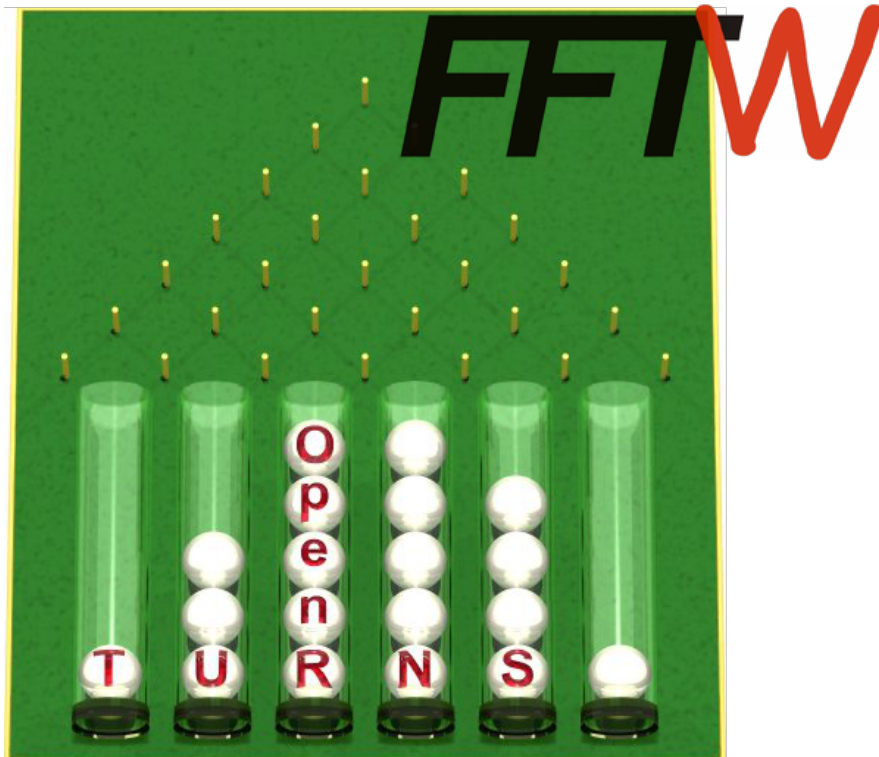# Documentation of the OpenTURNS-FFTW module

Documentation built from package otfftw-0.10

October 14, 2022

Abstract

The purpose of this document is to present the OpenTURNS-FFTW module.
This document is organised according to the Open TURNS documentation :

- a *Reference Guide which gives some theoretical basis,*

- a *Use cases Guide which details scripts in python (the Textual Interface langage of Open TURNS) and helps the User to learn as quickly as possible the manipulation of the otfftw module,*

- the *User Manual which details the otfftw objects and give the list of their methods,*

- the *Examples Guide which provides at the moment only one example performed with the otfftw module.*

Contents

1 Reference Guide

The OpenTURNS-FFTW library provides a bridge between the OpenTURNS library and the FFTW library, one of the most efficient implementation of the Fast Fourier Transform available to date. This library implements both the direct and inverse discrete Fourier transform.

More precisely, given a complex-valued sequence $(z_0, \ldots, z_{n-1})$, its direct discrete Fourier transform $\hat{z} = (\check{z}_0, \ldots, \hat{z}_{n-1})$ reads:

$$\hat{z}_k = \sum_{j=0}^{n-1} z_j e^{-2i\pi \frac{jk}{n}}$$

and its inverse discrete Fourier transform $\check{z} = (\check{z}_0, \ldots, \check{z}_{n-1})$ reads:

$$\check{z}_k = \frac{1}{n} \sum_{j=0}^{n-1} z_j e^{2i\pi \frac{jk}{n}}$$

which gives the relation $z = \hat{\check{z}}$.

It is worth noting that the FFTW library does not include the $\frac{1}{n}$ normalization factor for the inverse transform. The FFTW library provides an $\mathcal{O}(n \log n)$ complexity implementation of such transforms even for prime n, but the best performance is achieved when n is a power of 2.

2 Use Cases Guide

This section presents the main functionalities of the module $otfftw$ in their context.

2.1 Which python modules to import ?

In order to use the functionalities described in this documentation, it is necessary to import :

- the $otfftw$ module which links the *openturns* module.

