()` explicitly called inside the Python script may reduce the amount of memory needed during load balance algorithm, and can remove subtle inconsistencies (specially in Chimera).
A script devoid of `submit()` calls may be obtained automatically as the `<script>_nosubmit.py` logfile of :

```
elsa -f <script>.py --noexecution --nosubmit --logfile <script>_nosubmit.py
```

3.5.2 *run elsA (in mono-processor)*

When the load balance script is ready, run *elsA*, either with the sequential executable, or with the MPI executable with a single processor.

3.6 Parallel run

3.6.1 Without block splitting

If no block splitting occurs, just insert the following lines before calling `compute` ;

```
my_cfdpb.set('mpi_block2proc', 'example_block2proc_5')
```

3.6.2 With block splitting

If block splitting occurs, the easiest way is to use the driver script produced in step 3, `example_cfd.py`.
If the original script is written using the same philosophy as `example_cfd.py` (import of dedicated sub-script for topology, extraction definition, and motion definition), it may be possible to re-use the original script (see also description of `ParelsA.py`, *p. 46*).

3.7 Merge of extracted data files

3.7.1 PYTHONPATH setting

`PYTHONPATH` must be set with care during the merging step :

- `elsa_io.so` must be found;
- `merger.py` must be found;
- `merger` uses a special "fake" version of `elsA_user`, located in the same directory as `merger`, which must be found *before* the usual version.

So a typical setting of `PYTHONPATH` may be :

```
ELSA_PY=/home/elsa/Public/v3.2.06/Dist/lib/py
export PYTHONPATH=$ELSA_PY/Tools/Merger:\
                  $ELSA_PY/Tools:\
                  $ELSA_PY
```

Note that with such a `PYTHONPATH` setting, a standard *elsA* run does *not* work (since `elsA_user` is different from the standard one). In interactive mode, it is easy to forget which `PYTHONPATH` is used, so it is often a good idea to use different windows, one dedicated for load_balance / compute, one dedicated to the merging task.

3.7.2 *merger.run( )*

The only visible function provided by `merger` is `run`, which has two keyword arguments :
`run(sc1=script_extract, sc2=script_extract_split)`
`merger.run()` needs as input the names (without the '.py' extension) of :

- the original user-defined extraction sub-script; default name is `script_extract`.
- the name of the sub-script (default name : `script_extract_split`) produced during `print_script` (or during `print_script_topology`); if different from default, they must be given as keyword arguments when invoking `run`.

Let us give two examples.

- Example 1 : using default name :

```
python
import merger
merger.run()
```

- Example 2 : using user-defined name :

```
python
import merger
merger.run(sc1='extractToBeMerged', 'extractToBeMerged_split_5')
```

Let us show an example of `merger` output :

```
deimos>python
Python 2.4.3 (#4, Apr 24 2006, 19:27:23)
[GCC 3.2.3 20030502 (Red Hat Linux 3.2.3-47)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import merger
Numeric  not available
numpy  not available
numarray  available !
>>> merger.run(sc1='extractToBeMerged', sc2='extractToBeMerged_split')
    (1) Reading informations: path is ' Wksp/mach.tp ',
        extraction at ' cell ' and file type is ' fmt_tp '
    (2) Father areas Dict {structure is 'father area id : fatherBlock,
        ([i1,i2,j1,j2,k1,k2],[sons])' }:
        {0: [0, [1, 23, 1, 17, 1, 17], [0, 2, 3, 4, 5, 6, 8]],
         1: [1, [1, 12, 1, 9, 1, 9], [1, 7, 9, 10]]}
    (3) Merging ...
```

Merged files should be magically created : the merged file name is obtained from `<extractor>.file` with the addition of a `'.merged'` extension.

3.7.3 *Additional information*

3.7.3.1 bin_v3d *setting*

It is sometimes necessary to use `set_binv3d` to modify default binary "binv3d" treatment; in such situations, `elsa_io` must be imported first, for example :

```
deimos>python
Python 2.4.3 (#4, Apr 24 2006, 19:27:23)
[GCC 3.2.3 20030502 (Red Hat Linux 3.2.3-47)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import elsa_io
>>> elsa_io.set_binv3d('i8', 'r8')
int_form  :'i8'
real_form :'r8'
1
>>> import merger()
...
```

3.7.3.2 Restart files

If you use the automatic creation of restart files (`<cfdpb>.cfd_flow_out_loc` not equal to `'none'`), `extractor` objects are *not* explicitly defined in the user script, so there is no way for the merger to merge restart files. If you really need merging of restart files, you may use two tricks :

1. copy the extract script, let us say to `extractToBeMergedWithRestart.py`, and add an `extractor` definition for restart files :

   ```
   e_flow_out = extractor(name='e_flow_out')
   e_flow_out.set('file', 'restart_file')  # adjust restart_file
   e_flow_out.set('loc', 'cell')
   e_flow_out.set('var', 'conservative reok')
   ```

   Then give `extractToBeMergedWithRestart` as (1st) input argument to `merger.run()`.

2. the second way is to set `cfd_flow_out_loc='none'`, and define explicitly a restart file `extractor` in the extraction sub-script. Now, there is no need to keep two (slightly) different extraction scripts.

4. TRANSPREPARE

4.1 Location

transPrepare is located in:
`$ELSADIST/Dist/lib/py/Tools/TransPrepare`.

4.1.1 CVS

CVS location:
`/data/cvs/app`, CVS module name: `TransPrepare`

4.2 Environment requirements

`elsa_io` module (*cf.* 2) must be available.

4.3 Description

`transPrepare`, written by Robert Houdeville (ONERA/DMAE) is a Python package to interactively prepare the transition-related additional files
(`<boundary>.intermittency_file`, `<boundary>.trans_crit_file`)
for transition computation.

4.4 Principle of application

To explain how the package works, let us consider a problem needing `"trans_crit_file" or "intermittency_file"`. This problem is assumed to be described in the `"elsa_script.py"` file located in the `"home_trans"` directory.
In a first step, the `"elsa_script.py"` is analyzed to get the needed informations, and stored in a `.pickle` python format. In a second step, an interactive process is launched to build the `"trans_crit_file" or "intermittency_file"`.

4.4.1 First step

To analyze the `"elsa_script.py"`, `PYTHONPATH` must be set with care. The alternative version of `elsA_user` must be found first, in any case *before* the standard one. A typical `PYTHONPATH` may be:

```
export PYTHONPATH=$ELSADIST/Dist/lib/py/Tools/TransPrepare:\
                  $ELSADIST/Dist/lib/py
```

Then, run :

```
python elsa_script.py
```

The "`wallDescp.pickle`" file is now available.

