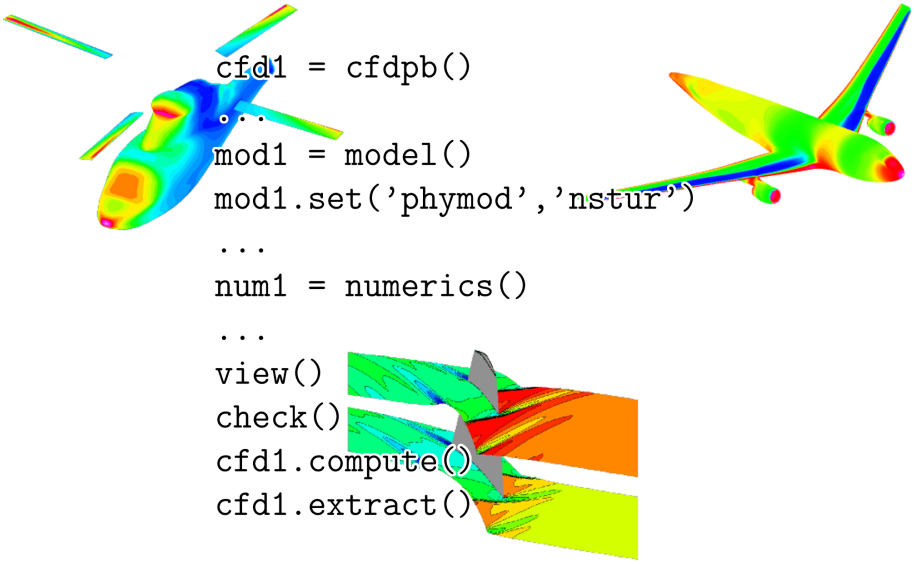
Development Process Tutorial



Quality	Author	For the reviewers	Approver
Function	Integration manager	Quality manager	Project head
Name	M. Gazaix	A.M. Vuillot	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	Sept 30, 2003	Creation
1.1	Oct 1, 2003	Delivery for v3.0
1.2	Jan 5, 2006	Delivery for v3.1
1.3	Mar 10, 2006	Minor correction of delivery for v3.1 before posting on elsA web
1.4	June 20, 2007	A. Gazaix-Jollès corrections
1.4	July 11, 2007	Delivery for v3.2

CONTENTS

1. INTRODUCTION

1.1 Document purpose

The purpose of this document is to help *elsA* developers. The document focuses on implementation level. Several aspects are covered:

- installation (chapter 2);
- porting *elsA* to a new computing environment (chapter 3);
- getting started (chapter 4);
- kernel development process (chapter 5).

This document is available through *elsA* Web site :
`http://elsa.onera.fr/ExternDocs/user/MDEV-03036.pdf`.
A companion document, "*elsA* Design and Implementation Tutorial",
`http://elsa.onera.fr/ExternDocs/user/MDEV-06001.pdf`,
is available to help developers in the understanding of the main features of *elsA* design.
See also the UML (Unified Modeling Language) documentation :
`https://elsa.onera.fr/ExternDocs/dev/uml/elsaDocUml_v3.0.01.html`.
Most *elsA* developers have to perform their own computations using *elsA*; *elsA* usage is not described in this document; instead consult :

- *elsA* User's Starting Guide :
  `http://elsa.onera.fr/ExternDocs/user/MU-03037.pdf`;
- *elsA* User's Reference Manual :
  `http://elsa.onera.fr/ExternDocs/user/MU-98057.pdf`.

1.2 *elsA* versions

Since the beginning of the *elsA* project in 1997, the major *elsA* reference versions, named deliveries, have been:

- v0 : September 1998
- v1.1 : January 2000
- v2.0 : November 2000
- v2.1 : June 2001
- v2.2 : May 2002

- v3.0 : December 2003
- v3.1 : December 2005
- v3.2 : July 2007 (planned)

Between deliveries, other *elsA* reference versions ("releases") are available, mainly for internal use.
A reference version is identified with a CVS tag[1]. The contents of each reference version can be consulted (in French for old releases) at *elsA* development page : `http://elsa.onera.fr/elsA/news/newsdev.html`.

1.3 *elsA* statistics

In this section, we present some statistics for *elsA* release 3.2[2] :

- 850 000 lines (600 000 lines if comments are removed)
- 726 C++ classes;
- 800 C++ header files (`.h`);
- 772 C++ implementation files (`.C`);
- approximately 1800 Fortran files (more than 1500 Fortran77 subroutines, and 300 F90 subroutines).

[1] for example `I3207c`
[2] statistics for the *elsA* kernel computed with `cp_line` (`api/Py/Tools`)

2. HOW TO INSTALL *elsA*

2.1 Building from source on Unix

2.1.1 Tools required

To build *elsA* from source, you need the following tools :

1. a C++ compiler;
2. a Fortran compiler;
3. Python correctly installed.

2.1.1.1 C++ and Fortran compilers

Table 2.1 gives information about *elsA* portability. If you have to port *elsA* to a new computing environment, see chapter 3, *p. 30*.

Remarks :

1. IA32 architecture : INTEL Pentium, AMD Opteron.
2. IA64 architecture : INTEL Itanium 2.
3. NEC SX: SX6, SX8.
4. `shared` =`yes`: shared library version available and tested.
5. `Ael` module cannot be compiled with `g77`. You can either switch to `g95` (`http://g95.sourceforge.net/`), or remove `Ael` module (see section 2.1.3.1 to configure *elsA* without `Ael`).

2.1.1.2 Python

Python (*www.python.org*) is used to build the *elsA* scripting interface : without Python, you will not be able to build and run the *elsA* executable; note however that most of the internal *elsA* libraries do **not** require Python (see section 2.1.9, *p. 21*), so that it should be possible to build and run unitary test cases, written entirely in C++, and possibly Fortran. Note that a complete Python installation is necessary (include header file `Python.h` and Python library, for example `libpython2.4.a` or `libpython2.4.so` must be available). If Python is not installed correctly on your machine, it is fairly easy to install (see section 2.1.1.3). *elsA* has been successively built with several versions of Python[1]. Presently, we strongly suggest to use version 2.2 or higher. To choose which Python version to use :

[1] beginning with version 1.5.2

Platform	**OS**	C++	**Fortran**	ELSAPROD	shared
IA32	GNU/Linux (SuSE, RedHat)	gcc (2.95.2, 3.2, 3.3,3.4)	g77	linux	yes
			g95	linuxg95	
		INTEL icc	INTEL ifort	intelIA32	yes
		PGI pgCC	PGI pgf90	pgi	yes
IA32	Windows (with Cygwin)	gcc	g77	linux	no
IA32em (x86_64)	GNU/Linux	INTEL icc	INTEL ifort	intelIA32em	yes
IA64 (Itanium)	HP-UX	aCC	f90	itanium	yes
	Linux (SGI Altix)	INTEL icc	INTEL ifort	intelIA64	yes
	Linux (BULL)	INTEL icc	INTEL ifort	bull	yes
	Linux (HP)	INTEL icc	ifort		yes
	Linux (NEC front-end)	gcc	g77	gnuIA64	no
SGI MIPS	IRIX	CC	f90	sgi	yes
SUN SPARC	Solaris	CC	f90	sun	no
HP PA-RISC	HP-UX	aCC	f90	hp	yes
IBM Power4-5	AIX	xlC	xlf	ibm	no
IBM PowerPC	MAC OS	xlC	xlf	macosx	no
HP Alpha	OSF	cxx	f90	dec	yes
Apple Mac	MacOS	gcc	g77	macos	no
NEC SX	SUPER-UX	sxc++	sxf90	nec	no
Fujitsu VPP		CC	frt	fuji	no

Table 2.1: *elsA* portability

- Building *elsA* with Python 1.5 is still working, but deprecated; moreover, some useful Python modules, most notably `numpy` [2], require newer Python version.
- *elsA* built with Python 2.2 has been widely tested, including NEC SX and Fujitsu VPP vector supercomputers;
- With Python 2.2 and 2.4, on NEC SX, and possibly on other platforms (HP-UX), it may be better to build Python with thread disabled. To do that, you must run the Python configure script with `-without_thread` option. As an other possibility, try to compile *elsA* source code with a '`enable multi-threading`' option, such as `-mt` on HP-UX.

2.1.1.3 Installing Python

If you have to install Python, take the following steps :

1. Download source from Python Web site (`http://www.python.org`).
2. Choose a working directory, let us say `Build_python`; this directory can be destroyed at the end of build process.
3. Choose where Python will be installed at the end of the installation process, let us assume `MY_PYTHON`[3]; `MY_PYTHON` is given to Python configure script through '`-prefix`' option ; Python executable will be installed in `$MY_PYTHON/bin`.
4. On some platforms (SGI IRIX, HP-UX, IBM AIX, SUN OS...), you will have to choose between 32- or 64-bit version; to do that, you may have to set some environment variable (`SGI_ABI` for IRIX, `OBJECT_MODE` for AIX), or you may choose to set the environment variable `CC` (`export CC='cc -64'`). Of course, *elsA* and Python should use the same choice! When possible, we suggest to use 64-bit, since it will provide the ability to compute very large problems;
5. Enter the following commands :

```
cd Build_python
gunzip Python-2.4.4.tgz; tar xvf Python-2.4.4.tar; cd Python-2.4
./configure --prefix=MY_PYTHON
make
make install
```

Now, to check that the installation is correct, the Python interpreter can be invoked :

```
export PATH=MY_PYTHON/bin:$PATH
python
```

[2] `numpy` is a re-implementation of older Python array modules, `Numeric` and `numarray`.
[3] for example `MY_PYTHON = $HOME/local`, or `/usr/local/elsA`

The output should look like to :

```
Python 2.4.4 (#5, Feb 12 2007, 11:31:02)
[GCC 3.4.4 20050721 (Red Hat 3.4.4-2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

2.1.1.4 Optional tools

- To build MPI parallel executable, MPI C header (`mpi.h`) and library (`libmpi`[4]) must be correctly installed;
- Some additional libraries are sometimes useful for Python additional modules (but never required, see section 2.5.3.4) :
  - `readline` library
  - `expat` library.
- **For developers only**: if you plan to **modify** *elsA* C++ <–> Python interface, you may have to install SWIG :
  `http://www.swig.org`
  (see section 2.3, *p. 23*).

2.1.2 How to get elsA source code

If you are eligible to get *elsA* source code, you can obtain it either through a standard Unix tarball, or through CVS. In both cases, you must first choose a working directory, let us say `WorkDir`, with at least 50 MegaBytes available[5]. You must set environment variable `ELSAWKSP`:

```
export ELSAWKSP=WorkDir
```

2.1.2.1 Unix tarball

You may ask an archive from elsa-infodev@onera.fr. Unpack the archive :

```
cd $ELSAWKSP   # Enter elsA top directory
gunzip elsA-$VERSION.src.tar.gz
tar xf elsA-$VERSION.src.tar
```

[4] or `libmpich`
[5] The amount of disk space required to build *elsA* is platform-dependent.

2.1.2.2 *From* CVS *repository*

If you plan to do some development work inside *elsA*, you should use CVS to access directly to the *elsA* CVS repository. To extract the *elsA* source code, you need to be registered, let us say `some_user`. Then, change into a valid directory [6], and enter the following command :

```
cvs -d:pserver:some_user@elsa.onera.fr:/data/cvs/ker co -P Kernel
```

Remarks :

1. *elsA* is designed with a modular approach, which allows to compile some of its components independently. When starting a new development, it is often a good idea to check out only the components to be modified, for example `Tur` if one plans to add a new turbulence model. For additional information related to this situation, see 2.2, *p. 22*.
2. In addition to the kernel source, CVS is used to manage the test base (unitary, non regression and validation), and the documentation (Theoretical Manual, Developer's Guide, User's Manual, Technical Notes). See also section 5.1, *p. 51*, and /ELSA/MU-00069[7] for complete *elsA* CVS repository information.

2.1.3 *elsA build*

2.1.3.1 *Selection of modules*

Several *elsA* modules are optional :

- Aeroelasticity module : `Ael`;
- Linearized RANS module : `Lur`[8];
- Shape optimization module : `Opt`;
- Load balance module : `Split`.

You can select which modules will be included in two ways :

- edit configuration file `DefConfig.h` (`$ELSAWKSP/Kernel/cfg/DefConfig.h`); use `#define` (`#undef`) to select (remove) modules;
- alternatively, you can use the `config` Makefile target[9] :

