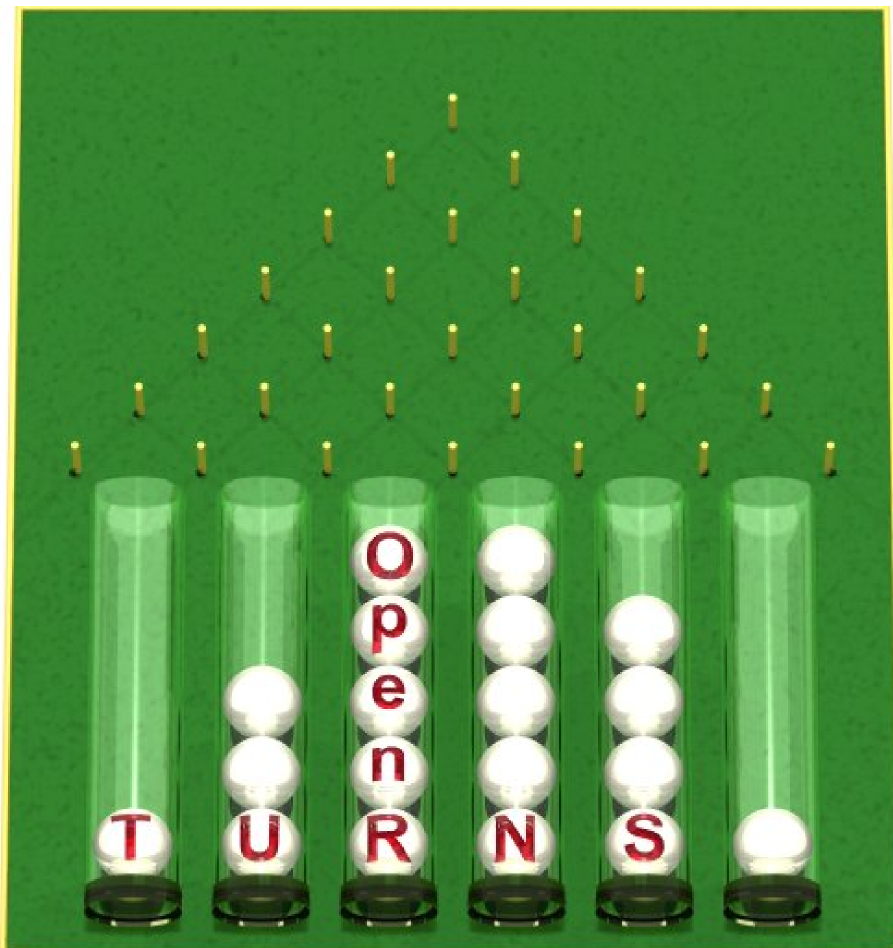
Open TURNS – Wrappers Guide

Open TURNS version 1.0

Documentation built from package openturns-doc-April2012

April 20, 2012



Open TURNS project: Coupling mechanisms with computational codes

Abstract

The purpose of this document is to present the latest features developed within the Open TURNS library regarding the interfacing mechanism with a computational code.
The latest advances offer users:

- greater simplicity of implementation with the use of macros and shared functions that cover a large part of the expressed needs;
- greater reliability, again thanks to these shared functions;
- faster development by reducing the amount of programming required from the user;
- simplified maintenance, again thanks to the reduction of the volume of code;
- improved performance by using parallel techniques in a few relevant spots.

©2005-2012 EDF - EADS - PhiMeca

Contents

1 Introduction 4

2 The different types of functions 5

3 The wrapper 6
3.1 Operating principle of a wrapper . . . 6
3.1.1 Why use a wrapper? . . . 6
3.1.2 Presentation of the wrapper . . . 8
3.1.3 The dynamic library . . . 8
3.1.4 The description file . . . 8
3.1.5 Invoking and loading a wrapper . . . 8
3.1.6 Step 1: Creation of a NumericalMathFunction . . . 9
3.1.7 Step 2: Search for the description file . . . 9
3.1.8 Step 3: Analysis of the description file . . . 10
3.1.9 Step 4: Search for the dynamic library . . . 10
3.1.10 Step 5: Loading the dynamic library . . . 10
3.1.11 Step 6: Completion of the NumericalMathFunction construction . . . 13
3.2 Example: Creating a minimal wrapper . . . 13
3.2.1 Required Items . . . 13
3.2.2 Starting from a template to generate the wrapper . . . 14
3.2.3 Compilation and installation of the wrapper . . . 15
3.2.4 Description file for the minimal wrapper . . . 15
3.2.5 Source code for the minimal wrapper . . . 18
3.2.6 Checking if the minimal wrapper is functioning . . . 20

4 The structure of a wrapper 21
4.1 Naming conventions . . . 21
4.1.1 In the wrapper's source file . . . 21
4.1.2 In the wrapper's description file . . . 22
4.2 Sequence of steps . . . 22
4.3 The processing/step combination . . . 23
4.4 The functions' signatures . . . 25
4.4.1 createState_ function . . . 27
4.4.2 deleteState_ function . . . 27
4.4.3 getInfo_ function . . . 28
4.4.4 init_ function . . . 28
4.4.5 exec_ function for the Function . . . 29
4.4.6 exec_sample_ function for the Function . . . 29
4.4.7 exec_ function for the Gradient . . . 30
4.4.8 exec_ function for the Hessian . . . 31
4.4.9 finalize_ function . . . 31
4.5 Static linking with FORTRAN 77 . . . 32
4.6 Computation with data samples . . . 34
4.7 Computation with a Gradient or a Hessian . . . 34

©2005-2012 EDF - EADS - PhiMeca

5 The wrapper as interface for the external code **36**
5.1 Transmitting values to the code . . . 36
5.2 Regular expressions . . . 36
5.2.1 General . . . 36
5.2.2 Implementation in the description file . . . 39
5.3 Isolating executions . . . 39
5.4 Transmission via files . . . 40
5.5 Transmission using command line arguments . . . 42
5.6 Writing an execution function for an external code . . . 43
5.7 Sharing the internal state . . . 44

6 Future developments **46**

A Functions from the WrapperCommon library **47**
A.1 Display and error functions . . . 47
A.2 Functions manipulating the exchange data structure . . . 48
A.3 Execution isolation functions . . . 49

B Macros from the WrapperMacros library **52**

C Description of the wrapper's development structures **55**
C.1 wrapper_linked_with_C_function directory . . . 55
C.2 wrapper_linked_with_F77_function directory . . . 56
C.3 wrapper_calling_shell_command directory . . . 56

D DTD for the wrapper's description file **57**

©2005-2012 EDF - EADS - PhiMeca

1 Introduction

Among the requirements expressed by the users during the initial design phase of the Open TURNS platform, the wish to undertake uncertainty treatment studies with any code, whether it be as simple as an analytical mathematical formula or a coupling involving several computational codes dedicated to the resolution of a very complex physical problem, was crucial.
Therefore, it was necessary to design a system that can be very flexible, efficient and scalable, in order to answer the users' needs while preserving some fundamental principles of software engineering, in particular the independence of the Open TURNS platform from the code with which it collaborates, and the separation of roles between developers and users.
What can be seen, at first, as a "squaring of the circle" type problem was achieved through this mechanism coupling Open TURNS with computational codes. This "wrapping" process relies on the use of many techniques that are sometimes complex: dynamic library loading, interface definition, delegation of processing, genericity, parallelism, etc.
However, much attention has been paid to simplifying or even hiding all of this, so as to make Open TURNS accessible to anyone wishing to design a wrapper for a computational code. The documentation reflects this care for simplification. However, some prerequisites are essential to fully understand it. This document assumes that the reader is familiar with programming, at least with traditional procedural languages such as FORTRAN and C. Although the Open TURNS platform is mainly written in C++, no specific knowledge of this language is required to design a wrapper: this is the first element of the above-mentioned simplification.
The technique suggested in Open TURNS tends to simplify itself as the product evolves. Some features are replaced, as the project goes along, by simplifications that hide many elements unnecessary to the developer. We invite the reader to always take advantage of these innovations, not least because they bring clarity and maintainability to the source code. However, nothing has been fundamentally modified since the first Open TURNS versions: what has been developed in the past continues to operate as long as all is recompiled with the new version of the platform.

©2005-2012 EDF - EADS - PhiMeca

2 The different types of functions

Complexity does not always come from the technical side. The spoken language is the first difficulty that must be overcome in computer science. In the case we are dealing with (i.e. the use of the Open TURNS platform to carry out uncertainty treatment studies), we are faced with the language's poor ability to express, through specific and distinct words, concepts that are very different from each other.
The difficulty that is hardest to tackle is the use of the word *function*, on which we rely quite frequently. It covers here at least three concepts.
The first meaning that we can associate with the term *function* is mathematical. In the field of the treatment of uncertainties, we are required to manipulate a mathematical function that is most often a physical model relying on input variables and producing output variables. Also known as a (broadly defined) *solver*, it is the component that performs the computation in the code with which the Open TURNS platform was coupled. As a mathematical object, this is a $\mathbb{R}^n \rightarrow \mathbb{R}^p$ function, where n and p repectively represent the number of input and output variables. When the computational code provides the gradient or the hessian of this function, Open TURNS is expected to be able to make good use of it. Gradient and hessian appear in Open TURNS as other mathematical functions of types $\mathbb{R}^n \rightarrow \mathbb{R}^p \times \mathbb{R}^n$ and $\mathbb{R}^n \rightarrow \mathbb{R}^p \times \mathbb{R}^n \times \mathbb{R}^n$.
To bring together these three mathematical functions linked by derivation relations in an object that allows easy handling by the users of the platform, we introduced a second concept of *function* which is mainly related to the field of uncertainty. This function is a computer object which rounds up the three mathematical objects into a consistent whole. This object makes it possible to tell the Open TURNS platform that this gradient function is indeed the gradient of that mathematical function.
The third concept associated with the term *function* is programming related. It is the concept of method in object programming, or of sub-program in procedural programming. This is the place to define the treatments to be carried out.
In this document, in order to distinguish these three concepts, we will use conventions designed to remove any ambiguity.
The concept of mathematical function is called either Function, or Gradient or Hessian (with a capital letter) depending on its role.
The concept of unifying function, bringing together the previous three functions, will be called NumericalMathFunction, NMF in short, so as to remain consistent with the naming of objects within the Open TURNS platform.
The concept of programming function will simply be called function. When we need to describe an example of such function, we will use lowercase and a bold font as follows. Example: **func_exec_m**.
We shall see later that these three function types are closely intertwined in the Open TURNS platform.

©2005-2012 EDF - EADS - PhiMeca

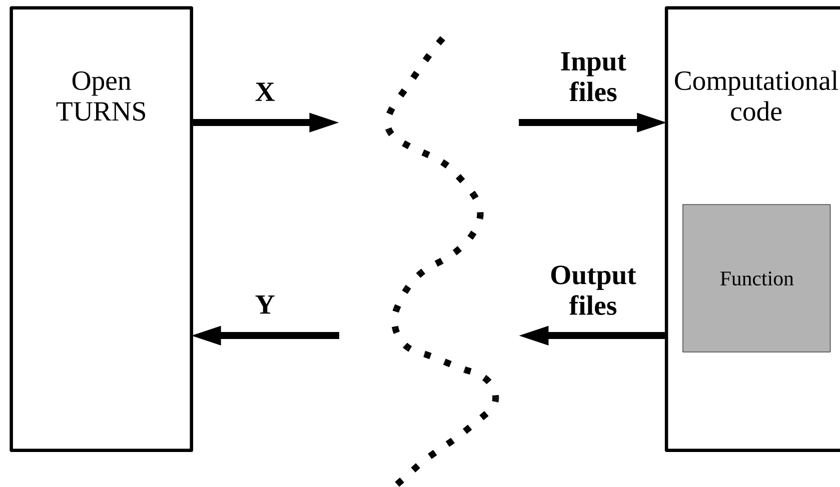


Figure 1: Adapting the interface between Open TURNS and the computational code

3 The wrapper

3.1 Operating principle of a wrapper

3.1.1 Why use a wrapper?

Among the goals that the Open TURNS platform attempts to reach, genericity is central. It is at the heart of the tool as it is at the heart of mathematics, which is the source of all the implemented algorithms. Therefore, an essential part of Open TURNS implies that the problem to be solved, and for which we wish to conduct an uncertainty study, can be modeled as a mathematical $\mathbb{R}^n \to \mathbb{R}^p$ function.
This model, which we chose to call Function, can be implemented in various ways.
It can be directly encoded within the library that forms the base of the platform. Currently, and even if it is easily feasible, no Function belongs to this category. The reason is primarily practical. Which Function can be hard-coded? Why this one rather than another? Given that the tool is generic and meant to be broadly disseminated, a choice made for our needs could be a problem in another context. This is why we refused to register the Functions at the library level.
It can also be coded within an analytical formula driven by a NumericalMathFunction. This will not be detailed here. It suffices to know that the said analytical formula is not hard-coded but passed as an argument of the NumericalMathFunction, which is able to interpret and execute it. Thus, the platform can work with almost any formula given by the user.
Advancing still further towards the externalization of the computation from the library, the Function can be implemented in a subroutine that will be dynamically (i.e. on demand) linked to the platform during computation.
Finally, it is possible to use this subroutine not to perform the computation, but to delegate this computation to an external entity, that is to say, a computational code (see Figure 3.1).
We are therefore in a situation in which Open TURNS only sees the Function hosted within the computational code as an object consuming a set of n values (a vector $\underline{X}$ belonging to $\mathbb{R}^n$) and producing a set of p values

©2005-2012 EDF - EADS - PhiMeca

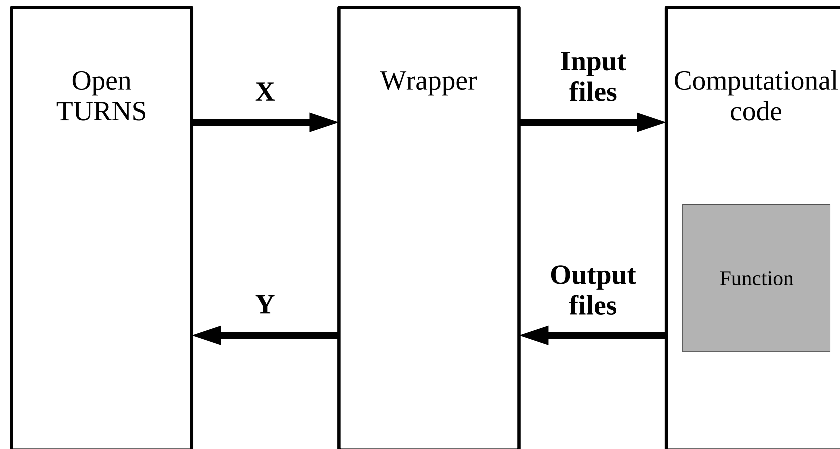


Figure 2: Coupling a computational code using a wrapper

(a vector $\underline{Y}$ belonging to $\mathbb{R}^p$), while the computational code usually expects to receive input files and produce output files.
These files contain much more information than the n or p values manipulated by Open TURNS. Moreover, they have a format that is specific to each computational code.
Therefore, it is difficult or even impossible for Open TURNS adapt to all situations. It is not possible either to impose a coupling mode and force the computational codes to follow it.
If we look closer, we notice that, on the one hand, Open TURNS expects the computational code to respect its interface, that is to say that it transmits the values through vectors $\underline{X}$ and $\underline{Y}$ and that, on the other hand, the computational code expects Open TURNS to send values through its interface, that is to say its files.
The "wrapper" is precisely the subroutine that converts the interface of one element into an interface of the other, carrying out the necessary transformations on the data. In this document, we will use the word wrapper, although we could also use the more explicit but less common term "interface adapter" (see Figure 3.2).
Of the three scenarios outlined above, the wrapper is only involved in the last two. This document therefore only addresses these two cases.
As an adapter, the wrapper is not meant to change the way the user sees the computational code with which they are working. The wrapper should thus be transparent in the exchange between Open TURNS and the external code. It behaves as an extension of the code towards the platform.
Beyond its role of interface adaptation, the wrapper also ensures the cooperation between a platform written in C++ and codes written in any other programming language (C, FORTRAN 77 or 90, Python, etc.). This will be detailed later on in this document.
It is also in charge of managing the execution of the computational code by invoking its commands or functions at the appropriate time.
In summary, the wrapper is a variable geometry interconnection element between Open TURNS and the computational code. It can be designed to provide basic or very advanced functionalities. Being specific to each code, it is its designer's responsibility to determine its capabilities.
This documentation is structured following a gradual progression. It begins by showing how to create a minimal

©2005-2012 EDF - EADS - PhiMeca

wrapper with the most basic behavior, which will, however, be adequate in many cases; then, as the reader advances in the text, they will be able to see how to add features enriching the interface. This progression can also be regarded as a design guide for the wrapper, which will be developed in stages.