4.4.2 *Second step:* `transiMainPy`

First of all, the "`trans_crit.py`"file must be copied in the working directory ("`home_trans`") to be customized. This file is used to set all the default parameters. The objective is now to create step by step a description file, called "`my_contour_file.py`". A default version of this file can be automatically generated to be customized (see below). At the end of the process, this file will contain the description of the various "`intermittency`", "`how`", "`origin`" and "`conta`" fields defined over each wall.
It is important to note that `elsa_io` is used to read the mesh and possibly the pressure fields. This means that `elsa_io` must be in `PYTHONPATH`; "`home_trans`" must also be in `PYTHONPATH` to access to "`trans_crit.py`".
Everything is now ready to run the tool :

```
python -c "import transiMainPy"
```

The first `initialization menu` is displayed :

```
 **** initialization menu ****
    default restart file = ./restart.pickle
0 : use this file
1 : use another file
2 : do not use restart file
enter your choice :
```

The first time you run the tool, there is no restart file, so the only possible response is "`2`". You may answer "`0`" or "`1`" if you have previously saved a `restart` file. Such a file contains all the data, including the wall coordinates, to run the tool. The reading of the `restart` file (in Python "pickle" format) is faster than the reading of the whole mesh. After entering your choice, the mesh and pressure files are read, the Tecplot file named `wallFld.plt` is generated and the second `initialization menu` is displayed :

```
 **** main menu ****
2 : prepare the model of the contour data file :
        ./my_contour_file.py
3 : write a restart file to store temporary state
4 : write elsA transition file(s)
5 : refresh Tecplot with new data from :
        ./my_contour_file.py
6 : finish
enter your choice :
```

After each response, you will be back in this menu. You can store the data in a `restart file` by answering "3". The Tecplot `wallFld.plt` file contains all the needed informations to prepare the transition files :

- `x y z` : coordinates of the nodes of the walls.
- `i3d j3d k3d` : "(i,j,k)" indexing of the nodes with the same convention as in the *elsA* Python description script. This indexing must be used to define the sub-windows in which you want to impose the values of the different fields.
- `pStat` : static pressure.
- `how`, `origin`, `inter`, `conta` : data fields as they will be in the `trans_crit_file`.
- `used` : "1" or "0" depending if the wall is used or not.

The names of the zones in Tecplot correspond to the names of the wall which must be used in `"my_contour_file.py"`. An example of this file including many comments is given below :

```
# example of user data file to prepare the "trans_crit_file" in elsA script

# Definitions :
# -----------
# window      : window as defined in Python elsA api.
# window_name : window name as in the elsA script (and Tecplot files)
#               For "new_boundary", the window name is of the form : w_BndName
# trans_var   : "trans_var" as in the elsa script
# undef_val   : default value for the corresponding field

# For each "trans_var", the contour zones are defined by subwindows (same
# convention [i1,i2,j1,j2,k1,k2] as in elsA, node indices starting at 1),
# and the value for the interior of the subwindow.
# The wall cell faces not included in the sum of the subwindows are
# set to "undef_val"
# The subwindows must not overlap for a given "trans_var" (this is checked).
# The subwindows must be inside the elsA wall windows (this is checked).
# Any of the [i1,i2,j1,j2,k1,k2] values can be replaced by "0" to
# get the actual bounds of the window.
#
# The construction of the present file can be checked step by step with
# transiGlob.py and Tecplot codes. "i,j,k" values and wall names can
# be obtained by "some" clicks in Tecplot.

# Look for "===>readUserFile::printAll" in transi_log.txt to check if the
# file is correctly interpreted

# Description of the command lines to make the "trans_crit_file" files.
# ---------------------------------------------------------------------
```

```
# 1) Selection of the walls to use
#    ------------------------------
#
# Only the selected walls are displayed in Tecplot and used to build
# the "trans_crit_file" files :
#
# select(<window_name_1>)
# select(<window_name_2>)
#     . . . . . . . .
#
# 2) Construction of the "trans_crit_file" files.
#    --------------------------------------------
#
# For each wall boundary, all the "trans_var" fields must be defined by
# giving rectangular subwindows :
#
# addField(<window_name>,<trans_var>,<undef_val>)
# addSubWin(<subwindow_bounds_1>,<trans_var_value>)
# addSubWin(<subwindow_bounds_2>,<trans_var_value>)
#     . . . . . . . .
# <subwindow_bounds> is a list of the form : [i1,i2, j1,j2, k1,k2]
# The subwindows correspond to the window defined by <window_name> until
# a new addField instruction is given.
#
# 3) Special keys
#    ------------
#
# For leading edge contamination criterion, the direction of the leading edge
# from fuselage to wing tip ("trans_l_edge_dir") is required. The information
# is # given using :
#
# setS(<key>,<string>)
#
# with <key> = "l_edge_dir" and <string> in :
# ['none', 'i_plus', 'i_minus', 'j_plus', 'j_minus', 'k_plus', 'k_minus']


# Example for "B-airfoil" in 5 domains
# ------------------------------------

# ------- selection of the walls to use --------------
select("w_wall1")
select("w_wall2")

# -------- window physWall1 ---------------------------
addField("w_wall1", "how", 2)
addSubWin([105, 130,  1, 1, 0, 0], 0)

addField("w_wall1", "origin", 0)
addSubWin([120, 121,  1, 1, 0, 0], 0)
```

```
# -------- window physWall0 -------------------------
addField("w_wall2", "how", 3)
addSubWin([1, 46,  0, 0, 0, 0], 1)
```

You can get a model of this file, by answering "2" in the second main menu. Be careful, if the file exists, it will be overwritten.
When everything seems correct, answering "4" will generate the `trans_pyth_script.py` file to insert in the *elsA* Python script and the `trans_crit_file` (or `intermittency_file`) to use for the computation.

4.5 Good practice

- The first Tecplot file is generated with all the walls of the configuration. Use the names of the zones to select the walls over which you want to build the fields ;
- Use as few walls as possible to prepare the fields. Only the selected wall will appear on the screen. For complex geometries, this allows to work separately with flat, flap or main wing body ;
- The construction of the fields can be done step by step using the refresh command to check the correctness of the data ;
- In case of problems, lines starting with "!!!" are displayed ;
- Many information are written in the `transi_log.txt` file. This file may be useful for debugging ;
- The pressure field may be used to locate the attachment lines (maximum of pressure). The origins of the transition lines must be on both sides of this maximum[1] ;
- The origins of the transition lines must correspond to "how" equal to `0` (laminar imposed) ;
- The direction of the transition lines must be in the flow direction as soon as "how" is equal to `2` or `3` (and therefore not necessarily where "how" is equal to `0`).

4.6 Restrictions

Remarks :

1. Presently, meshes must be given in `format='fmt_v3d'` or `format='fmt_tp'`.

[1]The automatic generation of the "origin" field from the pressure field is in progress.

2. Automatic coarsening (`<cfdpb>.coarsen_mesh`) may not work correctly.
3. The automatic generation of the "origin" field from the pressure field is in progress.

5. ADIM_LIB

5.1 Location

`adim_lib` is located in ;
`$ELSADIST/Dist/lib/py/Tools`.

5.1.1 CVS

CVS location ;
`/data/cvs/ker`, CVS module name ; `api`

5.2 Description

`adim_lib`, written by Robert Houdeville (ONERA/DMAE) is a Python module providing classes and methods to define a reference flow state from various combinations of pressure, density and Reynolds number. Some types of non-dimensionalization are coded which give most of the non-dimensional values and coefficients needed for an *elsA* computation.

5.3 Principle of application

5.3.1 `StateRef` class

`StateRef` defines the reference state of the flow using three independent quantities. The available combinations are presently :

key	used values
"M0_T0_Runit"	$M_\infty, T_\infty, R_{L_{Ref}} = \rho_\infty U_\infty L_{Ref}/\mu_\infty$
"PI_TI_Runit"	$P_i, T_i, R_{L_{Ref}}$
"M0_P0_T0"	$M_\infty, P_\infty, T_\infty$

The methods of the class are :

`printPhysConst` : print the values of the physical constants which are used.
`printFlowRefSI` : print the flow conditions in SI units.
`setPhysConst` : change the value of a given physical constant.
`getPhysConst` : get the dictionary of the physical constants.

