OpenTURNS modules development

Trainer : Sofiane Haddad
Airbus
sofiane.s.haddad@airbus.com

Developers training

OpenTURNS modules development

OpenTURNS modules

Module development - C++

Module development - Pure python module

OpenTURNS modules

OpenTURNS modules: objectives

OpenTURNS is a growing system developed by a small team. A constant problem is to assess the stability of the whole product, and in order to achieve this goal we introduced a notion of module, in order to insulate a core library dedicated to the definition of the abstract data model and to propose all the specific algorithms as optional modules. The core would evolve quite slowly, insuring its robustness, whereas the modules would have a more dynamic development model.
Another key objective is to provide a way to extend the existing platform with functionalities developed by teams that are reluctant to adopt the OpenTURNS development process. Within the module, the development team can adopt any coding rule or programming language he want, as long as the OpenTURNS interface is respected as well as the objects lifecycle.

Current modules - a non exhaustive list

Modules connected to the library and maintained by the consortium :

- otagrum : create a distribution from a Bayesian Network using aGrUM
- otfftw : Fast Fourier Transform algorithm (e.g. for stochastic processes) using FFTW
- otfmi : FMI models manipulation using PyFMI
- otmixmod : build mixtures of a multivariate Normal distribution from a sample
- otmorris : Morris screening method module
- otpmml : manages PMML files for meta-modeling exchanges
- otpod: A module to build Probability of Detection for Non Destructive Testing
- otrobopt: robust optimization
- otsubsetinverse: inverse subset simulation
- otsvm : Support Vector regression and classification with libsvm
- otwrapy : Python wrapper tools
- otsklearn : use OT surrogate models with the scikit-learn estimator API

Oldest modules:

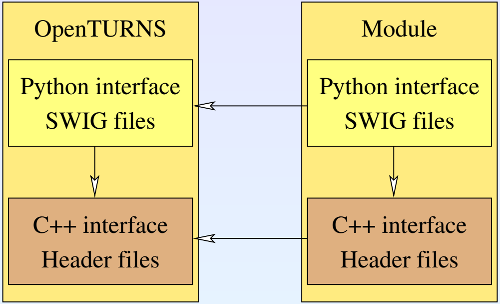
- otlm Linear model with stepwise strategies
- otlhs Optimal LHS (Monte-Carlo & Annealing)
- ...

Current modules - a non exhaustive list

Other modules connected to the library :

- otbenchmark : benchmark problems for reliability and sensitivity analysis
- othdrplot: high density region algorithm for functional outlier detection
- otsurrogate: surrogate models
- otusecases: use cases suitable for OpenTURNS (functions and datasets)
- otmarkov: simulates Markov chains (experimental)
- otsensitivity : sensitivity analysis with density based measures
- shapley-effects : compute Shapley effects
- batman : Statistical analysis for expensive computer codes made easy.
- otpod : Probability of Detection module.
- otgu : Gu method to make more robust the estimation kriging models.
- others : need to reactivate some oldest modules (`otgmm`, `pygosa`, `otfitting`, `othsic`) and others

OpenTURNS modules - C++



The principles
An OpenTURNS module is typically made of two parts:

- a C++ part that uses the OpenTURNS C++ interface in order to provide new specialization or to produce instances of the data model using new algorithms with no OpenTURNS counterpart.
- a Python part, the Python interface of the C++ part, often obtained using SWIG. In this case, this SWIG interface must use the OpenTURNS interface the same way its C++ interface uses the OpenTURNS C++ interface.

Module development

Module development

Step 1: copy and adapt an existing template

- Copy and rename the source tree of an example module (for example the Strange module) from the OpenTURNS source tree. The examples modules are located under the module subdirectory of OpenTURNS source tree:
  `git clone https://github.com/openturns/ottemplate.git MyModule template`
- Set the url of the remote (meaning you already created a repository):
  `git remote set-url origin https://github.com/USERNAME/REPOSITORY.git`
- Adapt the template to your module:
  `./customize.sh MyModule MyModuleClass`
  This command change the module name into all the scripts, and adapt the example class to this new name.

Step 2: develop the module

- Implement your module. You are free to use the rules you want, but if the final objective of the module is to be integrated in the official release of OpenTURNS, it is wise to adopt the OpenTURNS development process and rules.
- Build your module as usual:
```
mkdir build
cd build
cmake .. -DCMAKE_INSTALL_PREFIX=INSTALLDIR
-DOpenTURNS_DIR=OPENTURNS_INSTALLDIR/lib/cmake/openturns
make
```
- Create a source package of your module:
`make package_source`
It will create a tarball named mymodule-X.Y.Z.tar.gz (and mymodule-X.Y.Z.tar.bz2), where X.Y.Z is the version number of the module.