3.1.2 Presentation of the wrapper

Before turning to the next section, which will show how to use a wrapper, it is useful to present the elements that make it up.
From a strict computer science point of view, the wrapper consists of two files:

- a dynamic library compiled from a source file written in C language (eg **wrapper.so**)
- a file describing the library in XML (eg **wrapper.xml**).

Unless otherwise specified, when we speak of the wrapper in this document, we always refer to both of these files.

3.1.3 The dynamic library

The dynamic library contains the code performing all of the interfacing operations. It is loaded only when using the wrapper, which means that, when Open TURNS starts, it has no ability to interface with any computational code. It acquires this ability by loading a wrapper. It is possible to simultaneously load several different wrappers in order to work with several computational codes. However, the dynamic library load mechanism prevents the coexistence of multiple instances of the same wrapper. Once loaded, a wrapper cannot be loaded a second time unless it has been unloaded first. This is why dynamic libraries are also called shared libraries, and the corresponding files have the Unix extension **.so** for "shared object". This file can have any name but, in this document, we choose to call it **wrapper.so** for convenience when we talk about it in generic terms.

3.1.4 The description file

The previous section indicated that the wrapper could be gradually expanded with additional features. However Open TURNS can not easily infer, from the library that it loads, what these capabilities are and whether they should be used. That is why we add to the library file a description file written in XML which, as its name suggests, is there to inform the platform on the features offered by the wrapper library. For convenience and symmetry, we choose to name this file **wrapper.xml** in this document.

> *Note: The choice of names is only intended to clarify our text. However, it is very inappropriate to use these names for a real wrapper intended to be distributed to users. Not only wouldn't the name be explicit, but that wrapper would likely clash with a counterpart carrying the same name but responsible for something completely different.*
> *Therefore, we recommend naming the said files using the external code to be "wrapped". For example, the developer of a wrapper for the Octave software could name these files* **octave.so** *and* **octave.xml**.

3.1.5 Invoking and loading a wrapper

In order to gradually detail the way the wrapper operates, we will primarily focus on how it is used.
We adopt here the perspective of the user who seeks to carry out an uncertainty treatment study using a computational code that we will call **Code_X**. We do not concern ourselves yet with how to exchange variables with **Code_X**. This will be the subject of later sections.

©2005-2012 EDF - EADS - PhiMeca

All that interests us at this stage is to know that **Code_X**, seen from Open TURNS, behaves like a $\mathbb{R}^n \rightarrow \mathbb{R}^p$ Function. It will therefore be used as follows[1]:

```
>>> f = NumericalMathFunction ( "Code_X" )
```

A function **f** interfaced with **Code_X** is created using the coupling mechanism of Open TURNS.
Behind this very simple syntax hides a relatively complex series of operations that we will describe. We assume that the user is familiar with the object syntax of Python or C++ and with the standard classes of the Open TURNS library[2].
The NumericalMathFunction class allows to create programming objects mimicking the behavior of a real mathematical Function associated with its Gradient and its Hessian. It is thus possible to require an object of this type to evaluate itself at a given point or to provide the value of its gradient and its hessian at this point. This mechanism simulates the derivation relations between these three concepts.
The above syntax calls the constructor of the NumericalMathFunction class, passing a string to it as an argument, string indicating the code with which the user wants to interface. Following this call, and assuming that no error has occurred, the wrapper for **Code_X** will be loaded and the connection between the object **f** and the wrapper will be established.
This operation is carried out in six steps.

3.1.6 Step 1: Creation of a NumericalMathFunction

Everything starts with the creation of the NumericalMathFunction object within the Open TURNS library. Although the previous action is initiated from the Python language, it is immediately transcribed into C++ code executed within the library (see Figure 3.3).

3.1.7 Step 2: Search for the description file

The constructor of the NumericalMathFunction class therefore receives a string which it interprets as the filename of the wrapper's XML description file (but without its **.xml** extension). Thus the string **Code_X** matches the filename **Code_X.xml**.
After completing the name to restore its entirety, the file is searched on the computer's filesystem in a predefined list of directories (see Figure 3.4). These directories are, in order:

1. those contained in the **OPENTURNS_WRAPPER_PATH** environment variable, separated by a colon (**:**) and read from left to right (if the variable does not exist or if its content is empty, this step is ignored);
2. the directory **$HOME/openturns/wrappers** if available;
3. the wrapper installation directory of the library, **<openturns_dir>/share/openturns/wrappers**, where **<openturns_dir>** is the installation directory of the Open TURNS platform.

If the file cannot be found, an error is generated and the **f** object is not created, eg:

```
>>> f = NumericalMathFunction('NoFile')
Traceback (most recent call last):
File "<stdin>", line 1, in ?
```

[1] For the examples, we use the syntax of the Python text interface. The C++ syntax is very close and does not provide any additional information.
[2] For details about the classes, see the Open TURNS Reference Guide, available at **http://www.openturns.org** or the online help of the platform.

©2005-2012 EDF - EADS - PhiMeca

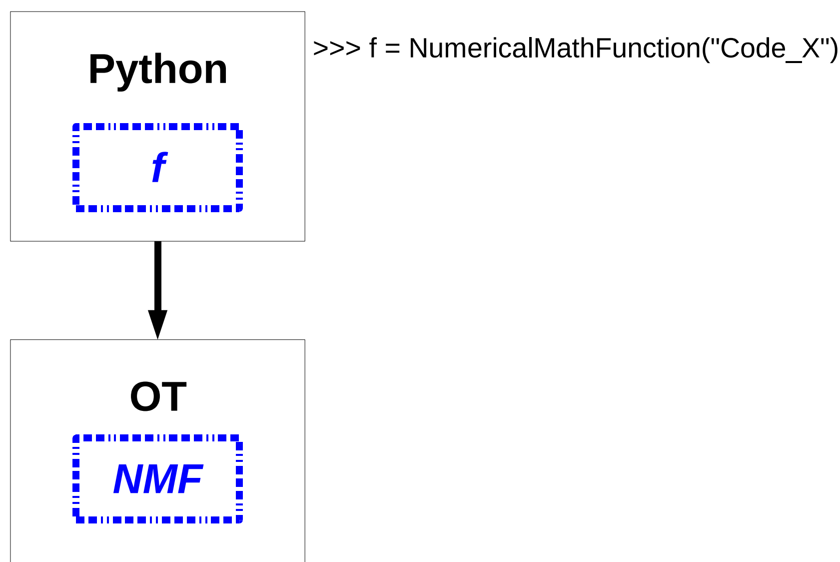


Figure 3: Creating a NumericalMathFunction

```
File "/local00/home/dutka/OpenTURNS/dutka/devel/build_3.4/install/lib/python2.
  4/site-packages/openturns/ot.py", line 11035, in __init__
this = _ot.new_NumericalMathFunction(*args)
RuntimeError: NoWrapperFileFoundException : FileNotFoundException : No file name
d 'NoFile.xml' was found in any of those directories : /local00/home/dutka/opent
urns/wrappers /local00/home/dutka/OpenTURNS/dutka/devel/build_3.4/install/share/
openturns/wrappers
```

3.1.8 Step 3: Analysis of the description file

However, if the file is present, it is read. Any reading error causes a program error and the **f** is not created. The details of this description file will be discussed later but, at this stage, it suffices to know that it contains the path to the **Code_X.so** dynamic library (see Figure 3.5).

3.1.9 Step 4: Search for the dynamic library

In turn, this library will be searched for in the same directories. If it is found, it is automatically loaded by Open TURNS(see Figure 3.6). Its absence causes an error.

3.1.10 Step 5: Loading the dynamic library

Once the library is loaded, it is tested to uncover some of its characteristics (see Figure 3.7). We will detail this point in section 4.4.3, which deals with the **getInfo_** [3] method.

[3] At this stage of the document, the real names of the wrapper's functions have not been described yet. In fact, the **getInfo_** method carries a more elaborate name, with other elements as prefixes and suffixes. This will be described later on in the document. We will use the shortened name in order to simplify the text, as long as it does not raise any ambiguity.

©2005-2012 EDF - EADS - PhiMeca

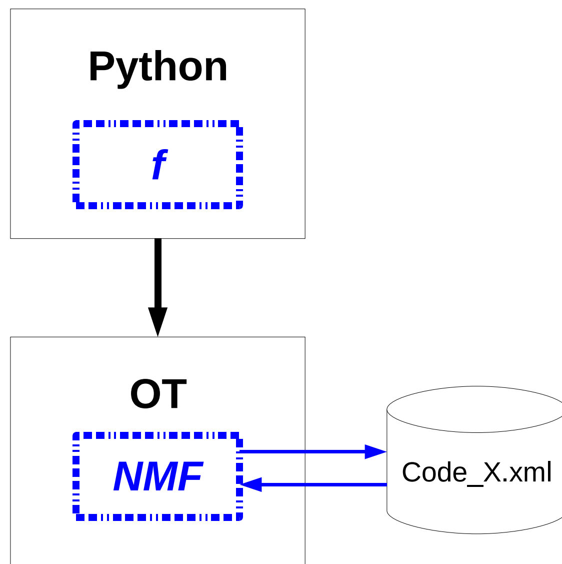


Figure 4: Search of the wrapper's description file

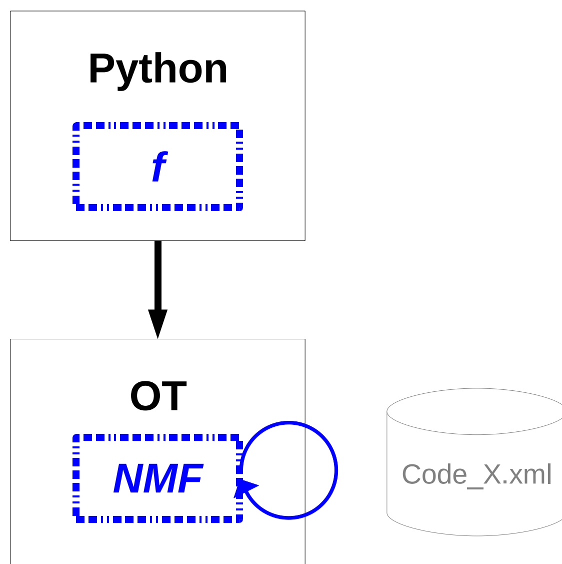


Figure 5: Reading and analysis of the wrapper's description file

©2005-2012 EDF - EADS - PhiMeca

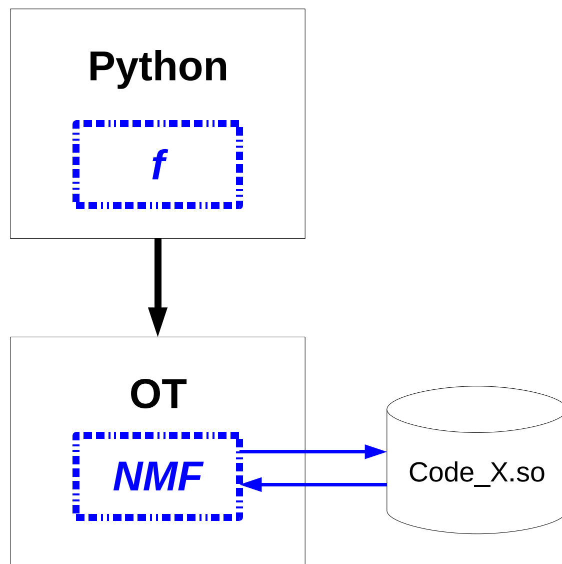


Figure 6: Search for the wrapper's dynamic library

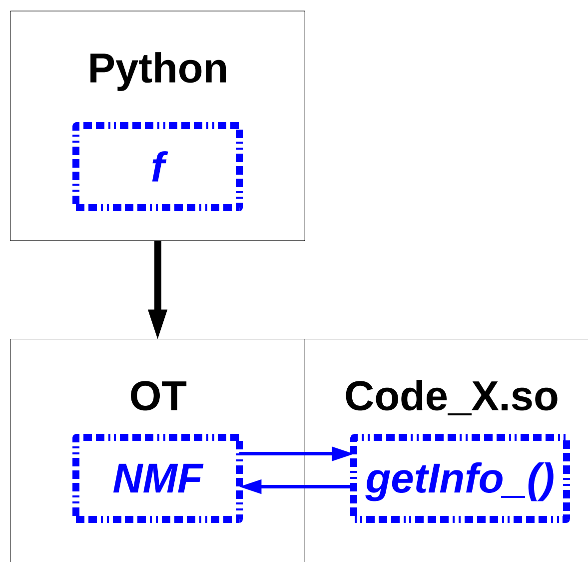


Figure 7: Loading and querying the dynamic library

©2005-2012 EDF - EADS - PhiMeca

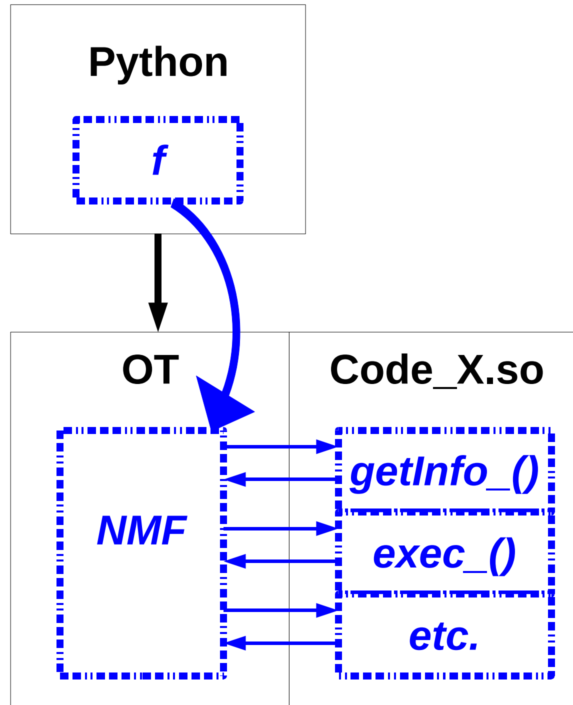


Figure 8: Completion of the NumericalMathFunction construction

3.1.11 Step 6: Completion of the NumericalMathFunction construction

Finally, Open TURNS maintains an internal link to the wrapper's dynamic library and returns a properly constructed and initialized NumericalMathFunction object to the user (see Figure 3.8). It is therefore possible to use it for the study of uncertainties.

3.2 Example: Creating a minimal wrapper

To make this document more concrete, this section shows how to create a "minimal" wrapper. It will allow to browse the description file (see Section 3.2.4) and the source code in detail (see Section 3.2.5). It will especially help distinguish the elements that *must* be present within any wrapper from elements that are optional and are only enriching features.

