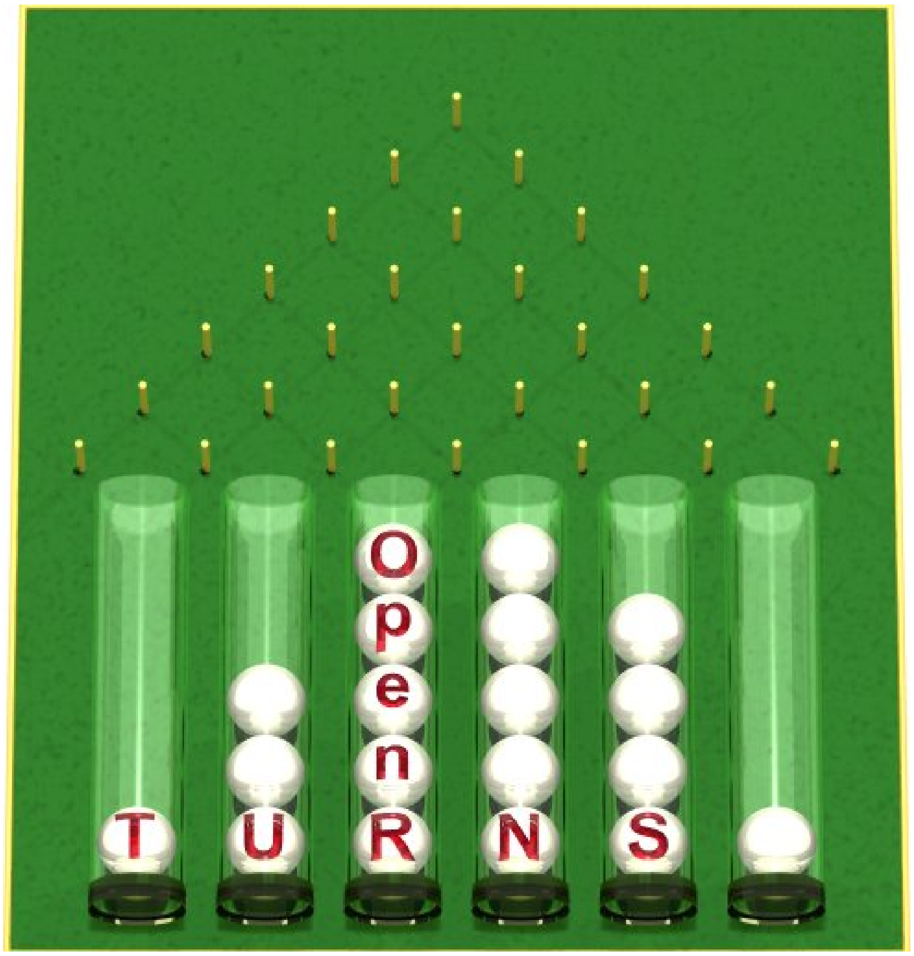
Use Cases Guide for the Textual User Interface

Open TURNS version 0.12.1

November 8, 2008



Contents

Introduction **3**

1 Probabilistic input vector modelisation **4**
1.1 Without samples on data . . . 4
1.1.1 UC : List of usual distributions . . . 4
1.1.2 UC : Creation of a truncated distribution . . . 15
1.1.3 UC : Creation of a copula and a composed copula . . . 16
1.1.4 UC : Creation of nD distribution from (marginals, copula) . . . 20
1.1.5 UC : Creation of a nD distribution from a Mixture . . . 22
1.1.6 UC : Manipulation of a distribution . . . 26
1.1.7 UC : Creation of the random input vector from a distribution . . . 33
1.2 With samples on data : manipulation on data . . . 34
1.2.1 UC : Import / Export data from a file at format CSV (Comma Separated Value) . . . 35
1.2.2 UC : Drawing Empirical CDF, Histogram, Clouds / PDF or superposition of two clouds from data . . . 36
1.2.3 UC : Do two samples have the same distribution : QQ-plot visual test, Smirnov numerical test . . . 41
1.2.4 UC : Are two scalar samples independent : ChiSquared test, Pearson test, Spearman test 43
1.2.5 UC : Particular manipulations of the Pearson and Spearman tests, when the first sample is of dimension superior to 1. . . . 45
1.2.6 UC : Regression test between two scalar numerical samples . . . 48
1.2.7 UC : Distribution fitting tests, numerical and visual validation tests : ChiSquared test, Kolmogorov test, QQ-plot graph . . . 49
1.2.8 UC : Normal distribution fitting test, visual validation tests (Henry line) and numerical validation tests in extreme zones (Anderson Darling test and Cramer Von Mises test) . 52
1.2.9 UC : Making a choice between multiple fitted distributions : Kolmogorov ranking, ChiSquared ranking and BIC ranking . . . 56
1.2.10 UC : PDF fitting by kernel smoothing and graphical validation : superposition of the empirical and kernel smoothing CDF . . . 59
1.2.11 UC : Building and validating a linear model from two samples . . . 63
1.2.12 UC : Statistical manipulations on data : min, max, covariance, skewness, kurtosis, quantile, empirical CDF, Pearson, Kendall and Spearman correlation matrixes and rank/sort functionnalities . . . 70
1.2.13 UC : Drawing one cloud . . . 74

2 Creation of the limit state function and the output variable of interest **75**
2.1 Creation of the limit state function . . . 75
2.1.1 UC : From an external wrapper with gradient and hessian implementations . . . 75
2.1.2 UC : From an analytical formula declared in line . . . 77
2.1.3 UC : Introducing some deterministic variables, using a LinearNumericalMathFunction . 78
2.1.4 UC : Introducing some deterministic variables, optimising memory and CPU time . . . 80
2.1.5 UC : Manipulation of a NumericalMathFunction . . . 82
2.2 Creation of the output variable of interest from the limit state function and the random input vector . . . 84
2.2.1 UC : Creation of the ouput random vector . . . 84
2.2.2 UC : Extraction of a random subvector from a random vector . . . 85