```
cd $ELSAWKSP/Kernel
make -f cfg/Makefile.mk config
```

[6] This directory corresponds to environment variable `ELSAWKSP`.

[7] "User's Guide to CVS and production on *elsA* project" `http://elsa.onera.fr/ExternDocs/user/MU-00069.ps.gz`

[8] `Lur` cannot be compiled if `Ael` is not selected.

[9] see also `make help_config`

2.1.3.2 *Build process*

Perform the following steps [10] :

1. Assuming `ELSAWKSP` is correctly set, enter into *elsA* root directory :
   `cd $ELSAWKSP/Kernel`

2. You must inform *elsA* Makefile system where the Python interpreter is located. To do that, you must set three environment variables[11] :
   `E_PPREFIX`, `E_PPREFIX1` and `E_PYVERSION`.

   ```
   # Example: Python executable is located in /usr/local/bin
   export E_PPREFIX=/usr/local
   export E_PPREFIX1=/usr/local
   export E_PYVERSION=2.2
   ```

3. Set the environment variable `ELSAPROD` : `ELSAPROD` must be a concatenation of a prefix, `PLATFORM`, and one or several optional suffixes. The allowed values for `PLATFORM` are (see Table 2.1, *p. 9*) :

   (a) `sgi` : SGI (IRIX);

   (b) `dec` : HP Alpha processor (Digital UX);

   (c) `hp` : HP-UX with PA-RISC architecture;

   (d) `itanium` : HP-UX with IA64 (Itanium) architecture;

   (e) `linux` (GNU/Linux, g++/g77 compilers); currently, gcc version 2.95 is known to work; to use gcc 3.2, you will have to use the line :

   ```
   E_CC=g++ -D_ELSA_COMPILER_GCC32_ \
   -DE_SCALAR_COMPUTER -D_E_USE_STANDARD_IOSTREAM_ -DE_RTTI
   ```

   in file `$ELSAWKSP/Kernel/cfg/prods/Make_linux.mk`.

   (f) `pgi` (GNU/Linux, PGI compilers);

   (g) `intelIA32` (GNU/Linux, INTEL IA32 architecture, INTEL compilers);

   (h) `intelIA32em` (GNU/Linux, INTEL IA32 extended memory architecture (64-bit), INTEL compilers);

   (i) `intelIA64` (GNU/Linux, INTEL IA64 architecture, INTEL compilers);

[10] In the following, we assume that you are running Korn shell (`SHELL = ksh`). If you use another shell, please modify accordingly the following shell instructions (if you have a `csh` -like shell, use `setenv` instead of `export`)

[11] These three environment variables are pre-defined for a small number of computers in file `Make_paths.mk`; local *elsA* administrator can decide to add the local hostname in `Make_paths.mk`. Note that `Make_paths.mk` takes precedence over externally defined `E_PPREFIX`, `E_PPREFIX1` and `E_PYVERSION`.

(j) `nec` : NEC SX (SX6, SX8; in that case, cross compilation is used (IRIX or GNU/Linux));

(k) `fuji` : Fujitsu VPP 700/3000;

(l) `ibm` : IBM Power4-5 (AIX);

(m) `powerpc` : IBM PowerPC (MAC OS);

(n) `macosx` : Apple (MAC OS);

(o) `sun` (Solaris);

(p) `cray` (CRAY SV1).

The allowed suffixes are (note that some suffixes are not meaningful on all platforms) :

(a) `i4` : integers use 4 bytes;

(b) `i8` : integers use 8 bytes; this is default on NEC;

(c) `r4` : floating point numbers use 4 bytes (sinple precision);

(d) `r8` : floating point numbers use 8 bytes (double precision); this is default on all platforms;

(e) `mpi` : MPI parallel mode. In MPI mode, we have tried to provide default values for location of MPI header (`mpi.h`) and MPI library; however, you may have to change these default values; to do that, edit the file `$ELSAWKSP/Kernel/cfg/prods/Make_$PLATFORM.mk`, and reset `E_MPIPATH_I` and `E_MPIPATH_L` :

```
#ifdef ___MPI
  E_MPIPATH_I = some_include_absolute_path
  E_MPIPATH_L = some_library_absolute_path
#endif
```

On SGI platform, you can choose between native MPI and MPICH (assuming that both are correctly installed), without any file manipulation[12]. To use MPICH instead of native MPI, you must set the environment variable `MPICH_ROOT`; for example :

```
export MPICH_ROOT=/usr/local/contrib/MPICH_DP1.2.2
```

If `MPICH_ROOT` is not set, *elsA* will try to use native MPI instead.

(f) `dbg` : `DEBUG` mode (much slower at run time!);

(g) `so` : shared (dynamically linked) libraries [13],[14];

[12] MPICH is also available on other platforms as well (Linux, SUN).

[13] not available for NEC SX

[14] please note that, for `ELSAPROD=intelIA64` and `ELSAPROD=sgi`, shared library is currently the default choice, (so, it is not necessary to specify explicitly the `so` suffix)

(h) `n64` : 64-bit addressing mode (available for AIX, HP-UX and IRIX);

(i) `n32` : 32-bit addressing mode (available for AIX, HP-UX and IRIX).

Examples of valid `ELSAPROD` :

- `export ELSAPROD=sgi_n32`
- `export ELSAPROD=hp_r4_mpi`
- `export ELSAPROD=ibm_dbg`

4. Now, you can actually build the *elsA* executable, starting from the 'master' Makefile :

   - `make -f cfg/Makefile.mk elsa`
     This command does several things :

     (a) it creates a symbolic link between `cfg/Makefile.mk` and `./makefile`;

     (b) in each sub-directory, or module (`Agt`, `Blk`...), it builds a local Makefile (called `Makefile.$ELSAPROD`); if necessary, you can re-build these local Makefiles through the command : `make depall`[15].

     (c) compilation and library build is performed; this can also be done through the command : `make sysall`. A library is created in each module, for example : `$ELSAWKSP/Kernel/src/Agt/.Obj/$ELSAPROD/libeAgt.a`; there is also a symbolic link in `$ELSAWKSP/Kernel/lib/$ELSAPROD` :

     ```
     bassgi07 [242] pwd
     /beasgi8a/mgazaix/v3207/Kernel/lib/sgi
     bassgi07 [243] ls -l libeTur.a
     lrwx------  libeTur.a -> ../../src/Tur/.Obj/sgi/libeTur.a
     ```

     Note that it is possible to build each individual library separately; for example : `cd $ELSAWKSP/Kernel/src/Agt; make sys`

     (d) link is performed in the special module `Api` (there is **no** `libeApi.a`); the executable file name is : `$ELSAWKSP/Kernel/src/Api/.Obj/Wrapper/$ELSAPROD/elsA.x`

     If everything goes as expected, after a few minutes, you should have the following messages :

     ```
     +++ Elsa    :  Add link to makefile
     +++ Elsa    :  Making public directories
     +++ Elsa    :   Add dir and links Fact
     +++ Elsa    :   Add dir and links Blk
     ...
     +++ Elsa    :  Done
     ```

[15] it is also possible to re-build only one local makefile, for example : `cd $ELSAWKSP/Kernel/src/Agt; make dep`

```
+++ Elsa    :  +++ Stage 1
+++ Elsa    :  Making all Makefiles
+++ Elsa    :   Generate makefile from scratch into Api
+++ Make    :      Platform flags for sgi
+++ Make    :      Api/Makefile.sgi
+++ Elsa    :   Generate makefile from scratch into Fact
+++ Make    :      Platform flags for sgi
+++ Make    :      Fact/Makefile.sgi
...
+++ Elsa    :  +++ Stage 2
+++ Elsa    :  Making all objects
+++ Elsa    :   Making into Api
+++ Elsa    :    This target is not applicable for Api
+++ Elsa    :    DO NOT WORRY ABOUT FOLLOWING EXIT...
*** Error code 1 (bu21)
+++ Elsa    :   Making into Fact
+++ Elsa    :    Compiling C++ .Obj/sgi/Base/FactDataBase.C
+++ Elsa    :    Compiling C++ .Obj/sgi/Base/FactBase.C
...
+++ Elsa    :  +++ Stage 3
+++ Elsa    :  Making API
+++ Elsa    :   Making into Api
+++ Elsa    :     Producing elsA.i
+++ Elsa    :    Copying headers...
+++ Elsa    :    Ok for elsA.i and headers
+++ Elsa    :    Pre-processing (by sed) Wrapper/DesBase.h
...
No Wrap generation (SWIG NOT used)
embed.i : Using Python 2.4
+++ Elsa    :    Compiling C++ .Obj/sgi/Wrapper/elsA_wrap.C
+++ Elsa    :    Link .Obj/sgi/Wrapper/elsA_wrap.x
+++ Elsa    :    Symlink exec to  .Obj/sgi_r8/elsa.x
+++ Elsa    :    Done
```

elsA build is now ended. The next section explains how to perform some simple tests.

If the *elsA* executable `$ELSAWKSP/Kernel/src/Api/.Obj/$ELSAPROD/Wrapper/elsA.x` is not built, then consult section 2.5.1, *p. 25*.

2.1.4 First runs

You are now able to launch the `elsA.x` executable, which is located in the directory : `$ELSAWKSP/Kernel/src/Api/.Obj/$ELSAPROD/Wrapper`.

Please note that if you use a shared executable, you must modify the environment variable `LD_LIBRARY_PATH`[16] :

```
export LD_LIBRARY_PATH=$ELSAWKSP/Kernel/lib/$ELSAPROD:$LD_LIBRARY_PATH
```

2.1.4.1 First test

The first test just launch the *elsA* interpreter, without any script file :

```
ksh > ./elsA
# ================================================================================
```

[16] or something equivalent on your system such as `LD_LIBRARY64_PATH`

```
# elsA v3.2.01d - Copyright (c) 1997-2006 by ONERA
# (IDDN.FR.001.370031.001.S.P.2001.000.10000)
# Built with Python library v2.3

-------------------------------------
| Additional Modules:               |
-------------------------------------
| Module Ael   : Aeroelasticity     |
| Module Opt   : Shape Optimization |
| Module Split : Load balancing     |
-------------------------------------

Size of Float   : 8 Bytes
Size of Integer : 8 Bytes

# ================================================================================
Python 2.3.4 (#4, Jan 24 2005, 11:16:41) [C] on irix6-64
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The Python banner and prompt should appear after some specific *elsA* build data (*elsA* version, Copyright, compiler options, precision mode). To quit the Python session, just type `'CTRL-D'`.

