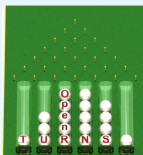


# OpenTURNS Developer training: first steps

Trainer : Régis LEBRUN  
Airbus  
[regis.lebrun@airbus.com](mailto:regis.lebrun@airbus.com)

Developers training



# OpenTURNS: first steps

1 Navigation in the source code

2 Library development

3 Module development

# Navigation in the source code

## The Uniform distribution

- Locate the class within the library source code;
- Follow its inheritance graph in order to explore the Bridge pattern;
- Locate the associated regression test;
- Execute the test;
- Locate its SWIG interface file and its associated Python module;
- Execute the associated python test.

## Library development 1/9

## Projects

- ① (\*) **InverseDistanceWeightingInterpolation** as a specialization of **EvaluationImplementation** (see `lib/src/Base/Func`). Given a set of data  $(x_i, y_i)_{i=1, \dots, N}$  in  $\mathbb{R}^n \times \mathbb{R}^p$ , the IDW interpolation is defined by:

$$\forall x \in \mathbb{R}^n, u(x) = \begin{cases} \frac{\sum_{i=1}^N w_i(x) y_i}{\sum_{i=1}^N w_i(x)} & \text{if } \forall i, d(x, x_i) > 0 \\ y_i & \text{if } \exists i, d(x, x_i) = 0 \end{cases} \quad (1)$$

where  $w_i(x) = \frac{1}{d(x, x_i)^p}$  and  $p > 0$  a given smoothness parameter.

The distance  $d$  can be the Euclidean distance, the 1-norm or the sup norm.

- ② (\*\*) **DiscreteIntegralCompound** as a specialization of **DiscreteDistribution** (see `lib/src/Uncertainty/Distribution`). Given the discrete distribution of a random variable  $N$  and the common discrete distribution of a sequence of iid integral valued random variables  $(X_i)$ , compute the distribution of the integral valued discrete random variable  $Y$  defined by:

$$Y = \sum_{i=1}^N X_i \quad (2)$$

## Library development 2/9

## Projects

Its generating function  $\phi_Y(z) = E[z^Y]$  is given by:

$$\forall z \in \mathbb{C}, \phi_Y(z) = \phi_N(\phi_X(z)) \quad (3)$$

and thanks to Poisson's summation formula for discrete distributions, we have for  $0 < r < 1$  and  $m \in \mathbb{N}^*$ :

$$\forall n \in \{0, \dots, m-1\}, p_Y(n) = \frac{1}{mr^n} \sum_{k=0}^{m-1} \phi_Y\left(re^{\frac{2i\pi k}{m}}\right) e^{-\frac{2i\pi kn}{m}} - e_d \quad (4)$$

where  $0 \leq e_d \leq r^m$  is the approximation error. For a given  $\epsilon > 0$  and  $m \in \mathbb{N}^*$ , set  $r = \sqrt[m]{\epsilon}$  and compute the FFT  $(\omega_0, \dots, \omega_{m-1})$  of the complex vector  $(\phi_Y(re^{\frac{2i\pi k}{m}}), \dots, \phi_Y(re^{\frac{2i\pi k}{m}}))$ . Then, the distribution is equal to the UserDefined distribution with locations  $\{0, \dots, m-1\}$  and probabilities  $\left(p_i = \frac{\Re(\omega_i)}{mr^i}\right)_{i=0, \dots, m-1}$ .

## Library development 3/9

## Projects

- ③ (\*) **ClenshawCurtis** integration algorithm as a specialization of **IntegrationAlgorithmImplementation** (see `lib/src/Base/Algo`). This integration algorithm allows to compute integrals of the form:

$$\begin{aligned} I(f) &= \int_a^b f(t) dt \\ &= \frac{b-a}{2} \int_{-1}^1 f\left(a + \frac{b-a}{2}(1+x)\right) dx \\ &\simeq \frac{b-a}{2} \sum_{k=0}^n w_k f\left(a + \frac{b-a}{2}(1+x_k)\right) \end{aligned}$$

where  $x_k = \cos \theta_k$ ,  $\theta_k = \frac{k\pi}{n}$  and  $w_k$  is given by:

$$w_k = \frac{c_k}{n} \left( 1 - \sum_{j=1}^{\lfloor n/2 \rfloor} \frac{b_j}{4j^2 - 1} \cos(2j\theta_k) \right) \quad (5)$$

## Library development 4/9

## Projects

where the coefficients  $b_j$  and  $c_k$  are given by:

$$b_j = \begin{cases} 1 & j = n/2 \\ 2 & j < n/2 \end{cases} \quad c_k = \begin{cases} 1 & k = 0[n] \\ 2 & k \neq 0[n] \end{cases} \quad (6)$$

for  $k = 0, \dots, n$ . An efficient FFT-based implementation of the computation of the weights and nodes is given in `fc1encurt.m`, another one (\*\*) in `1311.0445.pdf`.

