Python functions overview

Michaël Baudin

June 11, 2025

1 Python functions - Overview

Coding party OpenTURNS, March 2023

Michaël Baudin

The goal of this session is to get an overview of Python functions. We present the type of function, its purpose and the type of code that the user must provide.

[1]:
```
import openturns as ot
import openturns.viewer as otv
import numpy as np

ot.__version__
```

[1]: '1.19'

1.1 A generic framework

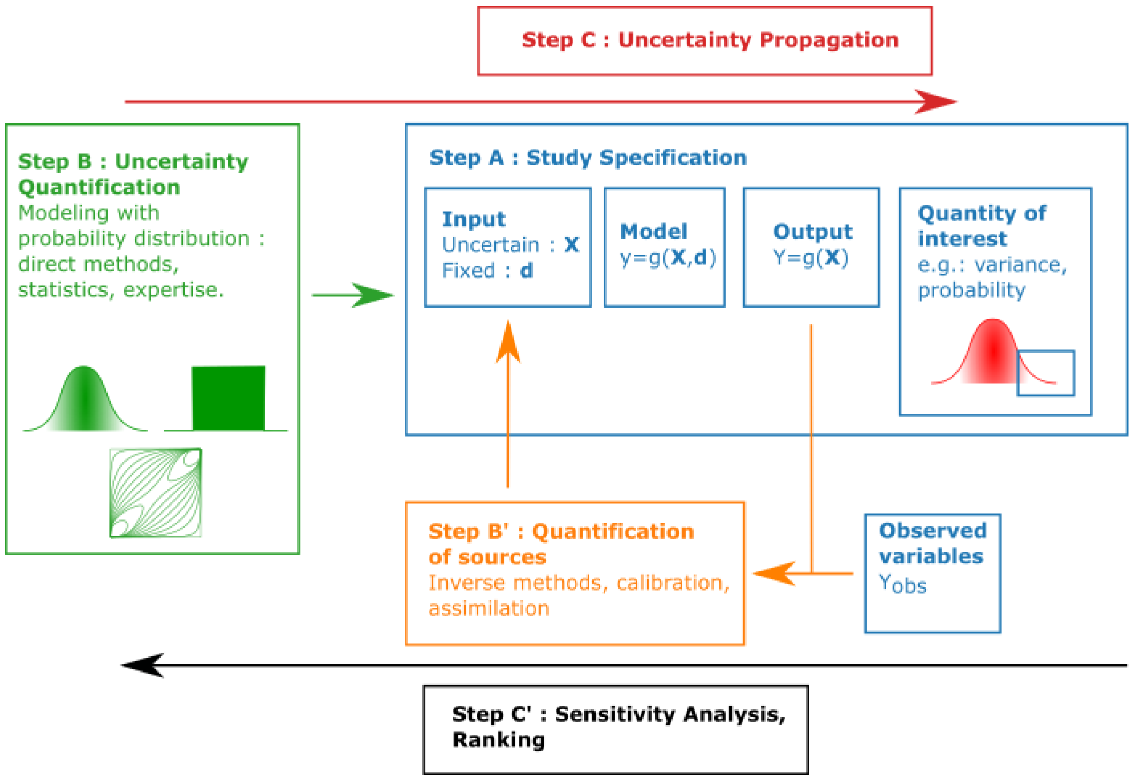


Figure 1. Defining the physical model g is the begining of any uncertainty study.

1.2 Definition of a Function in OpenTURNS

In OpenTURNS, a `Function` is defined from (see Function):

- an evaluation which has an input `Point` and returns an output `Point` (see Evaluation) ;
- the gradient which has an input `Point` and returns an output `Matrix` (see GradientImplementation) ;
- the hessian which has an input `Point` and returns and output `SymmetricTensor` (see HessianImplementation).

1.3 Different types of Python functions

Type	Purpose	Implementation
`PythonFunction`	Point to Point	`function`
`OpenTURNSPythonFunction`	Point to Point	`class`
`PythonFieldFunction`	Field to Field	`function`
`PythonPointToFieldFunction`	Point to Field	`function`
`PythonFieldToPointFunction`	Field to Point	`function`
`OpenTURNSPythonFieldFunction`	Field to Field	`class`
`OpenTURNSPythonPointToFieldFunction`	Point to Field	`class`

Type	Purpose	Implementation
`OpenTURNSPythonFieldToPointFunction`	Field to Field	`class`

Table 1. Different types of Python functions depending on the inputs and outputs and the code the user must provide.