3 Uncertainty propagation and Uncertainty sources ranking 86
3.1 Deterministic approach : Min/Max study 86
3.1.1 UC : Creation of a deterministic experiment plane 86
3.1.2 UC : Drawing an experiment plane in dimension 2 92
3.1.3 UC : Creation of a deterministic experiment plane in the physical space (type : Axial) where levels are proportionnal to the standard deviation of each component of the random input vector, and centered on the mean vector of the random input vector 92
3.1.4 UC : Creation of a random experiment plane 94
3.1.5 UC : Min/Max research of the output variable of interest from an experiment plane in the physical space (deterministic or random) of the input random vector and deterministic sensitivity of the output variable to the input vector at a particular point 95
3.2 Random approach : central uncertainty 96
3.2.1 UC : Correlation analysis on samples : Pearson and Spearman coefficients, PCC, PRCC, SRC, SRRC coefficients 96
3.2.2 UC : Moments evaluation from the Taylor variance decomposition method and evaluation of the importance factors associated 97
3.2.3 UC : Quantile estimations : Wilks and empirical estimators 98
3.3 Random approach : threshold exceedance 101
3.3.1 UC : Creation of an event in the physical and the standard spaces 101
3.3.2 UC : Manipulation of a StandardEvent 102
3.3.3 UC : Probability evaluation from FORM method and results associated : importance factors, reliability indexes, sensitivity on the FORM event probability and Hasofer-Lind reliability index 103
3.3.4 UC : Probability evaluations from SORM methods and results associated : importance factors, reliability indexes, sensitivity on the Hasofer-Lind reliability index 108
3.3.5 UC : Probability evaluation from the Monte Carlo simulation method, determination of the confidence interval of the probability and drawing of the convergence curve with the confidence curves 114
3.3.6 UC : Probability evaluation from Directional Sampling method, determination of the confidence interval and drawing of the convergence curve with the confidence curves . . 118
3.3.7 UC : Probability evaluation from Importance Sampling method centered on the design point issued from the FORM method, determination of the confidence interval and drawing of the convergence curve with the confidence curves 125

4 Construction of a response surface 130
4.1 UC : Linear and Quadratic Taylor approximations 130
4.2 UC : Linear approximation response surface by least squares method from a sample of the input vector and the real function 131
4.3 UC : Linear approximation response surface by least squares method from a sample of the input vector and a sample of the output vector 132

5 How to save and load a study ? 133
5.1 UC : How to save a study ? 134
5.2 UC : How to load a study ? 135

6 Annexe 1 : One example of a complete study 138
6.1 Presentation of the study case 138
6.2 The TUI File 139

Introduction

This guide aims at facilitating the use of Open TURNS through its textual User interface (TUI), by proposing numerous examples of TUI studies.

The presentation of the Use Cases Guide follows the methodology of uncertainty treatment presented in the scientific documentation of Open TURNS : examples are divided into four steps corresponding to the four steps of an uncertainty treatment study.

The example list presented here is not exhaustive but recovers most of the standard User needs. The TUI enables the User to perform much more functionalities of the *openturns* python library than those presented here : it is necessary for the User to refer to the complete python documentation of the *openturns* python library to have the whole list of what is possible to perform.

It is important to note that the python test files given in open source with the code source of Open TURNS are very useful : they provide to the User an example of the utilisation of each object of Open TURNS. The User is invited to refer to them : they will surely help him to write his study through the TUI with the right syntax.

In order to write a python file using fonctionalities proposed by the *openturns* python module, it is necessary to load the module in the python shell. If there is no danger to overload functionalities coming from other python modules, the loading command is :

```
from openturns import *
```

Otherwise, if some functionalities of the *openturns* python module might overload some functionalities coming from other python modules, it is preferable to launch the command :

```
import openturns
```

In that second case, each call to an *openturns* type must be accompagnied by the prefix *openturns*. For example, to create a *NumericalPoint* of dimension 2, the command is *myNumericalPoint = openturns.NumericalPoint(2)*.

In order to visualize graphics through the TUI, it is necessary to import the functionality *ViewImage* from the *openturns_viewer* module, thanks to the command :

```
from openturns_viewer import ViewImage
```

The command :

```
dir()
```

gives a general overview of the whole objects proposed by the *openturns* python library.

The command *help* gives detailed information on each object of the *openturns* python library. For example, to get information on the object *NumericalPoint*, the command is :

```
help(NumericalPoint)
```

or

```
help(openturns.NumericalPoint)
```

according to the way the *openturns* python module has been loaded.
In order to quit the *help* document, tape the key *q*.

If *myObject* is one instance of an *openturns* object, then the command :

```
myObject.
```

followed by the *Tabulation* key lists all the methods proposed by the object *myObject*.

In order to have some automatic completion of the *openturns* objects and their methods, it is necessary to type the following command in the current python session :

```
import readline
import rlcompleter
readline.parse_and_bind(''tab: complete'')
```

These commands may be written in the file *.pythonrc.py* put in the root repertory *$HOME* : it will be automatically taken into account for current python sessions.
Then, in order to complete and list all the *openturns* objects which begin by *Num*, the command is :

```
Num[TAB]
```

To list all the methods proposed by the *NumericalMathFunction* object, the command is :

```
NumericalMathFunction.[TAB]
```

where $[TAB]$ is the Tabulation touch.

1 Probabilistic input vector modelisation

The objective of the section is to model the probabilistic input vector, described with different ways, according to available data .
It corresponds to the step “Step B : Quantify the uncertainty sources” of the global methodology.

1.1 Without samples on data

1.1.1 UC : List of usual distributions

The objective of this section is to list all the usual distributions proposed by Open TURNS and to precise how each distribution is created, with its different arguments.

The different distributions proposed by Open TURNS are listed here after.