Python script for this use case :

```
from otfftw import *
```

2.2 Which python modules to import ?

In order to use the functionalities described in this documentation, it is necessary to import :

- the *openturns* python module which gives access to the Open TURNS functionalities,

- the $otfftw$ module which links the *openturns* module.

Python script for this use case :

```
# Load OpenTURNS to manipulate NumericalComplexCollection
from openturns import *
# Load the link between OT and FFTW
from otfftw import *
```

2.3 UC: Using the FFTW algorithm to perform discrete Fourier transforms

With the $otfftw$ module, it is possible to perform both direct and inverse discrete Fourier transforms using the high-performance fftw library. To perform such transforms, the needed data are:

Requirements	• a collection of complex values: *collection* **type:** NumericalComplexCollection • the index of the first element to be transformed: *first* **type:** UnsignedInteger • the size of the sub-sequence of values to be transformed: *size* **type:** UnsignedInteger
Results	• the transformed sequence : *transformedCollection*, **type:** NumericalComplexCollection

Python script for this use case:

```
from openturns import *
from otfftw import *

# Create the data
n = 16
collection = NumericalComplexCollection(n)
for i in range(n):
    collection[i] = (1.0 + i) * (1.0 - 0.2j)
first = 3
size = 8

print "collection=", collection
print "first=", first
print "size=", size

# Create a FFTW algorithm
myFFT = FFTW()

# Direct transform of the whole collection
transformedCollection = myFFT.transform(collection)
print "Direct transform of the whole collection=", transformedCollection

# Direct transform of a sub-sequence
transformedCollection = myFFT.transform(collection, first, size)
print "Direct transform of a sub-sequence=", transformedCollection

# Inverse transform of the whole collection
transformedCollection = myFFT.inverseTransform(collection)
print "Inverse transform of the whole collection=", transformedCollection

# Inverse transform of a sub-sequence
transformedCollection = myFFT.inverseTransform(collection, first, size)
print "Inverse transform of a sub-sequence=", transformedCollection
```

2.4 UC: Using the FFTW algorithm to speed-up spectral process simulation

The fftw library is much more efficient than the FFT library provided by OpenTURNS. Knowing this point, OpenTURNS has been designed such that the TTF implementation can be plugged at run time for the most demanding algorithms. One of these algorithms is the simulation of SpectralProcess processes.

Requirements	• a spectral normal process : *process* **type:** SpectralNormalProcess • a sample size : *size* **type:** UnsignedInteger
Results	• a sample of size *size* of the process : *sample,* **type:** SampleProcess

Python script for this use case:

```
from openturns import *
from otfftw import *
from time import time

# Create a discretized spectral normal process
dim = 1
n = 8
tg = RegularGrid(0.0, 1.0, n)
process = SpectralNormalProcess(CauchyModel(NumericalPoint(dim, 1), NumericalPoint(dim,

# Sample size
size = 3

# FFT algorithm
fft = FFTW()

process.setFFTAlgorithm(fft)

# Sample the process
sample = process.getSample(size)

print "sample=", sample
```

2.5   UC: Using the FFTW algorithm to speed-up spectral model estimation

The same way the FFTW class can be used to speed-up the SpectralNormal class, it can be used to speed-up the WelchFactory class.

Requirements	• a process sample : *sample* **type:** ProcessSample • a Welch factory : *factory* **type:** WelchFactory
Results	• a spectral model : *spectralModel*, **type:** UserDefinedSpectralModel

Python script for this use case:

```python
from openturns import *
from otfftw import *
from time import time

# Create a process sample
dim = 1
n = 8
tg = RegularGrid(0.0, 1.0, n)
process = SpectralNormalProcess(CauchyModel(NumericalPoint(dim, 1), NumericalPoint(dim,

# Sample size
size = 3

# Sample the process
sample = process.getSample(size)

# Welch factory
factory = WelchFactory()

# FFT algorithm
fft = FFTW()

# Use this fft in the spectral process
factory.setFFTAlgorithm(fft)

# Estimate the spectral model
spectralModel = factory.build(sample)

print "spectral_model=", spectralModel
```

3   User Manual

This section gives an exhaustive presentation of the objects and functions provided by the $otfftw$ module, in the alphabetic order.

3.1   FFTW

Usage :

$FFTW()$

Arguments : None

Value : a FFTW instance, it means an algorithm able to perform discrete Fourier transforms using the fftw library.

Some methods :

$transform$

Usage : $transform(collection)$

Usage : $transform(collection, first, size)$

Arguments :

$collection$: a NumericalComplexCollection of length n.

