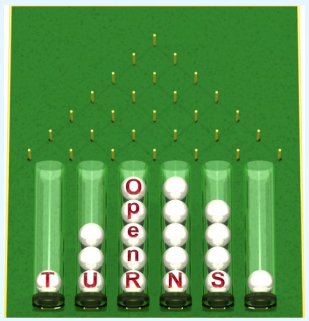
Library development

Trainer : Julien Schueller
Phimeca
schueller@phimeca.com

Developers training



CMake compilation infrastructure

CMake

Basics

CMake is an extensible, open-source system that manages the build process in an operating system and in a compiler-independent manner and covers:

- The detection and configuration aspects of the platform;
- The dependency management of the sources;
- The generation of parallel makefiles;
- The regression tests.

Some core ideas of CMake:

- All the configuration is done through a hierarchy of text files written in the CMake macro language.
- Unlike many cross-platform systems, CMake is designed to be used in conjunction with the native build environment.

The CMake compilation infrastructure

CMake

It consists configuration files placed in each source directory (called CMakeLists.txt files) and cover:

- *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
- *dependency detection* through a set of detection macros: the several .cmake files; such file includes the declaration of the several source files and associated header files, and make a recursive call to the subdirectories.
- *testing* using the associated CTest testing utility, responsible for running C++ and Python tests.

C++ development process

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 Developer 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.cxx and openturns/MyClass.hxx in a subdirectory of lib/src. The files must have the standard 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 to prevent from multiple inclusions.

```
#ifndef OPENTURNS_MYCLASS_HXX
#define OPENTURNS_MYCLASS_HXX
...
class OT_API MyClass ...;
...
#endif OPENTURNS_MYCLASS_HXX
```

See any pair of .hxx/.cxx files in the current directory and the coding section 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 CMake source configuration

Modify the CMakeList.txt file in some lib/src subdirectory:

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

At this point you can build the library with the new class:

```
make
```

Populate an existing concept

Step 2b: update the subdirectory header

Modify the OTsubdir.hxx file in lib/src/subdir/openturns/ to include the new header:
`#include "openturns/MyClass.hxx"`

Populate an existing concept

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.).

Populate an existing concept

Step 4: update the CMake test configuration

Modify the CMakeList.txt file in the lib/test directory:
`ot_check_test (MyClass_std)` At this point you can build and run the specific test:

```
make t_MyClass_std
./lib/test/t_MyClass_std
```

Populate an existing concept

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

Add an expected test result named t_MyClass_std.expout:
`./lib/test/t_MyClass_std > ../lib/test/t_MyClass_std.expout`
The content of this file is compared to the output of the cxx test executable when tests are run.
At this point you can run the specific test:
`ctest -R cppcheck_MyClass_std`

Populate an existing concept

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.

Tips and tricks

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.
- All the classes must include the `OT_API` macro in their header file in order to export their symbols for win32 targets.
- Implementation classes must provide a clone method.
- Use std::copy to mode chunks of contiguous data

Tips and tricks

Critical points

- The const correctness of the code is very important, both for the signature of the methods and for the temporary variables.
- Simple types must use OpenTURNS typedefs (Bool, UnsignedInteger, Scalar, Complex)
- Simple types attributes must be initialized in constructors.
- 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 section, 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!

Development of a new distribution

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

- From an algorithmic point of view, the minimum to do is to implement the `Scalar computeCDF(const Point & 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. `Scalar computePDF(const Point & point)`
  2. `Point getRealization()`
  3. `Scalar computeScalarQuantile(const Scalar prob, const Bool tail, const Scalar precision)`
  4. `void computeMean() const`
  5. `void computeCovariance() const`

SWIG

OpenTURNS Python bindings

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 developed in order to ease this binding, one of the most advanced being SWIG.

SWIG: Simplified Wrapper and Interface Generator

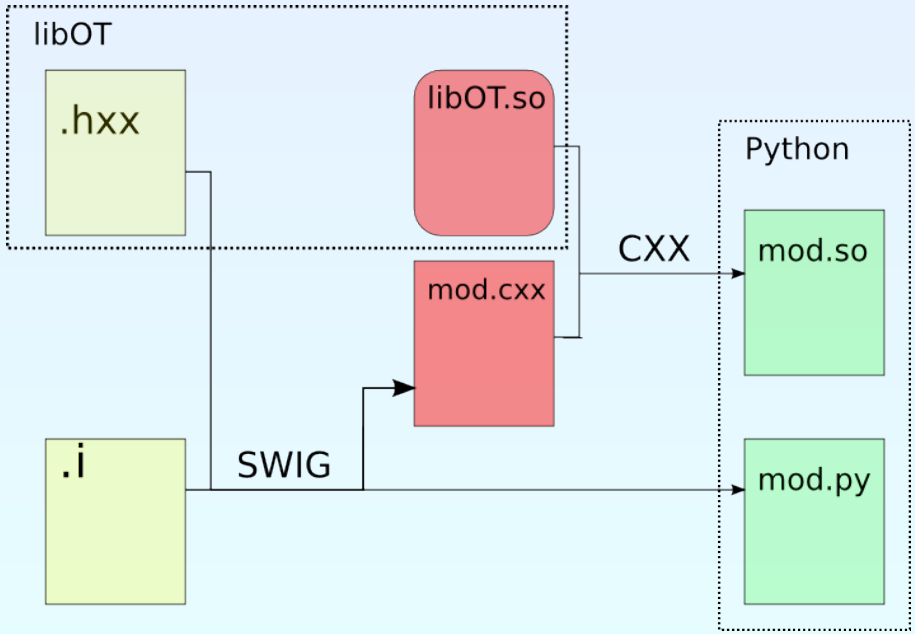
A tool to link C/C++ library with script languages

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.

SWIG: Simplified Wrapper and Interface Generator

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

SWIG needs a set of interface files (.i) with custom glue code.
SWIG parses the library headers and swig interface files to generates the corresponding module source yet to be compiled to produce a binary python module, see.
The process is split between several modules for modularity and to speed up compilation time with parallel builds.



Integration of the new class in the Python bindings

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

% {
#include "MyClass.hxx"
%}

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

Integration of the new class in the Python bindings

Step 10: integrate the SWIG interface file into the whole Python interface

- Locate in which of the Python submodule SWIG file you have to include MyClass.i (look for classes from the same source subdirectory)
- Add MyClass.i in the SWIG module source python/src/xxx_module.i
- Modify the CMakeLists file in python/src: add MyClass.i to the identified SWIG module

Integration of the new class in the Python bindings

Step 11: 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 expected result file t_MyClass_std.expout for the python test.
- Modify the CMakeLists file in python/test: `ot_pyinstallcheck_test (MyClass_std)`

Integration of the new class in the Python bindings

Step 12: write the docstring documention

Create a docstring file python/src/MyClass_doc.i.in:
This file contains the documentation of new methods (no need to document inherited methods).
It follows the Numpydoc docstring convention.

Integration of the new class in the Python bindings

Step 13: add an entry in the API manual

- Find the right .rst in in python/doc/user_manual;
- Add an entry in the restructured text format:

Integration of the new class in the Python bindings

Step 14: add an entry in the examples manual (optional)

- Add a .ipynb in a python/doc/examples subdirectory
- Add an entry in the restructured text file in that directory

Integration of the new class in the Python bindings

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) OT::
Collection<OT::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.

Integration of the new class in the Python bindings

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).