Below is a collection of relevant and interesting help pages.

Link	Type
Defining Python and symbolic functions	`PythonFunction`
Define a function with a field output	`PythonPointToFieldFunction`
Wrapper development	`coupling_tools`

Table 2. Help pages which present the use of Python functions.

1.4 Why may we use a Python function?

There are several reasons to use a Python function.

- Use **existing Python code** implementing a physical model g. This model may use `numpy`, `scipy` or a dedicated API (e.g. `AsterStudy` or `Telemac`).
- Achieve a **great flexibility** in the code, involving several other Python functions or classes.
- Provide the code to **another user**, with the possibility to **maintain the code** easily.
- Get **performance** by parallelizing the evaluation.

1.5 What is the PythonFunction necessary and useful?

Surprise for a Python user:

- Providing a `PythonFunction` class in a Python library may seem surprising.
- Other libraries (e.g. `scipy`) do not require that.

Discussion:

- The `PythonFunction` is **necessary**, because OpenTURNS is a C++ library accessible through SWIG.
- It is **useful**, because OpenTURNS provides many different types of functions so that each algorithm can be as efficient as possible.

Examples:

- The `ParametricFunction` class is useful for calibration algorithms.
- The `DistanceToDomainFunction` class is useful for HSIC indices.
- The `Point(or Field)toPoint(or Field)Function` is class is useful for stochastic processes.

1.6 The simplest possible example

[2]:
```
def mySimulator(x):
    y0 = x[0] + x[1] + x[2]
    y1 = x[0] - x[1] * x[2]
    y = [y0, y1]  # Will be converted to a Point by SWIG
    return y


gFunction = ot.PythonFunction(3, 2, mySimulator)
x = [1.0] * 3  # Will be converted to a Point by SWIG
print("x = ", x)
y = gFunction(x)
print("y = ", y)
```

```
x =  [1.0, 1.0, 1.0]
y =  [3,0]
```

1.7 Other ways to implement a function

We may also consider other ways to implement a function.

Class	Advantages	Drawbacks
`PythonFunction`	Flexible	Not always the fastest
`SymbolicFunction`	Exact gradient (not always), fast (not always)	Limited features

Table 3. Comparison of Python and symbolic functions.

Tool	Features	Pros/Cons	Link
Otwrapy	Multithread, distributed evaluation	Can be fast, depending on the situation	doc
Autograd	Automatic differentiation	Not always possible	doc
Jax	Automatic differentiation	Not always possible	doc
`coupling_tools`	Connect to an external program using files	Slow (depends on the speed of the disk)	doc

Table 4. Tools to consider when using a Python function.

More details on this topic :

- See `Coupling Tools.ipynb` in this repository for more details on `coupling_tools`.
- More details on `Otwrapy` and `Jax` LaTeXlater in the slides.

1.8 Benchmark

We consider the function $g : \mathbb{R}^3 \to \mathbb{R}^2$:

$$y_0 = x_1 + x_2 + x_3$$
$$y_1 = x_1 - x_2 x_3$$

for any $\boldsymbol{x} \in \mathbb{R}^3$.

- The input distribution has $\mathcal{N}(0, 1)$ independent marginals : generating the input sample is fast.
- We call the `getSample()` method of the output random vector.
- The evaluation is relatively fast.
- The vectorization is achieved using the `func_sample` keyword and the `numpy` library.