- ④ (\*) **Fejer1** integration algorithm as a specialization of **IntegrationAlgorithmImplementation** (see `lib/src/Base/Algo`). This integration algorithm is based on the nodes  $x_k = \cos \theta_{k+1/2}$  and weights:

$$w_k^{f1} = \frac{2}{n} \left( 1 - 2 \sum_{j=1}^{\lfloor n/2 \rfloor} \frac{1}{4j^2 - 1} \cos(j\theta_{2k+1}) \right) \quad (7)$$

for  $k = 0, \dots, n-1$ . There also exist fast implementations based on FFT or modified moments, see the references for Clenshaw Curtis.

## Library development 5/9

## Projects

- 5 (\*) **Fejer2** integration algorithm as a specialization of **IntegrationAlgorithmImplementation** (see `lib/src/Base/Algo`). This integration algorithm is based on the nodes  $x_k = \cos \theta_k$  and weights:

$$w_k^{f2} = \frac{4}{n} \sin \theta_k \sum_{j=1}^{\lfloor n/2 \rfloor} \frac{\sin((2j-1)\theta_k)}{2j-1} \quad (8)$$

for  $k = 0, \dots, n$ . There also exist fast implementations based on FFT or modified moments, see the references for Clenshaw Curtis.

- 6 (\*\*) **ClenshawCurtisProductExperiment** as a specialization of **WeightedExperiment**: same algorithm as for **ClenshawCurtis** but with adaptation to any weight function.
- 7 (\*) **MarshallOlkinCopula** as a specialization of **CopulaImplementation** (see `lib/src/Uncertainty/Distribution`). This copula is defined by:

$$\forall (u, v) \in [0, 1]^2, C(u, v) = \begin{cases} u^{1-\alpha} v & \text{for } u^\alpha \geq v^\beta \\ uv^{1-\beta} & \text{for } u^\alpha < v^\beta \end{cases} \quad (9)$$

where  $0 < \alpha, \beta < 1$ .



## Library development 6/9

## Projects

- 8 (\*) **GumbelCopula** as a specialization of `ExtremeValueCopula` (see `lib/src/Uncertainty/Distribution`). This copula already exists, but not as an extreme value copula. It is defined by its Pickand function:

$$\forall t \in [0, 1], A(t) = [t^\theta + (1 - t)^\theta]^{1/\theta} \quad (10)$$

where  $\theta \geq 1$ .

- 9 (\*) **GalambosCopula** as a specialization of `ExtremeValueCopula` (see `lib/src/Uncertainty/Distribution`). This copula is defined by its Pickand function:

$$\forall t \in [0, 1], A(t) = 1 - [t^{-\theta} + (1 - t)^{-\theta}]^{-1/\theta} \quad (11)$$

where  $\theta \geq 0$ .

- 10 (\*) **TawnCopula** as a specialization of `ExtremeValueCopula` (see `lib/src/Uncertainty/Distribution`). This copula is defined by its Pickand function:

$$\forall t \in [0, 1], A(t) = (1 - \psi_1)(1 - t) + (1 - \psi_2)t + [\{\psi_1 t\}^{1/\theta} + \{\psi_2(1 - t)\}^{1/\theta}]^\theta \quad (12)$$

## Library development 7/9

## Projects

- 11 (\*) **JoeCopula** as a specialization of `ExtremeValueCopula` (see `lib/src/Uncertainty/Distribution`). This copula is defined by its Pickand function:

$$\forall t \in [0, 1], A(t) = 1 - \left[ \{\psi_1(1 - t)\}^{-1/\theta} + \{\psi_2 t\}^{-1/\theta} \right]^{-\theta} \quad (13)$$

where  $\theta > 0$  and  $0 \leq \psi_1, \psi_2 \leq 1$ .

- 12 (\*\*) **ArchiMaxCopula** as a specialization of `CopulaImplementation` (see `lib/src/Uncertainty/Distribution`). Given an Archimedean copula with generator  $\psi$  and an extreme value copula with Pickand function  $A$ , an archimax copula  $C$  is defined by:

$$\forall (u, v) \in [0, 1]^2, C(u, v) = \psi^{-1} \left( \min \left( \psi(0), [\psi(u) + \psi(v)] A \left( \frac{\psi(u)}{\psi(u) + \psi(v)} \right) \right) \right) \quad (14)$$

It becomes (\*\*\*) if one wants to implement an efficient sampling algorithm.