The physical constants are :

"Gamma"	C_p/C_v	1.4	
"Pgp"	R	287.053	$m^2s^{-2}K^{-1}$
"Csuth"	S Sutherland	110.4	K
"Tsuth"	T_{ref} Sutherland	273.0	K
"Musuth"	S_{ref} Sutherland	$1.711\,10^{-5}$	$Kgm^{-1}s^{-1}$

5.3.2 TurRef *class*

TurRef class is useful to compute the turbulent quantities which must be given in the *elsA* script. At the creation, the following quantities must be given :

stateRef : StateRef instance
<turbMod> : turbulence model, see below
<Tu> : turbulence level (absolute value, not in percent)
<muRatio> : μ_t/μ

The available turbulence models (<turbMod> value) are :

"smith" : Smith $k-L$
"kepsjl", "chien" "kepsls" : $k-\varepsilon$ Jones-Launder, Chien, Launder-Sharma
"komega_wilcox" : $k-\omega$ Wilcox
"komega_menter" : $k-\omega$ Menter
"komega_kok" : $k-\omega$ Kok
"spalart" : $\tilde{\nu}$ Spalart-Allmaras model
"knut_spalart" : $k-\tilde{\nu}$ Spalart-Allmaras model
"kkl" : $k-kL$ Aupoix-Bézard model
"kphi" : $k-\varphi$ Aupoix model

5.3.3 AdimRef *class*

AdimRef class computes most of the non-dimensional quantities required in the *elsA* script. At the creation, the following quantities must be given :

stateRef : StateRef instance
turRef : TurRef instance
<adim_choice> : choice of the set of reference quantities
<Lref> : reference length (in meter) used in the mesh.

The choices of the set of reference quantities are the following :

`"RVT"`	: $\rho_\infty, V_\infty, T_\infty$
`"PVT"`	: $P_\infty, V_\infty, T_\infty$
`"PRT"`	: $P_\infty, \rho_\infty, T_\infty$
`"RAT"`	: $\rho_\infty, a_\infty, T_\infty$
`"RATI"`	: ρ_i, a_i, T_i
`"SI"`	: SI units

The methods of the class are :

`printRefSI`	: print the reference quantities used for the non-dimensionalization in SI units
`printRefAdim`	: print the non-dimensional reference quantities used for the non-dimensionalization
`getConservative`	: get the conservative quantities
`getTurVar`	: get the turbulent quantities
`getViscosity`	: get Sutherland constants

The arguments of the `getConservative` method are the two α and β angles and a key the value of which may be `"xz"` or `"xy"` to compute the velocity components according to :

$$\begin{aligned}
\text{"xz"} &: \left\{ \begin{array}{lcl} \rho u &=& \rho|U|\cos(\beta)\cos(\alpha) \\ \rho v &=& -\rho|U|\sin(\beta) \\ \rho w &=& \rho|U|\cos(\alpha)\sin(\beta) \end{array} \right. \\
\text{"xy"} &: \left\{ \begin{array}{lcl} \rho u &=& \rho|U|\cos(\beta)\cos(\alpha) \\ \rho v &=& \rho|U|\cos(\alpha)\sin(\beta) \\ \rho w &=& \rho|U|\sin(\beta) \end{array} \right.
\end{aligned} \tag{5.1}$$

5.4 Example

The file `$ELSADIST/Dist/lib/py/Tools/adim_info.py` is an example of application. This Python script may be copied, customized and run to have a better understanding of the library.

6. EPLOTX

6.1 Location

`eplotx` is located in :
`$ELSADIST/Dist/lib/py/Eplot`.

6.1.1 CVS

CVS location :
`/data/cvs/app`, CVS module name : `eplot`

6.2 Environment requirements

xmgrace must be installed.

6.3 Description

`eplotx`, written by Jean-Philippe Boin and Florian Blanc (CERFACS) is a Python script providing a simple interface to xmgrace , a public domain 2D plotting tool [1]. `eplotx` can be useful if Tecplot is not available [2]. An example of graphics generated by xmgrace is given in Figure 6.1.

6.4 Input

`eplotx` requires as input several Tecplot files :

- `resil1_all.tp` : residual (norm L_∞), whole configuration;
- `resil2_all.tp` : residual (norm L_2), whole configuration;
- `resicoeff*.tp` : integration of numerical flux over aerodynamic body surface;

These Tecplot files are generally produced at the end of some *elsA* run, for example :

```
# See Examples/eplotx_extract.py

ext_residl_1 = extractor('name='ext_residl_1')
ext_residl_1.set('norm',   NORM_L1)
ext_residl_1.set('file',   'resil1_all.tp')
ext_residl_1.set('format', 'fmt_tp')
```

[1] http://www.idris.fr/data/cours/visu/xmgrace/xmgrace.html
[2] for example in the context of University CFD formation

