Phimeca Engineering
Centre d'Affaires du Zénith
34 rue de Sarliève
63800 Cournon d'Auvergne
Tél. : +33 (0) 4 73 28 93 66 contact@phimeca.com
Fax : +33 (0) 4 73 28 95 78 www.phimeca.com

`otfmi`: simulate FMUs from `OpenTURNS`

Project documentation

Technical report

Sans remarque de leur part dans les huit jours suivant sa diffusion, ce document sera réputé « approuvé» par l'ensemble des destinataires.

Client :	EDF R&D, département MRI et STEP
Adresse :	6 quai Watier, 78401, Chatou
Affaire :	PMFRE-00997

Titre :	`otfmi`: simulate FMUs from `OpenTURNS`
Référence :	RT-PMFRE-00997-002A
Accessibilité :	Restricted
Mots clés :	`OpenTURNS`, Modelica, Python, functional mock-up interface (FMI), functional mock-up unit (FMU), `PyFMI`, `FMI lib`, `FMI++`.

Pré-diffusion

Diffusion

Michaël Baudin	EDF R&D MRI
Audrey Jardin	EDF R&D STEP
Anne-Laure Poppelin	EDF R&D MRI

Production

Rédacteur	Vérificateur	Approbateur
Sylvain Girard [illegible]	Julien Schüller [illegible]	Antoine Dumas [illegible]

Suivi des indices

Indice	Date	Évolutions
α	14/02/2017	Preliminary version
A	21/11/2017	Minor edits

Abstract

The functional mock-up interface (FMI) standard specifies a format for multipurpose, easy to build and reusable data interfaces to numerical models. A *functional mock-up unit* (FMU) is a black box defined by the FMI standard, akin to the *wrappers* familiar to the `OpenTURNS`' community.

The purpose of the `otfmi` Python module is to promote the use of the probabilistic approach with system models, in particular those written in Modelica, by enabling easy manipulation of FMUs with `OpenTURNS`. The `otfmi` module relies on `PyFMI`, a module for manipulating FMUs within Python.

This report describes the architecture of `otfmi` and the choices made for its development. It also provides installation instructions and validation examples.

For usage instructions, please refer to the user documentation (Girard, 2017).

Table of contents

1 Introduction

Most of the mathematical methods in OpenTURNS apply to numerical models considered as black boxes. Put another way, OpenTURNS interacts with a model through an *input and output data interface*, while the actual simulation of the model is left to a third-party software. Piloting the simulator is generally a matter of only a few lines of code. The core of the setup effort is building the data interface, also knowns as the *wrapper* in the OpenTURNS' jargon.

The *functional mock-up interface* (FMI) standard specifies a format for multipurpose, easy to build and reusable data interfaces (*Functional mock-up interface for model exchange and co-simulation* 2014; Modelica association, 2016[a]). A *functional mock-up unit* (FMU) is a black box defined by the FMI standard, actually a zip archive containing an XML file describing the variables of the model, and a set of possibly compiled C functions required for the simulation itself. An FMU can be quasi autonomous if a resolution algorithm is included among those functions. It is then fit for "co-simulation" usage. It may also rely on a third-party solver, to be provided at run time. In this case, only the "model exchange" framework of the FMI standard is available.

The purpose of the `otfmi` Python module is to promote the use of the probabilistic approach with system models, in particular those written in Modelica, by enabling easy manipulation of FMUs with OpenTURNS. A list of tools capable of compiling models into FMUs, or of piloting them is maintained by the Modelica association (Modelica association, 2016[b]). The `otfmi` module relies on one of them, `PyFMI` (Modelon, 2017[b]), which allows to manipulate FMUs within Python. It itself draws on the low level `FMI lib` library (Modelon, 2017[a]).

The core features of otfmi are:

1. loading an FMU in an object being integral part of OpenTURNS;
2. selecting inputs and outputs variables;
3. setting some initial values, possibly using a text file interface, to ease initialisation;
4. simulating the model for a single set of input values or a sample, possibly in parallel;
5. retrieving and store the simulation results.