3.2.1 Required Items

In order not to unnecessarily complicate this first example, we will limit the Function implemented in the wrapper to a simple $\mathbb{R}^3 \rightarrow \mathbb{R}^2$ analytical formula:

$$P \left\{ \begin{array}{l} y_1 = x_1 \times e^{(-x_2 \times x_2)} \\ y_2 = x_2 + x_3 \end{array} \right.$$

©2005-2012 EDF - EADS - PhiMeca

We call this Function **P**.
This Function is currently implemented directly in the source code of the wrapper. Later examples will show how this function can be extracted from the wrapper's body into an external code.
To create a wrapper based on **P** which can be used by Open TURNS, it is necessary to provide the library with the following elements:

- the path to the shared library that implements **P**, **P.so**;
- the list and description of variables exchanged between the wrapper and Open TURNS, namely input variables (x_1, x_2 and x_3) and output variables (y_1 and y_2) of the Function;
- the names of the Function within the wrapper, here **P**, because the same wrapper can technically implement multiple Functions;
- the name of the Gradient or the Hessian, if they are present in the wrapper, associated with the Function;
- information on how P is implemented in the wrapper.

In our example, we will provide neither the Gradient nor the Hessian for **P**, although this is quite feasible since P is very simple to derive. We will see later that the Open TURNS library adapts itself and produces on its own a Gradient and a Hessian with centered finite differences.

3.2.2 Starting from a template to generate the wrapper

It is not easy, for a wrapper designer, to produce a first functional draft based on the analytical formula alone. Therefore, all versions of Open TURNS offer by default a completely independent structure for wrapper development that allows to easily produce a wrapper.
Several development structures are offered in order to replace the old structure. The new structures are now available in the **share/openturns/WrapperTemplates/<wrapper_example>** subdirectory, where **<wrapper_example>** can be:

- **wrapper_calling_shell_command**: structure for the development of a wrapper launching the external code via a shell command line;
- **wrapper_linked_with_C_function**: structure for the development of a wrapper calling a C function;
- **wrapper_linked_with_F77_function**: structure for the development of a wrapper calling a FORTRAN 77 function.

We copy the **wrapper_linked_with_C_function directory** as **Pwrapper**:

```
> cp -r <openturns_dir>/share/openturns/WrapperTemplates/wrapper_linked_with_C
_function $HOME/Pwrapper
```

Then, just use the **customize** script which automatically renames the files with the name passed as an argument:

```
> ./customize P
```

©2005-2012 EDF - EADS - PhiMeca

3.2.3 Compilation and installation of the wrapper

Before going into detail on how to write the wrapper, we need to prepare the development environment which will conduct the compilation and installation of the library and of its XML file.

> *Note: To be able to use the* **Pwrapper** *user development directory, the developer must have access to three GNU Open Source tools known as* **autoconf**, **automake** *and* **libtool**. *These tools are commonly installed on all GNU/Linux distributions. However the installed versions may be older and it is always good to have the latest available update*[4].
> *The user must also have a C or C++ compiler in order to produce the wrapper's dynamic library. The GNU Open Source compiler* **gcc** *is a very good choice.*

The first step is to bootstrap the environment. To this end, the following command must be executed once:

```
> bootstrap
```

Then we determine which tools are available on the computer in order to compile and install the wrapper. This configuration step is also carried out only once:

```
> ./configure --with-openturns=<openturns_dir>
```

By default, the wrapper is installed in the directory **$HOME/openturns/wrappers** which is a standard search directory of Open TURNS.

> *Note: If the wrapper must be installed in a different directory, it is necessary to use the option* –**prefix** = <**Pwrapper_dir**> *where* <**Pwrapper_dir**> *is the directory where the wrapper will be installed (later on). When using the wrapper, the path to* <**Pwrapper_dir**> *should be positioned in the environment variable* **OPENTURNS_WRAPPER_PATH**.

When the source files needed to produce the wrapper are written (see sections 3.2.4 and 3.2.5), it will be possible to compile and then install the wrapper using the commands:

```
> make
> make install
```

3.2.4 Description file for the minimal wrapper

We begin by writing, or rather by modifying the description file **P.xml**.
This XML file is highly structured and must contain a certain number of tags. This rigid syntax is checked by a grammar (DTD) described in the **wrapper.dtd** file. This DTD is provided by default by Open TURNS[5].
Any error in the syntax or the structure of the XML file results in immediate rejection by the Open TURNS library and prevents from loading the wrapper.
The **P.xml** file contains the following tags:

<**wrapper**> This tag introduces the general description of the wrapper.

<**library**> This tag introduces the section dealing with the dynamic library as seen from the Open TURNS-standpoint.

<**path**> This tag allows to indicate to Open TURNS where to find the wrapper's dynamic library, **P.so** in our convention. It thus reads:

[4] See the website **http://www.openturns.org** to check the tool versions adapted to the Open TURNS version you are using.
[5] The file should normally be found in the <**openturns_dir**>/**share/openturns/wrappers** directory.

©2005-2012 EDF - EADS - PhiMeca

```
<wrapper>
<library>
<path>P.so</path>
...
```

The name of the dynamic library is marked by the opening tag **<path>** and the closing tag **</path>** following the standard XML syntax. Be careful not to introduce whitespaces or any other separator, for all characters between the opening and closing tags *will* be considered as the library name. The library will be sought in the same directory as the XML file (see Section 3.1.3) because the system assumes that both files are placed in the same directory.

If the library name is preceded by a relative path, this path will be relative to the above-mentioned search directories. Absolute paths are relative to the computer's file system.

<description> This tag introduces the definition of the Function, the Gradient, the Hessian and of the exchanged variables.

<variable-list> This tag introduces the list of input and output variables of the wrapper.

<variable> This tag allows to define a wrapper input or output variable as shown in the following example.

```
...
<description>
<variable-list>

<!-- three input variables -->
<variable id="x1" type="in" />
<variable id="x2" type="in" />
<variable id="x3" type="in" />

<!-- two output variables -->
<variable id="y1" type="out" />
<variable id="y2" type="out" />

</variable-list>
...
```

It is used to declare the three input variables (**x1**, **x2** and **x3**) and two output variables (**y1** and **y2**) of the **P** Function. The **id** attribute specifies the variable name and the **type** attribute, its nature (**in** for input and **out** for output, from the point of view of **P**). We remind here that the exchanged variables are doubles.

It is a good practice to list the variables in their natural order, input before output. This order is used internally by Open TURNS so as to determine to which variable it must assign the values of the input and output point components when evaluating the Function. Thus, the first component of the point will be assigned to the first listed variable, the second component to the second listed variable, and so on.

Each variable must have a unique identifier (**id**). If this is not the case, the description file is rejected.

We will see later on that the **<variable>** tags may be supplemented by sub-tags providing additional information.

©2005-2012 EDF - EADS - PhiMeca

<function>, <gradient>, <hessian> These three tags are used to respectively define the names of the Function, the Gradient and the Hessian. In our example of a minimal wrapper, only the Function is defined under the name **P**. The name is written between the opening and closing tags without spaces or any other superfluous character.

```
...
<function provided="yes">P</function>
<gradient provided="no" />
<hessian  provided="no" />
</description>
</library>
...
```

The **<function>**, **<gradient>** or **<hessian>** tags include a **provided** attribute that indicates whether the Function, the Gradient or the Hessian is implemented in the wrapper (**yes**) or not (**no**). In the latter case, the name need not be specified.

<external-code> This tag introduces the definition of the coupling between the wrapper and the external code.

<data> This tag introduces the definition of the data exchanged between the wrapper and the external code. In our minimal example, there really is no external code since the wrapper will play this role without resorting to some other object. This section will therefore remain empty.

The remaining information to be given to Open TURNS deals with the way the wrapper internally implements the Function. The presence of this information is justified for the development of a generic wrapper, but is useless in the example at hand. Yet it is still necessary to provide these pieces of information in order to have a valid description file. Future developments of Open TURNS will eliminate this dependence.

<wrap-mode> This tag indicates the way the wrapper internally implements the Function.

The **type** attribute indicates that the Function is directly coded within the wrapper (**static-link**). The sub-tags **<in-data-transfer>** and **<out-data-transfer>** specify that the variables must be passed to the Function as arguments.

<command> This tag describes the system command that calls the external code when it is fully independent and disjoint from the wrapper. Here, since the wrapper is minimal, no command is defined.

```
...
<external-code>
<data />

<wrap-mode type="static-link">
<in-data-transfer  mode="arguments" />
<out-data-transfer mode="arguments" />
</wrap-mode>

<command />

</external-code>
</wrapper>
```

©2005-2012 EDF - EADS - PhiMeca

Finally, the description file **P.xml.in** for our minimal wrapper looks like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE wrapper SYSTEM "@OPENTURNS_WRAPPER_DTD_PATH@/wrapper.dtd">

<wrapper>
<library>
<path>P.so</path>
<description>
<variable-list>
<variable id="x1" type="in"  />
<variable id="x2" type="in"  />
<variable id="x3" type="in"  />
<variable id="y1" type="out" />
<variable id="y2" type="out" />
</variable-list>
<function provided="yes">P</function>
<gradient provided="no" />
<hessian  provided="no" />
</description>
</library>
<external-code>
<data />
<wrap-mode type="static-link">
<in-data-transfer  mode="arguments" />
<out-data-transfer mode="arguments" />
</wrap-mode>
<command />
</external-code>
</wrapper>
```

3.2.5 Source code for the minimal wrapper

The second file needed to produce the wrapper is the source code of the dynamic library (**wrapper.c**). This source code is used to carry out the interface adaptation required by the data transfer between Open TURNS and the Function to implement. It could theoretically be written in any language evolved enough to perform such operations. But the fact that the wrapper's library is dynamically loaded, and that Open TURNS must search for symbols in it and be able to call them, requires the use of a computer interface based on C. This sets strong constraints on the languages that can be used to code the wrapper. Only C and C++ are easily usable for this task.

> *Note: This constraint does not mean that the developer should be a C or C++ expert to be able to develop a wrapper. The engineering work done in Open TURNS helped greatly simplify the writing of the wrapper, and any developer used to a procedural language (FORTRAN or other) is able to generate a wrapper. However, they must obey the rules imposed by Open TURNS.*

The first thing to do, in the source file **wrapper.c**, is to include the header file which provides definitions needed for compilation:

```
#include "Wrapper.h"
```

©2005-2012 EDF - EADS - PhiMeca

```
...
```

These two files are provided by default in the Open TURNS installation.
The file **Wrapper.h** is absolutely essential and should always be included in the source code of the wrapper. It gathers all the definitions that allow communication between the Open TURNS library and the wrapper. Without this file, no compilation of the wrapper is possible.
It also contains the declarations of functions that are practical and shared by many wrappers. These functions are part of the platform with which the wrapper connects itself. The use of these functions greatly simplifies the writing of wrappers by performing the most common actions in a simple and efficient way. This documentation largely relies on them. The first appendix describes the functions in this library.
Let's recall the definition of our **P** Function:

$$P \begin{cases} y_1 = x_1 \times e^{(-x_2 \times x_2)} \\ y_2 = x_2 + x_3 \end{cases}$$

Among the different techniques available to implement the **P** Function in the wrapper, we choose to isolate the computation in a separate C function which we call **myP**. The function being multidimensional $\mathbb{R}^3 \to \mathbb{R}^2$, the C function has been designed to accept an array of 3 doubles as input and an array of 2 doubles as output.
We assume that the arrays are already allocated with the proper size (respectively n and p) and that only the output values are to be modified.
We also use the C function return value to indicate a possible miscalculation. Here, since the calculation is always possible, the function systematically returns 0 to indicate that no error occurred (this is a C convention). In case of an error, such as a division by zero or an argument outside the Function's definition domain, the function needs to return a non-zero return code.

```
...
#include <math.h>

int myP ( double * x, unsigned long n,
double * y, unsigned long p )
{
  y[0] = x[0] * exp( - x[1] * x[1] );
  y[1] = x[1] + x[2];
  return 0;
}
...
```

In the example, we used the names **x** and **y** to refer to the input variables (**x1**, **x2** and **x3**) and output variables (**y1** and **y2**). However, seing that the C language requires to start numbering array indices from zero: **x[0]** corresponds to **x1**, **x[1]** to **x2**, etc.

Note: Inclusion of the **math.h** *file is justified by the use of the* **exp** *function.*

Having this **P** Function, we now need to interface it with Open TURNS. This is done using an auxiliary function that will ensure communication with the Open TURNS library.
This function is commonly called **exec_**, although its name is more complex as shown by the following example:

```
...
enum WrapperErrorCode func_exec_P ( void * p_state,
const struct point * inPoint,
struct point * outPoint )
```

```
{
  if ( myP (inPoint->data_,  inPoint->size_,
  outPoint->data_, outPoint->size_ ) ) return WRAPPER_EXECUTION_ERROR;

  return WRAPPER_OK;
}
```

The **exec_** function is responsible for calling the previously written **myP** function and passing to it the ad-hoc arguments that it will search within the data structures passed as arguments. These data structures are already allocated and filled (for those who may be) so that the functions of the wrapper simply have to fill them. No memory management is to be performed at this level in the wrapper, which makes it very simple to use.
We have seen that the **myP** function could, in case of failure, warn the calling code about it. This case is treated by the test (**if**) which returns a **WRAPPER_EXECUTION_ERROR** error code to Open TURNS. Otherwise the **WRAPPER_OK** code indicates that everything went smoothly.

3.2.6 Checking if the minimal wrapper is functioning

This being written, we saw that we only needed to compile and install the wrapper to run it (see sections 3.2.2 and 3.2.3). If the wrapper was installed in a standard Open TURNS directory or if the **OPENTURNS_WRAPPER_PATH** environment variable was correctly set, it is possible to test its basic operation by using a small Python script (eg **test_P.py**) which is executed as follows:

```
> python test_P.py
```

Congratulations! You have written the smallest possible wrapper and you have interfaced it with Open TURNS. After loading the entire Open TURNS library (**import**), the script creates a NumericalMathFunction type object (**P**) which loads the wrapper that we just built. It then evaluates this P object by passing a NumericalPoint (**x**) to it as an argument. This action causes the execution of the C function **myP**, computing the analytical formula written in the source code of the wrapper, and the return of the value of the computed NumericalPoint (**y**).
Essentially, the script **test_P.py** carries out the following series of actions:

```
>>> from openturns import *
>>> P = NumericalMathFunction ( "P" )
>>> x = NumericalPoint ( 3 )
>>> x[0] = 10.
>>> x[1] = 20.
>>> x[2] = 30.
>>> print x
 [...]
>>> y = P ( x )
>>> print y
 [...]
```

©2005-2012 EDF - EADS - PhiMeca

4 The structure of a wrapper

The rest of this document presents extensions to the mechanism described in the previous section, as well as the progressive externalization of the computation into an external code. Through these enhancements, we will see in detail the structure of the wrapper and the range of possibilities offered to the developer. If the example of the minimal wrapper allows to very quickly and easily test the interfacing of a formula with Open TURNS, the techniques described in this section will guide the reader towards features of progressive difficulty. It is important to consider that these features are mostly optional and are mainly improvements to the interfacing of the computational code with Open TURNS. Therefore, the developer can intentionally forget about them at first and start with a simple wrapper. The complexity will come later according to needs and developments of the computational code.
In short, the interfacing mechanism of a code with Open TURNS is scalable and the developer only needs to produce the functions necessary for their code.
This touches one of the basic criteria that have guided the development of the tool: genericity. Open TURNS has been designed as a tool that can interface with almost any computational code. But they can be very diverse and the wrapper for one code can be very different from the wrapper for another. Therefore, we had to invent a mechanism that satisfies both without excessively penalizing either one. This section will therefore lead us into considerations that the reader may, at first, see as complex or unnecessary. However, they are the result of specific requests for given codes. They are thus useful. Their complexity has been reduced as much as possible but it cannot always be hidden. The keyword for the reader here is to ignore what does not primarily concern them, in order keep the wrapper simple and suited to their needs.

4.1 Naming conventions

When the Open TURNS platform loads the wrapper's dynamic library (**wrapper.so**), it is confronted to an object that it must identify as a code wrapper and, moreover, as the wrapper for the right computational code. It may happen that another software also uses a library called **wrapper.so** and that this library be placed in an Open TURNS search path, in which case the loading succeeds but the library is unusable. It is also possible that the **wrapper.so** library is the wrapper for another code interfaced with Open TURNS. This happens when developers do not change the name of the library to match the name of their code: everyone uses the same name.
In order to detect some of these cases and avoid unnecessary computations run on unwanted codes, it is necessary to allow Open TURNS to check that the library being loaded is the expected one. This is done by requiring the wrapper to use names whose structure ensures (to some extent) an unambiguous identification.

4.1.1 In the wrapper's source file

The names of the proposed C functions are built on three elements:

- a prefix (**func_**, **grad_** or **hess_**) that indicates whether the function relates to the Function, the Gradient or the Hessian of the NumericalMathFunction;
- a central term which provides information on the treatment carried out by the C function; this will be detailed in Section 4.3;
- a suffix that specifies what Function, Gradient or Hessian is actually implemented by the C function.

An example is more eloquent. Consider again what has been shown in section 3.2.5 on the wrapper's source code. We used two C functions named **func_getInfo_P** and **func_exec_P**. The name of the **func_getInfo_P** function indicates that it informs Open TURNS about the parameters (**getInfo_**) of the Function (**func_**)

©2005-2012 EDF - EADS - PhiMeca

which implements the P Function (**P**), while the **func_exec_P** function name indicates that it carries out the computation (**exec_**) of the function (**func_**) which implements the P Function (**P**).
We will see later that the wrapper is divided into three main parts, the Function, the Gradient and the Hessian, which are very similar. Each of these parts is further divided into C functions, whose number may vary. These functions are distinguished from each other by the central term, which describes their role within the section.
The introduction of the name of the Function, Gradient or Hessian as a suffix results from a dual intent. First of all, it allows to clearly identify the computation carried out by the function and, therefore, to check that the loaded wrapper is the right one. It also allows to build a wrapper having multiple Functions, Gradients and Hessians in the same source code, in order to simultaneously have several implementations:

- of the same code, as is the case when an implementation is better than another, but both are useful, or that each implementation is used in different contexts; for example, each of them executes the code on a different machine or in a different version;
- of different codes; one might want to put in the same wrapper implementations related to different codes but most often executed together, as is the case when one sets up couplings.

4.1.2 In the wrapper's description file

The name of the Function is introduced by the string given between the opening tag <**function**> and the closing tag </**function**> of the wrapper's description file. It is important to not insert any whitespace, blank or other separator in the name because it would prevent Open TURNS from building a correct name and jeopardize the loading of the dynamic library wrapper.
Symmetrically, the name of the Gradient or Hessian is respectively introduced by the <**gradient**> and </**gradient**> tags or by the <**hessian**> and </**hessian**> tags, and it follows the same rules.

4.2 Sequence of steps

Section 3.1.3 has shown the procedure for loading a wrapper from the Open TURNS library. This initial stage is essential to any future use of the wrapper and, thus, of the code linked to it. The explanation given in this section therefore assume that this first stage is reached and that the NumericalMathFunction associated with the wrapper is operational. We will therefore examine the various activities undertaken during the use of the NumericalMathFunction.
Experience shows that the execution of a computation, even when it is simple, can be decomposed into a sequence of steps. This is more obvious when calling a computational code than an analytical formula, so we will use this first example as a guideline.
Assuming that the computational code to call in order to perform an Open TURNS computation is called **Code_X**, that it takes as input one (or more) data file(s) and produces as output one (or more) result file(s), which we will arbitrarily call **fich.dat** and **fich.res**, we can segment the following code call[6]:

```
> code_X < fich.dat > fich.res
```

as follows:

1. **code_X** reads its input file **fich.dat**;
2. **code_X** runs using the data read and produces a result;
3. **code_X** writes its result in the output file **fich.res**.

[6]The use of stream redirections, which is specific to Unix, is used here only to show the direction of the exchanges. Any other syntax would have been appropriate, since the idea remains the same.

©2005-2012 EDF - EADS - PhiMeca

Although not universal, this pattern is representative of a vast majority of computational codes.
When several computations based on **code_X** must be chained, as is almost always the case in uncertainty studies, we can see that these three steps will be repeated a number of times, up to several thousands. Yet it may be that certain steps are exactly repeated from one execution to the next. For example, a mechanics code may very well read several input files, a physical data file and a geometry file. If the probabilistic study covers only the physical data, reading the geometry file will be identical for each computation, but this reading can be highly time consuming. It is therefore interesting to isolate the processing in order to only chain the reading of the physical data and the computation itself.
Open TURNS enables a computational code to isolate one processing relative to another and to call it only as many times as is necessary. This, of course, assumes that the computational code has that option. In other cases, the processing will be carried out as a block, without the ability to segment stages.
This ability of Open TURNS to exploit the phasing of a code will maximize performance and achieve better uncertainty studies. However, it does not imply that Open TURNS only works with such segmented codes. This segmentation is only an optimization and traditional codes interface well with the techniques described in this document. They just consume more time to carry out the same studies.

4.3 The processing/step combination

This section will show that the segmentation of the computational code call can be broken down into seven separate steps. Only two of these steps are really necessary, the other being compensated when absent.
As shown in the previous section, the natural phasing exhibits three stages:

1. reading the input data, which can be likened to the preparation of a computation to come and that we will call *initialization*;
2. the actual computation, which we will call *execution*;
3. collecting the results and outputting them with the right format and at the right place, as well as freeing the resources consumed, which we will call *finalization*.

These three stages—initialization, execution and finalization—are the heart of any computation. They do not, however, play the same role. Users wishing to carry out many computations want to chain the executions but also want to benefit from the optimizations of Open TURNS, that is to say they do not want to run as many initializations and finalizations as executions. Therefore, the initialization and finalization are performed only once during the life of a NumericalMathFunction: initialization is carried out before performing the first run, while the finalization takes place after the last execution.
Thus, in the example below, three successive calls to the NumericalMathFunction **f**:

```
>>> f = NumericalMathFunction ( "Code_X" )
>>> # (We create three NumericalPoints i1 , i2 , i3 , correctly initialized)
>>> f(i1)
>>> f(i2)
>>> f(i3)
>>> # (End of script)
```

lead to the following calls:

f(i1) initialization of **f** and execution of **f** at point **i1**;

f(i2) execution of **f** at point **i2**;

©2005-2012 EDF - EADS - PhiMeca

f(i3) execution of **f** at point **i3** and finalization of **f**.

From a practical standpoint, these three steps are performed within the wrapper, each by a specific C function. Initialization relies on a C function whose central term is **init_**; execution relies on a C function whose central term is **exec_**; finalization relies on a C function whose central term is **finalize_**.
In the source code of a wrapper implementing the Function of a code named **code_X**, we therefore find the following functions: **func_init_code_X**, **func_exec_code_X** and **func_finalize_code_X**.
But since only the execution part of the Function is essential for the wrapper, the other two functions can be removed from the file. The functions **func_init_code_X** and **func_finalize_code_X** may not appear in the source code. In this case, the three phases—initialization, execution and finalization—are to be either gathered within the sole function **func_exec_code_X** or grouped within the computational code that does not distinguish them.
There is a variation in the pattern we just described. When the NumericalMathFunction is called with a sample (a collection of NumericalPoint), if the wrapper provides a C function whose central term is **exec_sample_**, then this term is called instead of the **exec_** function. If the **exec_sample_** function is absent, the **exec_** function is then called as many times as necessary to perform the same task. The initialization and finalization are still called once, respectively at the beginning and at the end of the processing.
Section 3.2.5 presented a function whose central term is **getInfo_**. This query function allows the wrapper to provide additional information to the Open TURNS library, information which would be available only when loading the wrapper. In particular, nothing prevents the wrapper from reading a file or connecting to a network service in order to obtain information necessary to its execution.
As it stands, the only piece of information required by the Open TURNS library is the size of the Function's input point and output point. Both values must be written in a pre-allocated structure of **WrapperInformation** type, which is passed as an argument to the **getInfo_** function. These sizes can either be hardcoded into the source code of the wrapper, which we do not recommend, or computed on the fly based on the content of the wrapper's description file. This is performed by the **getNumberOfVariables** function of the wrapper's library, described in Appendix A.1.
Five of the seven functions have been presented. The two remaining ones rely on more advanced programming concepts whose use is required by the performance and genericity constraints of the Open TURNS platform. Both goals pushed towards the use of parallelism and dynamic wrapper loading. However, the rules of parallel code development tend to ban the use of static variables, more commonly called global variables. These variables are often useful for storing information from one function call to the next and thus avoid computing again a value that was already generated. However, the use of static variables causes access conflicts when multiple simultaneous executions—concurrent executions—attempt to access these variables. Without any appropriate safeguards[7], the code is operating incorrectly.
On the other hand, the dynamic library loading mechanism implies that the wrapper, once loaded, can not be loaded a second time. Therefore, there is only one wrapper for the code in question installed in memory for a study, although it is quite possible to create multiple NumericalMathFunction instances based on the same wrapper. Indeed, the wrapper's library is an object shared between the different NumericalMathFunction objects of the study.
Thus, when we execute the following code:

```
>>> f = NumericalMathFunction ( "Code_X" )
>>> g = NumericalMathFunction ( "Code_X" )
```

the **f** NumericalMathFunction loads the **Code_X** wrapper, then the **g** NumericalMathFunction reuses the same wrapper without reloading it. All stages of initialization, execution, finalization will take place as expected even if the loading takes place only once.

[7]Access to the static variable happens in a critical section.

©2005-2012 EDF - EADS - PhiMeca

This causes the wrapper code, and thus the allocated memory space, to be shared by all NumericalMathFunctions of this wrapper. Indeed, the use of static variables in the wrapper will be the subject of access conflicts. The rule is as follows:

The use of static or global variables is forbidden within the wrapper.

How then can we retain information from one wrapper call to the next?
Open TURNS offers to solve this problem by asking the wrapper to allocate itself a memory space, called *internal state*, that will be associated to a NumericalMathFunction. The wrapper's developer is thus responsible for allocating the space, freeing it, initializing it and using it correctly. However, the developer cannot use any internal variable of the wrapper in order to retain the link—the pointer—with this memory space without falling into the trap of the static variable. Therefore, the Open TURNS library will be the one preserving the link for the developer, ensuring that the link to the memory space corresponds to the right NumericalMathFunction, i.e. the one that created it.
The creation of the internal state and its destruction are requested by Open TURNS from the wrapper at the right time, i.e. respectively when the NumericalMathFunction is created or deleted.
The wrapper can thus provide two C functions whose central terms are **createState_** and **deleteState_** to manage the internal state's lifecycle. These functions are optional. If they are absent, Open TURNS considers that there is no internal state. It then uses a null pointer.
Note that these functions are always provided together or not at all: there cannot be only one of them without the other.
The internal state being successively transmitted to all the C functions of the wrapper, it can also transmit information from the initialization stage to the execution and then the finalization. By default, the internal state is specific to each Function, Gradient and Hessian part. We will see in Section 5.7 how to ensure that the internal state is shared by all three blocks.
In summary, each part of the wrapper can provide the following C functions, listed in order of calling by the NumericalMathFunction:

1. **createState_**: creation of the internal state;
2. **getInfo_**: transmission of information from the wrapper to Open TURNS;
3. **init_**: preparation of the computation before the first execution;
4. **exec_** (or **exec_sample_**): computation execution; *this function is mandatory*;
5. **finalize_**: termination of the computation after the last execution;
6. **deleteState_**: destruction of the internal state.

4.4 The functions' signatures

Depending on its role, each function has a type of its own. By *type* we mean all of the function's input arguments, ordered and typed, as well as the type of its return value.
In the code wrappers designed for Open TURNS, the **exec_** functions all play the same role and they are thus of the same type. This allows to call them in a generic and uniform way, without having to know the processing it carries out.
The functions share the same type of return value. This type is always:

©2005-2012 EDF - EADS - PhiMeca

```
enum WrapperErrorCode func_... ( ... )
```

Note: The inclusion of the Open TURNS **WrapperInterface.h** *and* **WrapperCommon.h** *files provides the definitions for the symbols used in this section.*

This means that the function returns an error code indicating whether the processing was carried out properly or not. The Open TURNS library uses this code to handle failure situations, stop the computations that cannot succeed and warn the user of the potential problem.
This value of this code depends on the nature of the problem. Possible values are:

- **WRAPPER_OK**: processing was successful and the produced values are valid;
- **WRAPPER_MEMORY_ERROR**: the wrapper encountered a problem in memory management (allocation, deallocation, etc.).
- **WRAPPER_INITIALIZATION_ERROR**: the wrapper encountered a problem during the computational code's initialization phase;
- **WRAPPER_EXECUTION_ERROR**: the wrapper encountered a problem during the computational code's execution phase;
- **WRAPPER_FINALIZATION_ERROR**: the wrapper encountered a problem during the computational code's finalization phase;
- **WRAPPER_CANNOT_CREATE_STATE**: the wrapper can not create the internal state for a reason other than a memory management problem;
- **WRAPPER_CANNOT_DELETE_STATE**: the wrapper can not free the internal state for a reason other than a memory management problem;
- **WRAPPER_CANNOT_PROVIDE_INFORMATION**: the wrapper cannot provide Open TURNS with additional information;
- **WRAPPER_INTERNAL_ERROR**: the wrapper encountered an internal error (other than the previous errors);
- **WRAPPER_WRONG_ARGUMENT**: the wrapper was given an argument which is inconsistent with its settings;
- **WRAPPER_USAGE_ERROR**: the wrapper has received a configuration file which is incompatible with its internal capabilities;
- **WRAPPER_NOT_IMPLEMENTED**: the wrapper does not implement this feature.

This list may be supplemented in later versions of Open TURNS.
The rule regarding the return codes of the wrapper functions is:

The error code must be returned and its value must match the case met.

©2005-2012 EDF - EADS - PhiMeca

Normally, wrappers are not supposed to write on the standard output or standard error. However, for debugging purposes, it is convenient to use the **getErrorAsString** function of the wrapper's library in order to convert the error code into a readable message (see Annex A).
The signatures of the C functions are listed in the **WrapperInterface.h** file, but version 0.12.2 of the platform has brought many simplifications to the writing of wrappers wrapper writing, which we recommend using. The first of these simplifications consists in placing the name of the function implemented by the wrapper in a C macro:

```
#define WRAPPERNAME P
```

Thus **WRAPPERNAME** will be replaced by its value throughout the rest of the file.

4.4.1 createState_ function

The **createState_** function has the following prototype:

```
enum WrapperErrorCode
createState_ ( void ** p_p_state,
const struct WrapperExchangedData * p_exchangedData,
void * p_error )
```

It uses a **struct WrapperExchangedData** type parameter which contains information from the XML description file, formatted so as to be more easily used by a C source code. We will not go into detail on this structure, whose content may be subject to changes over time. Experienced developers will have recognized in the first parameter an opaque pointer. The value of this opaque pointer is positioned by the **createState_** function. Thus this function receives from the Open TURNS library the configuration described in the XML file and its role consists in allocating and producing an internal state which will be returned to Open TURNS. The content of this state is irrelevant for Open TURNS since the library only sees it as an opaque, i.e. typeless pointer. The third parameter is a pointer to another opaque structure, known by Open TURNS to manage errors inside wrappers. You do not need to care much about it, but you do have to pass this pointer "as is" to functions that may need it.
The last two parameters are *always* passed to the wrapper's functions. Therefore, from now on, we will not describe them anymore.
We recommend using the **CREATESTATE** macro to actually implement this function as shown below:

```
FUNC_CREATESTATE( WRAPPERNAME , {
  /* here you can allocate your internal state the way you want */
} )
```

4.4.2 deleteState_ function

The **deletestate_** function has the following prototype:

```
enum WrapperErrorCode
deleteState_ ( void * p_state,
const struct WrapperExchangedData * p_exchangedData,
void * p_error )
```

It takes as input the opaque pointer to the internal state, produced by the **createState_** function. Its role is to free the previously allocated space.

©2005-2012 EDF - EADS - PhiMeca

The allocated space *must* be freed.

This definition can be simplified as follows:

```
FUNC_DELETESTATE( WRAPPERNAME , {
  /* here you must deallocate the internal state you have created */
} )
```

4.4.3 getInfo_ function

The **getInfo_** function has the following prototype:

```
enum WrapperErrorCode
getInfo_ ( void * p_state,
struct WrapperInformation * p_info,
const struct WrapperExchangedData * p_exchangedData,
void * p_error )
```

Its role is to fill in the fields of the **struct WrapperInformation** structure. To achieve this, the function can rely on the content of the internal state, which is passed to it as an argument, and on the content of the **WrapperExchangedData** structure.
There are currently only two fields to fill in: the size of the Function's entry point (**inSize_**) and the size of the Function's exit point (**outSize_**). The **getNumberOfVariables** function substantially helps the developer with this task, as shown in the following example:

```
FUNC_INFO( WRAPPERNAME , {
  p_info->inSize_  = getNumberOfVariables(p_exchangedData, WRAPPER_IN);
  p_info->outSize_ = getNumberOfVariables(p_exchangedData, WRAPPER_OUT);
} )
```

4.4.4 init_ function

The **init_** has the following prototype:

```
enum WrapperErrorCode
init_ ( void * p_state,
const struct WrapperExchangedData * p_exchangedData,
void * p_error )
```

Its role is to prepare the computation before its first execution. All the elements required by this preparation are either statically known in the wrapper, or obtained through the internal state passed as an argument.
The way to prepare the computation varies a lot and entirely depends on the computational code that is interfaced with Open TURNS. An analytical formula has no requirements, whereas a networked computational code will probably require some preparation for the computation session. This function is the place for this preliminary work.
Most often, the **init_** function has nothing to do, in which case it can be completely removed. If, however, it were to be maintained in the source code, we must be careful to return a correct return code (**WRAPPER_OK**).

©2005-2012 EDF - EADS - PhiMeca

```
FUNC_INIT( WRAPPERNAME ,
{
  /* Write here the instructions performing the initialization */
} )
```

4.4.5 exec_ function for the Function

The **exec_** function for the Function has the following prototype:

```
enum WrapperErrorCode
func_exec_ ( void * p_state,
const struct point * inPoint,
struct point * outPoint,
const struct WrapperExchangedData * p_exchangedData,
void * p_error )
```

In addition to the internal state which is passed as an argument, this function receives a structure of type **struct point** containing the entry point of the function (**inPoint**), and an equivalent structure meant to contain the exit point (**outPoint**). It is important to note that these structures are already allocated and, as a consequence, it is neither necessary nor permitted to modify their memory storage in any way.
The **struct point** structure contains two fields:

- the size of the point in question (**size_**);
- the coordinates array of the point (**data_**).

The array is pre-allocated with the point size. No memory management is to be performed.
We remind here that C arrays start at index zero.
In the case of the entry point (**inPoint**), the coordinate values are automatically filled in by the Open TURNS library invoking the wrapper. However, the coordinates of the exit point (**outPoint**) are not initialized. The values stored in the exit point coordinate array are therefore invalid and cannot be used until they have been correctly positioned by the **exec_** function.
This function is the computational heart of the wrapper. It is responsible for calling the code to be interfaced according to the appropriate method. It may consult the content of the wrapper's description file in order to determine the type of interface and the instructions provided by the user to run the code. Before returning to the calling code, the exit point (**outPoint**) should be positioned.
It is important that this function correctly relays any error encountered during the execution of the code, so that the Open TURNS library can manage the problem.
This definition can be simplified as follows:

```
FUNC_EXEC( WRAPPERNAME ,
{
  /* Write here the instructions performing the computation */
} )
```

4.4.6 exec_sample_ function for the Function

The **exec_sample_** function for the Function has the following prototype:

©2005-2012 EDF - EADS - PhiMeca

```
enum WrapperErrorCode
func_exec_sample_ ( void * p_state ,
const struct sample * inSample ,
struct sample * outSample ,
const struct WrapperExchangedData * p_exchangedData ,
void * p_error )
```

In the way it operates, it is very similar to the **exec_** function except that it operates on a sample (**inSample**) placed in a **struct sample** structure and that it, in turn, produces a result sample (**outSample**).
The **struct sample** structure is constructed as a collection of **struct point** structures. In fact, it has two fields:

- the size of the sample, i.e. the number of present points (**size_**);
- the array of points (**data_**).

The structures are already allocated when entering the Function and no memory management is to be performed.
There are no **exec_sample_** for the Gradient and the Hessian.
This definition can be simplified as follows:

```
FUNC_EXEC_SAMPLE( WRAPPERNAME ,
{
  /* Write here the instructions performing the sample computation */
})
```

Very often, the user wants to take advantage of the multiprocessing capabilities of their hardware. The computation on samples is a good place to implement parallelism. But parallelism is a tricky matter and it may discourage the wrapper's developer. Therefore, we have defined a parallel version of the **exec_sample_** function, which can be called as follows :

```
FUNC_EXEC_SAMPLE_MULTITHREADED( WRAPPERNAME )
```

As its name indicates, the proposed **exec_sample_** function is multithreaded, which means that it calls the **exec_** function in parallel on machines with multiple processors or cores. For this to work properly, it is important to respect the rules concerning the implementation of parallel wrappers, especially the absence of static or global variables.

4.4.7 exec_ function for the Gradient

The **exec_** function for the Gradient has the following prototype:

```
enum WrapperErrorCode
grad_exec_ ( void * p_state ,
const struct point * inPoint ,
struct matrix * outMatrix ,
const struct WrapperExchangedData * p_exchangedData ,
void * p_error )
```

The only difference between the **exec_** function for the Function and the one for the Gradient is the type of structure returned. In the case of the Gradient, the function returns a **struct matrix** structure containing the Gradient matrix (**outMatrix**) at the considered point (**inPoint**).

©2005-2012 EDF - EADS - PhiMeca

The structures are already allocated and no memory management is to be performed.
The role of the function is to fill in the elements of the **outMatrix** matrix.
This example can be simplified as follows:

```
GRAD_EXEC( WRAPPERNAME ,
{
  /* Write here the instructions performing the computation */
} )
```

4.4.8 exec_ function for the Hessian

The **exec_** function for the Hessian has the following prototype:

```
enum WrapperErrorCode
hess_exec_ ( void * p_state,
const struct point * inPoint,
struct tensor * outTensor,
const struct WrapperExchangedData * p_exchangedData,
void * p_error )
```

The only difference between the **exec_** function for the Function and the one for the Hessian is the type of structure returned. In the case of the Hessian, the function returns a **struct tensor** structure containing the Hessian tensor (**outTensor**) at the considered point (**inPoint**).
The structures are already allocated and no memory management is to be performed.
The role of the function is to fill in the elements of the **outTensor** tensor.
This example can be simplified as follows:

```
HESS_EXEC( WRAPPERNAME ,
{
  /* Write here the instructions performing the computation */
} )
```

4.4.9 finalize_ function

The **finalize_** function has the following prototype:

```
enum WrapperErrorCode
finalize_ ( void * p_state,
const struct WrapperExchangedData * p_exchangedData,
void * p_error )
```

Its role is to finalize the computation after its last execution. All the elements necessary for this finalization are either statically known in the wrapper, or obtained through the internal state passed as an argument.
As with the **init_** function, the way to finalize a computation varies a great deal and entirely depends on the computational code interfaced with Open TURNS. An analytical formula requires nothing, whereas a computational code launched through CORBA will probably need to terminate the computation session. This function is the place for this work.
Most often, the **finalize_** function does nothing, in which case it can be completely removed. If, however, it were to be maintained in the source code, we must be careful to return a correct return code (**WRAPPER_OK**).
This definition can be simplified as follows:

©2005-2012 EDF - EADS - PhiMeca

4.5 Static linking with FORTRAN 77

History has it that FORTRAN 77 is the most commonly used language for writing scientific computational codes. However, it so happens that making C and FORTRAN 77 coexist is fairly easy. The developer of a wrapper for code written in FORTRAN 77 can thus benefit from this proximity to call subroutines[8] of the computational code directly from the wrapper's source coder. This way, the developer can benefit from the most efficient access to data while tightly controlling their code.
However, this is not possible with FORTRAN 90, which has completely changed the way to interface itself with the rest of languages.
Without going into the details of the interfacing techniques between C and FORTRAN 77

- any external symbol produced by the C compiler bears the same name in the source file and in the object file;
- any external symbol produced by the FORTRAN 77 compiler bears a different name in the object file and in the source file, but the translation algorithm is relatively simple (adding a variable number of underscore characters (_) as a prefix or suffix, changing the case for alphabetic characters, truncating the names of external symbols to 6 characters, etc.) and can be coded in a simple macro;
- the linking performed by the linker producing the wrapper maps identical symbols from object files (C and FORTRAN 77).

The situation is thus as follows: if one compiles C and FORTRAN 77 source files without noticing, chances are that the linking fails to produce the wrapper, because symbols on each side can not be matched. Which, in other words, means that it is not possible—unless by extraordinary chance—to call a FORTRAN 77 subroutine from C as is, or vice-versa.
It is therefore necessary to transform one of both symbols to match its counterpart in the other language. It turns out that this is much easier to achieve on the C side than on the FORTRAN 77 side. We will therefore proceed to transform the C symbols in order to make them identical to the FORTRAN 77 symbols.
To perform this operation in an easy, portable and maintainable way, we use the power of the **autoconf** tool which is already used to compile the wrapper (on this point, see section 3.2.2).
We only need to use the development structure, for which everything is ready, placed in the **share/openturns/WrapperTemplates/wrapper_linked_with_F77_function** directory (see Section 3.2.2).
It was assumed here that the FORTRAN 77 computational code was fully included in a file named **code.f**. If this was not the case, one must delve into the configuration details of the **autoconf**, **automake** and **libtool** tools, which is outside the scope of this document. The reader should refer to the online documentation available via the **info** command.
For illustration purposes, it is also assumed that the subroutine performing the computation in the **code.f** file has the following signature:

```
SUBROUTINE COMPUTE ( X, N, Y, P, RC )
REAL*8 X(*)
INTEGER*4 N
REAL*8 Y(*)
INTEGER*4 P
INTEGER*4 RC
```

This is a subroutine named COMPUTE taking five input arguments:

- **X** is an array of doubles containing the coordinates of the Function's entry point;

[8]By subroutine, we mean any compilation unit of **SUBROUTINE** type.

©2005-2012 EDF - EADS - PhiMeca

- **N** is the size of **X**;
- **Y** is an array of doubles containing the coordinates of the Function's exit point:
- **P** is the size of **Y**;
- **RC** contains a non-zero return value in case of an error during execution.

The last elements to edit are in the wrapper's source code (**wrapper.c**). However, before showing these modifications, we will first assume that the **exec_** function is written as follows:

```
FUNC_EXEC( WRAPPERNAME ,
{
  int status = 0;
  COMPUTE ( INPOINT_ARRAY,  & INPOINT_SIZE,
  OUTPOINT_ARRAY, & OUTPOINT_SIZE,
  status );

  if ( status ) return WRAPPER_EXECUTION_ERROR;
} )
```

This writing is obviously not correct: changes will have to be made to it in order to make it work, but it shows several interesting points.
First, we can see that writing an **exec_** function can be very simple, even if the processing performed by the COMPUTE subroutine is very complex.
It also shows that the structures passed as arguments to the wrapper's functions (**struct point**, but also **struct matrix** and **struct tensor**) can be easily interfaced with FORTRAN 77: this is a deliberate choice of the Open TURNS platform. All of the data structures called upon to be handled by FORTRAN 77 are compatible with its storage system (*column major*). They can thus be passed directly as arguments without further ado. This takes the simplification of wrapper writing one step further.
Finallyn the call to the FORTRAN 77 subroutine is implemented as a classic C function call. It is on this last element that the modifications will be performed. Instead of directly calling the name of the COMPUTE subroutine, we use a macro
The macro that allows us to change the name is called **F77_FUNC** and takes two parameters: the first is the name of the FORTRAN 77 subroutine, in lower case (**compute**) and the second is the same name in upper case (**COMPUTE**). We store the result of the call to **F77_FUNC** in a symbol that we call **COMPUTE_F77** by convention and to indicate that this is the name of the FORTRAN 77 subroutine **COMPUTE**.
Then the call to **COMPUTE** from the previous example is modified in a call to **COMPUTE_F77**.
This is reflected in the new code for the **exec_** function:

```
#define COMPUTE_F77 F77_FUNC ( compute , COMPUTE )

/* the FORTRAN subroutine prototype */
void COMPUTE_F77 ( double *, int *,
double *, int *,
int * );

FUNC_EXEC( WRAPPERNAME ,
{
  int status = 0;
  COMPUTE_F77 ( INPOINT_ARRAY,  & INPOINT_SIZE,
```

©2005-2012 EDF - EADS - PhiMeca

```
    OUTPOINT_ARRAY, & OUTPOINT_SIZE,
    status );

    if ( status ) return WRAPPER_EXECUTION_ERROR;
  } )
```

At this stage, we have a Function computed not in the wrapper's source code, but in a function written in FORTRAN 77 and linked to the wrapper's code at the time of linking. We are thus halfway through the externalization process of the computation relatively to the wrapper. It is quite possible to avoid having to compile the FORTRAN source code, as we have just shown, and to rather link with the wrapper an already compiled static or dynamic library.

4.6 Computation with data samples

If the computational code allows it, the wrapper can provide an **exec_sample_** function allowing the direct computation of samples. The importance of computing samples is to group in a single call to the code executions of the computational code that would otherwise occur as many times as there are points in the sample.
This may seem unnecessary when the function is directly linked to the wrapper (see sections 3.2.5 and 4.5) but it proves useful when each call to the computational code requires the creation of files or processes (**fork** parameter as the type of the <**wrap-mode**> tag in the description file). We can thus save much time. This is also essential when the code is executed on a server with a batch manager. Computations are grouped and submitted in batches.
As has been stated already, it is necessary that the code accepts a collection of points on which it will successively perform its computations. However, it is possible to encapsulate the code in a script (or any other program) that reads the content of the sample and produces the files and calls necessary for the code execution.
This documentation cannot detail all the possibilities of sample-based computation and it is the responsibility of the wrapper's developer to define the most appropriate procedure for their computational code.
In the absence of any **exec_sample_** function in the wrapper's source code, Open TURNS handles the sample computation by successively calling the **exec_** function which must be present. There is no need to code an **exec_sample_** function carrying out the same task in the wrapper because it would be redundant. Coding an **exec_sample_** function is only interesting insofar as the call to the computational code is grouped.

4.7 Computation with a Gradient or a Hessian

Some computational codes are able to provide all or part of the Gradient or Hessian of the Function at the considered point. These values can be retrieved to produce the matrix or tensor needed by Open TURNS for some algorithmic solving operations
Therefore, the wrapper can optionally provide a Gradient and a Hessian for the Function. This requires acting in two places.
The first modification occurs in the wrapper's description file, which must specify that it implements a Gradient or a Hessian by setting the **provided** attribute of the <**gradient**> or <**hessian**> tags to **yes**, and by defining the name of the Gradient or Hessian between the corresponding opening and closing tags. An example clarifies this point:

```
<function provided="yes">P</function>
<gradient provided="yes">P</gradient>
<hessian  provided="yes">P</hessian>
```

We used again here the example given in Section 3.2.4, after modifying it.

©2005-2012 EDF - EADS - PhiMeca

In doing so, we declare in this file that the wrapper defines a Function, a Gradient and a Hessian, all bearing the name **P**.
This name does not need be the same for all three parts of the wrapper but it improves the readability and understandability of the operation. It makes it clear that we provide the **P** Function, as well as **P**'s Gradient and **P**'s Hessian. We will see later why it may be interesting to *not* have the same name for these three elements.
The second change is in the wrapper's source code (**wrapper.c**) which must now include the C functions corresponding to the Gradient and Hessian parts of the wrapper.
As stated in Section 4.3, these functions vary in number as needed, but they must use the prefix **grad_** for the Gradient part and **hess_** for the Hessian part. We thus end up with at least two or three functions whose central term is **exec_**:

- **func_exec_P**;
- **grad_exec_P**;
- **hess_exec_P**.

The sequence of steps is similar, though independent for each part. By default, each part has its own internal state, completely isolated from the others. It may therefore be different from one part to another, even present in one and absent in the other two.
Each **exec_** function—there are three in this example—must execute the necessary call to fill in the data structure that is passed as an argument: **struct point** for the Function, **struct matrix** for the Gradient and **struct tensor** for the Hessian.
Since these functions are independent from each other, it is quite possible to use another code to compute the Gradient and the Hessian, if necessary through a completely different technique: analytical formula coded in the source, call to an external script, etc.
There are currently no **exec_sample_** functions for the Gradient or the Hessian.
The wrapper's source code can contain several different Function parts. An explanation can be, for example, that the developer does not want to design several wrappers for very simple formulas. They then have a wrapper which provides a range of Functions that can be selected with the content of the <**function**> tag of the description file, having if necessary several description files using the same wrapper (**wrapper.so**).
It is also possible to encode several Gradients (or Hessian) in the wrapper's source code. These may be linked to different Functions or to the same Function, which allows to have multiple implementations that can be selected using the description file.
As in the previous example, if the wrapper contains two implementations of P's Gradient, we may choose to name them P1 and P2 and write this version of the description file:

```
<function provided="yes">P</function>
<gradient provided="yes">P1</gradient>
<hessian  provided="yes">P</hessian>
```

or this one:

```
<function provided="yes">P</function>
<gradient provided="yes">P2</gradient>
<hessian  provided="yes">P</hessian>
```

When the wrapper does not provide a Gradient or Hessian, Open TURNS overcomes the problem by computing a Gradient or Hessian based on the Function by finite differences. One or the other may be absent independently.

©2005-2012 EDF - EADS - PhiMeca

5 The wrapper as interface for the external code

In the process of externalization of the computational code relatively to the wrapper, using an external code is the final step. It will address issues that were supported, until now, by statically linking the Function with the wrapper's code and that were therefore ignored. These issues include transmitting the variables' values to the code, using files or executing the command launching the computational code.

5.1 Transmitting values to the code

In the examples described so far, the variables' values were transmitted to the computational code through the arguments of the function called in the wrapper, whether this function be written in C or FORTRAN. This transfer was entirely supported by the compiler as long as the wrapper's developer had properly indicated the number, type and order of the variables to supply.
When the code is completely external to the Open TURNS platform, this mechanism cannot be generated by the compiler and one ends up having to manage many problems that were previously hidden:

- where must the variables be written so that the external code can read them?
- in which format should they be produced (text, binary, different format)?
- how many times does each variable appear?

However the vast majority of the computational codes we deal with use variables whose values are recorded in text format data files. The wrapper's library contains a set of predefined functions that can recognize and replace the values of these variables in text files.
There are many variables placed within binary files—e.g. mesh nodes or node fields—but the fact that this type of data is difficult to read, as well as the specific navigation and locating within these files, explain that they are not taken into account in the current version of Open TURNS. No other format is currently taken into account.
There are also computational codes for which some values may be transmitted over the command line. Open TURNS is also able to handle this situation. In this case, the only acceptable format is the text format.
The substitution mechanism for variables in files or on the command line can handle the presence of several variables, or even of the same variable several times. However, in the current state of things, the same variable can only be written in one format. This means that, for example, if a variable appears twice in the same file in two different formats, it can be recognized at both locations but it will be rewritten following a single format. This scenario is quite rare, it is currently not possible to distinguish two rewriting formats for the variable.
In summary, the mechanism of value transmission to the computational code is currently based on files or command lines in text format. Future developments may bring Open TURNS to support new file formats

5.2 Regular expressions

5.2.1 General

In the perimeter we have defined, the problem we are faced with is to identify and replace any variable, regardless of how it appears in the file or the command line. If we take the example of a physical variable, e.g. the pressure **P**, placed in the input data file for a computational code, this variable may appear in very different formats:

```
# temperature
T = 293 K
# pressure
P = 101300 Pa
```

```
# output
D = 1.5 l/s
```

or

```
293 101300 1.5
```

or else

```
293 1.013e5 1.5
```

Open TURNS can not force the computational codes with which it wishes to interface to change the format they use for writing data. It is thus necessary to use a technique that allows to recognize the value in (almost) all cases, at least the most common ones.
Regular expressions were designed to reach this goal and they are commonly present on all current platforms
Suppose that, in the following example:

```
# temperature
T = 293 K
# pressure
P = 101300 Pa
# output
D = 1.5 l/s
```

we try to extract the value of the pressure **P** to substitute a value calculated by Open TURNS. We could search for the value **101300** in the file, but this assumes that one knows the value in advance. This approach is valid if the data file changes little or not, and if the value appears only once in the file.
If the value appears multiple times or if the file is regularly modified, or automatically generated, identifying **101300** no longer works.
In reality, what we want, as a user, is to change the numerical value that is written on the line **P** = **<numerical value> Pa**. Regular expressions allow to describe this line in a shorter form

```
^P = \R Pa$
```

We find the expected terms **P** = and **Pa**, which frame a particular symbol **R**. This means that, at this spot, there must be a **real** numerical value. The two characters `^` and `$` respectively indicate the beginning and the end of the line.
Translated into natural language, the regular expression above can be read as *find a string starting at the beginning of the line* (`^`), *exactly followed by the string* **P** = *, then by a real numerical value, then by the string* **Pa** *and ending at the end of the line* (`$`).
It is then possible to find the value **101300** previously searched without knowing the value itself but only by describing how to find it using its structure (it is a numerical value) and its context (the strings **P** = and **Pa**).
The symbol **R** can recognize any writing format for real numerical values, whether they are written as integers (**101300**) or floating points (**101300.0**, **+1.013e+5**, etc.).
We can further improve the system by indicating that the whitespaces located between the character **P** and the character =, or anywhere elsewhere, may actually be whitespaces, tabs or any other character considered as a whitespace in a text file. This is done by replacing the whitespace with the symbol **S**, for separator (or space). The expression is then written:

```
^P\S=\S\R\SPa$
```

But it is common to place any number of whitespaces in the file, if only for readability or alignment. We can then specify that these (or any other character) is present in variable numbers. To achieve this, we have at our disposal five modifiers to set after the symbol or character considered:

- modifier ***** indicates that this character may be repeated zero or N times;
- modifier + indicates that this character may be repeated once or N times;
- modifier **?** indicates that this character may be repeated zero time or once;
- modifier **{N}** indicates that this character may be repeated N times;
- modifier **{N,P}** indicates that this character may be repeated between N and P times.

In this case the expression becomes:

```
^\S*P\S*=\S*\R\S*Pa\S*$
```

which allows us to identify the value in a file such as this one:

```
# pressure
P      = 101300  Pa
```

as well as this one:

```
# pressure
P=101300Pa
```

Note that we have added **S** symbols at the beginning and at the end of the line, in order to detect whitespaces at the beginning and at the end of the line.

> *Note: In addition to the symbol* **R**, *there is also the symbol* **I** *used to detect integers. All three symbols* **R**, **S** *and* **I** *are specific to the regular expressions library of Open TURNS.*

The entire standard grammar for extended regular expressions is available, which means that we have the following features:

- parentheses ( ) are used to group symbols into larger expressions and treat each term as a symbol in its own right (with its modifiers, etc.);
- the symbol | acts as a logical OR between two terms;
- brackets [ ] provide a list of characters (but not symbols) that can appear at one point.

Parentheses have an additional function. When a symbol, character or an expression involving symbols, characters and modifiers is set between parentheses, the value identified during reading is stored and can be used later. We will see later on that the value can be retrieved using the symbol **nnn**, where nnn is a positive integer (nnn = 1, 2, 3 ...) designating the parentheses group in question in the expression. The groups are numbered from left to right in order of their appearance.
Thus to retrieve the numerical value of the pressure in the file, we can use the following regular expression:

```
^\S*P\S*=\S*(\R)\S*Pa\S*$
```

The value will be placed in the symbol **1**.
As an exercise, if the data file was presented in the following format:

©2005-2012 EDF - EADS - PhiMeca

```
293 1.013e5 1.5
```

we would have recovered the temperature, pressure and flow using the following regular expression:

```
^\S*(\R)\S+(\R)\S+(\R)\S*$
```

respectively in the symbols **1**, **2** and **3**. Note the use of the + modifiers to separate the numerical values by spaces of variable size.

5.2.2 Implementation in the description file

Regular expressions allowing to locate the variable containing the pressure and temperature are declared in the **<regexp>** tags of each variable. Each string matching the regular expression is then replaced by the content of the **<format>** tag of the associated variable. This tag contains a string with one (and only one) descriptor **%E**, **%F** or **%G** (or **%e**, **%f**, **%g**) allowing to rewrite the numerical value of the variable with the decimal format expected by the external code. For a complete description of these format descriptors, please refer to the Unix man page for **printf**.
But developers used to programming in C or FORTRAN will recognize classic formats they are accustomed to. With the example of the pressure variable P, the string identified in the data file(s) using the regular expression will be replaced by the string contained in the format. For example, for a format **%20.16G**, the value of P will be written in 20 characters, including 16 decimals.

> *Note: Note that it is essential to provide the code with the maximum number of decimals that it can read (and that Open TURNS can produce) so as not to hinder the convergence of internal algorithms. Similarly, the computational code should produce values with maximum accuracy.*

In summary, if we were to substitute the values of pressure and temperature in the file described above, we would write variables in the wrapper's description file as follows:

```
<variable-list>
<variable id="P" type="in">
<name>Pressure</name>
<unit>Pa</unit>
<regexp>^\S*P\S*=\S*\R\S*Pa\S*$</regexp>
<format>P = %20.16G Pa</format>
</variable>
<variable id="T" type="in">
<name>Temperature</name>
<unit>K</unit>
<regexp>^\S*T\S*=\S*\R\S*K\S*$</regexp>
<format>T = %20.16G K</format>
</variable>
</variable-list>
```

5.3 Isolating executions

We saw in Section 3.3 that the wrapper functions could be called in parallel by multiple execution threads. This meant that the static and global variables were prohibited in the wrapper code

©2005-2012 EDF - EADS - PhiMeca

This problem arises again outside of the wrapper and the Open TURNS platform. Indeed, a careless design of the call to the external code may result in conflicts over access to the directories and to the code data files which then become shared resources.
Let's take the example of a computational code called **Code_X**, which reads a **fich.dat** input file and produces a **fich.res** result file in the working directory from which it is launched. Let's also assume that the code is simultaneously launched in the same directory: both instances of **Code_X** each produce a result file bearing the same name, in the same directory. Both files risk being corrupted and unreadable.
Similarly, if the input file is generated by the wrapper of **Code_X**, and if this wrapper is called in parallel, two files with different values will be produced in the same place. Again, the access conflict will cause corruption of the file contents.
The solution is to either rename the files so that each has a different name, or to write the files in different directories and launch each computation in one directory. The first solution is usually impossible to implement because the computational codes generally read files with specific names that cannot be changed. Accordingly, we adopted the second choice in Open TURNS. This assumes that the computational code can be run from any directory. There are few codes that do not allow this and it is always possible to recreate in the directory the environment they require.
We must also be careful here to create directories with unique names. Fortunately, the operating system offers standard features to perform this kind of operation. However, changing directories within the wrapper using the library function **changeDirectory** changes the working directory of all wrapper threads and destroys the parallelism. It is therefore advisable to avoid using this function starting from version 0.12.2. It is preferable to use the **runInsulatedCommand** function that performs the isolation without any disruption.

The *ChangeDirectory* function is obsolete and should no longer be used.

Open TURNS provides library functions to help write the wrapper:

- the **getCurrentWorkingDirectory()** function returns a string containing the name of the current execution directory. The returned string must be freed by the caller.
- the **createTemporaryDirectory(prefix, p_exchangedData)** function creates a temporary directory with a unique name and returns this name in a string. The directory is created in the **temporary-directory**, declared in the **<openturns_dir>/etc/openturns.conf**, prefixed with the **prefix** string.
- the **deleteTemporaryDirectory(tempDir, status)** deletes the temporary directory and its content if **status** has a non-zero value. **status** is usually the value returned by **runInsulatedCommand**, thus any execution error in the solver will allow to keep the files. The string **tempDir** is systematically deleted even if **status** is non-zero.

5.4 Transmission via files

The vast majority of external codes use files to communicate the input and output values for the computational code. Values are most often written as text (ASCII or equivalent) that can be modified either by user or by peripheral data tools. This is how physical or numerical parameters are transmitted for the computation.
It also happens that some codes use binary files to speed up and simplify the reading of complex data. This also significantly reduces the file size when there is a sizeable volume of data: mesh and CAD files are typically an example of binary files. We may also want to use this file format to retain maximum numerical precision on the data because no truncation or alteration happens during writing or reading.
As it stands, the substitution mechanism based on regular expressions can only work on text files. This is currently a limitation of the interfacing mechanism between external codes and Open TURNS, since it is not possible to read or write a probabilistic variable in binary files.

©2005-2012 EDF - EADS - PhiMeca

All files manipulated by the external code that must be processed by the wrapper, whether it be a copy in the temporary directory and/or variable substitution, must be declared in the <**data**> tag of the description file as follows:

```
<external-code>
<data>
<!-- An input file -->
<file id="data" type="in">
<name>The input data file</name>
<path>code_C1.data</path>
</file>

<!-- An output file -->
<file id="result" type="out">
<name>The output result file</name>
<path>code_C1.result</path>
</file>

</data>
 ...
</external-code>
```

This example shows how to declare a data file—of type **in**—and a result file—of type **out**. Data files are intended to be copied into the temporary directory because they will be modified before the computation begins. However, if the input files are never modified, we can get rid of the copy as long as the computation procedure, which will be launched by Open TURNS, knows where to find them. The result files are not copied in the initial directory
Each file must bear a unique identifier (**id**), a type (**in** or **out**). It may also have a name (**name**) that only serves as a commentary for users.
The path (**path**) is used to determine the location of the file relatively to the current directory where the wrapper is being run. This path can be relative or absolute. During the process of copying files in the temporary directory, no name substitution is possible. The file will thus be copied with the same name.
There is no limit on the number of input or output files.
Before version 0.12.2, data or result files were systematically read in order to substitute or read probabilistic variables. Occasionnally, however, some files could not be read, for example binary files, without causing an error. Therefore, version 0.12.2 introduces an additional tag called <**subst**>, indicating which variable is in which file. Only files declaring a non-empty <**subst**> tag will be read in search of variables. The variables appearing in the <**subst**> tag must be separated by commas without any other delimiter (whitespace or tab).
The following example shows how to declare the input variables (**P** and **T**) and output variables (**O1** and **O2**) in the files:

```
<external-code>
<data>
<!-- An input file -->
<file id="data" type="in">
<name>The input data file</name>
<path>code_C1.data</path>
<subst>P,T</subst>
</file>
```

©2005-2012 EDF - EADS - PhiMeca

```
<!-- An output file -->
<file id="result" type="out">
<name>The output result file</name>
<path>code_C1.result</path>
<subst>O1,O2</subst>
</file>

</data>
...
</external-code>
```

To easily perform the variable substitution and the file copy within the wrapper, Open TURNS offers several library functions:

- the **createInputFiles(tempDir, p_exchangedData, p_point)** function copies all the input files declared in the description file (whose content is carried by the **p_exchangedData** structure) in the temporary directory **tempDir** by substituting the variable values contained in **p_point**;
- the **readOutputFiles (tempDir, p_exchangedData, p_point)** function reads the content of the variables in the result files declared in the description file (whose content is carried by the **p_exchangedData** structure) and present in the temporary directory **tempDir**, and stores the values in **p_point**.

5.5 Transmission using command line arguments

In some rare cases, solvers require that the values of some variables be passed using command line options. Open TURNS is able to carry out the same kind of substitution from a model command line defined in the wrapper's description file (<**command**> tag). All that is needed is to create a pseudo-command line in which the values of variables appear in a form that can be detected by a regular expression, and to proceed with the variable substitution (<**regexp**> and <**format**> tags) in a conventional manner.
The following example shows how to substitute the **P** variable on the command line:

```
<variable-list>
<variable id="P" type="in">
<name>Pression</name>
<unit>Pa</unit>
<regexp>%:P:%</regexp>
<format>%20.16G</format>
</variable>
...
</variable-list>
...
<external-code>
<data>
<!-- NO input file -->
<!-- An output file -->
<file id="result" type="out">
<name>The output result file</name>
<path>code_C1.result</path>
<subst>O1,O2</subst>
```

©2005-2012 EDF - EADS - PhiMeca

```
</file>
</data>
...
<command>/here/is/myCode --someoption P=%:P:%</command>
</external-code>
```

The string %:P:%, very characteristically chosen, simply identifies the location of the substitution on the command line. It will be replaced by the value of variable in the %20.16G format as described in the **format** tag.
For the substitution to take place on the command line, we must explicitly call the library function **runInsulatedCommand(tempDir, p_exchangedData, p_point)** which requires the same parameters as the **createInputfiles** function seen in the previous section.

5.6 Writing an execution function for an external code

We saw in Section 4.4.5 that the execution function could be simplified to: (v. 0.12.2 and after)

```
FUNC_EXEC( WRAPPERNAME ,
{
  /* Instructions carrying out the computation */
} )
```

But we have not yet given an explicit content to this function. With the elements that have been presented in this section, it is now possible to describe what must be done in order to correctly call an external code.

```
FUNC_EXEC( WRAPPERNAME ,
{
  struct WrapperExchangedData * p_exchangedData =
  CAST(struct WrapperExchangedData * , p_state);
  int rc = 0;
  char * temporaryDirectory = 0;
  char * cmd = 0;
  char * currentWorkingDirectory = 0;

  /* We save the current working directory for a future come back */
  currentWorkingDirectory = getCurrentWorkingDirectory();

  /* We build a temporary directory in which we will work */
  temporaryDirectory = createTemporaryDirectory( "myPrefix", p_exchangedData);

  /* We create the external code's input files in the temporary directory */
  if (createInputFiles(temporaryDirectory, p_exchangedData, inPoint))
  return WRAPPER_EXECUTION_ERROR;

  /* The real computation is here */
  rc = runInsulatedCommand(temporaryDirectory, p_exchangedData, inPoint);

  /* Read the output values */
  if ( !rc )
  if (readOutputFiles(temporaryDirectory, p_exchangedData, outPoint))
```

©2005-2012 EDF - EADS - PhiMeca

```
    return WRAPPER_EXECUTION_ERROR;

    /* We kill the temporary directory if no error has occured */
    deleteTemporaryDirectory(temporaryDirectory, rc);

    free ( currentWorkingDirectory );
  }
```

Note that the heart of function is completely independent from the code that is being driven, with the exception of the prefix name for the temporary directory. That is why this function can be further simplified as:

```
FUNC_EXEC( WRAPPERNAME ,
FUNC_EXEC_BODY_CALLING_COMMAND_IN_TEMP_DIR( "myPrefix" ) )
```

5.7 Sharing the internal state

So far, the examples we have shown used what we called the "internal state" to store a data structure corresponding to the content of the wrapper's description file. This data structure was accessible through the **p_exchangedData** pointer.
We recall that the internal state can be accessed via a pointer named **p_state**, passed as argument to (almost) all functions of the wrapper. The information it contains is created within the **createState_** function but is retained by the Open TURNS library which is responsible for transmitting the pointer to each function call. The state plays the role of a static (or global) variable for the wrapper's functions, when this type of variable is prohibited.
Thus this internal state is the ideal place to store information that must be persistent from one call of the wrapper's functions to the next. It is quite possible to compute a piece of data in the **getInfo_** function, store it in the internal state which will be passed to the **exec_** function which then uses it.
We can also compute a value within the **exec_** function, store it in the internal state for the next call to this same **exec_** function. However, we must always bear in mind that the wrapper's functions, especially functions such as **exec_** or **exec_sample_**, may be called in parallel by the Open TURNS library. Therefore they must be designed to be reentrant and multithread-safe.
In the most common mode, the internal state is only shared within the wrapper functions corresponding to the same block, that is to say the same Function, the same Gradient or the same Hessian. This means that, by default, the internal state of the Function is completely different and disjoint from the internal state of the Gradient and of the Hessian. This is what we call the **specific** mode in the wrapper settings.
However, it is possible to combine these three internal states into one, which will be common to all three blocks Function, Gradient and Hessian. It is the **shared** mode in the wrapper settings. The internal state will be created and deleted by the **createState_** and **deleteState_** functions of the Function, and then passed as an argument to all the functions of all blocks. Again, we must be careful about concurrent access to the data contained therein. However, we must notice that the **createState_** and **deleteState_** functions of the Gradient and the Hessian will not be called at all.
Whichever mode gets chosen, the Open TURNS library ensures that the internal state will not be moved or relocated elsewhere in memory, which guarantees that the passed pointer may be accessed for reading without any special protection. However, any writing (and corresponding reading) must be protected.
Why share the internal state? When calling one block (for example the Function), some codes compute data related to another block (for example the Gradient). The shared internal state allows to store this information, which is unnecessary for the first block, so as to be used by the second block without being computed again. This saves considerable time and helps achieve a better accuracy on certain treatments.

©2005-2012 EDF - EADS - PhiMeca

We do not offer an example of shared internal state because situations are so varied and complex that a general solution is usually of little practical use. However, any wrapper developer with a good command of multithreaded and C programming can get by with this scenario without too much trouble.

©2005-2012 EDF - EADS - PhiMeca

6 Future developments

If the interfacing technique developed for Open TURNS has not significantly evolved in its principles since the inception of the platform, it has experienced many improvements that always strive for a simplification of the writing of wrappers, in order to make it accessible to as many users as possible and to enrich the library with new features.
The introduction of the wrapper library, of regular expressions and their shortcuts, or of macros, are part of the most emblematic recent developments.
Other developments, however, are in the pipeline and will be introduced over time. These include (the order being no indication of priority):

- variable search and substitution withinin binary files, especially MED-format files. These files often containing mesh elements, it would then be possible to turn geometric variables into probabilistic ones.
- batch execution of external codes on computational servers of any kind. In doing so, Open TURNS would be able to launch a large number of simulations on far more powerful machines, thereby enabling much better quality probabilistic studies.

©2005-2012 EDF - EADS - PhiMeca

A Functions from the WrapperCommon library

At the time of writing, the Open TURNS wrapper library contains a set of functions which can be directly used by wrapper developers. These functions are available to perform tasks shared by many wrappers and to save the developers the work needed to create them anew each time. They are regularly updated to follow the platform's evolution; therefore, by using them, the wrappers are systematically up to date regarding the platform's latest features.
We invite the reader to use them as much as possible. If functions are missing or not adapted, the reader wishing to improve the system can contact us through the official Open TURNS website (**http://www.openturns.org**) and inform us of the difficulties encountered. Any request will be welcomed, although it may not be immediately reflected in the source code.
The functions of the library can be divided into several categories:

- display and error functions;
- functions manipulating the structure of the data exchanged between the wrapper and the platform;
- execution isolation functions.

All of these functions are described in the header file **WrapperCommon.h**.

A.1 Display and error functions

The role of these functions is to help the wrapper development and to give information to the user on whether or not the wrapper is functioning correctly.
Some functions have a so-called "debug" version, which implies that they are called only when the wrapper was compiled with debugging[9] options. When the compiler is called in optimized mode without debugging, these functions are eliminated from the code so as to not slow down the execution. All of these functions are prefixed by **dbg_**. The version without the prefix exists and, if it is called, it will not be eliminated from the code.

```
const char * getErrorAsString(enum WrapperErrorCode errorCode)
```

This function translates an error code (**errorCode**) returned by a wrapper function into a message readable by the user. The return value points to a constant static area of the library which it is prohibited to modify, move or deallocate. This value is read-only.

```
void dbg_printMessage(const char * functionName, const char * message)
void printUserMessage(const char * functionName, const char * message)
```

Both of these functions send the **message** string to the log system of the Open TURNS platform, also indicating from which function (**functionName**) the message is coming.
Example:

```
printUserMessage( "func_exec_P", "This is the message");
```

The **printMessage** function is filtered by the log system as coming from the wrapper, while the **printUserMessage** function is filtered as coming from the user. By default and unless the log system was set up otherwise, the first one is not displayed while the second is. Depending on the importance of the information that the wrapper developer wants to send back to the user, they may choose either one of these functions.
All of the following functions are based on the **printMessage** function.

[9] This is achieved by defining the C macro **DEBUG** when configuring the wrapper: **./configure CFLAGS=-DDEBUG**.

©2005-2012 EDF - EADS - PhiMeca

```
void dbg_printEntrance(const char * functionName)
void dbg_printExit(const char * functionName)
```

Both of these functions indicate the entrance and exit of the wrapper function **functionName** to the log system.

```
void dbg_printState(const char * functionName, void * p_state)
```

This function prints in the log system the value of the **p_state** pointer as received by the **functionName** function.

```
void dbg_printWrapperExchangedData(const char * functionName,
const struct WrapperExchangedData * p_exchangedData)
```

This function prints in the log system the entire content of the data structure exchanged between the wrapper and the Open TURNS platform. This structure contains all the information read in the wrapper's description file (**wrapper.xml**) and information coming directly from the platform or the computer on which the wrapper is running.

```
void dbg_printPoint(const char * functionName, const struct point * p_point)
```

This function prints in the log system the value of the **p_point** point as received or returned by the wrapper's **functionName** function.

```
void dbg_printSample(const char * functionName, const struct sample * p_sample)
```

This function prints in the log system the value of the **p_sample** sample as received or returned by the wrapper's **functionName** function.

```
void dbg_printMatrix(const char * functionName, const struct matrix * p_matrix)
```

This function prints in the log system the value of the **p_matrix** matrix as received by the wrapper's **functionName** function.

```
void dbg_printTensor(const char * functionName, const struct tensor * p_tensor)
```

This function prints in the log system the value of the **p_tensor** tensor as received by the wrapper's **functionName** function.

A.2 Functions manipulating the exchange data structure

These functions simplify the reading, copying or destruction of the structure used for the exchange of data between the wrapper and the platform. This structure being relatively complex and its handling not necessarily easy for users or developers reluctant to use the C language, those functions are supposed to make things simple. Furthermore, this structure often evolves through the different versions of Open TURNS and it is clearly preferable to use those functions, which are updated with the developments, in order to keep the wrapper easily maintainable.

```
unsigned long getNumberOfVariables(const struct WrapperExchangedData
* p_exchangedData,
unsigned long type)
unsigned long getNumberOfFiles(const struct WrapperExchangedData
```

```
* p_exchangedData,
unsigned long type)
```

These functions scan the exchange data structure in order to determine the number of variables or files declared in the wrapper's description file. Since these variables or files can be of type either **in** or **out**, it is necessary to pass to these functions an argument of type either **WRAPPER_IN** or **WRAPPER_OUT**.

```
long copyWrapperExchangedData(struct WrapperExchangedData
** p_p_new_exchangedData,
const struct WrapperExchangedData
* p_exchangedData)
```

This function copies the content of the **p_exchangedData** structure in the memory location designated by **p_p_new_exchangedData**. All necessary memory allocations are made by the function[10], so that when the function exits, the two structures are identical but disjoint. The function returns a non-zero error code in case of an error. In this case, the **p_p_new_exchangedData** structure is unaffected.
The created structures must mandatorily be destroyed, before the final use of the wrapper, by a call to the **freeWrapperExchangedData** function.

```
void freeWrapperExchangedData(struct WrapperExchangedData * p_exchangedData)
```

This function frees all the memory allocated to an exchange data structure and destroys this structure. It is prohibited to use this function on the structure sent by Open TURNS to the wrapper. It should only be used on structures created by the wrapper for its own needs. This function accepts null pointers.

```
const char * getCommand(const struct WrapperExchangedData * p_exchangedData)
```

This function returns the command string as stated in the wrapper's description file. It is available starting from version 0.12.2.

```
unsigned long getNumberOfCPUs(const struct WrapperExchangedData
* p_exchangedData)
```

This function returns the number of virtual processors of the computer on which the wrapper is running. This number is determined when the platform is launched. It is available starting from version 0.12.2.

A.3 Execution isolation functions

These functions allow to properly manage the parallel execution of external codes by isolating them in individual temporary directories. Each function performs one task among the many required, including creating directories, transferring the execution within them, copying files, substituting variables or initiating execution. These functions are linked as described later in this guide. The reader should refer to the appropriate sections.

```
char * createTemporaryDirectory(const char * tempDirPrefix,
const struct WrapperExchangedData
* p_exchangedData)
```

This function begins by creating a unique temporary directory name based on the prefix (**tempDirPrefix**) and information contained in the exchange data structure, including the name of the general temporary directory (**temporary-directory**) listed in the Open TURNS configuration file **openturns.conf**. Then it proceeds to

[10] The function performs a deep copy.

©2005-2012 EDF - EADS - PhiMeca

create this directory. The directory name is returned by the function through a specially allocated character string. This string must be freed, in due course, with a call to the **deleteTemporaryDirectory** function. In case of failure, the return value is a null pointer.

```
void deleteTemporaryDirectory(char * tempDir, long executionStatus)
```

This function deletes the temporary directory whose path is contained in the **tempDir** string, as well as its entire content if the **executionStatus** parameter is zero. This parameter is most often the one returned by the function that causes the execution of the computational code (see the **runInsulatedCommand** or **system(3)** functions).
The string is systematically destroyed by this function even if the **executionStatus** parameter is non-zero.

```
long changeDirectory(const char * path)
```

This function moves the wrapper's execution in the directory specified in the **path** string. A non-zero return code is returned in case of an error and an explanatory message is sent to the platform's log system.

The *ChangeDirectory* function not being multithread-safe, it is recommended not to use it anymore. It is bound to disappear in a future version of the platform.

```
char * getCurrentWorkingDirectory()
```

This function returns the current execution directory path in a specially allocated string. This string must be deallocated in due course using the **free(3)** function.

```
long createInputFiles(const char * directory,
const struct WrapperExchangedData * p_exchangedData,
const struct point * p_point)
```

This function takes the data files listed in the exchange data structure and copies them in the **directory** folder (which must have been previously created), substituting variables in the ad-hoc files with the values contained in **p_point**. It returns a non-zero error code in case of an error.