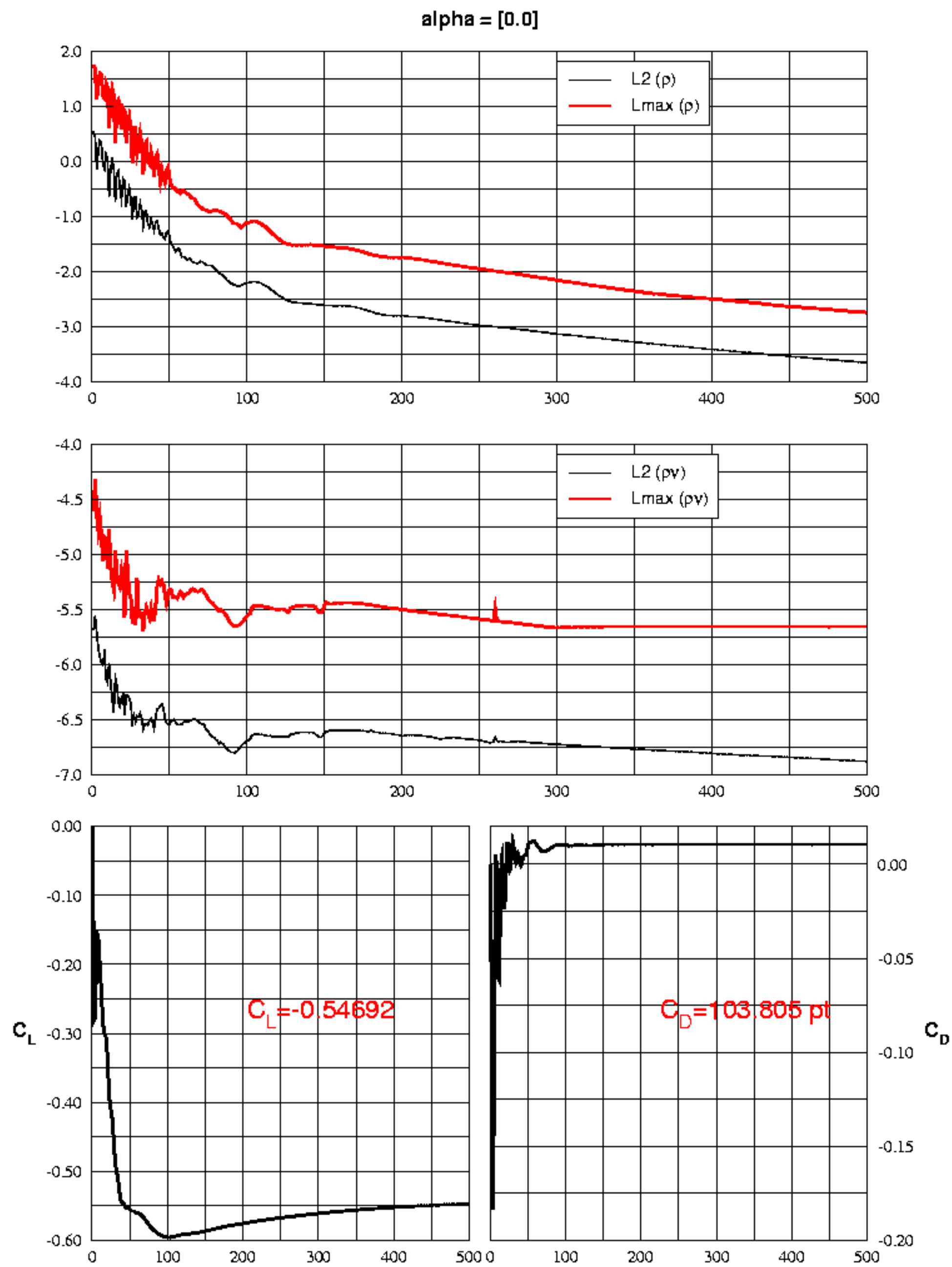


Figure 6.1: Example of `eplotx` output

```
ext_residl_1.set('var',    'residual_ro residual_roturl')

ext_residl_2 = extractor('name='ext_residl_2')
ext_residl_1.set('norm',   NORM_L2)
ext_residl_2.set('file',   'resil2_all.tp')
ext_residl_2.set('format', 'fmt_tp')
ext_resiext_residar',    'residual_ro residual_roturl')

ext_extrados_flux = extractor(F_Body, name='ext_extrados_flux')
ext_extrados_flux.set('loc',    'interface')
ext_extrados_flux.set('file',   'resicoeff_e.tp')
ext_residl_1.set('format', 'fmt_tp')
ext_extrados_flux.set('var',    'flux_rou flux_rov flux_row')
# Optional additional settings
ext_extrados_flux.set('fluxcoeff', coeff)
ext_extrados_flux.set('pinf',      pinf)

ext_extrados_flux = extractor(F_Body, name='ext_extrados_flux')
ext_extrados_flux.set('loc',    'interface')
ext_extrados_flux.set('file',   'resicoeff_e.tp')
ext_resiext_residormat', 'fmt_tp')
ext_extrados_flux.set('var',    'flux_rou flux_rov flux_row')
# Optional additional settings
ext_extrados_flux.set('fluxcoeff', coeff)
ext_extrados_flux.set('pinf',      pinf)

ext_intrados_flux = extractor(F_Body, name='ext_intrados_flux')
ext_intrados_flux.set('loc',    'interface')
ext_intrados_flux.set('file',   'resicoeff_i.tp')
ext_residl_1.set('format', 'fmt_tp')
ext_intrados_flux.set('var',    'flux_rou flux_rov flux_row')
# Optional additional settings
ext_intrados_flux.set('fluxcoeff', coeff)
ext_intrados_flux.set('pinf',      pinf)


# Optional extraction period
ext_residl_1.set('period', Period)
ext_residl_2.set('period', Period)
ext_extrados.set('period', Period)
ext_intrados.set('period', Period)
```

At runtime, the environment variable `EPLOT_DIR` is used to find xmgrace template files (in directory `$EPLOT_DIR/Xmgr_Template`).
Suggested setting :
`export EPLOT_DIR=$ELSADIST/Dist/lib/py/Tools`

6.5 eplotx options

`eplotx` options may be obtained with the '`-h`' option :

```
# eplotx : elsA direct residual visualisation.
# Version 1.0 - CERFACS 2006.

Usage : eplotx {option} launch in /RESIDUAL
        --> runs residual plots with Xmgrace
Options:
          -alpha AoA1 (AoA2...) (deg)
          -sref SREF
          -ref reference directory
          -free (full page)
          -y (CL=Cy)
          -h display this help
Need :
      SET THE PATH ON THE FIRST LINE TO YOUR CURRENT PTYHON PATH
      xmgrace damas residual script directory for eplotx:
      ENV VARIABLE : EPLOT_DIR
      $EPLOT_DIR/Xmgr_Template/
Examples :
      eplotx -alpha 2.0 -ref ../RESIDUAL_Ref
```

7. EPLOT

7.1 Location

`eplot` is located in :
`$ELSADIST/Dist/lib/py/Tools`.

7.1.1 CVS

CVS location :
`/data/cvs/app`, module name : `eplot`

7.2 Environment requirements

Tkinter must be installed.

7.3 Description

eplot , written by Jean-Philippe Boin and Florian Blanc (CERFACS), allows graphics display (with Tkinter) of residuals from *elsA* standard output.

7.4 Example

```
cat elsA_MPI_Pid_4196028_N_0
...
 -------------------------------------------------------------------
#dq  1 L2 = 5.3277084e-02 Linf = 1.0319864e+00  (  72  11   1 )    1
#dq  2 L2 = 1.4853927e-03 Linf = 3.5052698e-02  (  29  20   1 )    1
#dq  3 L2 = 1.2916068e-02 Linf = 3.5123805e-01  (  71  11   2 )    1
#dq  4 L2 = 2.7317121e-02 Linf = 3.3014163e-01  (  77  12   2 )    1
#dq  5 L2 = 1.3319253e-01 Linf = 2.5798143e+00  (  72  11   1 )    1
 -------------------------------------------------------------------
...
```

Then running :
`eplot.py elsA_MPI_Pid_4196028_N_0`
should open a new window (see Figure 7.1).

7.5 eplot options

`eplot` options may be obtained with the '`-h`' option :

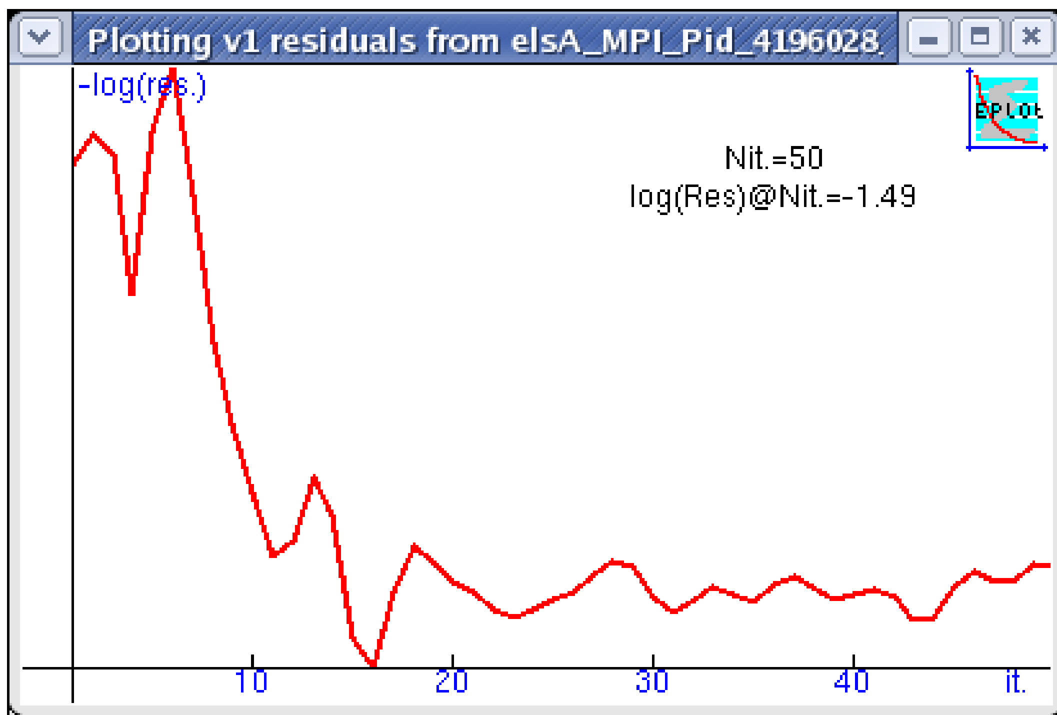


Figure 7.1: Example of `eplot` screenshot

```
eplot -h
################################################
# eplot : elsA direct residual visualisation. #
# Version 1.0 - CERFACS 2006.                  #
################################################
Usage : eplot {option} filename
     --> runs dynamical residual plots
Options:
     -var varnumber : number of the variable to plot (default 1)
     -dts : enables dts mode
     -h     : display this help
Example :
      eplot -var 6 elsa.out
```

8. EPMONITOR

8.1 Location

`EpMonitor` is located in :
`$ELSADIST/Dist/lib/py/Monitoring`.

