



Developing the OpenTURNS documentation

Joseph MURÉ - EDF R&D

February 14th-17th 2022

OpenTURNS Consortium

A few words about the development infrastructure (1/2)

Versioning system

- **Git** is a software versioning and a revision control system started by Linus Torvalds. It is used for the sources of the platform, the documentation, as well as for the development of modules.
- **GitHub** is a web-based hosting service for version control using Git. GitHub provides a bug tracker, pull requests (GitLab equivalent: merge requests), continuous integration, and other features.

Official repository and forks

The **official repository** of the library is <https://github.com/openturns/openturns>.

- All developers have a **fork** of that repository to host their development branches.
- Only the **integrator** (and a backup) has write access to the official repository.
- Modules are not part of the library, but some of them are maintained by the consortium: their repositories are hosted at <https://github.com/openturns> and they have the same development process as the library.
- Other modules are hosted elsewhere and do not need to follow the same development process.

A few words about the development infrastructure (2/2)

Structure of the library repository

The repository of the library contains source code for the C++ library, the Python interface and the documentation. Its layout is quite standard:

- A **master branch** that stores the source code of the upcoming version of the platform. The rule is to have only source code that passes all tests.
- Branches for **stable versions** and **tags for releases** and release candidates.
- Development branches are part of **contributor forks**.
- Development branches are merged with master via **pull requests**.
- At least **2 reviewers must approve** the PR, and unanimity is required.

Continuous integration

OpenTURNS makes extensive use of online continuous integration providers to **test the library** throughout the development process. Currently the following services are used to test the library:

- CircleCI: Linux and MinGW
- GitHub Actions: macOS and Windows

OpenTURNS development workflow



Overview of the OpenTURNS documentation

The OpenTURNS documentation is built using the Python [Sphinx](#) package.

One of the Continuous Integration machines (CircleCI `build-linux`) produces the documentation and publishes it online. Several versions are online:

- The version corresponding to the current release has [latest](#) in its URL:
`https://openturns.github.io/openturns/latest/`
- The version corresponding to the master branch has [master](#) in its URL:
`https://openturns.github.io/openturns/master/`
- Older versions are accessible with the version number:
`https://openturns.github.io/openturns/1.17/`

If you spot a mistake, make sure you are looking at `master` before correcting!

The built-in search engine distinguishes between 3 main sections:

- [Examples](#)
- [API](#) (user manual)
- [Theory](#)

You may also want to check out:

- [Contribute](#) (developer guide)
- [Common use cases](#) (added in 2020)

1. Theory
2. Examples
3. API (user manual)
4. Tips and tricks

Looking up the source file of a theory page

Let us consider the [Using QQ-plot to compare two samples](#) page:

https://openturns.github.io/openturns/master/theory/data_analysis/qqplot_graph.html

All Theory files are RST and the URL fits the source folder structure, so the relevant source file is: `python/doc/theory/data_analysis/qqplot_graph.rst`

There are great RST and Sphinx [cheatsheet](#) available, but here are the basics:

- A page can be [indexed](#) as `page_name` by writing `.. _page_name:` at the top.
- Because of this, that page can be [linked](#) to from anywhere in the doc with `:ref: 'page_name'`
- [Titles](#) are underlined (the number of underlining characters must match the title).
- [Inline math](#) is \LaTeX code with `$a=b$` replaced with `:math: 'a=b'`.
- [Display math](#) is \LaTeX code with `$$a=b$$` replaced with
$$.. math::$$
$$a = b$$
- [Macros](#) are defined in `python/doc/math_notations.sty`
- Try to use notations [consistent](#) with other theory pages.
- The [bibliography](#) can be found at `python/doc/bibliography.rst`

Adding a new theory page

Code of the main table of contents (python/doc/theory/theory.rst):

```
1  .. _theory:
2
3  =====
4  Theory
5  =====
6
7  The theoretical documentation.
8  This contains an in-depth description of all algorithms.
9
10 .. toctree::
11     :maxdepth: 2
12
13     data_analysis/data_analysis.rst
14     probabilistic_modeling/probabilistic_modeling.rst
15     meta_modeling/meta_modeling.rst
16     reliability_sensitivity/reliability_sensitivity.rst
17     numerical_methods/numerical_methods.rst
```

Table of contents of subdirectories are essentially the same.