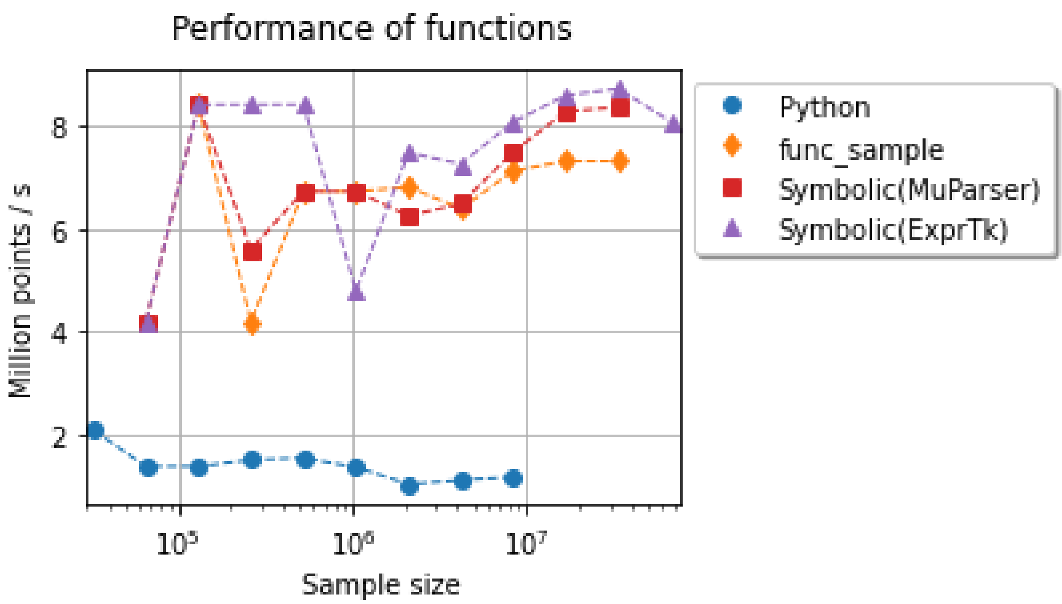


Figure 2. A benchmark of two Python functions compared to a symbolic function (see `python_benchmark.py`).

We see that, on this example, the two fastest methods are the symbol function and the vectorised Python functions.

Comments :

- See also the benchmark available in the doc.
- The performance of the symbolic function may depend on the backend `ExprTk` or `MuParser`. This can be configured using the `SymbolicParser-Backend` key of the `ResourceMap`.

[3]:
```
ot.ResourceMap.Set("SymbolicParser-Backend", "ExprTk")  # The default
ot.ResourceMap.Set("SymbolicParser-Backend", "MuParser")
```

1.9 How can PythonFunction be made fast?

The `PythonFunction` is based on the `OpenTURNSPythonFunction` than we will present later in the slides.

Depending on the `func_sample` and `n_cpus` options, we can make the evaluation faster.

- `func_sample`: evaluate the function on a `Sample` instead of a `Point` to vectorize the evaluations ;
- `n_cpus`: uses `multiprocessing` to make the evaluations parallel.

Here are different cases, depending the options specified by the user:

- If `func_sample` is undefined and `n_cpus` is undefined (i.e. the default), then the user-provided `func` function is used. It is not made parallel by OpenTURNS (but the user can do so).
- If `func_sample` is undefined and `n_cpus` is defined, then the implementation uses `multiprocessing`'s `Pool` to make the evaluation parallel (see the details in the private method `_exec_sample_multiprocessing_func`).
- If `func_sample` and `n_cpus` are both defined, then the implementation uses divides the `Sample` into sub-samples which are evaluated in parallel (see the details in the private method `_exec_sample_multiprocessing_func_sample`).

1.10 What is the memoryview class?

The `memoryview` object is the type of object we receive as input to a `PythonFunction`.

This class provides a lightweight object which prevent unnecessary object copies which can make the evaluation slower.

[4]:
```python
def mySimulator(x):
    print("Type of x : ", type(x))
    # dimension = x.getDimension()  # Fail
    y0 = x[0] + x[1] + x[2]
    y1 = x[0] - x[1] * x[2]
    y = [y0, y1]
    return y


gFunction = ot.PythonFunction(3, 2, mySimulator)
x = [1.0] * 3
print("x = ", x)
y = gFunction(x)
print("y = ", y)
```

```
x =  [1.0, 1.0, 1.0]
Type of x :  <class 'openturns.memoryview.Buffer'>
y =  [3,0]
```

The method `x.getDimension()` fails:

```
AttributeError: 'openturns.memoryview.Buffer'
```

```
object has no attribute 'getDimension'
```

From the doc:

"For efficiency reasons, these functions do not receive a `Point` or `Sample` as arguments, but a proxy object which gives access to internal object data. This object supports indexing, but nothing more. It must be wrapped into another object, for instance `Point` in `func` and `Sample` in `func_sample`, or in a Numpy `array`, for vectorized operations."