- Continuous distributions :

Name	probability density function	conditions	param. 1	param. 2
Beta	$\dfrac{(x-a)^{(r-1)}(b-x)^{(t-r-1)}}{(b-a)^{(t-1)}B(r,t-r)}\mathbf{1}_{[a,b]}(x)$	$r > 0,\ t > r,$ $a < b$	(r,t,a,b)	(μ,σ,a,b) $\begin{cases}\mu = a+(b-a)\frac{r}{t}\\ \sigma = (b-a)\frac{r}{t}\frac{\sqrt{t-r}}{\sqrt{r(t+1)}}\end{cases}$
Exponential	$\lambda e^{-\lambda(x-\gamma)}\mathbf{1}_{[\gamma,+\infty[}(x)$	$\lambda > 0$	(λ,γ)	-
Gamma	$\dfrac{\lambda}{\Gamma(k)}(\lambda(x-\gamma))^{(k-1)}e^{-\lambda(x-\gamma)}\mathbf{1}_{[\gamma,+\infty[}(x)$	$k > 0,\ \lambda > 0$	(k,λ,γ)	$(\mu,\sigma,\gamma),\ \begin{cases}\mu = \frac{k}{\lambda}+\gamma\\ \sigma = \frac{\sqrt{k}}{\lambda}\end{cases}$
Gumbel	$\alpha e^{-\alpha(x-\beta)-e^{-\alpha(x-\beta)}}$	$\alpha > 0$	(α,β)	$(\mu,\sigma)^1\ \begin{cases}\mu = \frac{\gamma_e^*}{\alpha}+\beta\\ \sigma = \frac{\pi}{\sqrt{6}}\frac{1}{\alpha}\end{cases}$
Histogram	$\displaystyle\sum_{i=1}^{i=n} h_i\mathbf{1}_{[x_i,x_{i+1}]}(x)/S$	$l_i = x_{i+1}-x_i$ $S = \sum_{i=1}^{n} h_i l_i$ $l_i \geq 0$	$(x_1,(h_i,l_i))$ $1 \leq i \leq n$	-
Logistic	$\dfrac{\exp\left(\frac{x-\alpha}{\beta}\right)}{\beta\left[1+\exp\left(\frac{x-\alpha}{\beta}\right)\right]^2}\mathbf{1}_{[\alpha,+\infty[}(x)$	$\beta > 0$	(α,β)	-
LogNormal	$\dfrac{e^{-\frac{1}{2}(\frac{log(x-\gamma)-\mu_l}{\sigma_l})^2}}{\sqrt{2\pi}\sigma_l(x-\gamma)}\mathbf{1}_{[\gamma,+\infty[}(x)$	$\sigma_l > 0$	(μ_l,σ_l,γ)	(μ,σ,γ), param. 3 : $(\mu,\frac{\sigma}{\mu},\gamma)$ $\begin{cases}\mu = e^{\frac{1}{2}\sigma_l^2+\mu_l}+\gamma\\ \sigma = (e^{\frac{1}{2}\sigma_l^2+\mu_l})\sqrt{e^{\sigma_l^2}-1}\end{cases}$
Non Central Student(*)	$p_T(x)$ given under the table	-	(ν,δ,γ)	-
Normal (nD)	$\dfrac{1}{(2\pi)^{\frac{n}{2}}(\det\underline{\underline{\Sigma}})^{\frac{1}{2}}}e^{-\frac{1}{2}(\underline{x}-\underline{\mu})^t\underline{\underline{\Sigma}}^{-1}(\underline{x}-\underline{\mu})}$	$\underline{\underline{\Sigma}} = \underline{\underline{\Lambda}}(\underline{\sigma})\underline{\underline{R}}\underline{\underline{\Lambda}}(\underline{\sigma})$, $\underline{\underline{\Lambda}}(\underline{\sigma}) = diag(\underline{\sigma})$, $\underline{\underline{R}}$ SPD, $\sigma_i > 0$	$(\underline{\mu},\underline{\sigma},\underline{\underline{R}})$ or $(\underline{\mu},\underline{\underline{\Sigma}})$	-
Student	$\dfrac{1}{\sqrt{\nu}B(\frac{1}{2},\frac{\nu}{2})}(1+\dfrac{(x-\mu)^2}{\nu})^{-\frac{1}{2}(\nu+1)}$	$\nu > 2$	(ν,μ)	-
Triangular	$\begin{cases}2\dfrac{x-a}{(m-a)(b-a)} & a \leq x \leq m\\ 2\dfrac{b-x}{(b-m)(b-a)} & m \leq x \leq b\\ 0 & \text{otherwise.}\end{cases}$	$a < m < b,\ a < b$	(a,m,b)	-
Truncated Normal	$\dfrac{\frac{1}{\sigma_n}\phi(\frac{x-\mu_n}{\sigma_n})}{\Phi(\frac{b-\mu_n}{\sigma_n})-\Phi(\frac{a-\mu_n}{\sigma_n})}\mathbf{1}_{[a,b]}(x)$	$\sigma_n > 0$	(μ_n,σ_n,a,b)	-
Uniform	$\dfrac{1}{b-a}\mathbf{1}_{[a,b]}(x)$	$a < b$	(a,b)	-
Weibull	$\dfrac{\beta}{\alpha}(\dfrac{x-\gamma}{\alpha})^{\beta-1}e^{-(\frac{x-\gamma}{\alpha})^\beta}\mathbf{1}_{[\gamma,+\infty[}(x)$	$\alpha > 0,\ \beta > 0$	(α,β,γ)	(μ,σ,γ) $\begin{cases}\mu = \alpha\Gamma(1+\frac{1}{\beta})+\gamma\\ \sigma = \alpha\sqrt{\Gamma(1+\frac{2}{\beta})-\Gamma^2(1+\frac{1}{\beta})}\end{cases}$

(*) Let's note that a random variable X is said to have a standard non-central student distribution $\mathcal{T}(\nu,\delta)$ if it can be written as:

$$X = \frac{N}{\sqrt{C/\nu}} \tag{1}$$

where N has the normal distribution $\mathcal{N}(\delta,1)$ and C has the $\chi^2(\nu)$ distribution, N and C being independent.

[1] Euler's constant $\gamma_e = -\displaystyle\int_0^\infty \log(t)e^{-t}dt$.

The non-central Student distribution in OpenTURNS has an additional parameter γ such that the random variable X is said to have a non-central Student distribution $\mathcal{T}(\nu, \delta, \gamma)$ if $X - \gamma$ has a standard $\mathcal{T}(\nu, \delta)$ distribution.