What holds for the **Theory** section also mostly holds for the **Contribute** (developer guide) and **Common use cases** sections.

1. Theory

2. Examples

3. API (user manual)

4. Tips and tricks

Looking up the source Python script of an example

The Examples section is produced by **Sphinx-gallery**, a Sphinx extension.

Let us consider the **Kriging with an isotropic covariance function** example:

https://openturns.github.io/openturns/master/auto_meta_modeling/kriging_metamodel/plot_kriging_isotropic.html

Example source files are **Python scripts**. The URL is still linked to the the source folder structure, but the `auto_` part is a Sphinx-gallery artifact. Its source file is:

```
python/doc/examples/meta_modeling/kriging_metamodel/  
plot_kriging_isotropic.py
```

- Use **# %%** to start a **new cell**.
- Write **Python code** as you **normally** would.
- Write **text as RST code**, but each RST line must start with **#**.
- A cell can contain both RST and Python code, but **RST must come first**.
- **RST rules are the same** as in the Theory section.
- A **blank line** interrupts RST code and **switches to Python**. For the remainder of the cell, lines starting with **#** are simply interpreted as Python comments.
- By default, the first image is used as **thumbnail picture**. To use e.g. the third:
sphinx_gallery_thumbnail_number = 3 (ideally right above the plot code)

Adding a new example

- Normally, you should not need to change `python/doc/examples/examples.rst`.
- Every subfolder has a `README.txt` file, which is actually an RST file that only contains a label for hypertext links and the subsection title.
- All source Python scripts are **automatically detected** and used to build HTML examples and the corresponding thumbnails, provided their name follows the `plot_xxx.py` template.
- **No need to index the files**, just put them in the appropriate subfolder.
- You can use the **docstring** at the start of the file not only to write the title of the example, but also the introductory text. This docstring is the only part of the script that is interpreted as RST code without a `#` at the start of every line.
- **Hypertext links** help users connect various aspects of the library:
 - `:class: '~.openturns.ClassName'` to link to the API doc of a class.
 - `:meth: '~.openturns.ClassName.method'` to link to the API doc of a class method.
 - `:ref: '_theory_page'` to link to a Theory doc page using its RST label.
 - `:doc: '/auto_category/subcategory/plot_xxx'` (no `.py`) to link to another example.
- Python scripts should be properly formatted according to the **PEP8**. Use **Black** to format them automatically if needed.
- Examples should **combine several classes** in order to **do something useful**.

Examples section: write interesting English (1/2)

In the Examples section, the English content should be an interesting explanation of the Python content.

Do not write:

```
1 # %%  
2 # Draw the function  
3  
4 n = 10000  
5 sampleX = im.distributionX.getSample(n)  
6 sampleY = im.model(sampleX)  
7 [...]  
8 plotXvsY(sampleX, sampleY)
```

Examples section: write interesting English (2/2)

Write instead:

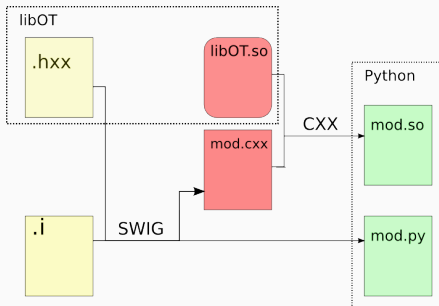
```
1 # %%
2 # In the next cell, we create a large sample of the input random
3 # vector and evaluate the corresponding output from the physical
4 # model. We then create a function which takes the input and output
5 # samples as input argument and creates a grid of scatter plots.
6
7 n = 10000
8 sampleX = im.distributionX.getSample(n)
9 sampleY = im.model(sampleX)
10 [...]
11 plotXvsY(sampleX, sampleY)
12
13 # %%
14 # We see that the variance of the output is mostly sensitive to the
15 # first variable :math:'X_1'.
16 # The variance of the output is not sensitive to the third
17 # variable, since the conditional variance is zero.
```

1. Theory
2. Examples
3. API (user manual)
4. Tips and tricks

SWIG: Simplified Wrapper and Interface Generator

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

SWIG parses the library headers (`.hxx`) and swig interface files (`.i`) to generate the corresponding module source that must be compiled to produce a binary Python module.