If we have to , we can convert the `memoryview` into a `Point`: this can make the evaluation slower, but can be necessary in some situations.

[5]:
```
def mySimulator(x):
    x = ot.Point(x)  # Convert to Point, but only if necessary
    print("Type of x : ", type(x))
    dimension = x.getDimension()  # Ok
    y0 = 0.0
    for i in range(dimension):
        y0 += x[i]
    y1 = x[0] - x[1] * x[2]
    y = [y0, y1]
    return y


gFunction = ot.PythonFunction(3, 2, mySimulator)
x = [1.0] * 3
gFunction(x)
```

```
Type of x :  <class 'openturns.typ.Point'>
```

[5]:
```
class=Point name=Unnamed dimension=2 values=[3,0]
```

1.11 What is the ParametricFunction for?

There are some cases when we want to create a function which has parameters, e.g. the gravity of Earth $g = 9.81\ m/s^2$. The `ParametricFunction` can be considered when the parameters is a vector of points:

- to perform Calibration using least squares : the parameter to calibrate is the parameter of the `ParametricFunction` ;
- to perform Bayesian calibration : the input of the `ParametricFunction` function is the parameter for which we want the *posterior* distribution ;
- to solve optimization problems with a parametric function (which avoids to create a new function each time the parameters change) ;
- etc.

We consider here the Ishigami test function.

1.11.1 References

- Ishigami, T., Homma, T. (1990, December). An importance quantification technique in uncertainty analysis for computer models. In Uncertainty Modeling and Analysis, 1990. Pro-

ceedings., First International Symposium on (pp. 398-403). IEEE.

[6]:
```
def ishigami(x):
    x0, x1, x2, a, b = x
    y0 = np.sin(x0) + a * np.sin(x1) ** 2 + b * x2**4 * np.sin(x0)
    y = [y0]
    return y


gFunctionFull = ot.PythonFunction(5, 1, ishigami)
a = 7.0
b = 0.1
xFull = [0.5, 1.0, 1.5, a, b]
y = gFunctionFull(xFull)
print(y)
```

```
[5.67865]
```

To propage the uncertainty through the Ishigami function, we can set the parameters a and b, so that the function only has `(x1, x2, x3)` as input.

Please use the Ishigami from the library for real simulations.

The next table presents the mapping from the variable name to its index. Notice that Python have indices that start from 0. The `gFunctionFull` function has no parameters.

Input variable	Input index
x_1	0
x_2	1
x_3	2
a	3
b	4

Parameter	Parameter index
$\emptyset$	$\emptyset$

Table 5. Input variables and parameters of the (full) `gFunctionFull` function.

[7]:
```
indices = [3, 4]
referencePoint = [a, b]
gFunction = ot.ParametricFunction(gFunctionFull, indices, referencePoint)
x = [0.5, 1.0, 1.5]
y = gFunction(x)
print(y)
```

```
[5.67865]
```

Manage parameters:

- We can use the `setParameter()` to set the parameters (and `getParameter()` to get them).
- The `parameterGradient()` returns the gradient of the function with respect to the parameters.

[8]:
```
print(gFunction.getParameter())
gFunction.setParameter([8.0, 0.2])
print(gFunction.getParameter())
```

```
[7,0.1]
[8,0.2]
```

The next table present the inputs and parameters of the parametric `gFunction` function. It has 3 inputs and 2 parameters.

Input variable	Input index
x_1	0
x_2	1
x_3	2

Parameter	Parameter index
a	0
b	1

Table 6. Input variables and parameters of the (parametric) `gFunction` function.

1.12 Why is the OpenTURNSPythonFunction class most powerful?

When the parameters cannot be stored as a single `Point`, we need a more powerful tool: the `OpenTURNSPythonFunction` class provides a way to implement a class so that we can provide any necessary parameter (whatever its type) to the evaluation.