We explicitate here the probability density function of the Non Central Student :

$$p_T(x) = \frac{\exp(-\delta^2/2)}{\sqrt{\nu\pi}\Gamma(\nu/2)} \left(\frac{\nu}{\nu + (x+\gamma)^2}\right)^{(\nu+1)/2} \sum_{j=0}^{\infty} \frac{\Gamma\left(\frac{\nu+j+1}{2}\right)}{\Gamma(j+1)} \left((x+\gamma)\delta\sqrt{\frac{2}{\nu+x^2}}\right)^j$$

- Discrete distributions :

Name	Distribution	conditions	param. 1
Geometric	$P(X=k) = p(1-p)^{k-1}$	$k \in \mathbb{N}^*$	p
MultiNomial (nD)	$P(\underline{X} = \underline{x}) = \frac{N!}{x_1! \dots x_n!(N-s)!} p_1^{x_1} \dots p_n^{x_n} (1-q)^{N-s}$	$0 \leq p_i \leq 1$ $x_i \in \mathbb{N}$ $q = \sum_{k=1}^{n} p_k \leq 1$ $s = \sum_{k=1}^{n} x_k \leq N$	$((p_k)_{1 \leq k \leq n}, N)$
Poisson	$P(X=k) = \frac{\lambda^k}{k!} e^{-\lambda}$	$k \in \mathbb{N}$	λ
User defined (nD)	$P(\underline{X} = \underline{x}_k) = p_k)_{1 \leq k \leq N}$	$0 \leq p_k \leq 1$, $\sum_{k=1}^{N} p_k = 1$	$(\underline{x_k}, p_k)_{1 \leq k \leq N}$

Furthermore, for all these 1D usual distributions, it is possible to truncate them within $[a, b]$, $[a, +\infty[$ or $]-\infty, b]$ (see UC.1.1.2).

Requirements	none
Results	• the random input distribution **type** : Distribution

The creation of each distribution is described in the following Python script :

```
## CONTINUOUS distributions

# Beta
    # Ppal Param : Beta(r, t, a, b)
    beta = Beta(2., 3., 0., 2.)
    # Param 1 : Beta(mu, sigma, a, b, 1)
    # Param 1 is coded by 1
    beta = Beta(2., 3., 0., 2., 1)
    # It is also possible to write :
    beta = Beta((2., 3., 0., 2., Beta.MUSIGMA)
    # Default construction ==> Beta(r, t, a, b)= Beta(2, 4, -1, 1)
    beta = Beta()
```

```

# Exponential
    # Ppal Param : Exponential(lambda, gamma)
    exponential = Exponential(1., 2.)
    # Default construction ==> Exponential(lambda, gamma) = Exponential(1.0,
        0.0)
    exponential = Exponential()

# Gamma
    # Ppal Param : Gamma(k, lambda, gamma)
    gamma = Gamma(3., 1., 2.)
    # Param 1 : Gamma(mu, sigma, gamma, 1)
    # Param 1 is coded by 1
    gamma = Gamma(3., 1., 2., 1)
    # It is also possible to write :
    gamma = Gamma(3., 1., 2.,Gamma.MUSIGMA)
    # Default construction ==> Gamma(k, lambda, gamma) = Gamma(1.0, 1.0, 0.0)
    gamma = Gamma()

# Gumbel
    # Ppal Param : Gumbel(alpha, beta)
    gumbel = Gumbel(1., 2.)
    # Param 1 : Gumbel(mu, sigma, 1)
    # Param 1 is coded by 1
    gumbel = Gumbel(1., 2., 1)
    # It is also possible to write :
    gumbel = Gumbel(1., 2.,Gumbel.MUSIGMA)
    # Default construction ==> Gumbel(alpha, beta) = Gumbel(1.0, 1.0)
    gumbel = Gumbel()

# Histogram
# Example : n = 3, x1 = 0.0 and
# (hi, li)_{i=1, ..., 3} = (1., 1.), (4., 2.), (2., 3.)
# The heights (hi) are automatically renormalized
    # Ppal Param : Histogram(x1, (hi, li)_{i=1, ..., n})
    collection = HistogramPairCollection(3)
    collection[0] = HistogramPair(1., 1.)
    collection[1] = HistogramPair(4., 2.)
    collection[2] = HistogramPair(2., 3.)
    histogram = Histogram(0., collection)

# Logistic
    # Ppal Param : (alpha, beta)
    logistic = Logistic(1., 2.)
    # Default construction ==> Logistic(alpha, beta) = Logistic(0.0, 1.0)
    logistic = Logistic()

# LogNormal
```

```
    # Ppal Param : LogNormal(mu_l, sigma_l,gamma)
    lognormal = LogNormal(1., 2., 1.)
    # Param 1 : LogNormal(mu, sigma, gamma, 1)
    # Param 1 is coded by 1
    lognormal = LogNormal(1., 2., 1., 1)
    # It is also possible to write :
    lognormal = LogNormal(1., 2., 1., LogNormal.MUSIGMA)
    # Param 2 : LogNormal(mu, sigma/mu, gamma, 2)
    # Param 2 is coded by 2
    lognormal = LogNormal(1., 2., 1., 2)
    # It is also possible to write :
    lognormal = LogNormal(1., 2., 1., LogNormal.MU_SIGMAOVERMU)
    # Default construction ==> LogNormal(mu_l, sigma_l,gamma) = LogNormal(0.0,
        1.0, 0.0)
    logNormal = LogNormal()

# Normal(1D)
   # Ppal Param : Normal(mu, sigma) = Normal(2.0, 1.0)
    normal1D = Normal(2.0, 1.0)
   # Default construction ==> 1D Normal distribution with zero mean and unit
       variance :
    normal1D_standard = Normal()

# Non Central Student
   # Ppal Param : NonCentralStudent(nu, delta, gamma) = NonCentralStudent(3.0,
       1.0, 0.0)
   nonCentralStudent = NonCentralStudent(3.0, 1.0, 0.0)
   # Default construction ==> NonCentralStudent(nu, delta, gamma) =
       NonCentralStudent(5.0, 0.0, 0.0)
   nonCentralStudent = NonCentralStudent()

# Normal (nD)
    # Ppal Param : Normal(mu, sigma, R)
    normal2D_1 = Normal(NumericalPoint(2, 1.), NumericalPoint(2, 2.),
        IdentityMatrix(2))
    # Ppal Param : Normal(mu, C)
    normal2D_2 = Normal(NumericalPoint(2, 1.), CovarianceMatrix(2))
    # 2D Normal distribution with zero mean and identity covariance matrix:
    normal2D_standard = Normal(2)

    # In order to create a Normal of dimension n
    # with 0 mean and Identity variance matrix
    normalStandardnD = Normal(n)

# Student
    # Param1 = Student(nu, mu)
    student = Student(3., 2.)
    # Default construction ==> Student(nu, mu) = Student(3.0, 0.0)
    student = Student()

# Triangular
    # Ppal Param = Triangular(a,m,b)
    triangular = Triangular(1., 2., 4.)
    # Default construction ==> Triangular(a, m, b) = Triangular(-1.0, 0.0, 1.0)
    triangular = Triangular()

# TruncatedNormal
    # Param1  = TruncatedNormal(mu_n, sigma_n, a, b)
    truncatednormal = TruncatedNormal(1., 2., -1., 5.)
    # Default construction ==> TruncatedNormal(mu_n, sigma_n, a, b) =
        TruncatedNormal(0.0, 1.0, -1.0, 1.0)
    TruncatedNormal = TruncatedNormal()


# Uniform
    # Param1 = Uniform(a,b)
    uniform = Uniform(1., 2.)
    # Default construction ==> Uniform(a,b) = Uniform(-1.0, 1.0)
    uniform = Uniform()

# Weibull
    # Param1 = Weibull(e, beta, gamma)
    weibull = Weibull(1., 2., 3.)
    # Param 1 = Weibull(mu, sigma, gamma, 1)
    # Param 1 is coded by 1
    weibull = Weibull(1., 2., 3.,1)
    # It is also possible to write :
    weibull = Weibull(1., 2., 3.,Weibull.MUSIGMA)
    # Default construction ==> Weibull(e, beta, gamma) = Weibull(1.0, 1.0, 0.0)
    weibull = Weibull()


## DISCRETE distributions

# Multinomial
    # Ppal Param : MultiNomial((p_i)_{i=1, ..., n}, N)
    distribution = MultiNomial(NumericalPoint(4, 0.25), 5)

# Geometric
    # Ppal Param : Geometric(p)
    geometric = Geometric(0.3)

# Poisson
    # Ppal Param : Poisson(lambda)
    poisson = Poisson(3)