Python **docstrings** are written in `_doc.i.in` files which supplement the **SWIG** `.i` files.

These docstrings are then read by **Sphinx** to build the API doc section.

Looking up the sources of docstrings

Let us consider the `LinearCombinationFunction` class and its source file:

```
https://openturns.github.io/openturns/master/user_manual/_generated/  
openturns.LinearCombinationFunction.html  
python/src/LinearCombinationFunction_doc.i.in
```

- Docstring sources are in `src` rather than `doc` because they are **part of the library**.
- A docstring is **just a string** `"...":` double quotes are thus forbidden inside, use `'...'` instead (not to be confused with `:math:`...``). Do not put any whitespace between the opening `"` and the first docstring character, nor between the last docstring character and the closing `"`.
- Docstrings follow **Numpydoc** style. **RST rules remain the same**.
- Example code lines start with `>>>` and are run during Python tests! If you call `print`, you need to **specify the expected output** or the test will fail.
- Numerical values can differ slightly between platforms. To prevent CI failures, you may replace the last digits with an **ellipsis**: `0.34023...` instead of `0.3402362897`
- Class inheritance extends to docstrings: `LinearCombinationFunction` inherits all its method docstrings from **FunctionImplementation**: look for the docstrings in `python/src/FunctionImplementation_doc.i.in`.
- `FunctionImplementation` has no HTML page, although it has many docstrings. Instead, those docstrings are used to produce the **HTML file** of **Function**. This holds for almost all Interface/Implementation couples.

Docstrings of a new class and/or method: SWIG part

To get the **docstrings** to appear in the **Python library**, you need to act on **python/src**:

- Create a **docstring file** `MyClass_doc.i.in`
- Declare the **class docstring**: `%feature("docstring") OT::MyClass`
- Declare a **method docstring**: `%feature("docstring") OT::MyClass::myMethod`
- Two ways are available to actually create the docstring:
 1. Write the docstring between double-quotes **right under the declaration**.
 2. **Make it a variable** so it can be used in several different places:

```
%define OT_MyClass_(myMethod)_doc
"..."
%enddef
```
- For a new class, **do not forget to declare**:
 - the docstrings in `MyClass.i`: `%include MyClass_doc.i`
 - the class in the appropriate `xxx_module.i`: `%include MyClass.i`
 - the class and its docstrings in the appropriate `xxx_module` section of `CMakeLists.txt`:
`MyClass.i MyClass_doc.i.in`
- For a new Interface/Implementation couple, define the docstrings in `XxxImplementation_doc.i.in` with **%define** to be able to reuse them in the Interface docstrings file `Xxx_doc.i.in`: see `Function_doc.i.in` and `FunctionImplementation_doc.i.in` for example.

Docstrings of a new class: Sphinx part

Sphinx looks at `python/doc/user_manual` for the structure of the API documentation.

For example, `LinearCombinationFunction` is referenced in `python/doc/user_manual/functions.rst`

To get your new class to appear in the **HTML API doc** section:

- Find the **RST file** of the appropriate subsection.
- **Insert the name** of your class where you want it to appear.
- Make sure it uses the correct **template**, otherwise use the `:template:` directive.

Templates are found in the `python/doc/_templates` folder. The main templates are:

- `class.rst_t`
- `classWithPlot.rst_t`

If you want to use `classWithPlot.rst_t`, you need to provide the code to generate the graph at the top of the page. For example, `LinearCombinationFunction` runs the code in `python/doc/pyplots/LinearCombinationFunction.py`.

Note that the plot code can be downloaded from the API doc page itself by clicking on *"Source code"*.

API example: always document the most significant method

In the API documentation, the **most significant method** should have an example within the main section, at the top of the help page. This main example should document the *flagship* of the class, that is, the most significant method(s) of the class.

For example, one of the main features of the **LinearEnumerateFunction** class is its evaluation operator, which is why the first example is:

```
1 >>> import openturns as ot
2 >>> enumerateFunction = ot.LinearEnumerateFunction(2, 0.5)
3 >>> for i in range(10):
4 >>>     print(enumerateFunction(i))
```

Contrast with examples from the **Examples** section, which **combine several classes** in order to do useful Uncertainty Quantification.