8.1.1 CVS

CVS location :
`/data/cvs/app`, CVS module name : `Monitor`

8.2 Environment requirements

Python extension module `Tkinter` must be installed.
xmgrace must be installed.

8.3 Description

`EpMonitor`, written by J.P. Boin and Y. Colin (CERFACS), allows to perform on-line monitoring of global data such as convergence residuals, lift or drag. `EpMonitor` is based on the visualization tool xmgrace and graphic interface Tkinter .

8.4 Usage

Let us give a simple example in which, for each iteration, residual and aerodynamic coefficients are plotted with xmgrace . These global data are obtained from the *elsA* kernel with dedicated methods [1] :

- `<extractor>.getLiftAero` [2];
- `<extractor>.getDragAero` [3];
- `<cfdpb>.getL2Residual`.

The `cfl` number is explicitly computed inside the Python script [4], and communicated to the *elsA* kernel by calling the `<cfdpb>.updateKernelCFL` method.
For each iteration, the *elsA* solver is called (`<conf>.advanceInnerLoop`). Then residual, aerodynamic coefficients and flight parameters are sent to monitor and

[1] `getLiftAero` and `getDragAero` are also used by the TargetLift tool.
[2] or `<extract_group>.getLiftAero`
[3] or `<extract_group>.getDragAero`
[4] instead of using `cfl_fct` hard-coded variation

controller objects (`<GraceMonitor>.update` and `<TkinterControl>.update`). The controller returns control integer values, depending on action on buttons; in this example, these control values are used to modify the `cfl` number ('CFL +' and 'CFL -' buttons). Another button allows to stop the computation [5]. The plots are refreshed every `REFRESH_GRAPHICS_PERIOD` iteration. (see also Figure 8.1).

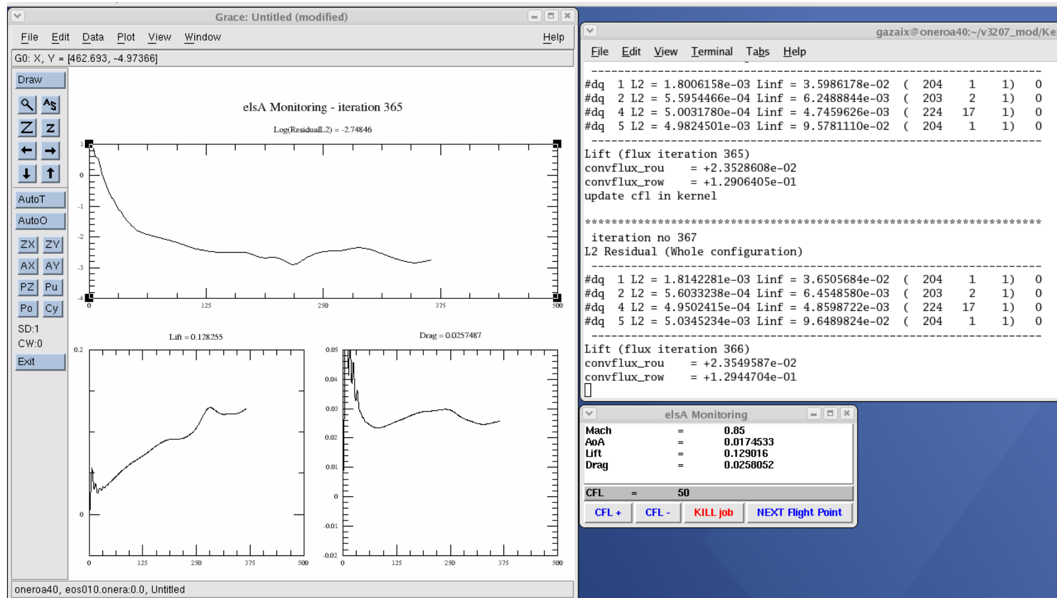


Figure 8.1: Example of `EpMonitor` usage

```
# See $ELSADIST/lib/py/Monitoring/Examples/monitor_naca.py
...
from elsA_user import *
conf = cfdpb(name='conf')
...
# insert problem definition here (geometry, numerics, model...)

MONITOR = 1
if MONITOR:

  # ------------------------------------
  # MONITORING
  # ------------------------------------
  from EpMonitor import *
  NITER     = 500
  monitor = GraceMonitor(itmax=NITER, resmin=-3)
  control = TkinterControl()
  REFRESH_GRAPHICS_PERIOD = 5
```

[5] Note that 'Kill job' action differs from a standard kill command ; indeed the time loop is interrupted, but all post-processing and extraction are performed.

```
  CFL1 =  1.0
  CFL2 = 20.0
  ITER2 = 50
  DCFL0 = CFL2 / float(ITER2)
  DCFL = DCFL0
  cfl = CFL1

  # -------------------------------------
  # TIME LOOP
  # -------------------------------------
  iter_tot = 0

  conf.preCompute()
  for i in range (1,NITER+1):
    iter_tot += 1
    conf.advanceInnerLoop()
    #
    # extraction res, lift and drag for monitoring
    lift = naca_extract.extract_lift.getLiftAero(alpha)
    drag = naca_extract.extract_lift.getDragAero(alpha)
    res  = conf.getL2Residual()
    # monitoring
    status,controler = control.update(Mach,alpha,lift,drag,cfl)
    monitor.update(iter_tot,res,lift,drag)
    # on the fly CFL control
    if cfl >= CFL2 or cfl < CFL1:
      DCFL = 0.0
    if controler[0] != 0:
      DCFL += (DCFL0*0.5)*float(controler[0])

    if i > 1:
      cfl += DCFL
      cfl = min(max(CFL1,cfl),CFL2)
      num.set('cfl', cfl)
      conf.updateKernelCFL()

    if i % REFRESH_GRAPHICS_PERIOD == 0:
      monitor.trace()

    if status < 1:
      break

  conf.posCompute()
  conf.extract()

  del monitor
  del control


else:
  conf.compute()
```

```
conf.extract()
```

8.5 Parallel computation

The monitor module works also in parallel MPI. All actions are performed by processor zero, selected with `get_proc`. To run correctly, the integer control values have to be shared between all processors. This is done with the method `broadcast_i` [6].

[6] `broadcast_i` is basically a wrapper of MPI function `MPI_Bcast(int)`

9. POLAR COMPUTATION : EPFLIGHTPOINT

9.1 Location

`EpFlightPoint` is located in :
`$ELSADIST/Dist/lib/py/Monitoring`.

9.1.1 CVS

CVS location :
`/data/cvs/app`, module name : `Monitor`

9.2 Description

`EpFlightPoint`, written by J.P. Boin and Y. Colin (CERFACS), can be useful to perform polar computations in a single *elsA* script; `EpFlightPoint` is able to generate each quantity needed for the initialization of a new flight point, taking into account different normalization options, thus allowing to update infinite state definition (class `FlightPointState`) and stop criteria (class `FlightPointCriteria`).

9.3 Usage

One must give the flight conditions :

- Mach number;
- angle of attack;
- Reynolds number;
- Normalization choice.