```

```
# User defined (nD), n=2
    # We create a collection of pair (xi, pi), i=1,2,3, each xi in R^2
    collection = UserDefinedPairCollection(3, UserDefinedPair(NumericalPoint(2), 0.0))

    # First pair : (x1 = (1.0, 1.5), p1 = 0.30)
    x1 = NumericalPoint(2)
    x1[0] = 1.0
    x1[1] = 1.5
    collection[0] = UserDefinedPair(x1, 0.30)

    # Second pair : (x2 = (2.0, 2.5), p2 = 0.30)
    x2 = NumericalPoint(2)
    x2[0] = 2.0
    x2[1] = 2.5
    collection[1] = UserDefinedPair(x2, 0.30)

    # Third pair : (x3 = (3.0, 3.5), p3 = 0.40)
    x3 = NumericalPoint(2)
    x3[0] = 3.0
    x3[1] = 3.5
    collection[2] = UserDefinedPair(x3, 0.40)

    # Create the UserDefined distribution
    distribution = UserDefined(collection)
```

The pdf of the usual distributions are drawned in Figures 1to 23.

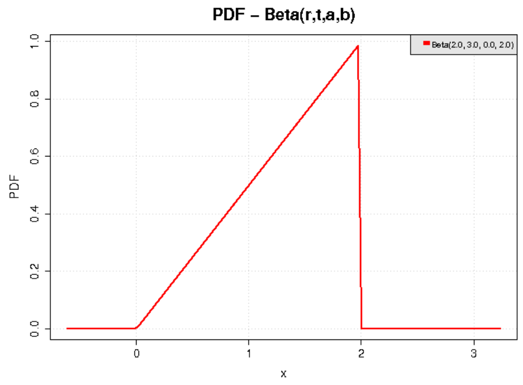


Figure 1: PDF of a distribution.

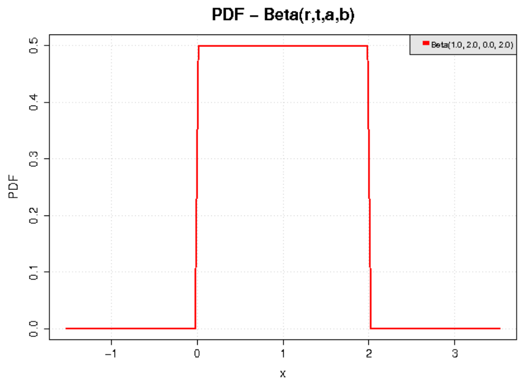


Figure 2: PDF of a Beta distribution.

The Histogram distribution explicited in the Use Case is drawn in Figures 24 and 25.

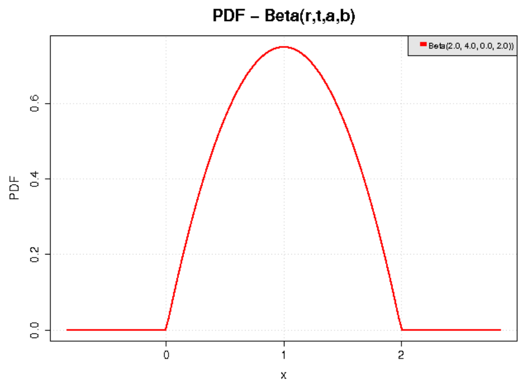


Figure 3: PDF of a Beta distribution.