- The constructor of the object can have any number or type of input arguments, as any `Class` object in Python.
- The calculations which are done in the constructor are done *"once for all"* at the creation of the object, which can make some calculations faster.

[9]:
```
class IshigamiFunction(ot.OpenTURNSPythonFunction):
    def __init__(self, a, b):
        super().__init__(3, 1)
        self.a = a
        self.b = b

    def _exec(self, x):
        y0 = (
            np.sin(x[0])
            + self.a * np.sin(x[1]) ** 2
            + self.b * x[2] ** 4 * np.sin(x[0])
        )
```

```
        return [y0]
```

[10]:
```
a = 7.0
b = 0.1
ishigamiFunction = IshigamiFunction(a, b)  # Create the object
gFunction = ot.Function(ishigamiFunction)  # Convert to a Function
x = [0.5, 1.0, 1.5]
y = gFunction(x)
print(y)
```

```
[5.67865]
```

Other examples:

- in the PRACE training a wrapper to the cantilever beam ;
- in otbenchmark, the Dirichlet test function for sensitivity analysis ;
- in otbenchmark, the flooding test case ;
- in otbenchmark, the Morris test function.

1.13 Otwrapy

The `otwrapy` package (available at github) provides a `Parallelizer` class that converts any `ot.Function` into a parallel wrapper using either `multiprocessing`, `ipyparallel`, `joblib` or `pathos`.

See the slides from Felipe Aguirre Martinez for Otwrapy details.

To install `otwrapy`, use conda (package):

```
$ conda install -c conda-forge otwrapy
```

The `Parallelizer` class parallelize a `Function`:

```
import otwrapy as otw
from otwrapy.examples.beam import Wrapper
parallelized_beam_wrapper = otw.Parallelizer(Wrapper())
```

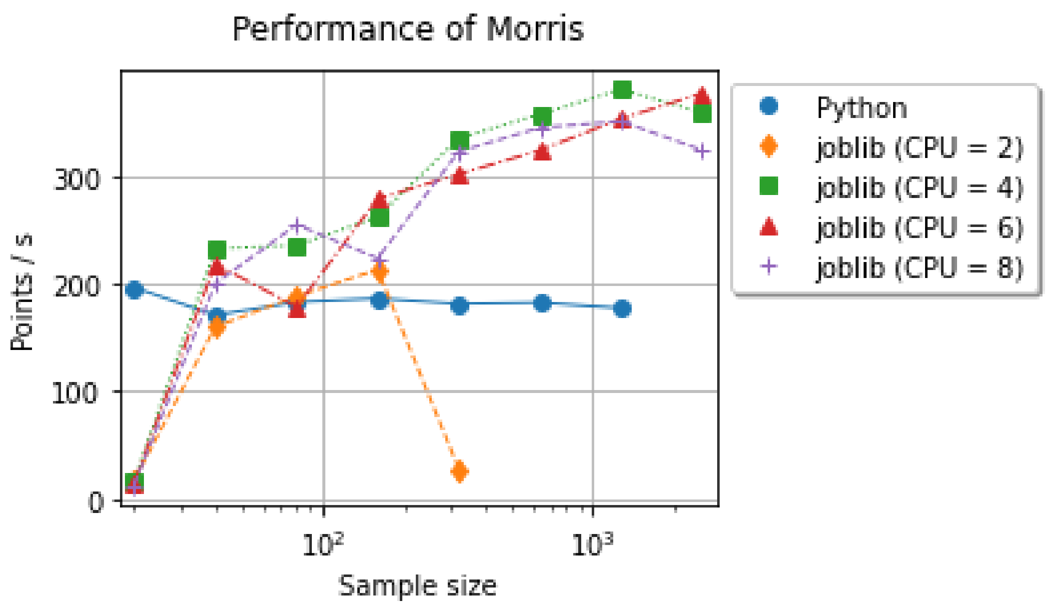


Figure 2. A benchmark of Morris function from `otmorris` using `otwrapy` (see the implementation details in `benchmark_Morris_otwrapy.py`).

We see that, on this example, the "joblib" backend improves the performance when the number of CPUs is in the range $[4, 6]$. Using more CPUs decreases the performance.