Step 3: documentation
Module documentation is very close to the API one:

- Developer guide : Architecture, validation
- SWIG documentation is to be completed (docstrings);
- Examples and API documentation;
- Theory (if needed)

Step 4: install and test the module

- Check that you have a working OpenTURNS installation, for example by trying to load the OpenTURNS module within an interactive python session:

  ```
  $python
  >>> import openturns as ot
  ```

  and python should not complain about a non existing openturns module.
- Since first step is ok, test your module within python:

  ```
  $python
  >>> import openturns as ot
  >>> import mymodule
  ```

  and python should not complain about a non existing mymodule module.
- You can now use your module as any python module.

Module development

Step 5: Maintenance
Each developer is responsible of his package should maintain it!

- We can rely on continuous integration tool accounting API changes
- He is in charge of bugs (except if underlying bug is in the API)

Exception : modules maintained by the consortium!

Step 6: Packaging
No specific rule concerning the packaging!

- Tarball generation
- Git
- conda...

Exception : modules maintained by the consortium!

Module development

And for a Python module?

Python eco-system:

- numpy/scipy : scientific usage
- scikit-learn : Machine-Learning algorithms
- pydoe, pydoe2 : Design of experiment
- chaospy : Polynomial chaos expansion
- pandas : Data analysis
- SALib : Sensitivity analysis library
- pymc3 : Fit bayesian models
- GPFlow, GPyTorch, TensorFlow, PyTorch, georges...
- statsmodels : Statistical tools
- $\cdots$ many many

Can my module be used by openturns algorithms?

For now, there is (almost) no way to use a Python object within the `OpenTURNS` C++ library, excepting:

- A Distribution;
- a Function;
- An experiment
- A RandomVector
- A FieldFunction
- ...

and use these one as "OpenTURNS classes".
The concept of "OpenTURNS Python module" is not very specific: we get some 'full python' packages such as `oticp`, `otwrap` , `otsklearn`. But these examples are in a way independent as they rely on the "OpenTURNS API" and the API is not able to 'integrate' them. For example :

- Optimization toolbox even if it relies on `openturns` objects;
- Surrogate models - Could not inherit from `MetaModelAlgorithm`
- kernels - Could not inherit from `CovarianceModel`
- ...

No existing (official) template

In comparison with C++, there is no template:

- Such a module might be some development using `openturns` classes or not;
- Such a development relies on external modules (`scikit-learn, rpy2` for example)
- Such a development makes others benefit from `openturns` in a friendly framework (`scikit-learn` for example)
- ...

Why ?

- Such a development relies on external modules (`scikit-learn, rpy2` for example)
- Such a development makes others benefit from `openturns` in a friendly framework (`scikit-learn` for example)

Non-official templates:

- Template done by Antoine Dumas
- new template

How to proceed?

Identify the needs

- Why do we need an Openturns interface?
- Do I need to link to an external module? If yes what kind of data or objects is required?
- What does my module require and provide?
- What kind of interaction with the API?

Step 1: copy and adapt an existing template (even if non official) or start from crash

- Copy and rename the source tree of an example module (for example the Strange module) from the OpenTURNS source tree. The examples modules are located under the module subdirectory of OpenTURNS source tree:
  `git clone https://github.com/sofianehaddad/ottemplatepython.git MyModule`
- Set the url of the remote (meaning you already created a repository):
  `git remote set-url origin https://github.com/USERNAME/REPOSITORY.git`
- Adapt the template to your module:
  `no customize.sh`
  Rename folder to the new module, update tests and doc.

Module development

Step 2: develop the module

- Implement your module. You are free to use the rules you want!
- Build & install your module as usual:

```
python setup.py build
python setup.py install
```

Module development

Step 3: documentation
Module documentation is very "easy".

- Document classes and method using python peps
- Developer guide : Architecture, validation
- Examples and API documentation (potentially notebooks)
- Theory (if needed)
- Build the doc: `python setup.py build_sphinx`

Module development

Step 4: install and test the module

- Since first step is ok, test your module within python:

  ```
  $python
  >>> import openturns as ot
  >>> import mymodule
  ```

  and python should not complain about a non existing mymodule module.
- You can now use your module as any python module.
- For the setup, use `setuptools, distutils`
- For the tests, rely on `openturns test`, `pytest`

Module development

Step 5: Maintenance
Each developer is responsible of his package should maintain it!

- We can rely on continuous integration tool accounting API changes
- He is in charge of bugs (except if underlying bug is in the API)

Step 6: Packaging
No specific rule concerning the packaging!

- Tarball generation
- Git
- conda/pip ...
- `python setup.py sdist --formats=gztar,zip`

Exception : modules maintained by the consortium!

Module development: conda deployement

User should generates a recipe in staged-recipes.

- Rely on existing template
- Fork the repository
- Have a look at the documentation
- Adapt to the module and propose pull-request

In case of pure python it is more simple! Have a look at tensap for example.
Once it is done, module is integrated to `conda-forge`