2.1.4.2 Second test

This test checks that `elsA.py` Python module is correctly installed. You have to set correctly the environment variable `PYTHONPATH` :

```
export PYTHONPATH=$ELSAWKSP/Kernel/api/Py
```

Then, exactly as in the first test, launch the *elsA* interpreter interactively, and, inside the interpreter loop, type (without any leading blanks!):

```
from elsA import *
dir()
```

You should see something similar to :

```
>>> dir()
['AbsoluteFrame', 'AbsoluteVel', 'Deformable', 'DesBase', 'DesBasePtr', 'DesBlock',
'DesBlockPtr', 'DesBndPhys', 'DesBndPhysPtr', 'DesBoundary', 'DesBoundaryPtr',
'DesCfdPb', 'DesCfdPbPtr', 'DesExtract', 'DesExtractGroup', 'DesExtractGroupPtr',
'DesExtractPtr', 'DesExtractor', 'DesExtractorPtr', 'DesFunction',
'DesFunctionPtr', 'DesGlobBorder', 'DesGlobBorderPtr', 'DesGlobWindow',
'DesGlobWindowPtr', 'DesInit', 'DesInitPtr', 'DesMask', 'DesMaskPtr', 'DesMesh',
'DesMeshPtr', 'DesModel', 'DesModelPtr', 'DesNumAutomesh', 'DesNumAutomeshPtr',
'DesNumChimera', 'DesNumChimeraPtr', 'DesNumImplicit', 'DesNumImplicitPtr',
'DesNumMultiGrid', 'DesNumMultiGridPtr', 'DesNumSpaceDisc', 'DesNumSpaceDiscPtr',
 'DesNumTimeInteg', 'DesNumTimeIntegPtr', 'DesNumerics', 'DesNumericsPtr',
'DesState', 'DesStatePtr', 'DesWindow', 'DesWindowPtr', 'ELSA_MAJOR_VERSION',
'ELSA_MICRO_VERSION', 'ELSA_MINOR_VERSION', 'E_1D', 'E_2D', 'E_3D', 'E_ADI',
'E_ASMSZL', 'E_AXI', 'E_BALDWIN', 'E_EXPLICIT', 'E_FIRST_ORDER_NO_SLOPE',
'E_K EPS', 'E_KL', 'E_KOMEGA', 'E_KO_JCKOK', 'E_LURELAXMAT', 'E_LURELAXSCA',
'E_LUSSORMAT', 'E_LUSSORSCA', 'E_MENTER', 'E_MICHEL', 'E_MINMOD', 'E_MKFLC2',
'E_NO_LIMITER', 'E_RELAXMAT_EULER', 'E_RELAXMAT_K4MAT', 'E_RELAXMAT_K4SCA',
'E_RELAXMAT_VISCOUS_3P', 'E_RELAXMAT_VISCOUS_5P', 'E_RELAXMAT_VISCOUS_SCA_3P',
'E_RELAXMAT_VISCOUS_SCA_5P', 'E_RELAXSCA_EULER', 'E_RELAXSCA_EULER_K4SCA',
'E_RELAXSCA_VISCOUS_3P', 'E_RELAXSCA_VISCOUS_5P', 'E_SLOPE_NULL', 'E_SPALART',
```

```
'E_SUPERBEE', 'E_VAN_ALBADA', 'E_VAN_LEER', 'E_X', 'E_Y', 'E_Z', 'Fixed',
'Mobile', 'RelativeFrame', 'RelativeVel', 'StaticDeformable', 'Undeformable',
'__builtins__', '__doc__', '__name__', 'built_with_Ael', 'built_with_Opt',
'built_with_Split', 'built_with_Xdt', 'getBlock', 'getCfdPb', 'getInstance',
'getNbProc', 'getProc', 'get_nb_proc', 'get_proc', 'invalid', 'isMPI', 'joinMatch',
'joinNearMatchCoarse', 'joinNearMatchFine', 'joinNoMatch', 'joinNoMatchLine',
'new_boundary', 'new_extract', 'new_extract_block', 'new_join', 'new_join_match',
'new_join_nearmatch', 'new_join_nomatch', 'new_join_nomatch_linem', 'new_window',
'print_e', 'types']
>>>
```

You can now enter *elsA* statements (using `class`, `method`, `constant`...) defined in `elsA.py` in an interactive session (for further details, please consult *elsA* User's Manual or *elsA* User's Starting Guide).

2.1.4.3 Third test

The third test case is convenient [17] to get a quick idea of *elsA* CPU efficiency, in serial and parallel (MPI) mode (see also 3.6, *p. 37*); it also gives a first example of *elsA* script files, which are simply plain valid Python script files. Get a copy of the script file `test_mpi_16block_ns_lu.py`, located in `$ELSAWKSP/Kernel/api/Py/Test`. Then, enter (in non MPI mode):

```
./elsA test_mpi_16block_ns_lu.py -n 2 -p 1 -s 10
```

You should obtain something close to :

```
# ================================================================================
# elsA v3.2.01d - Copyright (c) 1997-2006 by ONERA
#
# Production: IRIX64_oneroa1_6.5 - sgi_r8 - Apr_10,_2003_-_10:38:24
# C++ Compiler Option: -O2_-woff_all_-diag_error_1681_-LANG:ansi-for-init-scope=on
# Fortran Compiler Option: -O2

Size of Float   : 8 Bytes
Size of Integer : 8 Bytes

# ================================================================================
[('-n', '2'), ('-s', '10'), ('-p', '1')]
[]
SIZE  =  10
NITER =  2
NPROC =  1
Ghost Layer information: ific1 =  2, ific2 = 2
                         jfic1 =  2, jfic2 = 2
                         kfic1 =  2, kfic2 = 2

Cutoff used during computation: cutoff      = 1e-15
                                cutoff_geom = 1e-08
                                min_surface = 1e-30
                                min_volume  = 1e-30

========== Submit problem description : Begin ========================

Problem bench submitted
```

[17] Here an internally generated cartesian mesh is generated, thus eliminating the burden of taking care of mesh files.

```
...
...

======================================================================
 Begin computation
======================================================================

**********************************************************************
 iteration no 1

**********************************************************************
 iteration no 2


------------------------------------------
Time Loop CPU Time:
CPU Time (User) =  2.16 (s)
         (Sys)  =  0.06 (s)
------------------------------------------


======================================================================
 End Loop Computation
======================================================================


# elsA : normal run termination (0)
```

You can experiment with different values of the command line options :

1. '`-n`' controls the number of iteration, `NITER`
   (CPU time is directly proportional to `NITER`);
2. '`-s`' controls the problem size, `SIZE`;
   memory used will scale as the third power of `SIZE`;
3. '`-p`' controls the number of processors; to run in MPI mode :

   ```
   mpirun -np NB_OF_PROC ./elsA test_mpi_16block_ns_lu.py \
          -n 2 -s 10 -p NB_OF_PROC
   ```

2.1.5 elsA installation (Optional)

The last thing to do is to "install" the executable and the Python runtime configuration files :

```
export ELSADIST=$ELSAWKSP  # This is just an example;
                           # ELSADIST can be different from ELSAWKSP !
cd $ELSAWKSP/Kernel
make install
```

This command will copy all the files necessary to run *elsA* in the directory `$ELSADIST/Dist`. If the installation is successful, and if you do not plan to build other productions, you can safely remove the working directory tree `$ELSAWKSP/Kernel`:

```
rm -rf $ELSAWKSP/Kernel
```

To use the installed version, you must reset PYTHONPATH, taking into account ELSADIST, and it is probably convenient to adjust your PATH :

```
export PYTHONPATH=$ELSADIST/Dist/lib/py
export PATH=$PATH:$ELSADIST/Dist/bin/$ELSAPROD
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ELSADIST/Dist/bin/$ELSAPROD
```

The installation is now complete! Now, you should be able to run *elsA* (see *elsA* User's Starting Guide).

2.1.6 Building a new production in the same source tree

To build another production, you can use the **same** source tree. Just reset ELSAPROD, and enter :

```
cd $ELSAWKSP/Kernel; make elsa; make install
```

2.1.7 Optimization notes

Experienced developers may try to change some compiler switches to improve CPU performance. This can be done in two ways :

- globally, one can change compiler switches in file :
  $ELSAWKSP/Kernel/cfg/prods/Make_$PLATFORM.mk;
  the most important flags are :
  - CCCOPT : C++ compiler optimization;
  - FFFOPT : Fortran compiler option.

  Once the makefile template has been changed, you must rebuild the *elsA* libraries; to do that, enter :

  ```
  cd \$ELSAWKSP/Kernel; make cleanall; make depall; make sysall
  ```

  If you use shared libraries, that's all; if not, you still must rebuild the *elsA* executable :

  ```
  cd $ELSAWKSP/src/Api; make exec
  ```

  in both cases, you can, optionnaly, perform a new installation (`make install`). The command `'make depall'` can be quite time-consuming on some machines (slow disk access, NFS file system...), since it computes file dependencies (through a call to the Unix `'makedepend'` utility); it is possible to speed this phase by using `'make genall'` instead of `'make depall'` (note however that `'make genall'` does **not** compute file dependency).

- locally (inside a module directory, such as `$ELSAWKSP/Kernel/src/Agt`), one can edit the file `Makefile.$ELSAPROD.mk`, modifying flags (`CCCOPT` and `FFFOPT`). Then :

```
make clean; make sys
```

If you are using shared libraries, you can then put the modified library in your `LD_LIBRARY_PATH`. If not, you still have to re-build *elsA* executable : `cd $ELSAWKSP/Kernel/src/Api; make exec` and, optionnaly, `make install`

2.1.8 Documentation and indexing

It is often useful to generate doxygen documentation and indexing data, used by glimpse, vi and emacs (se also sections 4.2.1.3, *p. 41* and 4.2.1.4, *p. 41*). This can be done with makefile targets `installdox` and `indexing`.

2.1.9 Switching between different Python versions

If required, it is fairly easy to switch between different Python versions. To do that, you do **not** have to recompile everything ! Only five *elsA* components depend on `Python.h` : `Fact`, `Descp`, `Sio`, `Lur`, `Ael`.
These modules must be re-compiled[18]. So :

1. Reset *elsA* production environment variables related to Python, (`E_PYVERSION`, `E_PPREFIX` and `E_PPREFIX1`).

2. Re-build Python-dependent libraries :

```
cd $ELSAWKSP/Kernel/src/Fact
make clean; make sys

# Same thing for Descp, Sio...
```

3. Re-build *elsA* executable

```
cd $ELSAWKSP/Kernel/src/Api
make clean; make exec
make install # optional
```

[18] Note however that in many cases `Python.h` differences are fairly small, so that re-compilation is not strictly required.

2.2 Building *elsA* from locally modified components

elsA is designed with a modular approach, which allows to compile some of its components independently. When starting a new development, it is often a good idea to check out only the components to be modified (for example `Tur`[19] if one plans to add a new turbulence model). This technique provides several advantages :

- reduction of disk space;
- reduction of compilation time;
- improvement of *elsA* encapsulation and modularity through components.

To build a working version of *elsA* incorporating the locally modified features, it is necessary to set an additional environment variable, `ELSAPATH`, to inform the Makefile system of the location of the reference *elsA* installation.

```
export ELSAPATH=/home/elsa/Public/v3.2.07
cd WorkDir
mkdir Kernel; cd Kernel
cvs -d$CVSROOT co -P cfg  # can be replaced with a symbolic link
cvs -d$CVSROOT co -P Tur
cvs -d$CVSROOT co -P Fact
# building (locally) libraries: libeTur.so, libeFact.so...
make -f cfg/Makefile.mk sysall
```

The final step depends on the production used (dynamic or static link).

2.2.1 Production with shared library

In this case, it is often not necessary to re-build a new specific *elsA* executable; instead, at run time, it is much more convenient to reset `LD_LIBRARY_PATH`, for example :

```
export LD_LIBRARY_PATH=$ELSAWKSP/Kernel/lib/$ELSAPROD:$ELSAPATH/Dist/bin/$ELSAPROD
```

2.2.2 Production with static library

In this case, it is always required to re-build `elsA.x` (directory `Api` must first be checked out) :

```
cd $ELSAWKSP/Kernel
cvs -d$CVSROOT co -P Api
cd $ELSAWKSP/src/Api
make api
```

[19] and possibly a (hopefully) small number of additional related modules, for example `Fact`

2.2.3 Additional information

- If a new source file is added (C++ or Fortran), `Make_obj.mk` must be modified accordingly; note that `make dep` must be called *twice* to get correct dependency (generation of file `Makefile.$ELSAPROD`).
- If `Def` is not checked out, the libraries compiled locally have the same `config` (see 2.1.3.1, *p. 12*) setting as the reference installation.

  If `Def` is checked out locally, and `DefConfig.h` different from the reference version[20], then you must also check out every system depending upon `DefConfig.h` [21].
- If you have to modify C++ headers, then you must find which component must be checked out to avoid inconsistencies; for exemple, assuming that `TurBase.h` is modified :

```
cd $ELSAPATH/src
find . -name '*C' -exec grep -l '#include  *"Tur/Base/TurBase.h' \
  {} \; | cut -b 3-6 | sort | uniq

--> Ael/ Bnd/ Dtw/ Fact Fxd/ Lur/ Sou/ Tmo/ # Modules to be added
```

2.3 SWIG

2.3.1 SWIG *purpose*

Normally, for a standard *elsA* installation, SWIG is *not* required : this is because the only purpose of SWIG is to create some files, which are included in the source distribution. The next section provides more detailed information.