Installation instructions are provided in section 2 The architecture of the module and development choices are described in section 3. Finally, the functioning of the module is illustrated by two examples, presented in section 4.

For more in-depth usage instructions, please refer to the user documentation (Girard, 2017).

2 Installation instructions

The sources directory of the `otfmi` module contains a minimal documentation (`README.rst`), an installation script (`setup.py`), the programs of the module itself and example scripts in the `otfmi` folder, and some unit tests in the **`test`** folder. The other files are used to configure installation but should not require particular attention in normal usage.

Refer to the generic `OpenTURNS` documentation for details about its installation procedure. Using Anaconda, it is carried out with the command

```
conda install -c conda-forge openturns
```

Installation of `PyFMI` The `otfmi` module relies on `PyFMI` (Modelon, 2017[b]). On Windows, `PyFMI` can be installed using an installer program. Otherwise, relying on Anaconda (*Anaconda, Python distribution* 2017) is recommended. Indeed, compiling `PyFMI` from sources and sorting out its dependencies can be troublesome. With Python installed from Anaconda, `PyFMI` is installed by the following command:

```
conda install -c https://conda.binstar.org/chria pyfmi
```

Usage of `otfmi` without installation The module can be imported in a Python session as soon as the `otfmi` folder is listed in the `PYTHONPATH` environment variable. If you installed Python using Anaconda, the `otfmi` folder can be placed in any directory "monitored" by Anaconda, for instance `C:\Users\username\Anaconda\lib\site-packages` on Windows.

Installation of `otfmi` The `otfmi` module can be installed from sources using the classical distutils procedure. From the main folder, run the following command:

```
python setup.py install
```

On Windows, it is possible to avoid resorting to command line interface by using the installer program `otfmi-X.X.win32.exe` (`X.X` is the version number).

Removal of `otfmi` Removing `otfmi` if you installed it from the command line is just a matter of removing all created files. Usually, it is a single file, `otfmi-X.X-py2.7.egg` (`X.X` is the version number) located in the default directory for external module. The path to this directory depends on your Python installation. On Windows with Anaconda it is

C:\Users\username\Anaconda\lib\site-packages. The list of all created files can be retrieved by the following command

```
setup.py install --record list_file.txt
```

If you used the installer program on Windows, the standard removal procedure through the “Control panel” applies.

3 Architecture guide

An overview of the module architecture is given in section 3.1. The subsequent sections describe the main development choices and the way some difficulties were overcome.

3.1 Architecture overview

From the user perspective, the otfmi module provides two classes for loading and simulating FMUs:

- the *high level* class FMUFunction inherits from OpenTURNS' NumericalMathFunction, and is based on OpenTURNS' PythonFunction;
- the *low level* class OpenTURNSFMUFunction inherits from OpenTURNS' OpenTURNSPythonFunction.

They both give access to all the core features listed in the introduction. The low level class OpenTURNSFMUFunction provides a direct access to PyFMI's model object[1], and is more flexible. In particular, it allows to use input trajectories, and its parameters can be altered after instantiation. The parameters of the high level class FMUFunction cannot be altered after instantiation, and the only interface it maintains with the underlying PyFMI object is through optional keyword arguments to be passed to the __call__ method. However, being a daughter class of NumericalMathFunction, it is better integrated into the OpenTURNS ecosystem. For instance, it can be used to define an OpenTURNS' RandomVector.

From the developer perspective, FMUFunction is a NumericalMathFunction factory, that first instantiate an OpenTURNSFMUFunction and pass it to the NumericalMathFunction constructor. This design is directly derived from OpenTURNS' pair of classes PythonFunction and OpenTURNSPythonFunction.

