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 FMU from `OpenTURNS`

User 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 FMU from `OpenTURNS`
Référence :	RT-PMFRE-00997-003A
Accessibilité :	Restreinte
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 	Verificateur ?	Approbateur ?

Suivi des indices

Indice	Date	Évolutions
α	16/02/2017	Version préliminaire
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 and illustrates by examples the main classes and functions of `otfmi`. For installation instructions, validation examples and a description of the module's architecture from the developer point of view, please refer to the project documentation (Girard, 2017a).

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.

Section 2 describes and illustrates by examples the main classes and functions of `otfmi`. A complete working example on a use case is commented in section 3.

Embedded files

The code snippets provided as examples are embedded in text format into this pdf file in order to facilitate copying them. Embedded files are signalled by a symbol in the left margin.

2 User manual

As described in the project documentation (Girard, 2017b), there are three abstraction levels in `otfmi`:

1. the sub-module `otfmi.fmi`;
2. the low level class `OpenTURNSFMUFunction` from the main module `otfmi.otfmi`;
3. the high level class `FMUFunction` from the main module `otfmi.otfmi`.

All examples involving an FMU rely on those provided as example in the `otfmi/example/file/FMU` source folder. Retrieving the appropriate path to those FMUs is achieved with the following commands:

```
import otfmi.example.utility
# Getting the path to the "deviation" example
path_fmu = otfmi.example.utility.get_path_fmu("deviation")
```

This path can be replaced, with appropriate renaming of input and output variable, by any other leading to one of your own FMUs.

2.1 *Sub-module `otfmi.fmi`*

The lower abstraction level of the sub-module `otfmi.fmi` is not specific to `OpenTURNS`. It can be used for adding features missing to the two classes `OpenTURNSFMUFunction` and `FMUFunction`. In many cases however, the latter two should be sufficient.

2.1.1 *Loading an FMU*

Loading an FMU is the most basic feature of `otfmi`. It is directly mapped to the homonymous `PyFMI` function.

fmi_load_fmu.py

```
# Load an fmu
import otfmi.example.utility
path_fmu = otfmi.example.utility.get_path_fmu("deviation")

# Load an FMU using default mode (co-simulation if available)
```

```
model = otfmi.fmi.load_fmu(path_fmu)

# Load an FMU enforcing co-simulation mode
model = otfmi.fmi.load_fmu(path_fmu, kind="CS")

# Additional keyword arguments are passed on to pyfmi
model = otfmi.fmi.load_fmu(path_fmu, log_file_name="deviation.log")
```

2.1.2 Simulating an FMU

The function `otfmi.fmi.simulate` is mapped to the homonymous `PyFMI` function. It offers the additional feature of reading text files to set the initial values of variables.

fmi_simulate_fmu.py

```
# Load an fmu
import otfmi.example.utility
path_fmu = otfmi.example.utility.get_path_fmu("deviation")
model = otfmi.fmi.load_fmu(path_fmu)

# Simulate model with default values
result = otfmi.fmi.simulate(model)
print "y = %g" % result.final("y")
print "L = %g" % result.final("L")

# Use an initialization script
import time
temporary_file = "%s_initialization.mos" % time.strftime("%Y-%m-%d %H:%M:%S",
                                                         time.gmtime())
print temporary_file
with open(temporary_file, "w") as f:
    f.write("L = 300;")
result = otfmi.fmi.simulate(model, initialization_script=temporary_file)
print "y = %g" % result.final("y")
print "L = %g" % result.final("L")

# Additional keyword arguments are passed on to pyfmi
import numpy as np
result = otfmi.fmi.simulate(model, input=("L", np.atleast_2d([0, 200])))
print "y = %g" % result.final("y")
print "L = %g" % result.final("L")
```

The second simulation in the previous example uses an initialisation script to set initial values. It relies on the following lower level functions from `otfmi.fmi`: `apply_initialization_script`

, `parse_initialization_script`, and `parse_initialization_line`.

In the third and last call to `otfmi.fmi.simulate` in the previous example, the **`input`** argument has a somewhat convoluted form. The function `otfmi.fmi.parse_kwargs_simulate` is an intermediate to ease passing input values to the `otfmi.fmi.simulate` function.

fmi_parse_kwargs.py

```
# Load an fmu
import otfmi.example.utility
path_fmu = otfmi.example.utility.get_path_fmu("deviation")
model = otfmi.fmi.load_fmu(path_fmu)

# Format inputs automatically
kwargs_simulate = otfmi.fmi.parse_kwargs_simulate(
    value_input=[200], name_input=["L"], dimension_input=1, name_output=["y"])

result = otfmi.fmi.simulate(model, **kwargs_simulate)
print "y = %g" % result.final("y")

import pyfmi
try:
    print "L = %g" % result.final("L")
except pyfmi.common.io.VariableNotFoundError:
    print 'Only the values of the output variable "y" were stored.'
```

Formatting input arguments relies on the lower level function `otfmi.fmi.guess_time` and `otfmi.fmi.reshape_input`: they guess whether inputs are points or trajectories and adjust their dimensions accordingly.