2.3.2 Technical details

SWIG is used to generate **automatically** two files [22], both located in directory : `$ELSAWKSP/Kernel/src/Api/Wrapper` :

1. `elsA.py`, a Python module file which have to be imported by every *elsA* script file;
2. `elsA_wrap.C`, which wraps the Python interpreter.

By default, SWIG is not used, which means that `elsA.py` and `elsA_wrap.C` are not modified during the *elsA* build : the "current" version (for example CVS last version) are used. To invoke SWIG, just define the environment variable `E_SWIG`[23]; then :

[20] `$ELSAPTH/Dist/include/Def/Global/DefConfig.h`
[21] `Descp`, `Bnd`, `Sio`, `Api` (static), and possibly `Ael`, `Lur`, `Opt`
[22] if optional module `Opt` or `Ael` are selected, additional files are created : `elsA_Ael_wrap.C`, `elsA_Ael.py`, `elsA_Opt_wrap.C`, `elsA_Opt.py`
[23] or define `E_SWIG` inside `Make_paths.mk`; it may be useful to define also `E_SWIGOPT`

```
export E_SWIG=/usr/local/bin/swig # example
cd $ELSAWKSP/Kernel/src/Api
make  Wrapper/elsA.w
```

Remarks :

1. The command :

   ```
   cd $ELSAWKSP/Kernel/src/Api; make api
   ```

   invokes SWIG (if `E_SWIG` is defined).

2. It is probably a good idea to use the same Python version to build SWIG and to build *elsA*.

3. However, in most cases, files generated by SWIG can be used to build *elsA* with a different Python version.

4. On SGI, with IRIX, we have not been able to build a 64-bit version of SWIG. Fortunately, the 32-bit version works (SWIG only generates text files, so 32-bit or 64-bit does not matter).

5. Presently, we use version 1.3.31 of SWIG. Let us stress again that, since `elsA_wrap.c` and `elsA.py` are provided in the source distribution, SWIG is **NOT** required in standard *elsA* installation, as well as for most kernel developments.

2.4 Installing from a binary distribution

Instead of building *elsA* from source, it is sometimes useful, or even mandatory, to install *elsA* from a binary distribution :

- you do not have access to *elsA* source distribution;
- you do not have a working C++ or/and Fortran compiler;
- you are using a cross-compiler : compilation and link are done on a computer different from the runtime machine; in that case, you must install the *elsA* executable in some way on the runtime machine.

Currently, there is no sophisticated installation procedure. Set environment variable `ELSAHOME` and do :

```
cd $ELSAHOME
gunzip elsA_bin.tgz
tar xf elsA_bin.tar
```

In some cases, *elsA* binary distribution is delivered together with Python files. This is necessary if the target machine does not have a C++ compiler.
In other cases, Python has to be installed prior to *elsA*.
In both cases, be careful to set correctly `PYTHONHOME` (see also section 2.5.5, *p. 28*).

2.5 Troubleshooting

2.5.1 Build problems

2.5.1.1 Incorrect Makefile generation

The Makefile generation process is controlled by the script file `MakeMake.mk` (`$ELSAWKSP/Kernel/cfg/MakeMake.mk`) called by :

```
make -f cfg/Makefile.mk depall
```

It uses the C preprocessor, either directly :

```
cpp ${FLIST} -I${ERDIR}/cfg/prods ${ERDIR}/cfg/prods/Make_${EPLATFORM}.mk
```

or indirectly, through the C compiler :

```
cc -c ${FLIST} -I${ERDIR}/cfg/prods -E ${ERDIR}/cfg/prods/Make_${EPLATFORM}.mk
```

You may encounter problems if :

- C compiler is not properly installed;
- on some platform, the C compiler does not preprocess correctly files with `.mk` extension. To correct this, a possible solution is to set a symbolic link such as :

  ```
  ln -sf cfg/prods/Make_${EPLATFORM}.mk cfg/prods/Make_${EPLATFORM}.c
  ```

  This case occurs for instance when cross-compiling *elsA* on a Linux machine, to generate NEC specific compiled code.

2.5.1.2 Incorrect Python settings

If Python is incorrectly installed, or if environment variable `E_PYVERSION`, `E_PPREFIX`, `E_PPREFIX1` are incorrectly set, some files cannot be compiled. A convenient and fast check is given below :

```
cd $ELSAWKSP/Kernel/src/Fact
make  .Obj/$ELSAPROD/Base/FactDataBase.o
```

Look carefully to compiler messages about `Python.h` or `config.c`, such as :

```
deimos>make -f Makefile.intelIA64 .Obj/intelIA64/Base/FactDataBase.o
+++ elsA    :    Compiling C++       .Obj/intelIA64/Base/FactDataBase.C
../../include/Def/Global/DefPython.h(27): catastrophic error:
 could not open source file "Python.h" #include "Python.h"
```

If `FactDataBase.o` is not correctly produced, then it is useless to continue.

2.5.1.3 *Problem with iostream*

elsA uses the C++ `iostream` library to perform I/O. Unfortunately, the ISO C++ standardization for `iostream` has changed relatively recently. Without going into too much technical details, we have observed that most current implementations provided by compiler vendors can be classified in two categories :

- *classic* `iostream`;
- *standard* `iostream`.

In order to help somewhat to solve problems related to `iostream` compilation, you can sometimes solve them with the introduction (or removal) of the simple macro definition `_E_USE_OLD_IOSTREAM_` or `_E_USE_STANDARD_IOSTREAM_` in the template Makefile :
(`cfg/prods/Make_$PLATFORM.mk`). For example :

```
E_CC=g++ -D_E_USE_OLD_IOSTREAM_        # gcc2.95
E_CC=g++ -D_E_USE_STANDARD_IOSTREAM_   # gcc 3.2
```

If none of these 2 macros solve the compilation problem, you will have to modify *elsA* source (not recommended, please contact elsa-infodev@onera.fr).

2.5.2 *Known problems with Python*

- Installation of Python 2.2 on IBM AIX machines may be tricky.
- For installation of Python 2.3 on IBM AIX, in some cases we had to remove `socket` and `ssl` (possibly by modifying manually file `setup.py`).
- On NEC SX8, with Python 2.4, there is a header conflict : C++ `string` header must be included *before* `Python.h`. Helper script `patch_include.mk` solves this problem.

2.5.3 *Link time errors*

2.5.3.1 *INTEL* icc

With some installations of INTEL icc C++ compiler, it is sometimes necessary to add `-lstdc++` to `E_EXTERNLIBS`.

2.5.3.2 *Missing* `template` *library (PGI* pgCC *version 5)*

When using pgCC C++ compiler (`ELSAPROD=pgi`), there is a makefile bug related to C++ `template` instanciation, and library `libtemplate.a` is not created correctly, thus leading to a link time error [24]. To correct this problem :

[24] such as : "library `template` not found"

```
cd $ELSAWKSP/Kernel/lib/$ELSAPROD/Template
ar clq  libtemplate.a *.o
```

You can then try again to call the linker :

```
cd $ELSAWKSP/Kernel/src/Api
make exec
```

Remark : This problem does not occur with newer versions of pgCC.

2.5.3.3 Missing `libmass` library at link time (AIX)

With `ELSAPROD`=ibm_..., *elsA* uses the `libmass` and `libmassvp` libraries to speedup CPU computation. If these libraries are not correctly installed on your system, they can be easily downloaded (`http://www.ibm.com/support`).

2.5.3.4 Other link time problem (missing libraries)

On some platforms, *elsA* uses several additional libraries :

- `libz`
- `libexpat`
- `libreadline`
- `libcurses`

If these libraries are not available on your system, don't panic; just remove them by editing the template makefile `$ELSAWKSP/Kernel/cfg/prods/Make_$PLATFORM.mk` (for example, if you don't have `libz` and `libexpat`, remove '`-lz -lexpat`'). Then :

```
cd $ELSAWKSP/Kernel/src/Api
make dep  # rebuild local makefile
make exec # link
```

Only a small number of relatively minor features will be disabled.

2.5.4 Runtime errors

2.5.4.1 Checking configuration file access rights

elsA configuration files, which are Python scripts, should be readable. If not, even with a correct setting of `PYTHONPATH`, at run time you get a message such as :

```
Traceback (most recent call last):
  File "test1_mpi_16block_ns_lu.py", line 26, in ?
    from elsA import *
ImportError: No module named elsA
# elsA : normal run termination (1)
```

or maybe :

```
# elsA [2249] FATAL: API error
# info [2249] Bad module EpKernelDefVal.py
Please check \evname{PYTHONPATH},
check access permissions,
or corrupted files
(EpKernelDefVal.py, EpConstant.py).
Leaving error, code is 2249
# elsA : exit force
```

In such cases, check the file access mode (directory `$ELSADIST/Dist/lib/py`).

2.5.4.2 MPI runtime errors (incorrect environment)

With some versions of MPI (including MPICH), depending on the method that mpirun uses to start the processes, the environment variables such as `LD_LIBRARY_PATH` or `PYTHONPATH` may **not** be "sent" to the processes. For example, using shared libraries, the message may be something like :

```
Connection failed for reason: : Cannot assign requested address
```

One solution, admittedly not very elegant, is to set the required environment variables in the `.profile` (or `.cshrc`) file.
Another solution is simply to define the environment variables in the same command line, immediately before invoking *elsA* executable :

```
PYTHONPATH=$ELSAHOME/Dist/lib/py elsA.x test.py
```

Remark : To use several nodes of a HP Alpha cluster, it may be useful to remove the linker option '`-non_shared`'.

2.5.5 Python runtime errors

2.5.5.1 Incorrect PYTHONHOME

At runtime, *elsA* uses the Python runtime system; if the *elsA* executable was not built on the running machine, it is sometimes useful (specially for Python 1.5) to set the environment variable `PYTHONHOME`.

```
$ elsA.x
...
# ================================================================================
Could not find platform independent libraries <prefix>
Could not find platform dependent libraries <exec_prefix>
Consider setting $PYTHONHOME to <prefix>[:<exec_prefix>]
'import exceptions' failed; use -v for traceback
Warning!  Falling back to string-based exceptions
'import site' failed; use -v for traceback
Python 1.5.2 (#28, Oct 29 1999, 11:41:19) [C] on irix646
```

It is often possible to guess the correct `PYTHONHOME` :

```
$ which python
/home1/elsa/Tools/bin/python
$ export PYTHONHOME=/home1/elsa/Tools
```

Conversely, with Python 2.x, it is better to `unset` (or `unsetenv` in `csh`) `PYTHONHOME`.

2.5.5.2 Python runtime error on HP-UX Itanium

If you encounter Floating Point Error in the initialisation phase (`FactDataBase::FactDataBase`) of *elsA*, it may be useful to recompile *elsA*, with the following Python configure options :

```
CC='aCC -Ae +DD64' OPT='+DD64 +O2 +Onolimit +DSitanium2'
./configure --prefix=$HOME/python2 --without-threads
```

3. PORTING *ELSA* TO A NEW PLATFORM

This chapter can be safely skipped by most developers.

3.1 Introduction

Porting *elsA* to a new platform is usually a relatively simple task, because *elsA* source obeys several important rules:

1. *elsA* C++ source files do **NOT** use sophisticated C++ features such as:
   - exceptions;
   - `RTTI` (Run Time Type Identification)
   - "complex" template coding; we only use template code from Standard Library (STL, `iostream`), so that we usually do not have to care with `template` instantiation mechanism. As a result, in practice, we are able to compile *elsA* C++ source code with most C++ compilers.
2. *elsA* Fortran source files do **NOT** use Fortran90-only features [1]; so we can use any Fortran compiler (77, 90 or 95); this is quite useful on GNU/Linux machine, since it allows us to use the GNU `g77` compiler.
3. The platform-dependent code is **centralized** into a very small number of files; so porting to a new platform involves usually some minor modifications to this set of "configuration files", without touching any other files. This speeds up by a huge amount the time needed to achieve the porting task.
4. Python, which is used by *elsA* as its scripting language, is available on most computing platforms. In the following, we assume that Python is correctly installed (see also 2.1.1.3).
5. *elsA* uses a very small number of standard MPI routines to run in parallel mode.

