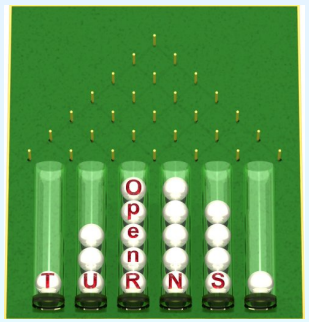
Compilation infrastructure (1/2)

Trainer : Julien Schueller
Phimeca
schueller@phimeca.com

Developers training



OpenTURNS compilation infrastructure

The autotools

The autotools are a set of tools that aim to ease the configuration and compilation of large software projects in the UNIX world. The objective is to generate Makefiles from a set of templates and the information gathered during a configuration step. The main tools are:

- *aclocal* in charge of the management of the several *detection macros* needed for the configuration of the project: the dependencies, the compilers and so on.
- *automake* in charge of producing parameterized Makefiles (*Makefile.in*) from template Makefiles (*Makefiles.am*).
- *autoconf* in charge of the parameterization of both the Makefiles and the sources of the project (notably for the conditional compilation of parts of the project). The main purpose of this tool is to produce a shell script (*configure*) based on a template (*autoconf.ac*) and the macros gathered by aclocal (*aclocal.m4*). This shell script converts the parameterized Makefiles (Makefile.in) into ready to use Makefiles.
- *autotest* in charge of the unit tests. Such a test is described as an association between a shell script command to be executed and a reference standard output and error output that is expected. The validation is done using a character-based comparison of the shell script output and the reference output, and through the return code of the shell script command.

Two compilation infrastructures

CMake

CMake is another compilation infrastructure with the same objectives as the autotools. All the configuration is done through a hierarchy of text files written in the CMake macro language, and a GUI is available to ease the creation of this hierarchy. The same topics are covered:

- *dependency detection* through a set of detection macros: the several .cmake files;
- *configuration* through a master configuration file: the top-level CMakeLists.txt file;
- *source organization* through a set of CMakeLists.txt files disseminated in the whole source tree: each such file includes the declaration of the several source files and associated header files, and make a recursive call to the subdirectories.
- *testing* using a mechanism that is not completely clear to me at this time...

Two main situations

There are two distinct situations in the development of additional capabilities of OpenTURNS:

- The addition of a new instance of an existing concept;
- The introduction of a new concept.

The associated development process shares the same principles in both cases, but the details are more involved in the second case.
Both cases are covered in the Contribution Guide documentation that comes with OpenTURNS, only the first situation will be covered here. We suppose that our extension consist in the creation of a new class called MyClass in an existing directory.

Populate an existing concept

Step 1: create the header file and the associated source file

Create MyClass.hxx and MyClass.cxx in the same directory. The files must have the standard OpenTURNS header, with a brief description of the class using the Doxygen format and the standard reference to the LGPL license.
For the header file MyClass.hxx, the interface must be embraced between the preprocessing clauses:

```
#ifndef OPENTURNS_MYCLASS_HXX
#define OPENTURNS_MYCLASS_HXX
...
your interface
...
#endif OPENTURNS_MYCLASS_HXX
```

to prevent from multiple inclusions.
See any pair of .hxx/.cxx files in the current directory and the OpenTURNS Coding Rules document as a guide for your development: the use of namespaces, case convention for the static methods, the other methods and the attributes, the trailing underscore for the attribute names to name a few rules.

Populate an existing concept

Step 2: update the automake file and the CMake file

Modify the Makefile.am file in the directory containing MyClass.hxx and MyClass.cxx:

- add MyClass.hxx to the `otinclude_HEADERS` variable
- add MyClass.cxx to the `libOTXXXXXX_la__SOURCES` variable, where XXXXXX is the name of the current directory.

Modify the CMakeList.txt file in the same directory:

- add MyClass.hxx using the instruction `ot_install_header_file ( MyClass.hxx )`
- add MyClass.cxx using the instruction `ot_add_source_file ( MyClass.cxx )`

Step 3: the source code of the test(s)

Create a test file t_MyClass_std.cxx in the directory lib/test. This test file must check at least the standard functionalities of the class MyClass. If relevant, some specific aspects of the class can be checked in specific other test files, such as the exceptional behaviour of the class or its functionalities in extrem configurations (large data set, hard to solve problems etc.).

Step 4: the autotest file(s) of the test(s)

Create an autotest file t_MyClass_std.at in the directory lib/test. This file describes the test, how to run it and what is the expected output (copy-paste the *validated* output of the test in the proper section of t_MyClass_std.at).
For the CMake infrastructure, there is no such step.

Step 5: update the automake file and the CMake file of the lib/test directory

- add t_MyClass_std (which is the name of the test executable) to the variable CHECK_PROGS or INSTALLCHECK_PROGS depending on the fact the test checks the correct behaviour of OpenTURNS independently of its installation or not. The several executables are organized following the library organization, you must follow this rule.
- add t_MyClass_std.at to the variable CHECK_TESTS or INSTALLCHECK_TESTS and in the correct set of autotest files, following the same rules than for the executable.
- Create a variable called t_MyClass_std_SOURCES and set its value to t_MyClass.cxx in the relevant set of sources.