The low level components required by otfmi's core features are all provided as methods of PyFMI's model object. Hence, the purpose of the module is mostly to accommodate

[1]By “PyFMI's model object”, we denote any of the pyfmi.fmi.FMUModelXXX classes, “XXX” being one of “ME1”, “CS1”, “ME2” and “CS2” depending on the FMI version (1 or 2), and usage mode, either model exchange (ME) or co-simulation (CS).

PyFMI's interface by setting default parameters and gathering methods into functions and methods that fit in the OpenTURNS's ecosystem. The otfmi.fmi sub-module is an additional intermediate abstraction layer between PyFMI and the two classes mentioned above.

Consider as an illustration the .simulate method of PyFMI's model object. This is in a sense the main function of PyFMI which performs the simulations and retrieve results. There is a simulate function in otfmi.fmi. It takes a model object as first argument and ends by calling its .simulate method. Beforehand, it resets the model and applies optional initialisation values. The low level class OpenTURNSFMUFunction also has a .simulate method. It parses and formats user provided inputs, and calls otfmi.fmi.simulate with the .model attribute of the class instance as first arguments. The .__call__ method of OpenTURNSFMUFunction relies on its .simulate method, as well as on its sample counterpart, .simulate_sample, for multiple simulations. Finally, the high level class FMUFunction relies on the .simulate and .simulate_sample methods of its underlying OpenTURNSFMUFunction object for its own .__call__ method

The two main classes, FMUFunction and OpenTURNSFMUFunction, are gathered in the sub-module otfmi.otfmi. Intermediate functions wrapping functions and methods from PyFMI are gathered in the sub-module otfmi.fmi. The sub-module otfmi.fmu_pool specifically deals with multiprocessing.

3.2 Input and output dimensions

The inputs and outputs of the black box functions represented by instances of OpenTURNS' NumericalMathFunction class are assumed to be scalar. Indeed, most of the mathematical methods available in OpenTURNS apply to scalar inputs and outputs. On the contrary, all Modelica models, even the static ones, have time series as inputs and outputs.

There is a TimeSeries class in OpenTURNS, but it does not fit very well with Modelica because it uses a fixed regular time-step. Its base class, FieldImplementation, is a generic template for the representation of higher dimension objects.

For now, it was decided to restrict the scope of otfmi to scalar inputs for the high level class FMUFunction . The low level class OpenTURNSFMUFunction supports input trajectories if the expect_trajectory Boolean input argument is set to True upon instantiation.

The outputs are assumed to be scalar for both classes. More precisely, the retrieved outputs are the final values of the simulations. A first step toward support of vector outputs would be to allow the user to provide a Python function for collapsing the simulation output time series into a set of scalars. Common application of this features would be time averaging, minimum or maximum, or projection on a low dimensional basis. This has been sketched in the current implementation in order to ease subsequent developments.

3.3 Multiprocessing

Straightforward application of simple high level multiprocessing setups with PyFMI fails haphazardly yielding somewhat cryptic trace-backs. The default method for handling simulation results in PyFMI uses temporary files and does not play well with multiprocessing. A simple workaround is to handle simulation results in memory. This is achieved simply by passing `options=dict(result_handling="memory")` to the `.simulate` method of PyFMI model objects.

The high level functions of the standard `multiprocessing` (*multiprocessing Python module* 2017) module rely on serialisation with the `pickle` module (*pickle Python module* 2017). This is the second hurdle in the way to easy multiprocessing: class instances and methods (such as a model object and its `.simulate` method) cannot be pickled (*What can and cannot be pickled?* 2017). The `pathos` module is a non-standard, more versatile, equivalent to `multiprocessing` relying on `dill`, an extension of `pickle`, which *can* serialise class instances and methods.

It was decided however to stay with the standard `multiprocessing` module to avoid a non trivial dependency and facilitate installation of `otfmi`. This required to draw on the lower level functions to sidestep the pickling issue. It resulted into the `otfmi.fmu_pool` sub-module which is heavily inspired upon the homonymous module from Marco Bonvini's *EstimationPy Python module* (2016).