`FlightPointState` updates all *elsA* description objects (`model`, `numerics`, `state`, `boundary`). This task is already done with the Airbus tools EDM from EGAT2.0. Here, the purpose is to have a similar module which can be used directly inside the *elsA* script. Thus, it makes it possible to perform several computations with different conditions (flight points) in a single driver script. Furthermore this module allows to change the current normalization.
The module is used inserting the following lines :

```
from EpFlightPoint import *
FPstate = FlightPointState()
# flight point definition: Python dictionary
newFP = {'mach':0.72, 'alpha':2.0, 'adim':'srvt'}
FPstate.update(newFP)
```

First the previous values are read from description objects and stored in a dictionary. Then, this dictionary is updated with new values given by user. All quantities are calculated with respect to the normalization and description objects are updated.

9.3.1 *Updating* elsA *description object data*

First all the quantities are calculated setting a normalization and using generic formula. Then the *elsA* objects are updated if necessary. The turbulent model is up to now restrained to a maximum of two transport equations. Only Spalart, $k - Wilcox$ and $k - Smith$ models are treated.

9.4 Example

An example of polar computation using `EpFlightPoint` and `EpMonitor` is given below (see also Figure 9.1).

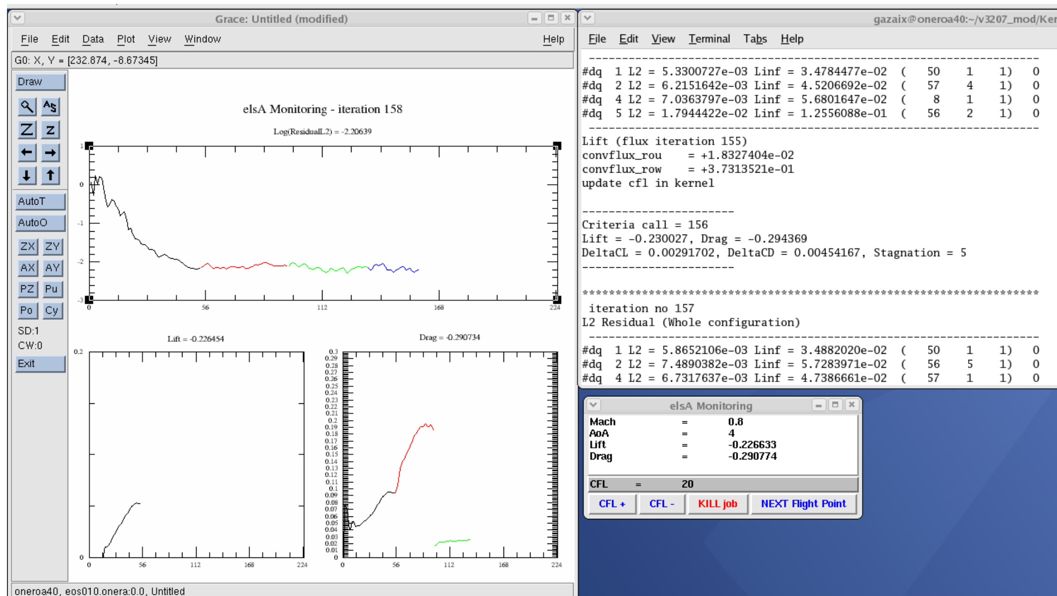


Figure 9.1: Example of `EpFlightPoint` usage

```
# See $ELSADIST/lib/py/Monitoring/Examples/monitor_naca_polar.py
...
# --------------------------------------
# FLIGHT POINT MODULE
# --------------------------------------
from EpFlightPoint import *
FPstate = FlightPointState(sref=162.0)
FPconv  = FlightPointCriteria()
```

```
# -------------------------------------
# GRAPHICS MONITORING
# -------------------------------------
from EpMonitor import *
NITER    = 150
monitor = GraceMonitor(itmax=NITER,
                       resmin=-3,  resmax=1,
                       liftmin=0., liftmax=.2,
                       dragmin=0., dragmax=.3)
                       control = TkinterControl()
control = TkinterControl()
REFRESH_GRAPHICS_PERIOD = 5

# -------------------------------------
# FLIGHT POINT LOOP
# -------------------------------------
NITER = 150
Alpha=[1.0,2.0,3.0,4.0]
NPOINTS = len(Alpha)
Mach   = 0.8
QLIFT = 0.01
QDRAG = 0.01
STAG_ITER = 10


CFL1 =  1.0
CFL2 = 20.0
ITER2 = 57   # ???
DCFL0 = CFL2 / float(ITER2)
DCFL = DCFL0
cfl = CFL1

iter_tot = 0

# --------------------------------------
# 'state' dummy initialisation
# FlightPointState.update() to be improved
# --------------------------------------
stateInf = DesState('stateInf')
stateInf.setF('ro',  1.)
stateInf.setF('rou', 1.)
stateInf.setF('rov', 0.)
stateInf.setF('row', 0.)
stateInf.setF('roe', 1.)

# -------------------------------------
# ANGLE OF ATTACK LOOP
# -------------------------------------
for point in range (1,NPOINTS+1):
  FP = {'mach':Mach,'alpha':Alpha[point-1],'rey':1.413e6}
```

```
    FPstate.update(FP)
    FPconv.reset()
    num.set('inititer',iter_tot+1)
    DCFL=DCFL0

    # ------------------------------------
    # TIME LOOP
    # ------------------------------------
    conf.preCompute()
    for i in range (1,NITER+1):
       iter_tot += 1
       conf.advanceInnerLoop()

      # extraction res, lift and drag for monitoring convergence
      lift = naca_extract.extract_lift.getLiftAero(Alpha[point-1])
      drag = naca_extract.extract_lift.getDragAero(Alpha[point-1])
      res  = conf.getL2Residual()
      # monitoring
      status,controler = control.update(Mach,Alpha[point-1],lift,drag,cfl)
      monitor.update(iter_tot,res,lift,drag,point-1)

      # on the fly CFL control
      if cfl >= CFL2 or cfl < CFL1:
        DCFL = 0.0
        if controler[0] != 0:
          DCFL += (DCFL0*0.5)*float(controler[0])

        if i > 1:
          cfl += DCFL
          cfl = min(max(CFL1,cfl),CFL2)
          num.set('cfl', cfl)
          conf.updateKernelCFL()

      if i % REFRESH_GRAPHICS_PERIOD == 0:
        monitor.trace()

      if FPconv.eval(lift,drag,QLIFT,QDRAG,STAG_ITER) or status < 1:
        break


      conf.posCompute()
      conf.extract()
      conf.set('reuse','active')
      # interruption
      if status == -1:
        break
      else:
        control.reset()

  del monitor
  del control
```

10. TARGET-LIFT COMPUTATION : TARGET_LIFT

10.1 Location

`target_lift` is a Python–*elsA* description class providing a simple interface to target lift computations.
Its implementation is in `py/Products/TargetLift`, mainly in the `EpTargetlift.py` Python module file.

10.2 Environment requirements

As `target_lift`is part of the Python–*elsA* user interface, it is automatically available.
It is presently documented in the *elsA* User's Reference Manual, `/ELSA/MU-98057`, as an example of Python-level programming for interested users.

10.3 Description

This tool provides a lift → incidence solver, for a specified point or a collection of specified points.

10.4 Principle of application

An outer Newton iteration is wrapped around the standard *elsA* iteration, for which the `compute()` call is automatically managed.