The `otfmi.fmi.strip_simulation` function simply extracts final values from simulation result objects.

2.1.3 Auxiliary methods

The `otfmi.fmi` also provides a few convenience functions wrapping some common `PyFMI` operations:

- `get_name_variable` gets the list of a model's variable names.
- `get_causality` gets the causality of a model's variables.
- `get_fixed_value` gets the values of a model's variables whose variability is `"fixed"`.
- `set_dict_value` use a dictionary to set the values of some variables of a model.

2.2 Low level class OpenTURNSFMUFunction

The low level class `otfmi.OpenTURNSFMUFunction` requires a path to an FMU and a list of input names corresponding to variables of the FMU. Passing additionally names of output variable allows to streamline the results (the default behaviour is to consider all variables as outputs). Calling the class instance with a vector simulates one point. Calling the class instance with an array simulates a sample, possibly using several cores in parallel.

The following script illustrates standard usage of the class.

lowlevel.py

```
# -*- coding: utf-8 -*-
# Load an fmu
import otfmi.example.utility
path_fmu = otfmi.example.utility.get_path_fmu("deviation")
model = otfmi.fmi.load_fmu(path_fmu)

# Query variable names
print "The FMU has the following variables:"
print otfmi.fmi.get_name_variable(model)

# Instantiate a low level FMU function
function = otfmi.OpenTURNSFMUFunction(
    path_fmu, inputs_fmu=["E", "F", "L", "I"], outputs_fmu="y")


# Simulate points
print "\nSimulate points"
print "y = %g" % function([3.0e7, 30000, 200, 400])[0]
print "y = %g" % function([3.0e7, 30000, 250, 400])[0]
print "y = %g" % function([3.0e7, 30000, 300, 400])[0]

# Simulate sample
print "\nSimulate sample"
list_result = function([[3.0e7, 30000, 200, 400],
                        [3.0e7, 30000, 250, 400],
                        [3.0e7, 30000, 300, 400]])

for result in list_result:
    print "y = %g" % result[0]


# Simulate sample in parallel
print "\nSimulate sample in parallel"
list_result = function([[3.0e7, 30000, 200, 400],
```

```
                          [3.0e7, 30000, 250, 400],
                          [3.0e7, 30000, 300, 400]], n_cpus=2)

for result in list_result:
    print "y = %g" % result[0]

# Incorrect names
import pyfmi.common.io
try:
    function = otfmi.OpenTURNSFMUFunction(
        path_fmu, inputs_fmu=["E", "F", "L", "I", "invalid_name"],
        outputs_fmu="y")
except pyfmi.common.io.VariableNotFoundError as e:
    print ("\nVariable names are checked on instantiation: '%s' is invalid." %
           e.message)

# Multiple outputs
print("\nMultiple outputs")
function = otfmi.OpenTURNSFMUFunction(
    path_fmu, inputs_fmu=["E", "F", "L", "I"], outputs_fmu=["y", "L", "y"])
for name, value in zip(function.getFMUOutputDescription(),
                       function([3.0e7, 30000, 200, 400])):
    print "%s = %g" % (name, value)

# Access to the pyfmi model object
print "Pyfmi's model object: %s" % function.model
```

2.3 High level class `FMUFunction`

The high level class `otfmi.FMUFunction` operates similarly as its low level counterpart. The main difference is that its `__call__` method does not accept additional arguments. Hence, parameters, such as the number of cores to use for parallel computing, should be set on instantiation. Additionally, it does not provide any access to the underlying `PyFMI` model object. On the other hand, it can be used by `OpenTURNS`' high level algorithms, as will be seen in section 3.

The following script illustrates standard usage of the class.