3.4 Solver selection

The co-simulation mode may be used to provide the solver along with the model. In `otfmi`, it is assumed that a user providing a solver expects it to be used instead of those provided by PyFMI. Therefore, the default mode for loading FMUs is co-simulation, contrary to PyFMI that uses model exchange when both modes are available. If co-simulation is not available, the FMU is loaded in the model exchange mode. It is possible to force using one or the other by passing a `kind` argument to the loading function, or at instantiation of either of the two classes.

4 Usage examples

The `otfmi` module was applied to two test cases : a toy model representing the deviation of a cantilever beam, and a more realistic thermal-hydraulic model of the secondary loop of a pressurised nuclear power plant. The cantilever beam test case, presented in section 4.1, was used to ascertain the exactitude of the results derived from FMU simulations by comparing them to a pure Python reference. The nuclear power plant test case, presented in section 4.2, demonstrate the applicability of `otfmi` to real scale models, with particular attention devoted to initialisation.

4.1 Deviation of a cantilever beam

In this academic use case taken from the OpenTURNS documentation (Dutfoy et al., 2009; OpenTURNS consortium, 2016), a load is applied to a cantilever beam. The load (F), beam Young's modulus (E), length (L) and section modulus (I) are uncertain. The objective is to estimate the probability that the deviation exceeds a given threshold. The quantile estimation is performed by straightforward Monte Carlo sampling.

The analytical formula for the deviation has been programmed in Modelica as follows (annotations where removed for better legibility).

```
model deviationpoutre
  "Model from here: http://doc.openturns.org/openturns-latest/html/
   ExamplesGuide/cid1.xhtml#cid1"
  output Real y;
  input Real E (start=3.0e7);
  input Real F (start=3.0e4);
  input Real L (start=250);
  input Real I (start=400);
equation
  y=(F*L*L*L)/(3.0*E*I);
end deviationpoutre;
```

This test case is available on Linux 64 bits (FMU for model exchange and co-simulation compiled with OpenModelica) and Windows32 bits (FMU for co-simulation compiled with Dymola).

Probabilistic model The four uncertain inputs of the deviation model were modelled by a Gaussian copula with Gaussian marginals. The objective is to estimate the probability of exceedance of given threshold on a single scalar output, the vertical displacement, or deviation, y.

Running the test There is a demonstration script in otfmi (otfmi.example.deviation) comparing the estimation with pure Python and using an FMU based on the Modelica model shown above. In can be run interactively with the following commands

```
from otfmi.example import deviation
deviation.run_demo()
```

or by executing the script located in the sources directory : otfmi/example/deviation.py.

The two estimations are equal up to machine precision.

With the random seed and target coefficient of variation set by default in the example, the Monte Carlo estimation stops after a total of 4400 simulations.

The computation time, on the laptop used for developing `otfmi`, is in the tenths of seconds for the pure Python reference, about 5 s with the OpenModelica FMU on Linux and up to 10 min with the Dymola FMU on Windows.

It must be noted that about 14 % of the total computation time is devoted to resetting the model, and another 43 % to initialising it. With realistic models, the pre-processing overhead would be much lesser with respect to actual simulation time.

4.2 Thermal balance "partial BIL100"

The thermal power of pressurised nuclear reactors is quantified by several means of differing precision and frequency. The BIL100 is a thermal balance of the secondary loop at full load performed every 30 days with finely calibrated high precision sensors. In the following test case, a physical model of a simplified (hence the adjective "partial") secondary loop is used to estimate the uncertainty of the BIL100 induced by measurement uncertainties. The BIL100 is simulated by computing a stationary state of the secondary loop. The model is therefore purely algebraic. The uncertainty propagation is performed by straightforward Monte Carlo sampling.

The thermal power balance is the difference between:

- the sum of the power produced by the steam generators (the 1300 MW power plants have 4 of them);
- the sum of the power of all primary component, excepted the steam generators and the reactor (it is conventionally fixed to 17 MW for the 1300 MW class power plants).