1.14 Jax

Jax is the new Autograd.

- can automatically differentiate native Python and NumPy code.
- uses XLA to compile and run your NumPy programs on GPUs and TPUs.
- can differentiate through loops, branches, recursion, and closures, and it can take derivatives of derivatives of derivatives.
- supports reverse-mode differentiation (a.k.a. backpropagation) via grad as well as forward-mode differentiation, and the two can be composed arbitrarily to any order.

Three main functions:

- `jit()`: speeding up your code,
- `grad()`: taking derivatives,
- `vmap()`: automatic vectorization or batching.

From the Quickstart :

```
import jax.numpy as jnp
from jax import grad, jit, vmap

def sum_logistic(x):
  return jnp.sum(1.0 / (1.0 + jnp.exp(-x)))

x_small = jnp.arange(3.)
```

```
derivative_fn = grad(sum_logistic)
print(derivative_fn(x_small))
```

1.15 Other Python objects

Class	Purpose	Link
PythonRandomVector	Simulate a random vector	Link
PythonDistribution	Define a distribution	Link

Table 6. Other Python objects.

1.16 PythonRandomVector

The `PythonRandomVector` class can be used to implement the `getRealization()` method for an object for which the distribution is not necessarily known.

Other examples:

- implement a NormalTruncatedToBall
- simulate a Markov chain
- implement a BoxConstrainedNormal

We would like to estimate the PDF of the biased sample variance.

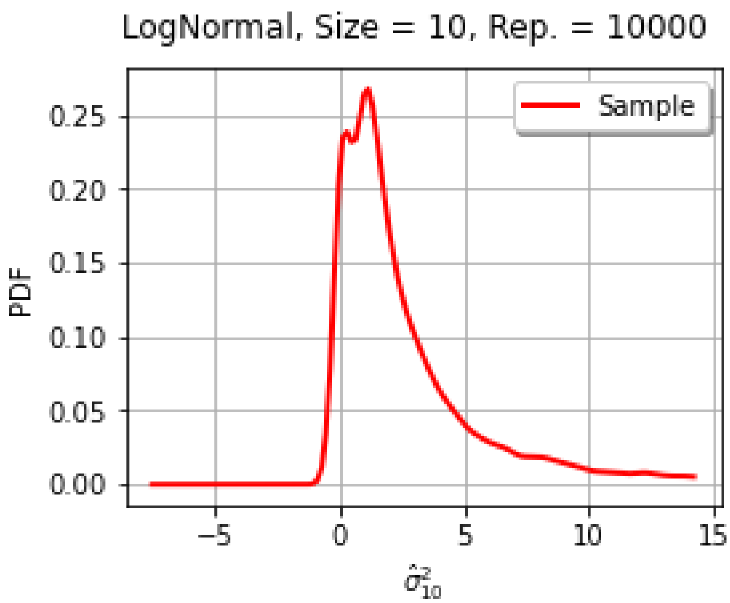
[11]:
```
class BiasedSampleVariance(ot.PythonRandomVector):
    def __init__(self, distribution, sample_size):
        super().__init__(1)
        self.setDescription(["$\hat{\sigma}^2_{%d}$" % (sample_size)])
        self.sample_size = sample_size
        dimension = distribution.getDimension()
        self.distribution = distribution

    def getRealization(self):
        sample = self.distribution.getSample(self.sample_size)
        sample_variance = sample.computeCenteredMoment(2)[0]
        return [sample_variance]
```

[12]:
```
def plot_sample_by_kernel_smoothing(distribution, sample_size, repetitions_size):
    myRV = ot.RandomVector(BiasedSampleVariance(distribution, sample_size))
    sample_variance = myRV.getSample(repetitions_size)

    graph = ot.KernelSmoothing().build(sample_variance).drawPDF()
    graph.setLegends(["Sample"])
    name = distribution.getClassName()
    graph.setTitle("%s, Size = %d, Rep. = %d" % (name, sample_size,␣
 ↪repetitions_size))
    return graph
```

```
repetitions_size = 10000
view = otv.View(
    plot_sample_by_kernel_smoothing(ot.LogNormal(), 10, repetitions_size),
    figure_kw={"figsize": (4.0, 3.0)},
)
```



1.17 PythonDistribution

The `PythonDistribution` class defines a distribution.