highlevel.py

```
# -*- coding: utf-8 -*-
# Load an fmu
import otfmi.example.utility
path_fmu = otfmi.example.utility.get_path_fmu("deviation")
```

```
model = otfmi.fmi.load_fmu(path_fmu)

# Query variable names
print "The FMU has the following variables:"
print otfmi.fmi.get_name_variable(model)

# Instantiate a high level FMU function
function = otfmi.FMUFunction(
    path_fmu, inputs_fmu=["E", "F", "L", "I"], outputs_fmu="y")


# Simulate points
print "\nSimulate points"
print "y = %g" % function([3.0e7, 30000, 200, 400])[0]
print "y = %g" % function([3.0e7, 30000, 250, 400])[0]
print "y = %g" % function([3.0e7, 30000, 300, 400])[0]

# Simulate sample
print "\nSimulate sample"
list_result = function([[3.0e7, 30000, 200, 400],
                        [3.0e7, 30000, 250, 400],
                        [3.0e7, 30000, 300, 400]])

for result in list_result:
    print "y = %g" % result[0]


# Simulate sample in parallel
print ("\nWith the high level object 'FMUFunction', the number"
       " of cores is selected at instantiation.")
function_parallel = otfmi.FMUFunction(
    path_fmu, inputs_fmu=["E", "F", "L", "I"], outputs_fmu="y", n_cpus=2)


print "\nSimulate sample in parallel"
list_result = function_parallel([[3.0e7, 30000, 200, 400],
                                 [3.0e7, 30000, 250, 400],
                                 [3.0e7, 30000, 300, 400]])

for result in list_result:
    print "y = %g" % result[0]

# Incorrect names
```

```
import pyfmi.common.io
try:
    function = otfmi.FMUFunction(
        path_fmu, inputs_fmu=["E", "F", "L", "I", "invalid_name"],
        outputs_fmu="y")
except pyfmi.common.io.VariableNotFoundError as e:
    print ("\nVariable names are checked on instantiation: '%s' is invalid." %
           e.message)

# Multiple outputs
print("\nMultiple outputs")
function = otfmi.FMUFunction(
    path_fmu, inputs_fmu=["E", "F", "L", "I"], outputs_fmu=["y", "L", "y"])
for name, value in zip(function.getFMUOutputDescription(),
                       function([3.0e7, 30000, 200, 400])):
    print "%s = %g" % (name, value)
```

3 Use case guide: 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).

The Python code for this use case is given in appendix A.

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 example 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`.

Comments on the code The blocks "Identifying the platform" (lines 11 to 17) and "FMU model" (lines 72 to 90) locate the example FMU file corresponding to your platform. The actual loading of the FMU is performed by instantiation of a `otfmi.FMUFunction` on lines 85 to 86.

The code from lines 19 to 45 was directly taken from the `OpenTURNS` documentation (*Deviation of a cantilever beam* 2017). It builds a probabilistic model of the four uncertain inputs using standard `OpenTURNS` functions.

The block "Python model (reference)" (lines 47 to 70) defines a Python function simulating the beam's bending. It instantiates an `openturns.PythonFunction`, a daughter class of the ubiquitous `OpenTURNS` class `openturns.NumericalMathFunction`. The `otfmi.FMUFunction` class is the exact counterpart of `openturns.PythonFunction` for dealing with FMUs, and has a very similar structure.

The `create_monte_carlo` function (lines 94 to 119) defines the event whose probability we want to estimate, namely the exceedance of a deviation threshold. The `coefficient_variation` argument controls the precision of the estimation: lowering it increases precision but calls for more simulations.

The `run_monte_carlo` function (lines 121 to 144) performs the actual estimation by performing simulations until the estimate's coefficient of variation is lower than the target specified with `create_monte_carlo`.

The `run_demo` function performs the probability estimation with both models, FMU based and pure Python, and compares the results.

The `if __name__ == "__main__":` protected block (lines 194 to end) offers the interface for running the example from command line.

Appendix A Code of the use case "deviation of a cantilever beam"

deviation.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Copyright 2016 EDF. This software was developed with the collaboration of
# Phimeca Engineering (Sylvain Girard, girard@phimeca.com).
"""Estimate a threshold exeedance probability with both Python and FMU models.
The physical model represents the deviation of a cantilever beam subjected to
a load. The probability that the deviation exceeds a given threshold is
estimated by straightforward Monte Carlo sampling.
"""

# Identifying the platform
import platform
key_platform = (platform.system(), platform.architecture()[0])
# Call to either 'platform.system' or 'platform.architecture' *after*
# importing pyfmi causes a segfault.
dict_platform = {("Linux", "64bit"):"linux64",
                 ("Windows", "32bit"):"win32"}

# Define the input distribution
import numpy as np
import openturns as ot

E = ot.Beta(0.93, 3.2, 2.8e7, 4.8e7)
F = ot.LogNormal(3.0e4, 9000.0, 15000.0,  ot.LogNormal.MUSIGMA)
L = ot.Uniform(250.0, 260.0)
I = ot.Beta(2.5, 4.0, 310.0, 450.0)

# Create the Spearman correlation matrix of the input random vector
RS = ot.CorrelationMatrix(4)
RS[2,3] = -0.2

# Evaluate the correlation matrix of the Normal copula from RS
R = ot.NormalCopula.GetCorrelationFromSpearmanCorrelation(RS)

# Create the Normal copula parametrized by R
mycopula = ot.NormalCopula(R)

# Create the input probability distribution of dimension 4
inputDistribution = ot.ComposedDistribution([E, F, L, I], mycopula)
```