```
long readOutputFiles(const char * directory,
const struct WrapperExchangedData * p_exchangedData,
struct point * p_point)
```

This function takes the result files listed in the exchange data structure and reads them in the **directory** folder (which must exist) to find the declared variables. The values of these variables are stored in the **p_point** structure. It returns a non-zero error code in case of an error.

```
char * makeCommandFromTemplate(char * command,
const struct WrapperExchangedData
* p_exchangedData,
const struct point * p_point)
```

This function replaces, in the **command** string, the variables declared in the exchange data structure with their values in the **p_point** structure. It deallocates **command** and returns a string containing the new modified command. This string must be freed in due course by a call to **free(3)**.

```
char * insulateCommand(char * command, const char * temporaryDir)
```

©2005-2012 EDF - EADS - PhiMeca

This function changes the command (**command**) in order to make the directory change to **temporaryDir** multithread-safe. It deallocates **command** and returns a string containing the new modified command. This string must be freed in due course by a call to **free(3)**. The new string thus has the following behavior:

1. Change directory to **temporaryDir**;
2. Execute **command**;
3. Return to the original directory.

```
long runInsulatedCommand(const char * directory,
const struct WrapperExchangedData * p_exchangedData,
const struct point * p_point)
```

Using the exchange data structure, this function replaces, in the command read in the wrapper's description file, the variables declared in that same file by their values in the **p_point** structure. It then launches the execution of the substituted command in the **directory** temporary directory. This function is multithread-safe. The returned value is non-zero if an error occurs during the command execution.

©2005-2012 EDF - EADS - PhiMeca

B Macros from the WrapperMacros library

Starting with version 0.12.2 of the Open TURNS platform, a **WrapperMacros.h** file offers a set of C macros which can greatly simplify the task of writing wrappers. These macros entirely support interfacing the wrapper with the Open TURNS platform, which means the developer only needs to write the functions' content purged from any programming clutter.
This, however, requires to make some choices in terms of how the wrapper operates. These macros thus assume that the internal state only contains one copy of the exchange data structure. In most cases, this should not be a problem. Occasionnally, the developer will not be able to rely on these macros, and they will have to use conventional function calls and the functions from the wrapper's library.

```
CAST (type, arg)
```

This macro converts the **arg** argument into the specified type.

```
SET_INFORMATION_FROM_EXCHANGED_DATA( p_exchangedData )
```

This macro fills a structure named **p_info** based on data contained in the **p_exchangedData** exchange data structure.

```
GET_EXCHANGED_DATA_FROM( pointer )
```

This macro converts the **pointer** argument into an exchange data structure called **p_exchangedData**.

```
COPY_EXCHANGED_DATA_TO( pointer )
```

This macro copies the content of the exchange data structure called **p_exchangedData** into the **pointer** argument.

```
DELETE_EXCHANGED_DATA_FROM( pointer )
```

This macro destroys the exchange data structure contained in the **pointer** argument.

```
CHECK_WRAPPER_MODE( mode )
```

This macro checks that the mode specified for the wrapper in the description file is indeed **mode** or returns a **WRAPPER_USAGE_ERROR** error.

```
CHECK_WRAPPER_IN( mode )
```

This macro checks that the **in** parameter specified for the wrapper in the description file is indeed **mode** or returns a **WRAPPER_USAGE_ERROR** error.

```
CHECK_WRAPPER_OUT( mode )
```

This macro checks that the **out** parameter specified for the wrapper in the description file is indeed **mode** or returns a **WRAPPER_USAGE_ERROR** error.

```
PRINT( message )
```

This macro sends the **message** string message to the platform's log system.

```
INPOINT_ARRAY
OUTPOINT_ARRAY
```

©2005-2012 EDF - EADS - PhiMeca

These macros return pointers to the data tables contained in the **point** structures of the execution functions.

```
INPOINT_SIZE
OUTPOINT_SIZE
```

These macros return the size of the data tables contained in the **point** structures of the execution functions.

```
INPOINT_COORD( i )
OUTPOINT_COORD( i )
```

These macros return the value of the ith data table contained in the **point** structures of the execution functions. Following the C convention, the index numbering starts at zero.

```
FUNC_INFO( name, code )
FUNC_CREATESTATE( name, code )
FUNC_DELETESTATE( name, code )
FUNC_INIT( name, code )
FUNC_EXEC( name, code )
FUNC_EXEC_SAMPLE( name, code )
FUNC_FINALIZE( name, code )
GRAD_INFO( name, code )
GRAD_CREATESTATE( name, code )
GRAD_DELETESTATE( name, code )
GRAD_INIT( name, code )
GRAD_EXEC( name, code )
GRAD_FINALIZE( name, code )
HESS_INFO( name, code )
HESS_CREATESTATE( name, code )
HESS_DELETESTATE( name, code )
HESS_INIT( name, code )
HESS_EXEC( name, code )
HESS_FINALIZE( name, code )
```

These macros define the functions after which they are named with the **name** suffix (which must be the same as the one stated in the wrapper's description file), and whose active part is described by **code**.
The **name** argument is a standard C identifier, but not a string. Therefore, it must not be placed in quotes. Similarly, **code** is a set of C instructions that we take care to write between an opening brace and a closing brace, in order to avoid any interpretation problem.
It is often useful to declare a wrapper-specific macro which bears the name of the Function, of the Gradient and of the Hessian one wants to code. This can be done as follows:

```
#define WRAPPERNAME myCode
FUNC_INIT( WRAPPERNAME, {} ) /* Nothing to do in func_init_ */
FUNC_EXEC( WRAPPERNAME,
{
  /* Here we enter what to do in func_exec_ */
} )
```

A set of opening and closing braces without any content is used to indicate that nothing is to be executed in this function.

©2005-2012 EDF - EADS - PhiMeca

```
FUNC_EXEC_SAMPLE_MULTITHREADED( name )
```

This macro replaces the classical **FUNC_EXEC_SAMPLE** macro and provides an implementation based on the **func_exec_** function, whether it be declared through the **FUNC_EXEC** macro or manually, in order to perform a multithreaded sample calculation. It is necessary that the function **func_exec_** be multithread-safe.

```
FUNC_EXEC_BODY_CALLING_COMMAND_IN_TEMP_DIR( prefix )
```

The execution of the external code by the use of files and of a call to a command isolated in a temporary directory is so common that this macro replaces the code that a developer should write in their **FUNC_EXEC** macro. Only the prefix, which will allow to easily find the temporary execution directory, needs to be specified. This macro is used as follows:

```
#define WRAPPERNAME myCode
FUNC_EXEC( WRAPPERNAME,
FUNC_EXEC_BODY_CALLING_COMMAND_IN_TEMP_DIR( "myCode" ) )
```

©2005-2012 EDF - EADS - PhiMeca

C Description of the wrapper's development structures

There are three directories containing the templates used to interface Open TURNS with an external code written in C, in FORTRAN 77 or available as a command that can be called from the shell.
These directories share the same structure described here. The specifics of each directory are outlined in the relevant sections.
The developer interested in any of these directories must start by copying it entirely in a personal directory. We will assume that the copy is made in the developer's home directory (**$HOME**) that the produced wrapper will be called **myWrapper**.

```
> cp -r <template_dir> $HOME/myWrapper
```

Then, after ensuring that they have the **autoconf**, **automake** and **libtool** tools, the developer enters the newly copied directory and initializes the development environment. This step is performed only once.

```
> cd myWrapper
> sh customize myWrapper
> sh bootstrap
```

> *Note: The* **customize** *script modifies a number of files in which the name of the wrapper (***myWrapper** *in our case) must appear.*

At this stage, a new **configure** file has been created in the directory. This file will allow to continue with the procedure. This executable requires, however, two pieces of information:

- the name of the installation directory for the produced wrapper. By default, this wrapper will be written in the directory **$HOME/openturns**, which is a standard directory for Open TURNS to search for wrappers. However, if you choose to install the wrapper elsewhere, you will have to declare this path in the **OPENTURNS_WRAPPER_PATH** environment variable.
- the Open TURNS installation directory, i.e the directory providing access to **lib/openturns/libOT.so**. We will call this directory **openturns_dir**.

```
> ./configure --with-openturns=<openturns_dir>
```

This command produces the **Makefile** files needed to compile the wrapper.

```
> make
> make install
```

All of the directories contain a file called **wrapper.c** defining symbols for functions expected by the Open TURNS library. In some cases, these functions are written as macros or as a traditional C code.
Finally, a Python test file allows to control the minimal functioning of the produced wrapper.

```
> python test.py
```

C.1 wrapper_linked_with_C_function directory

This directory contains two additional files that encode a hypothetical function written in C and serving as the external code:

©2005-2012 EDF - EADS - PhiMeca

myCFunction.h the header file declaring the function;

myCFunction.c the source file defining the function.

The function here is called **myCFunction** and shows how to pass the input vector (**inPoint**) as an argument and how to retrieve the result in an output vector (**outpoint**). We remind here that no memory allocation is needed.
The **wrapper.c** file imports the definition of **myCFunction** using the directive:

```
#include "myCFunction.h"
```

It explicitely calls this function in the **FUNC_EXEC** macro.

C.2 wrapper_linked_with_F77_function directory

This directory contains only one additional file encoding a hypothetical function written in FORTRAN, which acts as the external code:

code.f the source file defining the function.

The function here is called **COMPUTE** and shows how to pass the input vector (**inPoint**) as an argument and how to retrieve the result in an output vector (**outpoint**). We remind here that no memory allocation is needed.
Since there is no header file allowing to import the **wrapper.c** file where the **COMPUTE** function is declared, and since it is necessary to adapt the symbols between both languages C and FORTRAN, we must call the **F77_FUNC** macro defined by **autoconf**. This macro produces a new symbol to be called **COMPUTE_F77**, which will be called directly in the **FUNC_EXEC** macro.

C.3 wrapper_calling_shell_command directory

This directory contains two files used to simulate an external code called from the shell command line:

code_C1.c the self-standing source code processing the numerical input data;

code_C1.data the file containing the numerical input data expected by **code_C1**.

At compile time, the **code_C1.c** file will be used to produce the **code_C1** executable, which takes two files as arguments on its command line:

code_C1.data the name of the input file containing the numerical data;

code_C1.res the name of the output file containing the numerical results.

```
> ./code_C1 code_C1.data code_C1.res
```

The macros listed in the **wrapper.c** file, and the file and variable description in the **myWrapper.xml.in** file, induce that the call to the **code_C1** external code is correctly performed, as well as the data transfer for input and output.

©2005-2012 EDF - EADS - PhiMeca

D DTD for the wrapper's description file

This file allows to check the syntax of the wrapper's description file.

```
<?xml encoding="ISO-8859-1"?>
<!-- LastChangedBy   : $LastChangedBy: dutka $ -->
<!-- LastChangeddate : $LastChangedDate: 2008-09-22 11:34:11 +0200 $ -->

<!ELEMENT wrapper           (library,external-code) >
<!ELEMENT library           (path,description) >
<!ELEMENT description       (variable-list,function,gradient,hessian) >
<!ELEMENT external-code     (data?,wrap-mode,command) >
<!ELEMENT data              (file*) >
<!ELEMENT file              (name?,path,subst?) >
<!ELEMENT variable-list     (variable*) >
<!ELEMENT variable          (comment?,unit?,regexp?,format?) >
<!ELEMENT function          (#PCDATA) >
<!ELEMENT gradient          (#PCDATA) >
<!ELEMENT hessian           (#PCDATA) >
<!ELEMENT name              (#PCDATA) >
<!ELEMENT path              (#PCDATA) >
<!ELEMENT subst             (#PCDATA) >
<!ELEMENT regexp            (#PCDATA) >
<!ELEMENT format            (#PCDATA) >
<!ELEMENT unit              (#PCDATA) >
<!ELEMENT comment           (#PCDATA) >
<!ELEMENT wrap-mode         (in-data-transfer,out-data-transfer) >
<!ELEMENT command           (#PCDATA) >
<!ELEMENT in-data-transfer  (#PCDATA) >
<!ELEMENT out-data-transfer (#PCDATA) >

<!ATTLIST file
id ID #REQUIRED
type (in|out) #REQUIRED>

<!ATTLIST variable
id ID #REQUIRED
type (in|out) #REQUIRED
computed-gradient (yes|no) #IMPLIED>

<!ATTLIST wrap-mode
type (static-link|dynamic-link|fork) #REQUIRED
state (shared|specific) #IMPLIED>

<!ATTLIST in-data-transfer
mode (files|pipe|arguments|socket|CORBA) #REQUIRED>

<!ATTLIST out-data-transfer
mode (files|pipe|arguments|socket|CORBA) #REQUIRED>
```

©2005-2012 EDF - EADS - PhiMeca

```
<!ATTLIST function
provided (yes|no) #REQUIRED>

<!ATTLIST gradient
provided (yes|no) #REQUIRED>

<!ATTLIST hessian
provided (yes|no) #REQUIRED>
```

©2005-2012 EDF - EADS - PhiMeca

Index

central term, 4, 8, 9, 18–21, 23, 24
changeDirectory, 39, 49
compilation, 14, 17, 54
copyWrapperExchangedData, 48
createInputFiles, 41, 49
createState_, 24
createTemporaryDirectory, 39, 48

data structure, 4–9, 15, 18, 22, 26, 28–30
deleteState_, 24
deleteTemporaryDirectory, 39, 49
description file, 7–10, 12, 14–17, 21, 23, 26, 28, 33, 38, 41, 43, 56
DTD, 14, 17, 56
dynamic library, 7, 9, 11, 12, 14, 15, 17, 20, 21, 34

exec_, 23
exec_sample_, 23, 24, 43
execution, 6, 19–25, 38

finalization, 22–25
finalize_, 23
FORTRAN, 3, 6, 13, 17, 31, 38, 54
freeWrapperExchangedData, 48
Function, 4, 5, 8, 12, 13, 15–18, 20, 21, 23, 24, 31, 33, 43
function, 4, 5, 8, 13, 18–21, 23–25

getCommand, 48
getCurrentWorkingDirectory, 39, 49
getErrorAsString, 26
getInfo_, 23
getNumberOfCPUs, 48
getNumberOfFiles, 48
getNumberOfVariables, 23, 27, 48
Gradient, 4, 8, 13, 15, 16, 20, 21, 24, 29, 33, 43
gradient, 4, 8

Hessian, 4, 8, 13, 15, 16, 20, 21, 24, 29, 30, 33, 43
hessian, 4, 8

init_, 23
initialisation, 24
initialization, 22, 23, 25
installation, 8, 14, 18
installation directory, 8, 13, 14, 39, 54
insulateCommand, 50
internal state, 18, 24–28, 30, 43

macro, 26–30, 32, 42, 51
makeCommandFromTemplate, 49

NumericalMathFunction, 4, 5, 8, 9, 12, 19–24

prefix, 4, 18, 20, 23
printEntrance, 47
printExit, 47
printMatrix, 47
printMessage, 46
printPoint, 47
printSample, 47
printState, 47
printTensor, 47
printUserMessage, 46
printWrapperExchangedData, 47

readOutputFiles, 41, 49
regular expression, 6, 36
regular expressions, 37–39, 41, 45
return code, 19, 25
runInsulatedCommand, 39, 42, 50

sample, 23, 29, 33
substitution, 39, 41, 45

wrapper, 3, 5–8, 10–21, 23, 24, 31, 33, 38, 45, 46
wrapper search, 8, 14, 19, 54
wrapper source code, 13, 17, 19, 20, 23, 31, 33, 34, 54
Wrapper.h, 17
WrapperCommon.h, 25, 46
WrapperInterface.h, 25, 26