Summary of the previous parts

Two tools are involved in the creation of the documentation.

Sphinx:

- Theory files: `python/doc/theory`
- Example files: `python/doc/examples`
- API tables of contents: `python/doc/user_manual`
- API plots: `python/doc/pyplots`
- Math notations are defined in `python/doc/math_notations.sty`

SWIG:

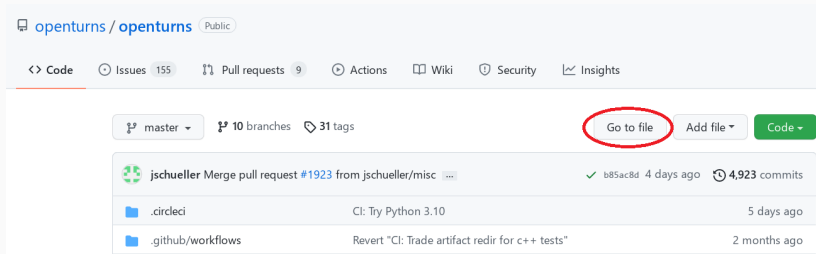
- API docstrings: `*_doc.i.in` files in `python/src`

Docstrings are in `python/src` because they are part of the Python library and not only the HTML documentation.

1. Theory
2. Examples
3. API (user manual)
4. Tips and tricks

Quickly find the appropriate source file

The GitHub “*Go to file*” button does wonders to find the source file you need.



- API: simply type the class name.
- Examples: scroll down to “*Download Python source code*” for the script name.
- Theory: click on “*Show source*”, the exact `.rst` file name appears in the URL.

Write clear commit messages: examples

```
1 Create a quick start example for points and samples
```

```
1 Add new ResourceMap keys for Mesh checking
```

```
2
```

```
3 We add a new RM key for mesh checking. Default value is set to False  
4 as the method might be costly.
```

```
1 Doc: Fix keyword in graph example
```

```
2
```

```
3 Closes #1742
```

```
1 Fixed GaussianNonLinearCalibration
```

```
2
```

```
3 The estimation of distribution of the observations error was  
4 incorrect. It was created using a zero mean to compute the Cholesky  
5 factor of its (known) covariance matrix in order to compute the  
6 residual function, but was not updated once the residual function  
7 was known.
```

```
8
```

```
9 Closes #1529
```

Write clear commit messages: the rules

- First line: a **title** of 50 characters or less.
- Second line: **blank**
- Next lines: **body** of the message with unlimited length.
- If your commit **closes an issue**, use one of the following keywords in the body:
 - Close #....
 - Closes #....
 - Fix #....
 - Fixes #....
 - Solve #....
 - Solves #....

Some text editors like Vim use syntax coloring to help you remember these rules.

Closing keywords are recognized by GitHub: when your commit enters the master branch, the corresponding issue is **automatically closed**.

Remember to also use these **keywords in the first message** of your **pull request**, so that GitHub registers the PR as fixing the issue.

The first message of the PR should **summarize the changes** made by the commits of the PR.

Tip for reviewers: always look at the HTML output, not the source file!

CircleCI `build_linux` builds the doc in addition to the library when a PR is submitted.

For easy access, click on the “*Details*” link of the `build_linux` artifact “check”.

✓ All checks have passed 5 successful checks			Hide all checks
✓	 <code>.github/workflows/build.yml / macos (pull_request)</code>	Successful in 60m	Details
✓	 <code>.github/workflows/build.yml / windows (pull_request)</code>	Successful in 42m	Details
✓	 <code>ci/circleci: build_linux</code> — Your tests passed on CircleCI!		Details
✓	 <code>ci/circleci: build_linux artifact</code> — Link to 0/html/index.html		Details
✓	 <code>ci/circleci: build_mingw</code> — Your tests passed on CircleCI!		Details

- The `build_linux` artifact pseudo-check “succeeds” if, and only if, the `build_linux` check succeeds. This means that a test failure could cause a red cross to appear in front of both of them, even though the doc was actually successfully generated.
- If the person submitting the Pull Request has CircleCI set up on their fork, then the `build_linux` artifact link is always broken regardless of whether or not the doc was successfully built. You can still access it through the “*Artifacts*” tab in CircleCI after clicking on the `build_linux` “*Details*” link.