3.2 Introducing a new platform

1. The first thing to do is to choose a prefix for the new platform, let us say `xxx`; at build time (*cf.* 3.5, *p. 36*), you have to set environment variable `ELSAPROD` to `xxx` (or maybe `xxx_mpi`, `xxx_r4`, `xxx_mpi...`).
2. Then we must create the template makefile :

   `$ELSAWKSP/Kernel/cfg/prods/Make_xxx.mk`.

   It is often useful to start from an existing, hopefully similar, template makefile.

[1] except in `Ael` module

3. To speed up the porting process, in a first stage, it is often convenient to remove the optional *elsA* components : see section 2.1.3.1, *p. 12*.

3.2.1 Compiler choice

3.2.1.1 Set compiler options

You must edit `$ELSAWKSP/Kernel/cfg/prods/Make_xxx.mk`, in order to define the C++ compiler, `E_CC`, including its options, `E_CCCOPT` and `E_CCCFLAGS`. Similarly, you must define the FORTRAN compiler, `E_F90`, including its options, `E_FFFOPT` and `E_FFLAGS`.
You may also have to specify:

- linker options: `E_LDFLAGS`;
- additional libraries: `E_EXTERNLIBS`.

3.2.1.2 C++ standard conforming macros

If the C++ compiler is not fully standard compliant, you may have to specify several symbols:

- `E_NO_COVARIANT_RETURN` : Adding "covariant return type" was the first modification of the C++ language approved by the standards committee. This is a fancy way of saying that the virtual function of a derived class can now return an instance of the derived class when the base class virtual function it is overriding returns an instance of the base class. Example:

```
class GeoGrid : public GeoGridBase
{
...
#ifdef E_NO_COVARIANT_RETURN
  /** */
  virtual const GeoMetricsBase* getMetric() const;
#else
  virtual const GeoMetrics*     getMetric() const;
#endif
}
```

- `E_RTTI` : will replace C++ qualified `dynamic_cast` by C casts; this will have negligible impact on CPU efficiency.

```
#ifdef E_RTTI
#define E_DYNAMIC_CAST(a) dynamic_cast<a>
#else
#define E_DYNAMIC_CAST(a) (a)
#endif
```

3.2.1.3 CPU optimization

Two preprocessor symbols control CPU optimization:

- `_E_FORTRAN_LOOPS_` :
  for some loops, two implementations, Fortran and C++, are available. Defining `_E_FORTRAN_LOOPS_` will select the Fortran version (recommended).

- `E_SCALAR_COMPUTER` :
  Some Fortran subroutines exist in two different versions;
  defining `E_SCALAR_COMPUTER` will select the code optimized for scalar (cache-based) computing platforms, instead of code optimized for vector supercomputers.

3.2.1.4 Fortran file preprocessing

elsA Fortran files (extension `.for`) must be preprocessed. We have to address three cases :

1. the Fortran compiler is able to preprocess `.for` files: there is nothing special to do;

2. the Fortran compiler is unable to preprocess `.for` files, but it can preprocess Fortran files with other extensions (for example, `.F`). In that case, the *elsA* build system will generate symbolic links, so that the compiler can do the preprocessing job; to allow the build system to generate the required links, you have to add three lines in the template Makefile (`E_USE_CPP_FOR_FORTRAN`, `E_REQUIRE_FORTRAN_CPP_EXT`, `E_FOREXT`) :

```
E_USE_CPP_FOR_FORTRAN     = false
E_REQUIRE_FORTRAN_CPP_EXT = true
E_FOREXT=F
```

3. the Fortran compiler is unable to preprocess correctly *elsA* Fortran files; in that case, you have to use the `cpp` preprocessor, with several macros defined, for example:

```
E_CPPF90C=/usr/bin/cpp -D_ELSA_COMPILER_XXX_ \
         -traditional $(E_DOUBLEDEF)
E_USE_CPP_FOR_FORTRAN     = true
E_REQUIRE_FORTRAN_CPP_EXT = false
E_FOREXT=f  # extension of pre-processed Fortran files
```

Remark : *elsA* assumes that the Fortran 90 compiler is able to pre-process correctly Fortran 90 (extension `.f90`) files.

3.2.2 MPI settings

If you want to build a MPI executable, you must set `E_MPIPATH_I`, `E_MPIPATH_L`, `MPICCCFLAGS`, and possibly `MPIEXTERNLIBS`[2].

3.3 Insulation of *elsA* from platform-dependent features

Before entering the compile stage, you will have probably to make minor changes to three files:

- `Def/Global/DefCompiler.h`
- `Def/Global/DefIostream.h`
- `Def/Global/DefFortranOpDir.h`

In addition, in MPI mode, you may have to modify two other files:

- `Pcm/Base/PcmDefMpi.h`
- `Def/Global/DefTypes.h`

3.3.1 DefCompiler.h : STL (and string) insulation

STL platform dependent C++ code is centralized in the C++ header :
`$ELSAWKSP/Kernel/src/Def/Global/DefCompiler.h`.
You must define the preprocessor symbols `E_STD` and `_DEF_USE_USING_`, to control if STL classes (`vector`, `list`, `map`...) are in `namespace std::` or not (i.e in global namespace `::`).
In some rare cases, you may have to do similar things with the `string` class.
In some situations, you may have to change the definition of preprocessor symbol `E_CONST_ITERATOR`; defining `E_CONST_ITERATOR` to `iterator` (instead of `const_iterator`) may solve compilation errors (but decrease somewhat type safety).

3.3.2 DefIostream.h : iostream insulation

`iostream` dependent code is centralized in C++ header :
`$ELSAWKSP/Kernel/src/Def/Global/DefIostream.h`.
In most "old" C++ compilers, the `iostream` library was not included in the `std::` namespace. Actually, it was the old AT&T Cfront compiler library, included through the `iostream.h` header. Now, recent compilers require the inclusion of the `iostream` header (without extension). But even with this up-to-date inclusion, some `iostream` libraries are not within `std::` namespace. For example, with SGI CC v7.30, the

[2] in some cases, it can be useful to use mpiCC; since there is no MPI calls from Fortran in *elsA*, mpif90 is usually not useful

"old" header `iostream.h` is included by default ; to include `iostream` requires `'-LANG:std'` compiler option.
The `_E_USE_OLD_IOSTREAM_` and `_E_USE_STANDARD_IOSTREAM_` macros are used to make the code more readable. These macros should be defined in the template makefile, `Make_$PLATFORM.mk` .
In some (hopefully rare) cases, one may have to modify also `DefFstream.h` and `DefStringStream.h`.

3.3.3 DefFortranOpDir.h: Fortran directive

Fortran directive optimization (mostly for vectorization) are centralized into one file: `$ELSAWKSP/Kernel/src/Def/Global/DefFortranOpDir.h`. You must set `FOR_OPT_DIR_NODEP_E` and `FOR_OPT_DIR_NOLOOPCHG_E` here. Additionally, if you introduce a new directive, consider adding a line here, so that it can be used on other platforms as well, if necessary.

3.3.3.1 MPI insulation

Check that MPI settings are correct
(`Pcm/Base/PcmDefMpi.h` and `Def/Global/DefTypes.h`).

3.4 Basic type sizes

3.4.1 Specifications

elsA has to be usable in single and double precision mode (floating point numbers coded with 4 or 8 bytes); additionally, some platforms put restrictions on integer sizes (4 or 8 bytes). We wish to have the most flexible system; of course this system must be fully portable (no dependency upon some specific compilation options, such as `'-autodbl'`). The following sections decribe the solution implemented in *elsA*; however, in most cases, you will not have to modify anything.

3.4.2 C++ basic type sizes

In order to achieve the requirements described in the previous section, *elsA* C++ code does not directly use `float` (or `double`), `int` (or `long`) and `bool` keywords. Instead, we use `E_Float`, `E_Int` and `E_Bool`, which are defined through a set of `typedef`, inside C++ header `$ELSAWKSP/Kernel/src/Def/Global/DefTypes.h`. For `Real` (`float` or `double`) entities:

```
#ifdef E_DOUBLEREAL
  typedef double  E_Float;
  #ifdef E_MPI
    #define E_PCM_FLOAT MPI_DOUBLE
  #else
    #define E_PCM_FLOAT sizeof(E_Float)
```

```
  #endif
#else
  /** float/double */
  typedef float   E_Float;
  #ifdef E_MPI
    /** MPI float */
    #define E_PCM_FLOAT MPI_FLOAT
  #else
    #define E_PCM_FLOAT sizeof(E_Float)
  #endif
#endif
```

For Integers (`int` or `long`) entities:

```
#ifdef E_DOUBLEINT
  #if !defined(_ELSA_COMPILER_NEC_)     && !defined(_ELSA_COMPILER_HP64_) \
   && !defined(_ELSA_COMPILER_ITANIUM_) && !defined(_ELSA_COMPILER_DEC_)
#   if defined(_ELSA_COMPILER_CRAY_)
#   define E_Int long
#   define E_Bool long
#   else
    typedef long long E_Int;
    typedef long long E_Bool;
#   endif
    #ifdef E_MPI
    #  define E_PCM_INT MPI_LONG_LONG
    #else
    #  define E_PCM_INT sizeof(E_Int)
    #endif
  #else
    typedef long E_Int;
    typedef long E_Bool;
    #ifdef E_MPI
      #define E_PCM_INT MPI_LONG
    #else
      #define E_PCM_INT sizeof(E_Int)
    #endif
  #endif
#else
  /** C++ integer (int or long) */
  typedef int  E_Int;
  typedef int  E_Int;
  /** C++ boolean. May be changed to 'bool' in future releases. */
  typedef int  E_Bool;
  #ifdef E_MPI
  /** MPI integer */
  #  define E_PCM_INT MPI_INT
  #else
  #  define E_PCM_INT sizeof(E_Int)
  #endif
#endif
```

The size of float (real) and integer variables used inside *elsA* is thus determined by the definition of the macros `E_DOUBLEREAL` and `E_DOUBLEINT`, in `Make_$PLATFORM.mk`, (definition which depend of the value of `ELSAPROD`), combined with `DefTypes.h` (which is included by virtually any *elsA* C++ body files).

3.4.3 Fortran basic type sizes

Fortran files use the same insulation technique: every Fortran file must include the Fortran header `$ELSAWKSP/Kernel/src/Def/Global/DefFortran.h`:

```
#ifdef E_DOUBLEINT
#  define INTEGER_E INTEGER*8
#else
#  define INTEGER_E INTEGER*4
#endif

#ifdef E_DOUBLEREAL
#  define REAL_E REAL*8
#else
#  define REAL_E REAL*4
#endif
```

Fortran files do not use REAL or INTEGER directly; instead, they use `REAL_E` and `INTEGER_E`, which are then transformed during the preprocessing phase.
Of course, if you make any modification to `DefTypes.h` and/or `DefFortran.h`, you must be careful: `E_Float` and `REAL_E` **MUST** have the same size, as well as `E_Int` and `INTEGER_E`.

3.5 Building *elsA*

Now, it is time to try to build *elsA*, with the new value of `ELSAPROD`. Basically, you have to follow the procedure described in section 2. In fact, if you have never installed *elsA* before, it is probably a good idea to perform all the steps of a standard installation on a platform already available.