This model has been used to validate the prototype of intrusive interface between `OpenTURNS` and OpenModelica developed within the ITEA OPENPROD project. Details about the physical modelling can be found in the project report (Jardin and Bouskela, 2012).

This test case is available on Windows 32 bits (FMU for co-simulation compiled with Dymola).

Probabilistic model We consider in this example six measurements with uncertainties modelled by independent Gaussian random variables:

- pressure, flow rate and temperature of the secondary feed water (3 variables);
- flow rate in the dryer–superheater and turbine groups (2 variables);

- pressure at the turbine bypass.

The model output is the thermal power of the nuclear reactor, as described above.

Running the test Two FMUs have been prepared for this example. One FMU has hard-coded guess values for the state variables critical for solving the initialisation problem. It can be run as is, without additional user input. The other FMU has only default values and cannot be initialised without providing relevant start values. Initialisation can be done using .mos scripts, namely text files with lines of the form "`variable_name = value;`".

There are two demonstration scripts in `otfmi` (`bil100_without_initialization_script` and `otfmi.example.bil100_with_initialization_script`). The former has the hard-coded guess values, while the latter relies on an input text file for initialisation. They can be run interactively with the following commands

```
from otfmi.example import (bil100_without_initialization_script,
                           bil100_with_initialization_script)
bil100_without_initialization_script.run_demo()
bil100_with_initialization_script.run_demo()
```

or by executing the scripts located in the sources directory. The `otfmi.example.bil100` script gathers the functions common to the two examples. Running it with its default parameters is equivalent to running `bil100_without_initialization_script`.

A sample of 100 sets of input values is generated and simulated with the FMU. Both FMUs yield identical results: an average thermal power of 3790.66 MW with a standard deviation of 34.91 MW. The total simulation time on the laptop used for developing the module is about 10 s.

References

Dutfoy, Anne et al. (2009). 'OpenTURNS, an Open Source initiative to Treat Uncertainties, Risks 'N Statistics in a structured industrial approach'. In: *41èmes Journées de Statistique, SFdS, Bordeaux*.

Jardin, Audrey and Daniel Bouskela (2012). *Handling of uncertainties on EDF models of energy systems*. Tech. rep. OpenProd R6.20b. EDF.

Functional mock-up interface for model exchange and co-simulation (2014). version 2.0. Modelica association.

OpenTURNS consortium (2016). *Examples Guide for the Text User Interface*. Tech. rep. Airbus, EDF, IMACS, Phimeca. URL: http://doc.openturns.org/openturns%C2%ADlatest/html/ExamplesGuide.

Girard, Sylvain (2017). *otfmi: simulate FMUs from OpenTURNS: User documentation*. Tech. rep. RT-PMFRE-00997-003. Phimeca.

Anaconda, Python distribution (2017). URL: http://continuum.io/downloads (visited on 14/02/2017).

EstimationPy Python module (2016). URL: http://lbl-srg.github.io/EstimationPy/ (visited on 13/12/2016).

Modelica association (2016[a]). *Functional mock-up interface*. URL: https://www.fmi-standard.org/start (visited on 17/11/2016).

– (2016[b]). *Functional mock-up interface*. URL: https://www.fmi-standard.org/tools (visited on 17/11/2016).

Modelon (2017[a]). *FMI library*. URL: http://www.jmodelica.org/FMILibrary (visited on 10/02/2017).

– (2017[b]). *PyFMI Python module*. URL: http://www.jmodelica.org/page/4924 (visited on 10/02/2017).

multiprocessing Python module (2017). URL: https://docs.python.org/2/library/multiprocessing.html/ (visited on 06/02/2017).

pickle Python module (2017). URL: https://docs.python.org/2/library/pickle.html (visited on 06/02/2017).

What can and cannot be pickled? (2017). URL: https://docs.python.org/2/library/pickle.html#what-can-be-pickled-and-unpickled (visited on 06/02/2017).