11. PARELSA

11.1 Location

`ParelsA.py` is located in :
`$ELSADIST/Dist/lib/py`

11.1.1 CVS

CVS location :
`/data/cvs/ker`, CVS Module name : `api`

11.2 Description

In the context of *elsA* industrialization, in which the end-user does not have to deal with parallel specific tuning, it may be convenient to use the `ParelsA` Python module (available in standard *elsA* distribution), or to write a similar tool. The idea is to put invariant data in "invariant" scripts :

- main (driver) script) ;
- physical boundary data (for ex. `bndphys.py`) ;
- extraction definition is done in a separate script.

Conversely, data depending on load balancing (i.e. depending on block splitting) is put in separate scripts :

- topology ;
- block-specific data [1] ;
- allocation of block to processors.

11.3 `ParelsA` description

In order to use `ParelsA`, the main (driver) script must contain the main numerical choices, and imports the other sub-scripts ; this driver is used both in load_balance mode and in in parallel execution mode ; in this way, there is no file manipulation, thus greatly reducing risk of errors. The trick is to choose the name of the non-invariant scripts in a consistent way, depending on the number of processors, such that the driver can "guess" the actual script name, so avoiding any editing of script by users. Control of execution is done through environment variable `ELSA_MODE` and command line option '`-p`' :

[1] such as motion definition

1. First step : load_balance mode :

   ```
   export ELSA_MODE=load_balance~; elsA.x driver,py -p NPROC
   ```

   `<cfdpb>.load_balance((NPROC)` is called, splitted files are produced (`ParelsA` manages directory creation), and required parallel scripts are automatically produced, with names computed (by `ParelsA`) such that the main driver script will be able to import them correctly in the parallel computing step.

2. Second step : compute mode :

   ```
   export ELSA_MODE=compute; mpirun -np NPROC elsA.x driver,py -p NPROC
   ```

Let us sketch the driver main script :

```
# ---------------------------------------------------------
# driver.py
# ---------------------------------------------------------

from elsA_user import *
from EpConstant import *

import ParelsA

dic = ParelsA.getData('Mesh', flow_dir='Flow_ini', script_topo='topology', \
                       script_block='block', script_proc='proc')
MPI_Nproc               = dic['MPI_Nproc']
NPROC                   = dic['NPROC']
ELSA_MODE               = dic['ELSA_MODE']
script_topo             = dic['script_topo_i']
script_proc             = dic['script_proc_i']
script_block            = dic['script_block_i']
script_topo_l           = dic['script_topo_l']
script_proc_l           = dic['script_proc_l']
script_block_l          = dic['script_block_l']
script_extract_split_l = dic['script_extract_split_l']
mesh_dir_i              = dic['mesh_dir_i']
flow_dir_i              = dic['flow_dir_i']
dist_dir_i              = dic['dist_dir_i']
bnd_dir_i               = dic['bnd_dir_i']
mesh_dir_l              = dic['mesh_dir_l']
flow_dir_l              = dic['flow_dir_l']
dist_dir_l              = dic['dist_dir_l']
bnd_dir_l               = dic['bnd_dir_l']


# =====
# cfdpb
# =====
```

```
conf = cfdpb(name='conf')

conf.set_nb_error_max(200)
conf.set('cpu_max', 82944.0)
conf.set_binv3d('i4', 'r8')

conf.set('config', '3d')
conf.set_block_creation_mode('automatic')
conf.set('automatic_block_gen', 'db_directory')

# ..........................
# Global ('cfd_') definition
# ..........................

conf.set('cfd_mesh_dir',        mesh_dir_i)
conf.set('cfd_flow_ini_dir',    flow_dir_i)

# conf.set('cfd_init_state', 'sta1')

# Here we choose to use a specific 'dedicated' restart extractor
# ==> merging of restart file easier
conf.set('cfd_flow_out_loc', 'none')


conf.set('cfd_flow_ini_var',    'conservative')
conf.set('cfd_coeffmutinit', 1.000000000000e-01)

# Walldistance initialization: from file
conf.set('cfd_walldist_ini',      'active')
conf.set('cfd_walldist_ini_dir',  dist_dir_i)
conf.set('cfd_walldist_ini_file', 'dist')


# =====
# Model
# =====
FluidModel = model(name='FluidModel')
...

# ========
# Numerics
# ========
NumericalScheme = numerics(name='NumericalScheme')
...


#========
# import  (bndphys, topology)
#========
import bndphys
```

```
print 'importing %s' %script_topo
__import__(script_topo)

__import__(script_block)

#==========================================
# Block allocation to processors (// MPI)
#==========================================

if get_nb_proc() > 1:
  # (Script script_block2proc.py requires exactly 8 proc)
  # (see also environment variable 'ELSA_MPI_MODULO_PROC' and/or
  #  attribute 'MPI_MODULO_PROC')
  conf.set('mpi_block2proc', script_proc_l)
#------------------------------------------


# Post-processing
import jam_kl_rotwall_extract


#========================
# Compute / Load_balance
#========================

if ELSA_MODE == 'load_balance':
  conf.set('script_topology',              script_topo_l)
  conf.set('script_block2proc',            script_proc_l)
  conf.set('script_extract_split',         script_extract_split_l)
  conf.set('split_mesh_dir',               mesh_dir_l)
  conf.set('split_flow_dir',               flow_dir_l)
  conf.set('split_walldist_dir',           dist_dir_l)
  conf.set('split_bnd_file_dir',           bnd_dir_l)
  conf.set('split_bnd_nsrotwall_omg_dir',  bnd_dir_l)

  conf.load_balance(NPROC)

  conf.split_init_file()

  conf.print_script_topology  (script_topo_l)
  conf.print_script_block2proc(script_proc_l)
  conf.print_script_block_user(script_block_l)


else:
  conf.compute()
  conf.extract()

# End script file driver.py
```

11.4 Use of environment variable ELSA_MPI_MODULO_PROC

In many cases, it may be convenient to use ELSA_MPI_MODULO_PROC environment variable. ELSA_MPI_MODULO_PROC allows to use scripts computed (during load balance step) with load_balance(NPROC_lb) with, at run time, a **lower** number of processors, NPROC_rt. The only restriction is :

```
NPROC_rt =  NPROC_lb / ELSA_MPI_MODULO_PROC
```

This can be useful in two ways :

- Different block splittings lead to different convergence behaviours, since *elsA* implicit algorithms work on a "block-implicit" basis. Conversely, using a single block splitting (obtained with the highest planned NPROC) insures identical results for different NPROC_rt ; so depending on the number of processors available at run time (varying platform load for example), different NPROC_rt can be selected, without impairing reproducibility.
- In some cases, asking for load_balance(NPROC_lb) whith NPROC_lb higher than the planned NPROC_rt can lead to a better load balancing.

12. ELSA_DIGEST

12.1 Location

`elsA_digest` is located in:
`$ELSADIST/Dist/lib/py/Tools`.

12.1.1 CVS

CVS location:
`/data/cvs/ker`, CVS module name: `api`

12.2 Environment requirements

Python interpreter must be available.

12.3 Description

`elsA_digest` may be used to get some kind of information from existing *elsA* Python scripts. The idea is to overload *elsA* description classes, in order to achieve a specific purpose, different from the standard *elsA* way.
`elsA_digest` can be seen as a "common denominator", factorizing common code which can be used in different contexts, and thus removing the burden of parsing *elsA* syntax [1].

Remark : Instead of using `elsA_digest`, it is also possible to start from the original Python interface Python–*elsA* (`elsA_user...`) and extend / specialize it. This approach is powerful, but requires at least a basic understanding of Python–*elsA*; in many situations, use of `elsA_digest` may be more convenient.