3.5.1 Troubleshooting

3.5.1.1 Begin with `Agt` module

In order to avoid many cryptic compiler messages, we suggest to begin with the compilation of only one module, `Agt` (the smallest one).

```
cd $ELSAWKSP/Kernel/src/Agt
make sys
```

3.5.1.2 Checking Makefile generation

It is sometimes useful to check that the Makefiles are correct. The first thing to do is to look at the actual compilation command (C++ and Fortran); a convenient way is to use the '`-n`' Makefile option; for example:

```
> cd $ELSAWKSP/Kernel/src/Agt
> make .Obj/$ELSAPROD/Transfo/AgtTransfo.o -n

echo "+++ Elsa    : " "  Compiling C++" .Obj/sgi_r8/Transfo/AgtTransfo.C
if [ x = x ]; then
CC  -DE_SCALAR_COMPUTER -D_ELSA_COMPILER_SGI_ -D_E_USE_OLD_IOSTREAM_
    -64 -DE_RTTI -I../../include -DE_DAMAS
    -DE_DOUBLEREAL -DE_DOUBLEINT
    -D_E_FORTRAN_LOOPS_ -DNDEBUG -DE_MEMORY
    -ansiW -diag_suppress 1429,1521 -O2 -woff all
    -LANG:ansi-for-init-scope=on -LANG:exceptions=off
    -o .Obj/sgi_r8/Transfo/AgtTransfo.o
```

```
    -c .Obj/sgi_r8/Transfo/AgtTransfo.C;
    ...

> make .Obj/$ELSAPROD/Transfo/AgtTransfoGenF.o -n

echo "+++ Elsa    : " " \
Compiling Fortran" .Obj/sgi_r8/Transfo/AgtTransfoGenF.for
...
f90 -D_ELSA_COMPILER_SGI_ -64 -cpp -I../../include   -64 -i8 -r8
    -O2 -DE_DOUBLEREAL -DE_DOUBLEINT
    -c .Obj/sgi_r8/Transfo/AgtTransfoGenF.for
    -o .Obj/sgi_r8/Transfo/AgtTransfoGenF.o
...
```

3.5.1.3 elsA main() function

In some situations, such as global initialization, you may have to modify *elsA* `main()` function:

- If you are not using SWIG (this should be the "normal" situation), you must edit file `Api/Wrapper/elsA.C`.
- If you are using SWIG, you have to modify `Api/Wrapper/elsAembed_template.i`: the Makefile system (see `Api/Make_obj.mk`) *builds* `elsA.C` *from* elsAembed_template.i.

3.5.1.4 Link unresolved references

If the linker requires multiple pass, you may have unresolved references. Try to uncomment the following line (file `cfg/Make_lib.mk`):

```
# E_ELSALIBS=$(E_ELSA_SLIBS) $(E_ELSA_SLIBS) $(E_ELSA_SLIBS)
```

3.6 CPU time measurement

To obtain accurate estimation of *elsA* efficiency, you can use several methods:

1. Use the `time` Python module :

   ```
   cfd = DesCfdPb()
   ...
   import time
   t1 = time.clock()
   cfd.compute()
   t2 = time.clock()
   print "CPU time = ", t2-t1
   ```

   This method is quite useful, but:

   - it overestimates slightly the computing time, since it includes the time needed to build the kernel objects;

- Python `time` module cannot be used for large values of elapsed time (internal overflow).

So usually, prefer the other methods.

2. *elsA* prints internally the time spent in the main iterative loop. This eliminates completely startup time, so it is more accurate. It uses Unix `times`, or, when available (NEC SX), a more accurate timer, `syssx`. If the new platform provides such timing function, it is easy to use it: just edit `Def/Sys/DefCPUTime.h` and `Def/Sys/DefCPUTime.C`.

3. For MPI computations, *elsA* uses `MPI_Wtime` to get very accurate timings.

To benchmark a new computing platform, it may be useful to use the Python script : `test_mpi_16block_ns_lu.py`,
located in directory `Kernel/api/Py/Test`. See section 2.1.4.3, *p. 18* for additional details.

4. DEVELOPING INSIDE *ELSA* SYSTEM : GETTING STARTED

4.1 Introduction

To decrease the learning time necessary to develop inside *elsA*, let us give some suggestions :

1. Read this document.
2. Consult *elsA* web site, especially :
   `http://elsa.onera.fr/elsA/dev/guide.html`
3. Consult the reference documentation, especially the UML model :
   `http://elsa.onera.fr/elsA/doc/refdoc.html`
4. Have a look to the (automatically extracted from *elsA* source) doxygen documentation :
   `https://elsa.onera.fr/elsA/dev/doc/document.html`
5. Be sure to have access to some good books about Object-Oriented Software Development :
   - Bertrand Meyer: Object-Oriented Software Construction (2ed);
   - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software.
6. Some C++ books may be helpful; we recommend:
   - Bjarne Stroustrup: The C++ Programming Language (`http://www.research.att.com/~bs/3rd.html`);
   - Harvey M. Deitel, Paul J. Deitel: C++ How to Program (`http://www.prenhall.com/deitel/`);
   - Scott Meyers: Effective C++ (`http://www.aristeia.com/books_frames.html`);
   - Scott Meyers: More Effective C++;
   - Scott Meyers: Effective STL;
   - Robert B. Murray: C++ Strategies and Tactics;
   - Bruce Eckel: Thinking in C++ (Vol 1) (`http://64.78.49.204/TICPP-2nd-ed-Vol-one.zip`);
   - Bruce Eckel: Thinking in C++ (Vol 2) (`http://64.78.49.204/TICPP-2nd-ed-Vol-two.zip`);

- John Lakos: Large-Scale C++ Software Design (`http://www.awprofessional.com/catalog`).

7. Some useful links :

   - Unified Modeling Language (UML) : `http://www.uml.org`;
   - Standard Template Library Programmer's Guide : `http://www.sgi.com/tech/stl`;

4.2 Useful tools

In this section, we describe briefly some tools, which may be useful in order to reduce the learning time for new *elsA* developers.
Without any tools, it would be a lengthy and boring task for newcomers to grasp the technical contents of the software. Even if the "*elsA* Design and Implementation Tutorial", associated with the UML Design documentation, give many useful information, it must be admitted that the understanding of *elsA*, at source level, is not easy. For example, it can be frustrating to find where a specific type (`class`, or `typedef`), a function, a preprocessor `macro` (such as `#define`), or an `enum`, is declared or defined.
In the following, we describe briefly several tools that have been found useful over the years by *elsA* developers.

4.2.1 Navigating elsA source code

4.2.1.1 doxygen

doxygen (`http://www.stack.nl/~dimitri/doxygen/`) is a powerful documentation system for C++ code:

- The documentation is extracted directly from the sources,

  which makes it much easier to keep the documentation consistent with the source code.

- Generated on-line HTML documentation, which can be browsed by any navigator, is very useful to quickly find your way in *elsA* source tree.

- Relations between the various elements (files, types, functions, modules, namespaces) are visualized by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are generated automatically (thus avoiding any human error!).

- Fortran file are not fully supported; however, Fortran routine calls inside C++ code are recognized, and Fortran source files can be browsed.

Please visit the documentation generated by doxygen from *elsA* source code : `https://elsa.onera.fr/ExternDocs/dev/doxygen/html/hierarchy.html`.

4.2.1.2 Doc++

`Doc++` was available several years before doxygen. Since doxygen is more powerful, its usage inside the *elsA* project is now deprecated. Switching from Doc++ to doxygen was relatively easy, since doxygen recognizes the documentation convention of Doc++.

4.2.1.3 glimpse

glimpse builds a keyword index in advance for very fast searching. Uncommon words will be found rapidly even in a very large fileset, up to several Gigabytes. Common words (with 100's or 1000's of matches) will take longer, but if the number of hits returned can be limited, even those will be very fast.
glimpse is available on most SGI machines (and can probably be installed on other platforms[1]). For each new production, it is a good idea to (re-)build the index[2], for example:

```
glimpseindex -o -H Some_Directory -n $ELSAWKSP/Kernel/src
alias -x glimpse='glimpse -H Some_Directory'
```

Then, to retrieve all the occurences of, let's say, `BndPhys`:

```
glimpse BndPhys
```

In practice, glimpse is **much much** faster than Unix `grep`.

4.2.1.4 Use of "tag" index files

A tag is an identifier that appears in a "tags" file. It is a sort of label that can be jumped to. For example: each function name can be used as a tag. The "tags" file has to be generated by the Unix program ctags, before the tag commands can be used. This tag file allows these items to be quickly and easily located. Tag index files are supported by numerous editors (vi, vim, emacs, nedit...), which allow the user to locate the object associated with a name appearing in a source file and jump to the file and line which defines the name.
For each new production, it is a good idea to (re-)build the tag files[3], for example (the options to use may depend of the versions of [ce]tags used):

[1] `http://www.icewalkers.com/Linux/Software/517090/Glimpse.html`
[2] this is done with `make indexing`
[3] this can be done with : `make indexing`

```
cd $ELSAWKSP/Kernel
find $ELSAWKSP/Kernel/src  \
     \( -name '*\.for' -o -name '*\.[hC]' -o -type f  \) \
 | ctags - --c++ --define  --globals --members \
         --typedefs-and-c++ --no-warn

find $ELSAWKSP/Kernel/src  \( -name '*\.for' \
     -o -name '*\.[hC]'  \) | grep -v Obj | etags - -C
```

Now, with `vi/vim` editor :

- With the '`:tag some_tag`' command the cursor will be positioned on the definition of `some_tag`.
- If you see a call to a function and wonder what that function does, position the cursor inside of the function name and hit `CTRL-]`. This will bring you to the function definition: the keyword on which the cursor is standing is used as the tag; if the cursor is not on a keyword, the first keyword to the right of the cursor is used.
- An easy way back is with the `CTRL-T` command.

For emacs users, please consult emacs documentation.

4.3 Non regression tests

The purpose of a non regression test is to check that, after some code modification, a specific feature is still working[4]. In *elsA*, non regression tests perform a small number of iterations (5 or 10, usually). The computed residuals are then compared (with `diff`) with the residuals produced by the reference version[5]. The amount of memory, and CPU time can also optionally be compared[6]. A large number of non regression tests already exists[7]. Any new development must provide one (or several) new non regression tests. Since non regression tests are passed very often, it is quite important to keep their computing cost (memory and CPU) as low as possible. Before any integration, developers must pass the complete suite of non regression tests.
Consult *elsA* web site for additional information about `Reg` :
`https://elsa.onera.fr/elsA/dev/dev/env.html#moel`

4.3.1 How to run regression tests

The non regression tests are stored in a separate CVS repository, `Reg` (see section 5.1). The regression test environment is controlled with a specific Makefile.

[4] it is also very helful when porting *elsA* to a new computing environment

[5] output files produced by `extractor` objects are also checked, using `cmp`, if the file name respects the naming convention :`Wksp/script_name*data_[0-9][0-9]*`

[6] see `compare_memory.sh` and `dif_memory.sh` in `Reg/Tools`

[7] approx. 600 sequential tests, and 60 parallel (MPI) tests

4.3.1.1 *Checking out (CVS) regression scripts*

To check out the reference Python script :

```
export CVSROOTREG=...
cd SomeWorkDirectory
cvs -d $CVSROOTREG co -P Reg_core
cd Reg
```

Remark : Instead of checking out `Reg_core`, it is also possible to check out the entire `Reg` repository; however, doing this checks out all the existing references[8], which can take quite a long time, and wastes a lot of disk space[9]

4.3.1.2 *Setting the executable version to be tested*

You must edit `SomeWorkDirectory/Reg/Makefile`, and set correctly `ELSAROOT`, `ELSAPROD` and `ELSAVERSION`.

4.3.1.3 *Setting the reference*

In many cases, a reference already exists :

- For `intelIA64` (and `intelIA64_mpi`) and `sgi` (and `sgi_mpi`), reference is stored in CVS in a sub-module[10] :

  ```
  cd SomeWorkDirectory/Reg
  cvs co -P Ref_intelIA64
  # cvs co -P Ref_intelIA64_mpi_2proc
  # cvs co -P Ref_sgir8
  # cvs co -P Ref_sgir8_mpi_2proc
  ```

  Then edit `Makefile` and set correctly `OLDREF`.

- In other situations, a reference exists[11], but has not been stored in CVS. Set `OLDREF` accordingly.

- In some cases, you want to compare the results produced by two different *elsA* executables, `elsA1.x` and `elsA2.x`[12]. You must first run the regression tests with `elsA1.x`, put the results in reference[13], and then re-run the regression tests with `elsA2.x`.