# Library development 8/9

## Projects

- 13 (\*) **SquaredNormal** as a specialization of `ContinuousDistribution` (see `lib/src/Uncertainty/Distribution`). If  $X$  is distributed according to the  $\mathcal{N}(\mu, \sigma)$  distribution,  $Y = X^2$  is distributed according to the squared normal distribution with parameters  $\mu$  and  $\sigma > 0$ . This distribution has already been implemented in Python, see `SquaredNormal.py`.
- 14 (\*\*) **ConditionalEventDistribution** as a specialization of `ContinuousDistribution` (see `lib/src/Uncertainty/Distribution`). Given the joint distribution of an  $(m + n)$  dimensional random vector  $(X, Y)$  and an  $m$  dimensional interval  $I$  such that  $\mathbb{P}(X \in I) > 0$ , it is the distribution of  $Y$  knowing that  $X \in I$ . This distribution has already been implemented in Python, see `ConditionalEventDistribution.py`. It becomes (\*\*\*) if one wants to implement an efficient simplification mechanism.
- 15 (\*\*\*) Extend archimedian copulas from 2-d to  $n$ -d. Given a 2-d Archimedean copula with generator  $\psi$ , implement its  $n$ -d counterpart using:

$$\forall (u_1, \dots, u_n) \in [0, 1]^n, C(u_1, \dots, u_n) = \psi^{-1}(\psi(u_1) + \dots + \psi(u_n)) \quad (15)$$

The main difficulties are the architecture of this extension and the implementation of an efficient sampling algorithm.

## Library development 9/9

## Projects

- 16 (\*\*) **BlockComposedDistribution** as a specialization of **DistributionImplementation** (see `lib/src/Uncertainty/Distribution`). Given a collection of distributions  $D_1, \dots, D_n$  of dimensions  $d_1, \dots, d_n$ , it is the distribution of the random vector  $(X_1, \dots, X_n)$  of dimension  $d_1 + \dots + d_n$  where  $X_i$  is distributed as  $D_i$  and  $X_1, \dots, X_n$  are independent. It becomes (\*\*\*\*) if one wants to propagate this new distribution in every places it could go within the library.
- 17 (\*) Extend **SolverImplementation** and **Solver** to the resolution of systems of nonlinear equations and provide a generic implementation using the **LeastSquaresProblem** class. The solutions  $x^*$  of a nonlinear system of equations  $f_1(x) = 0, \dots, f_n(x) = 0$  where  $x = (x_1, \dots, x_n)$ , if they exist, have to be found in the set of solutions of the following least-squares problem:

$$x^* = \arg \min \sum_{j=1}^n f_j^2(x) \quad (16)$$

for which many solvers are available in OpenTURNS.

# Module development 1/2

## Projects

- 18 (\*) or (\*\*) **CloudMesher**: mesh generation over a cloud of points using kernel mixture, pca, rotation, then levelset mesher on an interval
- 19 (\*) **UniformSphereRandomVector** as a specialization of `RandomVectorImplementation` (see `lib/src/Uncertainty/Model`). This random vector is distributed uniformly on the sphere of center  $c \in \mathbb{R}^n$  and radius  $r > 0$ . The sampling is done using the fact that  $Y/\|Y\|$  is uniformly distributed over  $S_{n-1}$ , the unit sphere in  $\mathbb{R}^n$ , if  $Y$  is an  $n$  dimensional random vector with independent  $\mathcal{N}(0, 1)$  components.
- 20 (\*) **UniformBallRandomVector** as a specialization of `RandomVectorImplementation` (see `lib/src/Uncertainty/Model`). This random vector is distributed uniformly on the ball of center  $c \in \mathbb{R}^n$  and radius  $r > 0$ . The sampling is done using the fact that  $Y/\sqrt{\|Y\|^2 + Z}$  is uniformly distributed over  $B_{n-1}$ , the unit ball in  $\mathbb{R}^n$ , if  $Y$  is an  $n$  dimensional random vector with independent  $\mathcal{N}(0, 1)$  components, and  $Z$  is  $\mathcal{E}(1)$  independent from  $Y$ .
- 21 (\*) **UniformSimplexRandomVector** as a specialization of `RandomVectorImplementation` (see `lib/src/Uncertainty/Model`). This random vector is distributed uniformly in the simplex given by  $n + 1$  points in  $\mathbb{R}^n$ . The sampling is done using the fact that  $Y$  is uniformly distributed over the standard simplex in  $\mathbb{R}^n$  if it follows the Dirichlet distribution with parameter  $(\theta_1 = 1, \dots, \theta_n = 1)$ .

## Module development 2/2

### Projects

- 22 (\*\*) **SmoliakExperiment** as a specialization of `WeightedExperiment` (see `lib/src/Uncertainty/Algorithm/WeightedExperiment`). This design of experiment is obtained by interfacing the `smolpack` C library. A possible name for the module is **OTSmolpack**.
- 23 (\*\*) **CubaIntegration** as a specialization of `IntegrationAlgorithmImplementation` (see `lib/src/Base/algo`). This algorithm is obtained by interfacing the `cuba` C library. A possible name for the module is **OTCuba**.
- 24 (\*\*) **HIntLibIntegration** as a specialization of `IntegrationAlgorithmImplementation` (see `lib/src/Base/algo`). This algorithm is obtained by interfacing the `HIntLib` C++ library, see <https://github.com/JohannesBuchner/HIntLib>. A possible name for the module is **OTHIntLib**.