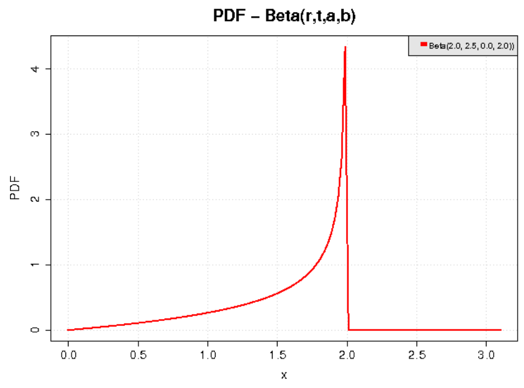


Figure 4: PDF of a Beta distribution.

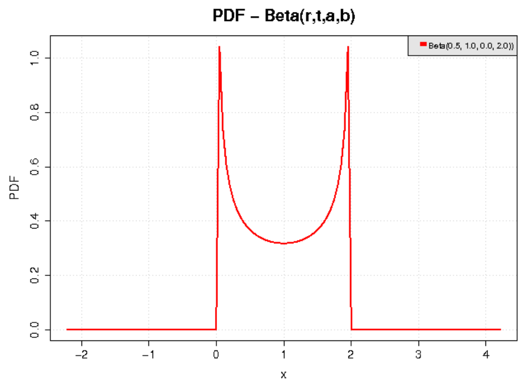


Figure 5: PDF of a Beta distribution.

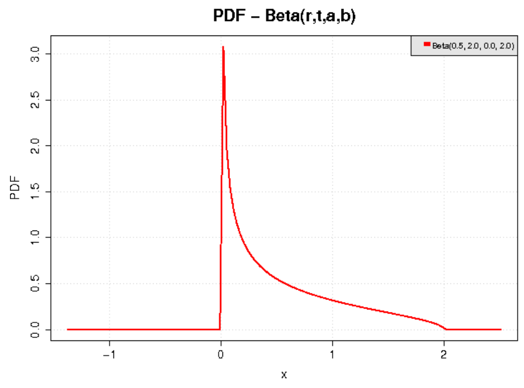


Figure 6: PDF of a Beta distribution.

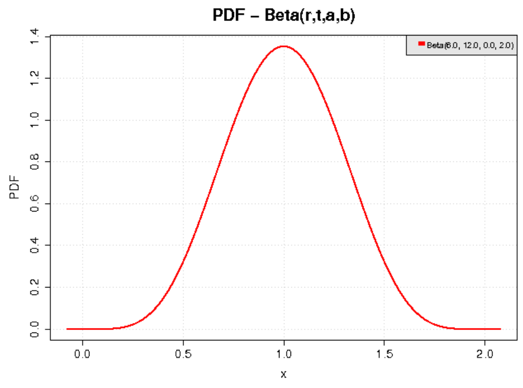


Figure 7: PDF of a Beta distribution.

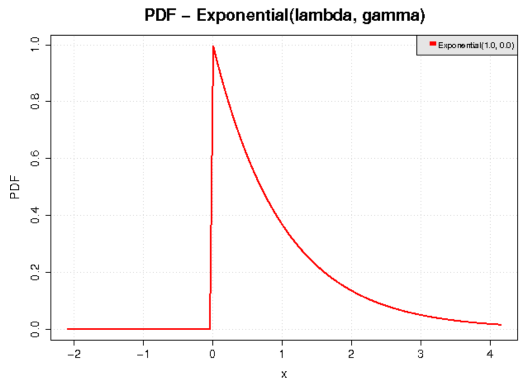


Figure 8: PDF of a Exponential distribution.

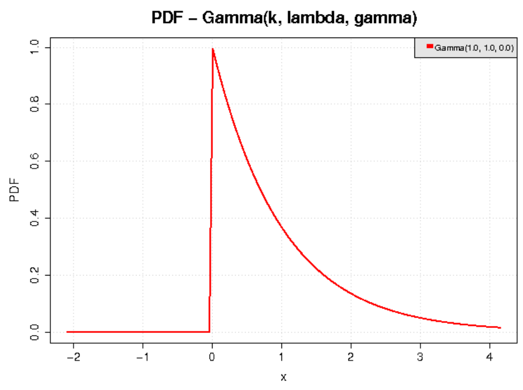


Figure 9: PDF of a Gamma distribution.

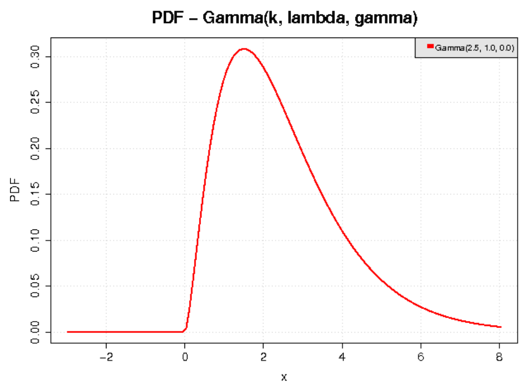


Figure 10: PDF of a Gamma distribution.

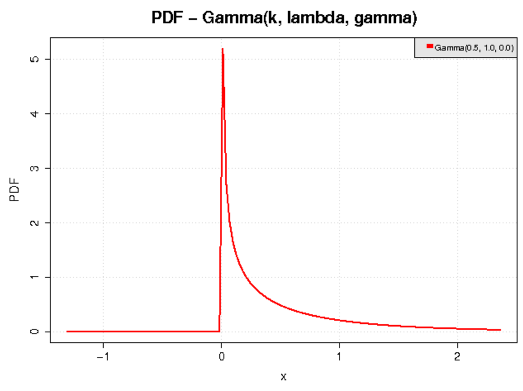


Figure 11: PDF of a Gamma distribution.

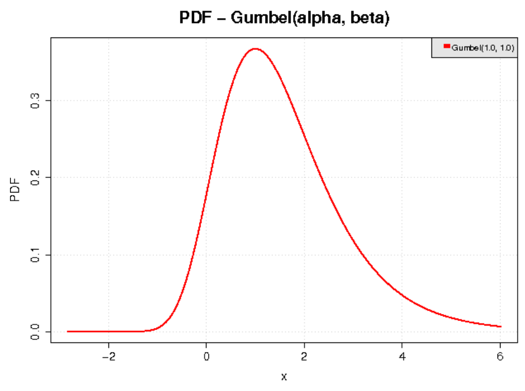


Figure 12: PDF of a Gumbel distribution.