[8] most of them useless in the current context
[9] appprox. 600 MBytes
[10] currently, four references are available through CVS : `Ref_intelIA64`, `Ref_intelIA64_mpi_2proc`, `Ref_sgir8`, `Ref_sgir8_mpi_2proc`
[11] for example, it has been installed by the local *elsA* expert
[12] for example compiled with different optimization options
[13] using `VNREF` and `make putrefPY` (or `make putrefPY_PARA`)

```
# setting ELSAROOT... to run elsA1.x
make
# set VNREF
make putrefPY TAG=...
# reset OLDREF to VNREF, and  ELSAROOT... to run elsA2.x
make
```

4.3.1.4 Running regression tests

- To run all the sequential tests, just enter :
  `make`

- To run all the parallel tests, just enter :
  `make PY_PARA`
  Currently, parallel MPI cases run with 2 (default) or 5 processors[14] [15].

- To run test cases contained in a single test directory :
  ```
  make PY_CASES=Nozzle/Axi          # sequential
  make PY_PARA PY_CASES_PARA=Vega # parallel
  ```

- To run several test directories :
  `make PY_CASES="Nozzle/Axi Nozzle/Base"`

Remarks :

1. Depending on compiler options and processor floating point arithmetic implementation, results computed by *elsA*, with exactly the same source code, can be different on two different platforms. However, those differences are usually very small, so that if an "exact" reference" is not available, it may be convenient to use another one[16].

2. Python regression scripts have to read additional data files (mesh, init or boundary files). Currently, these files are not stored in a CVS directory. The location of the root directory can be adjusted by setting `ROOT_DB` in the regression Makefile.

3. Several runtime environment variables can be tuned in order to track machine-dependent code[17] :

   - `ELSA_INIT_ARRAY_F` : initialization of C++-allocated real arrays;
   - `ELSA_IEEE_MODE`[18] : control IEEE exception behaviour.

[14] look at `NPROC_REG` inside Python script

[15] note that the name of the CVS directory for parallel regression tests, `Ref_sgir8_mpi_2proc` and `Ref_intelIA64_mpi_2proc` are inconsistent

[16] for example, `intelIA64`, `bull`, `intelIA32` and `intelIA32em` are nearly identical

[17] see *elsA* User's Reference Manual for a complete description of `ELSA_INIT_ARRAY_F` and `ELSA_IEEE_MODE`

[18] currently, active only for `sgi` and `intelIA64`

In some cases, it can be useful to relax the runtime environment constraints[19].

4.3.2 Adding new regression tests

- To add a new sequential regression test in an existing sub-directory, put the new script inside the sub-directory, and edit file `00LIST_TEST_CASE_PY` .
- To add a new parallel regression test in an existing sub-directory, put the new script inside the sub-directory, and edit file `00LIST_TEST_CASE_PARA` .
- To add a new regression directory, you must modify `PY_CASES`[20] in the regression makefile.

4.4 Validation Data Base

The validation tests are stored in a separate CVS repository (see section 5.1). The validation data base provides a wide range of test cases (approx. 200). Two platforms are regularly considered in this validation : NEC SX for sequential test cases, IBM Power4-5 for parallel (MPI) test cases. Any development introducing a new functionality must provide one validation test case (including the script, the mesh and all files needed for the computation, some tecplot macros in order to visualize the results, and reference results) which will be introduced in the validation repository.
The validation data base can be consulted :
`http://elsa.onera.fr/elsA/validation/valid.html`.
For each test case, the script file is available, as well as examples of post-processed results (Tecplot graphics) and performance measurements (CPU and memory).

4.5 Unitary test cases

C++ unitary test cases can be used to check the behaviour of class member functions and FORTRAN subroutines. They can be written during the class coding phase by the class developer. Detecting errors early, during unitary test runs, can save a lot of time see 4.4, *p. 45*). Moreover, unitary test cases often provide useful insight to others programmers upon the correct use of a new class.
The end result of a unitary test case should be a boolean :

- TRUE if everything is ok;
- FALSE if something is unexpected.

[19] notably for some test cases related to the `Opt` component
[20] or PY_CASES_PARA in MPI

The following lines present a very simple example testing the `normeL1` and `normeL2` methods of the `FldField` class :

```
const E_Int nval=5;
const E_Int nfld=2;
const E_Int dim = nval*nfld;
FldArrayF valeur2(dim);
FldFieldF field1(nval,nfld);
FldArrayF norme1(nfld);
FldArrayF norme2(nfld);

E_Int nbErrors = 0;

// test of norms ------------------------------------------------------
//
for (i=0; i<dim; i++)
{
  valeur2[i] = E_Float(i-5);
}

field1.setAllValuesAt(valeur2);

field1.normeL1(norme1);
field1.normeL2(norme2);

if (!fEqual(norme1[0], 3.))
  nbErrors++;

if (!fEqual(norme1[1], 2.))
  nbErrors++;

if (!fEqual(norme2[0]*norme2[0], 11.))
  nbErrors++;

if (!fEqual(norme2[1]*norme2[1], 6.))
  nbErrors++;

if (nbErrors)
{
  cerr << " nbErrors " << nbErrors << endl;
  return E_False;
}
else
{
  return E_True;
}
```

Remark : Due to time constraints, *elsA* unitary tests are no more updated on a regular basis. They can still be useful to provide some examples.

4.6 A simple *elsA* application

An application is a procedural sequence of instructions where the developer describes the problem to be solved by creating objects and sending messages to them. A typical application can involve the following steps :

1. creation of a mesh;

2. initialization;
3. creation of boundary conditions;
4. resolution of the problem;
5. postprocessing.

In practice, the application takes the form of a C++ `main()` function, which must be written by the developer. Once successfully compiled, it is linked with the class libraries to end up with an executable file. Then the developer can run the executable file and obtains the output.

4.6.1 Example from the classical shock tube problem

As an example, we give here the main lines necessary to build objects used to compute a shock tube problem (to simplify, most objects used in the time integration algorithm have not been taken into account) :

```
// ============================================================================
//               Project: elsA - DSNA/ELSA - Copyright (c) 20039 by ONERA
// Type   : <1357438245 9689> C++ Unitary Test File
// File   : SimpleTest.C
// Vers   : $Revision: 1.1 $
// Chrono : No  DD/MM/YYYY Author   V   Comments
//          1.6 11/05/2003  AJ      3.0 Creation
// ============================================================================
#include "Rhs/Base/RhsEquation.h"

#include "Lhs/Base/LhsNone.h"

#include "Eos/Base/EosSysEq.h"
#include "Eos/Base/EosIdealGas.h"

#include "Def/Support/DefMain.h"

#include "Geo/Grid/GeoGrid.h"
#include "Geo/Grid/GeoWindowStruct.h"
#include "Geo/Grid/GeoCfdField.h"

#include "Blk/Base/BlkMesh.h"
#include "Blk/Compose/BlkElemBlock.h"

#include "Fxc/Centered/FxcCenter.h"
#include "Fxc/Centered/FxcScaNumDiss.h"

#include "Bnd/Phys/BndSupOut.h"

// ============================================================================
E_Int main()
{
  E_Int DIM=10;
  E_Int NITER = 10;
  cerr << "CUBE dimension = " << DIM << " X " << DIM << " X " << DIM << endl;
  cerr << "Number of ITERATION = " << NITER << endl;
  cerr << "============================================" << endl;
  cerr << endl;

  // Coeff for grid construction:
```

```
E_Int im=DIM;
E_Int jm=DIM;
E_Int km=DIM;
E_Float dx=ONE/DIM;
E_Float dy=ONE/DIM;
E_Float dz=ONE/DIM;

// ------------------------------------------------------------------------
// 1. Creation of mesh, block, grid objects:
// ------------------------------------------------------------------------
BlkMesh mesh(im, jm, km, 0., 0., 0., dx, dy, dz);

BlkElemBlock block(mesh);
E_Int gridLevel=1;
GeoGrid& grid = *(block.accessGrid(gridLevel));

// ------------------------------------------------------------------------
// 2. Initialization of the aerodynamic field
// ------------------------------------------------------------------------

E_Float val1 = 1.;
E_Float val2 = 2.5;

FldCellF wCons(ncell,eqNb);
for (E_Int wi=0; wi<ncell; wi++)
{
  wCons(wi, 1) = val1;
  wCons(wi, 2) = 0.;
  wCons(wi, 3) = 0.;
  wCons(wi, 4) = 0.;
  wCons(wi, 5) = val2;
}
// Perturbation to avoid residual=0 (and unwanted end of iteration !)
wCons(ncell/2,2)=0.1;

// Creation of the container which stores the aerodynamic solution
E_Int order = 1;
E_Int ncell = grid.getNbCell();
E_Int eqNb  = 5;
GeoCfdField cfdF(order, ncell, eqNb);
grid.setGeoCfdField(&cfdF);

for (E_Int i=0;i<=order;i++)
  cfdF.initSolution(wCons,i);

// ------------------------------------------------------------------------
// 3. Creation of fluxes and boundary conditions
// ------------------------------------------------------------------------

// Jameson's scheme
FxcCenter operCen;

// Artificial (numerical) dissipation --------------------
E_Float k2    = 0.5;
E_Float k4    = 0.016;
E_Float sigma = 1.0;
FxcScaNumDiss::sensorType sensor = FxcScaNumDiss::pressure_velocity;
FxcScaNumDiss operDiss(k2,k4,sensor,sigma);

// Complete fluxes with boundary conditions
// (for example "inactive" condition)

// create 6 windows (GeoWindowStruct):
GeoWindowStruct w1(  1,  1,  1,DIM,  1,DIM);
GeoWindowStruct w2(DIM,DIM,  1,DIM,  1,DIM);
```

```
GeoWindowStruct w3(  1,DIM,  1,  1,  1,DIM);
GeoWindowStruct w4(  1,DIM,DIM,DIM,  1,DIM);
GeoWindowStruct w5(  1,DIM,  1,DIM,  1,  1);
GeoWindowStruct w6(  1,DIM,  1,DIM,DIM,DIM);

BndSupOut bndS1(grid, w1);
BndSupOut bndS2(grid, w2);
BndSupOut bndS3(grid, w3);
BndSupOut bndS4(grid, w4);
BndSupOut bndS5(grid, w5);
BndSupOut bndS6(grid, w6);


// -------------------------------------
// 4. Creation of the physical model
// -------------------------------------

E_Float gam = 1.4;
EosIdealGas eos(gam);

// -------------------------------------
// 5. Creation of numerical objects used
//    in the time integration algorithm
// -------------------------------------

// Choice of monogrid:
TmoMonoLevel level(E_True);

// Choice of the multistep time integration algorithm:
TmoRKutta tmoRK;
tmoRK.freezing(Steady);

// -------------------------------------------------
// 6. Creation of the system of equations to solve
// -------------------------------------------------

// Characterization of the system of equations:
EosSysEq desCfdSys(eos,eosgl_meanFlow,eos_euler);

// TmoSystem: stands for a system of equations:
TmoSystem sys1(desCfdSys,eosel_meanFlow);

// ----------------------------------------------------------------
// 7. Creation of the numerical problem to solve with all its data
// (the system of equations,
// the block where this system has to be solved,
// the numerical ingredients,
// the physical modelization choice)
// ----------------------------------------------------------------

TmoPbElem pbElem(eos, desCfdSys, eosel_meanFlow, level, tmoRK, TmoSolverBase::impl);
pbElem.addBlock(block, TmoLevel::Undefined);
pbElem.addRhsOper(operCen, block, desCfdSys, eosel_meanFlow);
pbElem.addRhsOper(operDiss, block, desCfdSys, eosel_meanFlow);

E_Boolean flag = E_True;
flag &= pbElem.addBoundary(bndS1, operCen, block);
flag &= pbElem.addBoundary(bndS2, operCen, block);
flag &= pbElem.addBoundary(bndS3, operCen, block);
flag &= pbElem.addBoundary(bndS4, operCen, block);
flag &= pbElem.addBoundary(bndS5, operCen, block);
flag &= pbElem.addBoundary(bndS6, operCen, block);

flag &= pbElem.addBoundary(bndS1, operDiss, block);
flag &= pbElem.addBoundary(bndS2, operDiss, block);
```

```
    flag &= pbElem.addBoundary(bndS3, operDiss, block);
    flag &= pbElem.addBoundary(bndS4, operDiss, block);
    flag &= pbElem.addBoundary(bndS5, operDiss, block);
    flag &= pbElem.addBoundary(bndS6, operDiss, block);


    sys1.addTmoPbElem(pbElem);


    [......]
  }

  // ===== Tmo/Driver/Test/TmoDriverTest === Last line ===
```

5. DEVELOPMENT PROCESS

In this chapter, we will only be interested in developments which have to be integrated, for whatever reason. Obviously, there may exist developments which will never enter into the integration process (research purpose).