For the CMake infrastructure, add the line ot_installcheck_test (MyClass_std) in the relevant section of the CMakeLists.txt file.

Step 6: update the autotest infrastructure

Add t_MyClass_std.at to the file check_testsuite.at or installcheck_testsuite.at using the same rule than for the Makefile.am modification.
If the test checks functionalities available after the installation of OpenTURNS, use the installcheck_testsuite.at file as your test is a post-installation test, else use the check_testsuite.at file.
There is no such step in the CMake infrastructure.

Step 7: validation

If the validation of your class involved advanced mathematics, or was a significant work using other tools, you can add this validation in the validation/src directory.

- copy all of your files in the validation/src directory.
- modify the Makefile.am file by appending the list of your files to the dist_validation_DATA variable.

Document the new class

Step 8: update the documentation

The documentation must be written in English, using LaTeX. For an addition to the C++ library, you may have to update the following documents in the OpenTURNS documentation source tree:

- Add an entry in the document src/ArchitectureGuide/OpenTURNS_ArchitectureGuide.tex if your class has a significant impact on the library architecture.
- Add an entry in the document src/WrappersGuide/OpenTURNS_WrappersGuide.tex if your class has a significant impact on the way OpenTURNS interfaces external codes.
- Add an entry in the document src/ReferenceGuide/OpenTURNS_ReferenceGuide.tex if your class add a new concept not already described in the reference guide. Your entry must take the form of a specific description using the same template than the other descriptions.

Critical points

- All the classes must include the `CLASSNAME` macro (defined in Base/Common/Object.hxx) in their header file in order to benefit from the (basic) introspection mechanisms. The associated `CLASSNAMEINIT` macro must be used in the corresponding source file.
- All the class corresponding to persistent objects must instantiate a static parameterized factory in their source file.
- In order to improve the readibility of the source code, the needed classes that are not in the current namespace must be aliased using a typedef. These typedef must be wisely separated between those in the header file and those in the source file.
- The const correctness of the code is very important, both for the signature of the methods and for the temporary variables.
- All the object arguments must be passed using const references. The use of non const references to make side effects must be limited as much as possible.
- Most of the coding rules are described in the Coding Rules Guide, but you can infer the rules by looking at the existing code. The key point is that the only difficult points should be the conception and the algorithms, not the indentation or the coding style!

Practical case: adding a new distribution to the C++ library

- Each trainee has to implement a new distribution in the C++ library, this distribution being chosen without replacement in an urn containing a dozen of distributions.
- From an algorithmic point of view, the minimum to do is to implement the `NumericalScalar computeCDF(const NumericalPoint & point)` method.
- From a development process point of view, each trainee is expected to go through at least the 6 first steps.
- The other methods should be added in the following order:
  1. `NumericalScalar computePDF(const NumericalPoint & point)`
  2. `NumericalPoint getRealization()`
  3. `NumericalScalar computeScalarQuantile(const NumericalScalar prob, const Bool tail, const NumericalScalar precision)`
  4. `void computeMean() const`
  5. `void computeCovariance() const`

C++/Python binding

A user-friendly interface for the OpenTURNS library

OpenTURNS is intended to be used for complex industrial application. It means the ability to pilot complex simulation softwares, but also complex probabilistic modelling and involved strategies for uncertainty propagation. A typical graphical user interface does not provide the flexibility to address such needs, so OpenTURNS is proposed to the user as a Python module.

Python is a full-featured object oriented programming language, and allows for complex scripting of functionalities comming from numerous modules. A typical uncertainty propagation study can be fully implemented using OpenTURNS only, but it can be easier to delegate some treatments to other graphical, statistical or numerical packages. For complex studies, it is the only way to do the job.

The standard extension mechanisms proposed by Python to bind an external library are very low level mechanisms. It is mainly a C interface through which all the types are lost: the arguments are mainly `void *` pointers, and a lot of transtyping is required in order to make the things work.

Several higher level tools have been developped in order to ease this binding, one of the most advanced being SWIG.

SWIG: Simplified Wrapper and Interface Generator

A tool to link C/C++ library with script languages

SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages. SWIG is used with different types of target languages including common scripting languages such as Perl, PHP, Python, Tcl and Ruby. The list of supported languages also includes non-scripting languages such as C#, Common Lisp (CLISP, Allegro CL, CFFI, UFFI), D, Go language, Java, Lua, Modula-3, OCAML, Octave and R. Also several interpreted and compiled Scheme implementations (Guile, MzScheme/Racket, Chicken) are supported. SWIG is most commonly used to create high-level interpreted or compiled programming environments, user interfaces, and as a tool for testing and prototyping C/C++ software. SWIG is typically used to parse C/C++ interfaces and generate the 'glue code' required for the above target languages to call into the C/C++ code.