The two mandatory methods are:

- `getRange()`;
- `computePDF()`.

Implementing other methods can improve speed and accuracy.

We would like to easily see the asymptotic distribution of the sample variance.

[13]:
```
class SampleVarianceAsymptoticDistribution(ot.PythonDistribution):
    def __init__(self, distribution, sample_size):
        super().__init__(1)
        self.distribution = distribution
        self.sample_size = sample_size
        asymptotic_mean = self.distribution.getCenteredMoment(2)[0]
        asymptotic_variance = (
            self.distribution.getCenteredMoment(4)[0] - asymptotic_mean**2
        )
```

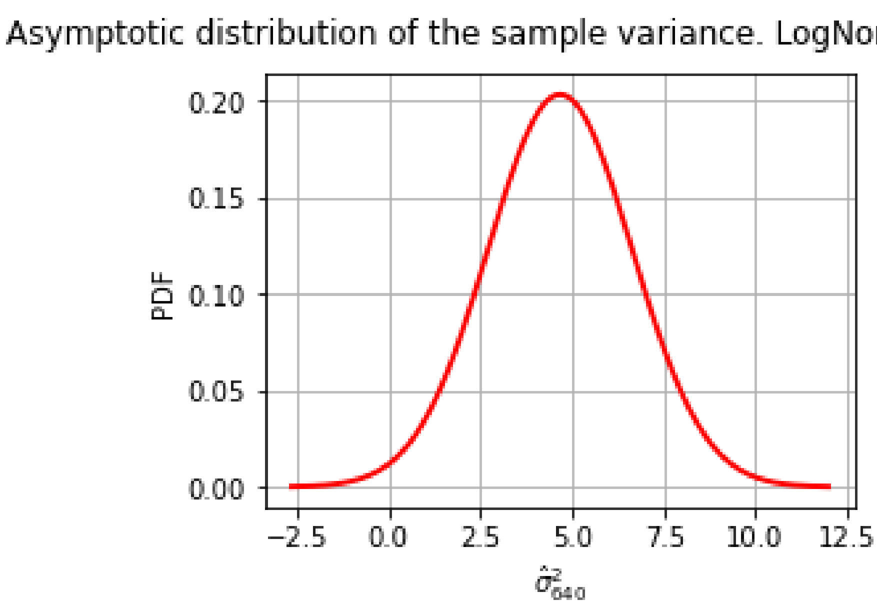
```
        asymptotic_sd = np.sqrt(asymptotic_variance) / np.sqrt(self.sample_size)
        self.asymptotic_distribution = ot.Normal(asymptotic_mean, asymptotic_sd)

    def computePDF(self, x):
        y = self.asymptotic_distribution.computePDF(x)
        return y

    def computeCDF(self, x):
        y = self.asymptotic_distribution.computeCDF(x)
        return y

    def getRange(self):
        return self.asymptotic_distribution.getRange()
```

[14]:
```
asymptoticDistribution = SampleVarianceAsymptoticDistribution(
    ot.LogNormal(), 640
)  # Create the object
distribution = ot.Distribution(asymptoticDistribution)  # Convert to Distribution
graph = distribution.drawPDF()
graph.setTitle("Asymptotic distribution of the sample variance. LogNormal, n =␣
 ↪640")
graph.setLegends([""])
graph.setXTitle("$\hat{\sigma}_{640}^2$")
view = otv.View(
    graph,
    figure_kw={"figsize": (4.0, 3.0)},
)
```



1.18 What's next?

Please consider the exercises in this repository:

- `Python_function_exercises.ipynb` : exercises on `PythonFunction` ;
- `Coupling_tools.ipynb`: on `coupling_tools` sub-module to connect to an external program using files ;
- `Parametric_function.ipynb`: practical hands-on exercises on the `ParametricFunction` and `OpenTURNSPythonFunction` classes ;
- `Symbolic_function.ipynb`: exercises on `SymbolicFunction`.
- `python_benchmark.py`: a benchmark with Python functions.