5.1 Team work for a common version

Although developers come from different CFD or software cultures and have to implement very different developments (for example adding new CFD capability, or extending Python interface), they have to integrate their development in the **common unique** version of *elsA*.
To achieve this, developers have to respect procedures and rules; this is mandatory for an integration agreement.
The first point a developer has to be convinced of is that his development doesn't only consist of some source lines, but also of:

- documentation:
  - manuals and technical notes :
    CVS `:pserver:user@elsa.onera.fr:/data/cvs/doc`;
  - source documentation.
- test cases belonging to one of the three following test bases:
  - `Apps` : the C++ unitary test base :
    CVS `:pserver:user@elsa.onera.fr:/data/cvs/apps/test`;
  - `Reg`: the (Python) regression test base (*cf.* 4.3, *p. 42*) :
    CVS `:pserver:user@elsa.onera.fr:/data/cvs/reg`;
  - `Val` : the (Python) validation test base (*cf.* 4.4, *p. 45*) :
    CVS `:pserver:user@elsa.onera.fr:/data/cvs/val`.

The second point is that his development will be integrated in a **common** code and will necessarily interact with other capabilities issued from other developments. That means that the developer has to:

1. acquire a basic knowledge of *elsA*, particularly the part to modify or extend, by:
   - reading the source code;
   - understanding the existing design solutions.
2. respect all rules and procedures (of source code, but also of tests):

- respect the coding rules and implementation choices :
  /ELSA/MDEV-03050 : *elsA* Programming rules
  (`http://elsa.onera.fr/elsA/doc/refdoc.html#MDEV-03050`)
- respect as much as possible a global coherency in design solutions :
  /ELSA/MDEV-06001 : Design and implementation tutorial
  (`http://elsa.onera.fr/elsA/doc/refdoc.html#MDEV-06001`)

3. search for simplicity, clarity, efficiency;
4. deliver all information: documentation, source code, test cases;
5. take into account remarks issued from the integration review.

This development process necessarily leads to a "more or less" heavy development phase, but benefits in term of reduced maintenance costs are obvious. These rules should lead to a globally coherent software, well documented and tested, simple and versatile enough to take multi-applications into account and allow various developers to perform their own task. Even if this process is not sufficient to reach this goal, it is nevertheless necessary. We describe in the following sections the different stages of development process and the different associated tasks the developer has to perform.

5.2 Different developer's profile

Coding in *elsA* doesn't mean necessarily controlling everything in the source code. As in any large software, the developer has to accept ignoring completely some parts, not knowing every detail in others. Object-oriented programming, based on interfaces, facilitates using the code while ignoring large parts of it. Here is an attempt to identify three "types" of *elsA* developer.

- A "beginner" developer can be usually viewed as a CFD developer: he usually knows about only a limited subset of the kernel. He may have to:
  - use CFD classes and methods,
  - extend, modify an algorithm,
  - introduce Fortran subroutines,
  - specialize or generalize existing C++ classes,
  - propose some extension to the Python interface in order to make new feature accessible to users,
  - introduce C++ unitary test cases,
  - extend and/or introduce Python integration or validation test cases.

  He has to acquire the knowledge of the module impacted by the development, and a minimum knowledge of the kernel design.

- A "kernel" developer understands most of the kernel architecture. He has to introduce new concepts, or improve existing ones. He has to introduce his own capability from the script file up to the internal objects of the kernel; so, he has to be able to use the description objects (`Descp`) and most of the time a limited part of the Factory (`Fact`), but he doesn't have to understand all the source code of these two modules. If his design work may have wide repercussions on other functionnality, he has to acquire a quite complete global understanding of the kernel. It is of his responsability to estimate which part he can ignore, which other he has to understand. In that case, he should interact with other developers (elsa-dev@onera.fr), or developer support (elsa-infodev@onera.fr), and propose new solutions before implementing them. Support of design documentation or technical notes should allow this interactive work.

- An "*elsA* application" developer knows the whole architecture, including Python scripting interface; he understands creation, destruction, association of objects and all transverse mechanisms, proposes Python extensions and evolutions.

5.3 Development process

The development process can be splitted in five successive stages.

5.3.1 Definition of the specifications

First of all, specifications of the new development should be defined and the developer support (elsa-infodev@onera.fr) has to be informed that a new development has been planned for a necessary coordination of the *elsA* evolution.
During this first stage, the developer has to define the specifications in the following way:

1. write theoretical basis of the development (Theoretical Manual contribution),

2. define validation test cases (Validation contribution),

3. write the corresponding user interface documentation: usage description, new key words, advices to users, ... (User's Manual contribution).

5.3.2 Design

The second stage is the time of design elaboration. Achieving a good design will reduce implementation, test, and maintenance time.
This stage is important because each decision made for a specific development can have wide repercussion when the code is used by others. Among the issues to be worked out are:

- Interfaces: what services and access are provided? The aim is to provide services that are convenient, with enough functionnality to be easy to use, but not so much as to become unwieldy.
- Information hiding: which information is visible and which is private? The interface must provide access to services while hiding details of the implementation, so they can be changed without affecting users.

During the design stage, the developer has to write a short design documentation describing his design choices and the retained solution. This documentation will be necessary to enrich the UML class model documentation and the "Developer's Guide". It will participate to the elaboration of the common design experience and will be very useful for other developers in similar situations.
During this stage, we recommand to interact with other developers and with the developer support, specially for a novice developer.

5.3.3 Implementation

After design comes implementation. But in many cases these two stages are not so clearly separated; in fact, most of the time, design and implementation evolve together in an iterative cycle.
The implementation stage consists of:

1. writing documented source lines:
   - modifying existing code;
   - introducing new CFD capability in the kernel.
2. testing the development by mean of :
   - new C++ unitary test cases; these tests will verify the correct coding of the CFD capabilities, specially the Fortran routines;
   - new Python integration test cases; these tests will verify the whole design of the new development, the correct creation of all useful objects, the general coherency and memory management of the development.

   These tests are very important to insure maintenance of the new added feature.
3. modifying all test cases impacted by the new development;
4. checking no-regression of Python integration tests.

5.3.4 Validation

Everything is now ready to perform the validation computations defined in the first stage.

5.3.5 Integration review

The last stage of the process is the integration review.
Before integration, all elements produced for a development (documentation, source code and tests cases) are reviewed by reviewers. This is very useful to check the completeness of all elements and conformity with the development rules. It improves the homogeneity of the source code and extends knowledge of the source code by the developers. Furthermore, it is very important to detect most errors and imperfections as soon as possible so as to minimize the cost of their consequences.
As soon as the development is ready for integration, the developer has to inform the developer support, and ask for an integration review. The information to communicate is described on the web site (see the template for mail to be used to ask for an integration review) :
`http://elsa.onera.fr/elsA/dev/reviews.html`.
This review is mandatory and consists of checking that:

1. all software elements have been furnished : source code, documentation, new test cases to enrich both `Reg` (regression test base) and `Val` (validation test base);
2. the code of the CVS workspace is ready for integration:
   - it has been updated, all conflicts have been removed,
   - all debug code has been suppressed;
3. the no-regression `Reg` test base has been checked; in some situations, it is also necessary to run the `Val` test base.
4. coding rules and implementation choices have been respected;
5. the implementation corresponds to the described design;
6. the new test cases are pertinent (check and validate properly the development);

Moreover, the review has to evaluate:

- the code quality: simplicity, clearness, coherency;
- the design solutions and their impact on other developments.

In some cases, additional questions can be asked to the developer, such as:

1. give an estimation of the performance and the vectorization state of the development; in that case, the developer will have to provide "profilers", memory usage for different computations;
2. check the portability; in that case, the developer will have to perform some tests on different computers.

If all the checked points are declared correct by the reviewers, the development can be integrated and enrich all the repositories; conversely, if some checked points are not judged satisfactory by reviewers, corrections and modifications have to be made by the developer.
We have describe here the standard development process, with its successive steps. But most of the time, the developer has to iterate through this process; refactoring is particularly important, because requirements can change, software needs to be extensible, developer experience grows. It is a part of the every developer's daily business and object-oriented programming should make it easier.

5.4 Development support and documentation

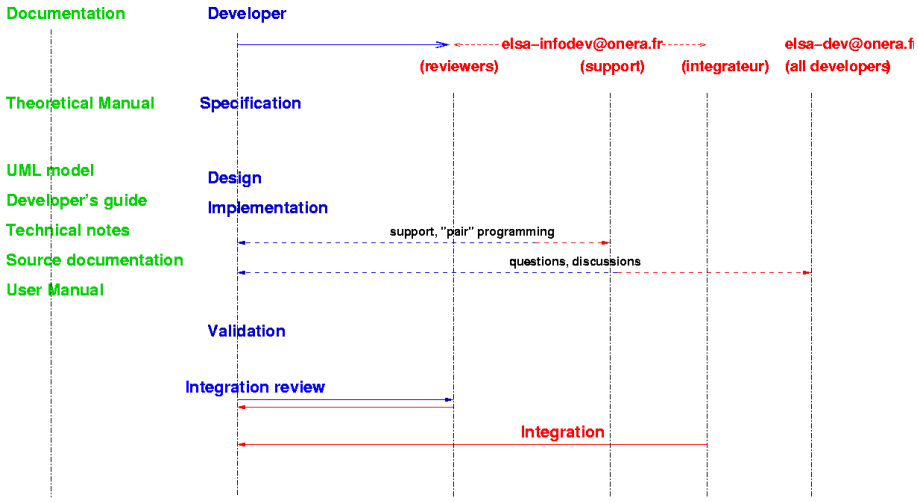
Several communication means are available to receive and send information about *elsA*.
First of all, the address :
elsa-infodev@onera.fr
centralizes all information about developments, integration reviews, integration, and has to be informed of each new developer and of each new planned development. All question or remark a developer wants to communicate to the *elsA* team has to be sent to this address. Introducing a new developer to elsa-infodev@onera.fr is for him the entry point to access to CVS, receive access passwords to the private sections of the Web site dedicated to developers, and receive all electronic or paper information intended for developers.
Support to the developers is ensured by the *elsA* team and covers:

- tutorship,
- maintenance of the developer documentation,
- concrete help to beginner developers through pair programming,
- follow-up of any development if needed,
- coordination of design and implementation choices,
- information by mail to all developers about: new production versions, tips for development, problems detected in reference versions;
- support for debugging reference versions,
- coordination of integration reviews.

The address `elsa-infodev@onera.fr` has to be used to contact the developer support.
A developer discussion list is also at the disposal of the developers to share their experience and questions about developing in *elsA*. The address of this mailing list is

elsA-dev@onera.fr. It is possible to subscribe on the *elsA* web site to the Developer discussion list) :
`http://elsa.onera.fr/elsA/dev/guide.html`
and consult the archives of all mails sent to this adress.
Finally, information dedicated to the developers is available on the *elsA* web site and propose different sections:

- News/Developers for general information, especially about reference versions;
- Development/Tips for development advice;
- Development/Validation for intermediate version validation record;
- Development/Known bugs for bugs detected in release versions;
- Development/Problem Tracking for developers information about use problem tracking;
- Development/Reviews for integration review information;
- Development/Environment for hardware and software development environment.



Empty page

Direct access to index's alphabetical section headings :

– Z – *p.* 66

INDEX

– D –

– E –

DIFFUSION SCHEME

Software Secretariat Archives
Redactors
elsA developers

END of LIST