```python

# Give a description of each component of the input distribution
inputDistribution.setDescription( ("E", "F", "L", "I") )

# Create the input random vector
inputRandomVector = ot.RandomVector(inputDistribution)

# Python model (reference)
def deviationFunction(x):
    """Python version of the physical model.

    Parameters:
    ----------
    x : Vector or array with individuals as rows, input values in the
    following order :
      - beam Young's modulus (E)
      - load (F)
      - length (L)
      - section modulus (I)

    """

    E=x[0]
    F=x[1]
    L=x[2]
    I=x[3]
    y=(F*L*L*L)/(3.*E*I)
    return [y]

model_py = ot.PythonFunction(4, 1, deviationFunction)
model_py.enableHistory()

# FMU model
import otfmi
from pyfmi.fmi import FMUException
import sys

import os


path_here = os.path.dirname(os.path.abspath(__file__))
try:
    directory_platform = dict_platform[key_platform]
```

```
    path_fmu = os.path.join(path_here, "file", "fmu",
                            directory_platform, "deviation.fmu")
    model_fmu = otfmi.FMUFunction(
        path_fmu, inputs_fmu=["E", "F", "L", "I"], outputs_fmu="y")
except (KeyError, FMUException):
    print ("This example is not available on your platform.\n"
           "Execution aborted.")
    sys.exit()

model_fmu.enableHistory()

def create_monte_carlo(model, inputRandomVector, coefficient_variation):
    """Create a Monte Carlo algorithm.

    Parameters:
    ----------
    model : OpenTURNS NumericalMathFunction.

    inputRandomVector : OpenTURNS RandomVector, vector of random inputs.

    coefficient_variation : Float, target for the coefficient of variation of
    the estimator.

    """

    outputVariableOfInterest = ot.RandomVector(model, inputRandomVector)
    # Create an Event from this RandomVector
    threshold = 30
    myEvent = ot.Event(outputVariableOfInterest, ot.Greater(), threshold)
    myEvent.setName("Deviation > %g cm" % threshold)

    # Create a Monte Carlo algorithm
    myAlgoMonteCarlo = ot.MonteCarlo(myEvent)
    myAlgoMonteCarlo.setBlockSize(100)
    myAlgoMonteCarlo.setMaximumCoefficientOfVariation(coefficient_variation)

    return myAlgoMonteCarlo

def run_monte_carlo(model, coefficient_variation=0.20):
    """Run Monte Carlo simulations.

    Parameters:
    ----------
```

```
    model : OpenTURNS NumericalMathFunction.

    coefficient_variation : Float, target for the coefficient of variation of
    the estimator.

    """

    # Setup Monte Carlo algorithm
    myAlgoMonteCarlo = create_monte_carlo(model, inputRandomVector,
                                          coefficient_variation)

    # Perform the simulations
    myAlgoMonteCarlo.run()

    # Get the results
    monte_carlo_result = myAlgoMonteCarlo.getResult()
    probability = monte_carlo_result.getProbabilityEstimate()

    return probability

def run_demo(seed=23091926, coefficient_variation=0.20):
    """Run the demonstration

    Parameters:
    ----------
    seed : Integer, seed of the random number generator. The default is
    23091926.

    coefficient_variation : Float, target for the coefficient of variation of
    the estimator.

    """
    import time

    ot.RandomGenerator.SetSeed(seed)
    time_start = time.time()
    probability_py = run_monte_carlo(
        model_py, coefficient_variation=coefficient_variation)
    elapsed_py = time.time() - time_start

    ot.RandomGenerator.SetSeed(seed)
    time_start = time.time()
    probability_fmu = run_monte_carlo(
```

```
            model_fmu, coefficient_variation=coefficient_variation)
    elapsed_fmu = time.time() - time_start

    title = "Threshold exeedance probability:"
    print "\n%s" % title
    print "-" * len(title)
    justify = 20
    print "Full python: %f".rjust(justify) % probability_py
    print "FMU: %f".rjust(justify) % probability_fmu

    relative_error = (abs(probability_py - probability_fmu) / probability_py)

    from numpy import finfo
    if relative_error < finfo(float).eps:
        print "Relative error is below machine precision."
    else:
        print "Relative error: %e" % relative_error

    title = "Computation time in seconds:"
    print "\n%s" % title
    print "-" * len(title)
    justify = 20
    print "Full python: %f".rjust(justify) % elapsed_py
    print "FMU: %f".rjust(justify) % elapsed_fmu

if __name__ == "__main__":
    import sys
    try:
        coefficient_variation = float(sys.argv[1])
    except IndexError:
        coefficient_variation = 0.20

    run_demo(coefficient_variation=coefficient_variation)
```