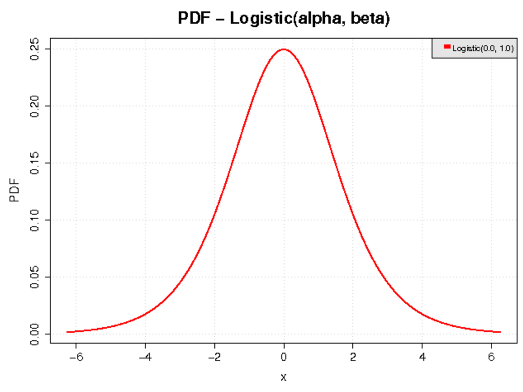


Figure 13: PDF of a logistic distribution.

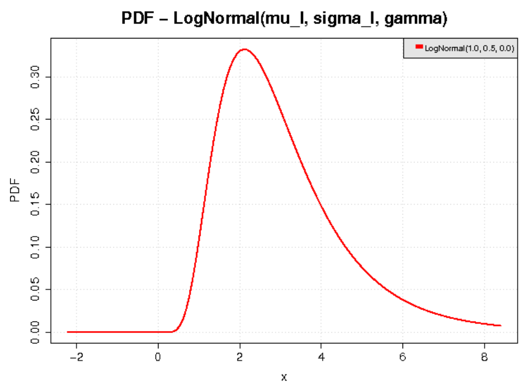


Figure 14: PDF of a LogNormal distribution.

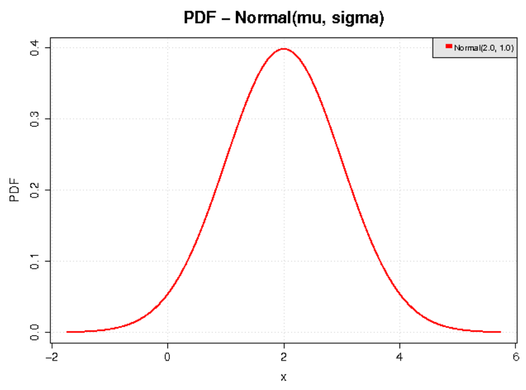


Figure 15: PDF of a Normal distribution.

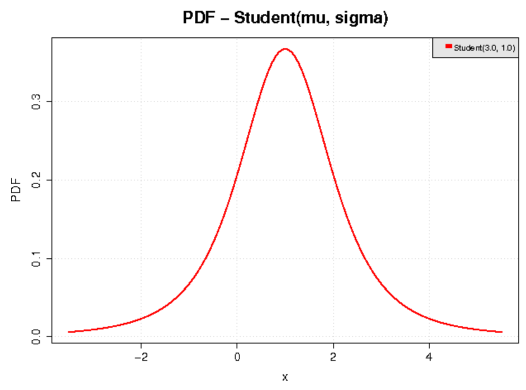


Figure 16: PDF of a Student distribution.

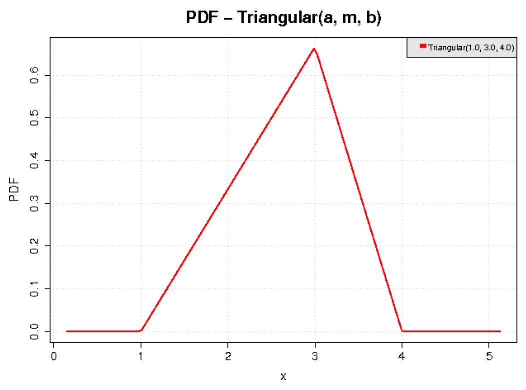


Figure 17: PDF of a Triangular distribution.

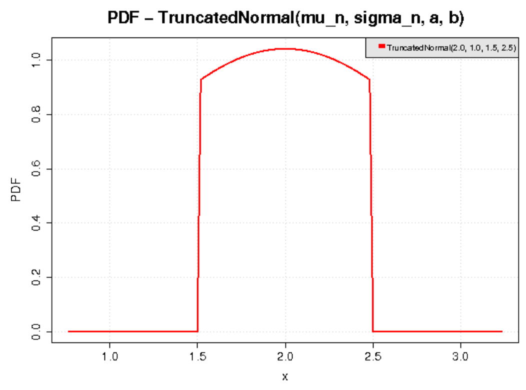


Figure 18: PDF of a TruncatedNormal distribution.

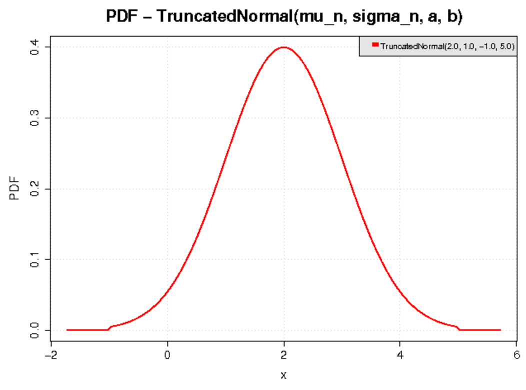


Figure 19: PDF of a TruncatedNormaldistribution.

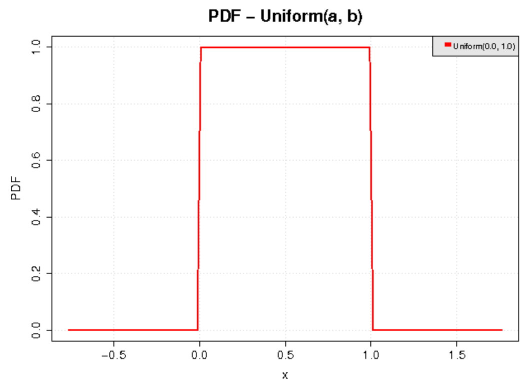


Figure 20: PDF of a Uniform distribution.