$first$: an integer, the index of the first element of $collection$ to be taken into account. We must have $first < n$.

$size$: an integer, the size of the sub-sequence of $collection$ to be transformed. We must have $first + size \leq n$.

Value : a NumericalComplexCollection, containing the direct discrete Fourier transform of the whole collection for the first usage and of the sub-sequence starting at position $first$ and of length $size$ for the second usage.

$inverseTransform$

Usage : $inverseTransform(collection)$

Usage : $inverseTransform(collection, first, size)$

Arguments :

$collection$: a NumericalComplexCollection of length n.

$first$: an integer, the index of the first element of $collection$ to be taken into account. We must have $first < n$.

$size$: an integer, the size of the sub-sequence of $collection$ to be transformed. We must have $first + size \leq n$.

Value : a NumericalComplexCollection, containing the inverse discrete Fourier transform of the whole collection for the first usage and of the sub-sequence starting at position $first$ and of length $size$ for the second usage.

4 Examples Guide

We present here an example of an uncertainty propagation study based on a representation of the uncertainty thanks to a normal process $X(t)$. We define the input normal process using a spectral model, then we sample this model and propagate it through a linear model $L(x)$. Then, we recover the spectral model of the output process $Y(t)$ based on the output sample. This study is FFT intensive, and all the computations will be done using the FFTW object.

The input process is of dimension 1, and based on a normalized Cauchy spectral model defined by a spectral density $S(f)$ such that:

$$\forall f \in \mathbb{R}, \quad S_X(f) = \frac{2}{1 + (2\pi f)^2}$$

The linear model $L(x)$ is simply a scaling transformation:

$$\forall x \in \mathbb{R}, \quad L(x) = \alpha x$$

The theoretical output process $Y(t)$ is also normal, stationnary and its spectral density is given by:

$$\forall f \in \mathbb{R}, \quad S_Y(f) = \frac{2}{1 + (2\pi f)^2}$$

A graphical comparison between this theoretical spectral density and the reconstructed one is given on figure 1.

The speed-up with respect to the default OpenTURNS FFT algorithm is:

- a sampling time divided by a factor of 1.14

- an estimation time divided by a factor of 2.31

on an Intel QuadCore Q9300 at 2.53GHz based laptop running Linux Mandriva 2010.2

4.1 Python script

```python
from openturns import *
from otfftw import *
from math import *
from time import time

# Create the input process
# first , the time grid
tMin = 0.0
tStep = 0.5
nStep = 2002
timeGrid = RegularGrid(tMin, tStep, nStep)
# second , the process
inputProcess = SpectralNormalProcess(CauchyModel(), timeGrid)

# Create the FFT algorithm
myFFT = FFTW()
inputProcess.setFFTAlgorithm(myFFT)
```
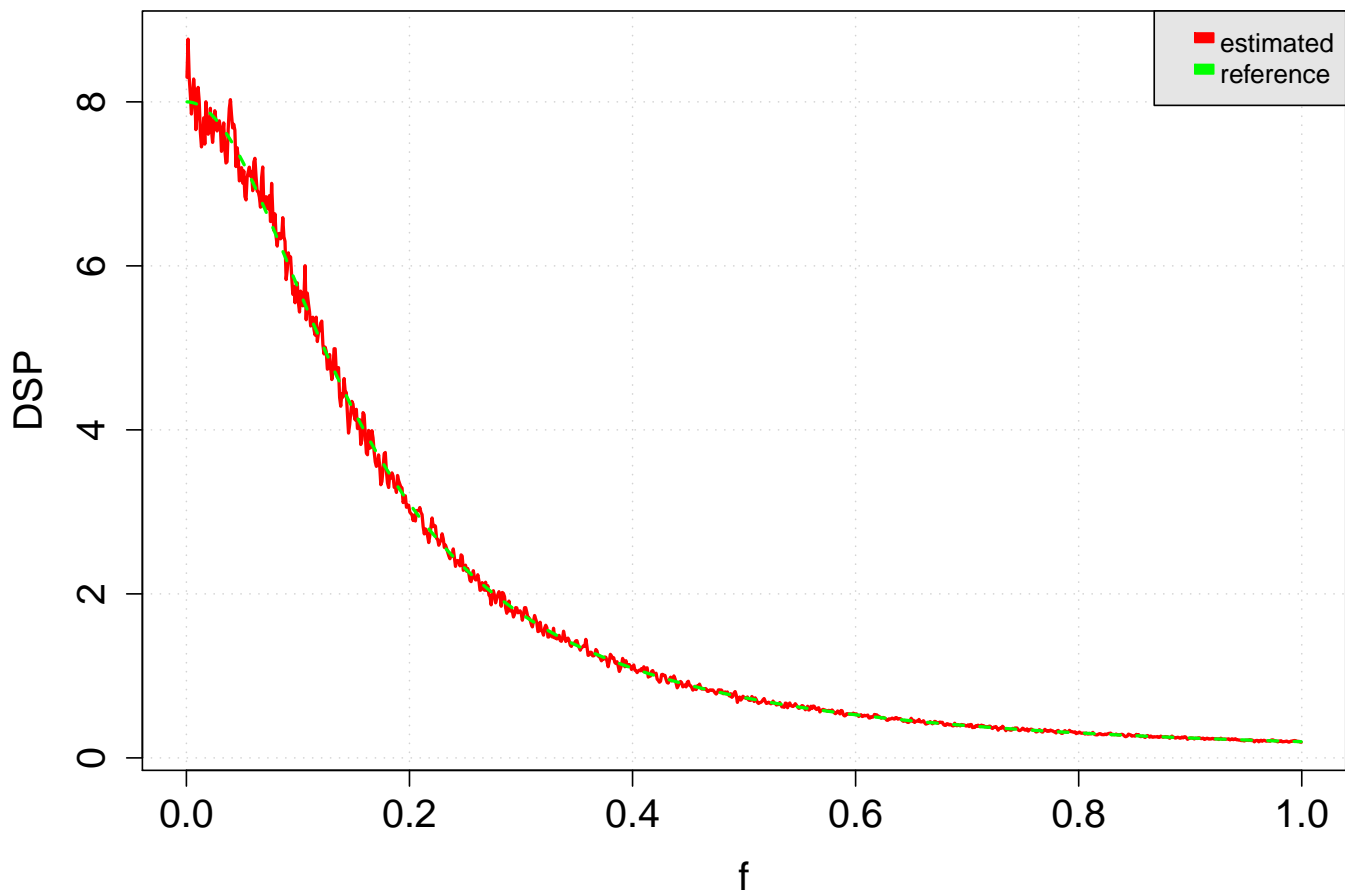
Output DSP comparison

Figure 1: Estimated spectral density (red) and theoretical one (green) of the output process

```
# Create the linear model
alpha = 2.0
model = SpatialFunction(NumericalMathFunction("x", str(alpha) + "*x"))

# Create the output process
outputProcess = CompositeProcess(DynamicalFunction(model), inputProcess)
outputProcess.setTimeGrid(timeGrid)

# Sample the output process
size = 1000
t0 = time()
sample = outputProcess.getSample(size)
print "sampling_time=", time() - t0, "s"

# Build an estimation of the output spectral density
```

```
factory = WelchFactory()
factory.setFFTAlgorithm(myFFT)
t0 = time()
outputSpectralModel = factory.build(sample)
print "estimation_time=", time() − t0, "s"


# Graphical comparison of the output spectral models
referenceOutputSpectralModel = CauchyModel([alpha], [1.0])
frequencyGrid = outputSpectralModel.getFrequencyGrid()
nFrequency = frequencyGrid.getN()
dataEstimated = NumericalSample(nFrequency, 2)
dataReference = NumericalSample(nFrequency, 2)
for i in range(nFrequency):
    f = frequencyGrid.getValue(i)
    dataEstimated[i, 0] = f
    dataReference[i, 0] = f
    dataEstimated[i, 1] = outputSpectralModel.computeSpectralDensity(f)[0, 0].real
    dataReference[i, 1] = referenceOutputSpectralModel.computeSpectralDensity(f)[0, 0].
g = Graph("Output_DSP_comparison", "f", "DSP", True, "topright")
estimated = Curve(dataEstimated)
estimated.setColor("red")
estimated.setLegendName("estimated")
estimated.setLineWidth(2)
g.add(estimated)
reference = Curve(dataReference)
reference.setColor("green")
reference.setLegendName("reference")
reference.setLineStyle("dashed")
reference.setLineWidth(2)
g.add(reference)
Show(g)
g.draw("DSPComparison", 640, 480, GraphImplementation.PDF)
```