12.4 A first example

Let us assume that for some reason, we must find all the `boundary` objects of type `walladia`. This can be done easily by writing a "fake" `elsA_user`, which, after importing `elsA_digest`, redefines (overloads) `compute`:

[1] For example, parsing *elsA* scripts to get the configuration topology requires a treatment of `DesBoundary`, `boundary`, `new_boundary`, `new_join`, `new_join_nomatch`..., which can be quite complex. An alternative, relying of auxiliary 'log' file (`script_log.py`) is not very robust, since it breaks down if the log-file format changes.

```
# ----------------------
# elsA_user.py
# ----------------------
from elsA_digest import *
# Redefinition class DesCfdPb   ------------------------------------
class DesCfdPb(DesCfdPb):
  def compute(self):
    storeList = []
    print "\n===>compute :\n"
    for bnd in dict_boundary.values():
      if bnd._type  == 'walladia':
        print '<boundary> walladia found: %s' %bnd._name
        storeList.append(bnd)

# --- End script elsA_user.py ---
```

12.5 More complicated examples

12.5.1 Python `Split` module

A prototype load balancing tool, `Split`, written in Python, is in development. `Split` uses `elsA_digest` to get the configuration data (topology, mesh files. . .).

12.5.2 `transPrepare`

`transPrepare` (*p. 22*) also uses `elsA_digest` internally.

13. TRANSFORMMESH

13.1 Location

13.1.1 CVS

`/data/cvs/app`, CVS module: `transformMesh`

13.2 Description

The `transformMesh` tool provides transformations (translation, rotation, symmetry, homothety…) and restrictions on mesh files.

13.3 Example

```
#================================================================================
# File example for TransformMesh module use
#================================================================================
from elsA_user import *
from EpConstant import *
from MeshTransform import *
from math import *

rae2822Chim = cfdpb(name='RAE')

#--------------------------------------------------------------------
# MESH & BLOCKS
#--------------------------------------------------------------------
mshRae = mesh(name='mshRae')
mshRae.set('file','/data/Public/DCFD/rae2822/2dom/rae2822-211x40-xy.mai')
mshRae.set('format','fmt_v3d')
mshRae.submit()

blkRae = block(name='blkRae')
blkRae.attach(mshRae)
blkRae.submit()

mshCart = mesh(name='mshCart')
mshCart.set('file','/data/Public/DCFD/rae2822/1dom/rae2822-cart-149x129x2-xy.mai')
mshCart.set('format','fmt_v3d')
mshCart.submit()

blkCart = block(name='blkCart')
blkCart.attach(mshCart)
blkCart.submit()

#--------------------------------------------------------------------
# TRANSFORMATION
```

```
#------------------------------------------------------------------

mshRae_API = mshRae._bind

save(mshRae_API,"mshRae.tp","fmt_tp")

subzone(mshRae_API,(1,1,1),(211,40,1))
save(mshRae,"mshRae_subzone.tp","fmt_tp")

oneovern(mshRae_API,(2,2,1))
save(mshRae_API,"mshRae_oneover2.tp","fmt_tp")

homothety(mshRae_API,(0.,0.,0.),0.5)
save(mshRae_API,"mshRae_homothety.tp","fmt_tp")

translate(mshRae_API,(0.,1.,0.))
save(mshRae_API,"mshRae_translate.tp","fmt_tp")

rotate(mshRae_API,(0.,1.,0.),(0.,0.,1.),1.57)
save(mshRae_API,"mshRae_rotate.tp","fmt_tp")

symetrize(mshRae_API,(0.,0.4,0.),(1,0,0),(0,0,1))
save(mshRae_API,"mshRae_symetrize.tp","fmt_tp")
```

Remark : The above example is transitional : references to `mshRae_API` should be replaced by `mshRae` in future versions.

14. PYTHON MODULE PYSPLIT

14.1 Location

14.1.1 CVS

`/data/cvs/app`, CVS module: `PySplit`

14.2 Description

`py_split` is a preliminary rewriting in Python of the internal C++ kernel Split module is available.

INDEX

_elsa_io, 8
_elsa_io.so, 8
MPI, 40
Python, 51
Split, 55
TargetLift, 37
Tecplot, 9, 13, 23, 24, 26, 31
Tkinter, 35, 37
Voir3D, 9
array, 12
bin_v3d, 10, 21
eplot, 35
polar, 41
refresh, 26
transPrepare, 22
transformMesh, 7
xmgrace, 31, 33, 37
-h (command-line option), *33*, *35*
-p (command-line option), *46*

adim_lib (Python module), 28
AdimRef, 4, 29
advanceInnerLoop, 37

bndphys, 17
boundary, 15, 41, 51
broadcast_i, 40
byteorder_v3d, 10

cfd_flow_out_loc (cfdpb.), *21*
cfl (numerics.), *37, 38*
cfl_fct (numerics.), *37*
coarsen_mesh (cfdpb.), *27*
command-line option, 33, 35, 46
compute, 19, 51

DesBoundary, 51

elsA_digest (Python module), 7, 51, 52
elsa_io (Python module), 7–9, 11, 13, 21–23
elsa_io.py, 8
elsa_io.read, 9
elsa_io.readAll, 10
elsa_io.so (Python module), 19
elsa_io.write, 9
ELSA_MODE (environment variable), *46*
ELSA_MPI_MODULO_PROC (environment variable), *5*, *50*
elsA_user (Python module), 19, 22, 51
ELSADIST (environment variable), *8*
ELSAPROD (environment variable), *8*
environment variable, 4, 5, 8, 19, 22, 23, 33, 46, 50
EpFlightPoint (Python module), 7, 41, 42
eplot (Python module), 7, 35, 36
EPLOT_DIR (environment variable), *33*
eplotx (Python module), 7, 31–33
EpMonitor (Python module), 7, 37, 38, 42
EpTargetLift (Python module), 7
extract, 14, 16
extract_group, 14
extractor, 14–16, 21

family (boundary.), *14, 15*
file (extractor.), *20*
FlightPointCriteria, 41
FlightPointState, 41
FlightPointState (Python module), 41
fmt_tp (extractor.format.), 26
fmt_v3d (extractor.format.), 26
format (extractor.), *26*

get_proc, 40
getDragAero, 37
getL2Residual, 37
getLiftAero, 37
globborder, 14, 16
globwindow, 14, 16

intermittency_file (boundary.), *22*

jtype (boundary.), *16*

load_balance, 16, 17

merger (Python module), 7, 9, 13, 19, 20
model, 41
monitoring, 37

new_boundary, 15, 17, 51
new_join, 17, 51
new_join_nearmatch, 17
new_join_nomatch, 17, 51
nomatch (boundary . jtype .), 16
nomatch_linem (boundary . jtype .), 16
none (cfdpb . cfd_flow_out_loc .), 21
numarray (Python module), 8, 13
Numerics (Python module), 8, 13
numerics, 41
numpy (Python module), 8, 9, 13

on-line monitoring, 37

ParelsA (Python module), 7, 46, 47
ParelsA.py (Python module), 19
pickle, 22
print_script, 17, 20
print_script_bndphys_user, 17
print_script_topology, 16, 17, 20
py_split (Python module), 9, 55
PySplit (Python module), 7
Python>–elsA, 45, 51
PYTHONPATH (environment variable), *4*, *8*, *19*, *22*, *23*

run, 20

set_binv3d, 10, 21
set_funit, 10
Split (Python module), 52
split_init_file, 18
state, 41
StateRef, 4, 28

target_lift (Python module), 45
Tkinter (Python module), 37
trans_crit_file (boundary .), *22*
transformMesh (Python module), 53
transiMainPy (Python module), 23
transPrepare (Python module), 7, 9, 22, 52
TurRef, 4, 29

update, 38
updateKernelCFL, 37

Empty page

DIFFUSION SCHEME

Software Secretariat Archives

elsA team

At users' request

END of LIST