Python, C++ and SWIG

Some of the (numerous) features of the C++ language have no equivalent in the Python language. Thus, there is a choice to be made on how to expose these features in Python. Two specific features are of interest in the OpenTURNS context:

- nested classes (a classe that is defined inside another class);
- parametric classes (no template concept in Python)

The new versions of SWIG (2.0.0 and later versions) are supposed to improve the support of these features, which means that they propose a standard way to expose these features without the help of the developer.
Whereas it is often acceptable to abandon the nested classes in the C++ part without compromising too much the architecture, the parametric classes are more problematic. Some clues will be given in the development process part of this course.

Step 9: create the SWIG interface file

In order to make the new class visible in the OpenTURNS Python module, you have to create a specific SWIG interface file, namely the file MyClass.i in the python/src directory. In most situations, it should be as simple as:

```
// SWIG file MyClass.i
// Author :  $LastChangedBy:  dutka $
// Date :  $LastChangedDate:  2007-03-07 15:50:39 +0100 (mer.  07 mars 2007) $
// Id :  $Id:  Triangular.i 345 2007-03-07 14:50:39Z dutka $

%{
#include "MyClass.hxx"
%}

%include MyClass.hxx
namespace OpenTURNS { namespace NameSpace1 { namespace NameSpace2 {
%extend MyClass { MyClass(const MyClass & other) {
return new OpenTURNS::NameSpace1::NameSpace2::MyClass(other);
} } }}}
```

supposing that your class is in the namespace `OpenTURNS::NameSpace1::NameSpace2`.

Integration of the new class in the TUI

Step 11: integrate the SWIG interface file into the whole Python interface

- Modify the Makefile.am file in python/src: add MyClass.i to the variable OPENTURNS_SWIG_SRC
- Locate in which of the Python submodule SWIG file you have to include MyClass.i (look for the file corresponding to the last level of namespace of your class)

Integration of the new class in the TUI

Step 12: test the new class in the Python module

- Create a test file t_MyClass_std.py in the directory python/test. This test implements the same tests than t_MyClass_std.cxx, but using python.
- Create an autotest file t_MyClass_std.atpy that has the same role than t_MyClass_std.at, but for the python test.
- Modify the Makefile.am file in python/test:
  - add t_MyClass_std.py to the variable PYTHONINSTALLCHECK_PROGS. The several executables are organized following the library organization, you must follow this rule.
  - add t_MyClass_std.atpy to the variable PYTHONINSTALLCHECK_TESTS.

Step 12: document your new class in the TUI documentation

Comment your python test as a new use-case in the document src/OpenTURNS_UseCasesGuide/UseCasesGuide.tex following the generic format of this document:

- describe the inputs of your use-case.
- extract code snippets that show the user interaction with your class.
- add the relevant keywords to the index.

Gives a description of your class in the document src/UserManual/OpenTURNS_UserManual.tex

- following the general form of this document, fill-in the sections but only describe the methods the user is intended to use (forget the most computer programming inclined methods).
- give some reminders of theoretical aspects if needed, in the form of an equation or a short (1 or 2 sentences) mathematical explanation. Give a pointer to the relevant reference guide section.

Pitfalls, tips and tricks

Python does not support nested classes. As such, SWIG does not propose any automatic mechanism to expose such classes in Python. The solution retained in OpenTURNS is to typedef the instanciations of the parametric classes to explicit new classes. Example:

- In the C++ library:

```
template <class T> class Collection
typedef Collection< Distribution > DistributionCollection;
```

- In the SWIG interface file:

```
% template(DistributionCollection) OpenTURNS::Base::Type::
Collection<OpenTURNS::Uncertainty::Model::Distribution>;
```

For the nested classes, no reasonable solution has been found: we had to unnest the class in the SWIG interface file, creating C++ source code to be maintained in the SWIG interface. We decided to do this job in the C++ library instead.

Automatic conversion between C++ types into Python types

The automatic conversion of types is needed both to ease the writing of OpenTURNS scripts by Python users. Two distinct cases are of concern with OpenTURNS:

- The automatic conversion between Python lists/arrays and OpenTURNS collections;
- The automatic promotion of implementation classes into interface classes.

The first point is adressed both at the Python level and the C++ level:

- A set of parametric wrapping methods are defined in a C++ header (see PythonWrappingFunction.hxx in python/src);
- All the parametric classes are extended at the SWIG level with constructors from Python objects, using these wrapping methods.

The second point is due to the lack of capabilities of SWIG to identify correctly the Bridge pattern and use the existing constructors in order to perform the automatic conversions. It results in a need to make these conversions explicitely in the Python scripts, which is not natural for a Python programmer. The solution retained in OpenTURNS is to use the `typemap` service of SWIG and the wrapping methods in order to make these conversions automatic for the Python programmer (see Distribution.i in python/src).