Build the documentation with the library

Pass the following option to CMake in order to build the doc with the library:

`-DSPHINX_FLAGS="-j6"` (to build with 6 cores)

Pass the following option to CMake in order to not build the doc:

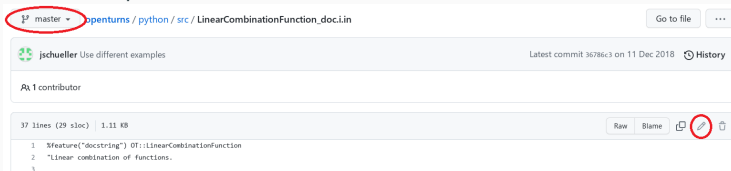
`-DUSE_SPHINX=OFF`

Building the doc takes a while if you build from scratch. It is much shorter afterwards.

Quick patch to the doc (1/2)

You can patch the doc directly from GitHub:

1. Go to the **master** branch of the official repository:
<https://github.com/openturns/openturns>
2. Find the file you want to edit (with “**Go to file**” for example).
3. Click on the pencil button.



4. Make your change and click on “**Preview changes**” to make sure you have not made any mistake.
5. Scroll down to write the title (with **50 characters or fewer**) and the body of your commit message. You **cannot edit more than one file** in a GitHub commit.
6. Click on “**Propose changes**” to:
 - Create a new patch-1 (or more) branch on your fork.
 - Validate the commit.

Quick patch to the doc (2/2)

7. GitHub will ask if you want to create a **Pull Request**. If you accept, you are done!
8. If you want to **add more commits** to your newly created branch, close the tab.
9. Go to **your fork** and select the **patch-1** (for example) branch from the menu.
10. **Edit** a file as before and create a **new commit**.
11. After you have created all the commits you want, create a **Pull Request**.

Advantage of this method:

- Do everything **remotely**. No need to worry about local/remote branches.

Drawbacks of this method:

- Only **one file** can be changed **per commit**.
- **No way to inspect** the HTML output before pushing.

Build select parts of the documentation without the library with `docfast.py`

The Python script `utils/docfast.py` can build selected parts of the library and only requires OpenTURNS to be installed, not necessarily compiled.

Prerequisites: python 3.8+, sphinx, sphinx-gallery, numpydoc, openturns, matplotlib.

```
$ docfast.py build-sphinxonly smirnov_test.rst plot_smoothing_mixture.py
```

- `build-sphinxonly` is the name of the **build folder**: HTML output files will be stored in `build-sphinxonly/install`
- `smirnov_test.rst` is the source file of the **Kolmogorov-Smirnov two samples test** theory page.
- `plot_smoothing_mixture.py` is the source file of the **Bandwidth sensitivity in kernel smoothing** example.

Advantages of this method:

- **Fast**: the command above only builds one example and one theory page.
- Does **not** require a **full compilation** environment.

Drawbacks of this method:

- If the installed version of OpenTURNS is older than master, examples relying on **new classes or methods will fail**.
- Since it does not use SWIG, **how can it deal with docstrings?**

Docstring handling with docfast.py

- Recall: docstring source files are stored in python/`src` as `_doc.i.in` files.
- Sphinx does not deal with them: that is part of SWIG's job.
- In a normal build, SWIG creates the Python library with the docstrings.
- Sphinx reads the docstrings in the Python library, not its source files.

Since Sphinx reads the docstrings of the installed OpenTURNS Python library, docstring.py attempts to patch them with the changes made to the python/`src`/`*_doc.i.in` source files.

By default, it relies on `git diff`: this only takes unstaged changes into account. You can pass an additional argument to `git diff` with the `--diff` option:

```
$ docfast.py --diff master ... uses git diff master instead.
```

Regardless, to tell docfast.py which API page(s) it needs to build, you need to pass the name of a file from python/`doc`/`user_manual`. They are RST table of content files.

```
$ docfast.py [--diff master] functions.rst plot_smoothing_mixture.py
```

Unfortunately, docstring patches can fail in various ways.

To disable them and build from the installed docstrings, use the `--nodiff` option:

```
$ docfast.py --nodiff functions.rst plot_smoothing_mixture.py
```