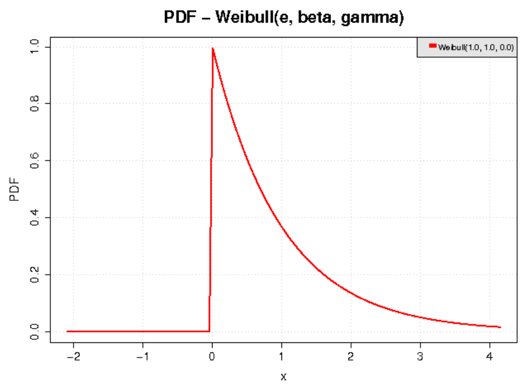


Figure 21: PDF of a Weibull distribution.

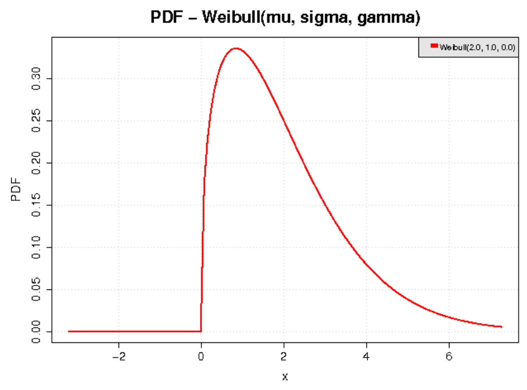


Figure 22: PDF of a Weibull distribution.

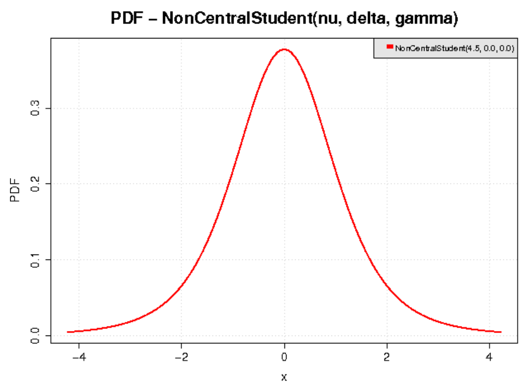


Figure 23: PDF of a Non Central Student distribution.

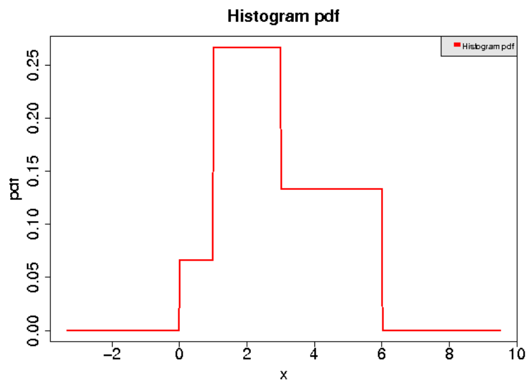


Figure 24: PDF of an Histogram distribution

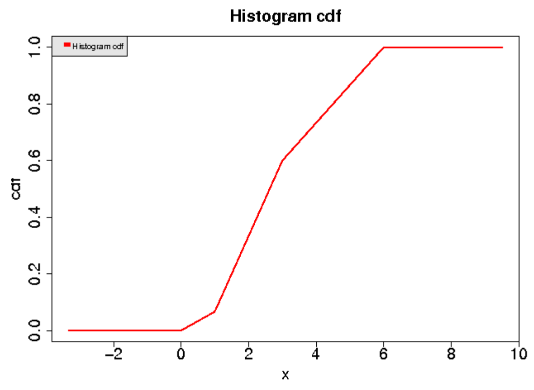


Figure 25: CDF of an Histogram distribution

1.1.2 UC : Creation of a truncated distribution

The objective of the US is to truncate a 1D distribution already defined. Open TURNS enables to truncate the distribution in its lower area, or its upper area or in both lower and upper areas. After having truncated a distribution, it is possible to recuperate the initial distribution thanks to the method *getDistribution()*.

Let's consider X a random variable whith respectively F_X and p_X its cumulative and probability density functions, and $(a, b) \in \mathbb{R} \cup \pm\infty$. The random variable $Y = X/[a, b]$ which is the random variable X given that $X \in [a, b]$ is defined by the following cumulative and probability density functions F_Y and p_Y :

$$\forall y \in \mathbb{R}, F_Y(y) = Prob(X < y \,/\, X \in [a, b]) = \left| \begin{array}{ll} 1 & \text{for } y \geq b, \\ 0 & \text{for } y \leq a, \\ \dfrac{F_X(y) - F_X(a)}{F_X(b) - F_X(a)} & \text{for } y \in [a, b] \end{array} \right.$$

$$\forall y \in \mathbb{R}, p_Y(y) = \left| \begin{array}{ll} 0 & \text{for } y \geq b \text{ or } y \leq a \\ \dfrac{1}{F_X(b) - F_X(a)} p_X(y) & \text{for } y \in [a, b] \end{array} \right.$$

Requirements	• some lower and upper bounds : *myLowerBound, myUpperBound* **type** : reals • a 1D distribution : *myEntireDistribution* **type** : a Distribution which implementation is UsualDistribution or ComposedDistribution or Mixture
Results	• a distribution : *myTruncatedDistribution* **type** : a TruncatedDistribution

Python script for this UseCase :

```

# CASE 1 : Truncate the distribution whithin the range $[myLowerBound, myUpperBound]$
  myTruncatedDistribution = TruncatedDistribution(Distribution(
      myEntireDistribution), myLowerBound, myUpperBound)


# CASE 2 : Truncate the distribution whithin the range $[myLowerBound, \infty[$
    or $[myLowerBound, max[$ if
# myEntireDistribution was already bounded by $max$
  myTruncatedDistribution = TruncatedDistribution(Distribution(
      myEntireDistribution), myLowerBound, TruncatedDistribution.LOWER)


```

```
# CASE 3 : Truncate the distribution whithin the range $[-\infty, myUpperBound[$
    or $[min, myUpperBound[$ if
# myEntireDistribution was already bounded by $min$
  myTruncatedDistribution = TruncatedDistribution(Distribution(
      myEntireDistribution), myUpperBound, TruncatedDistribution.UPPER)

# Recuperate the initial distribution
  initialDistribution = myTruncatedDistribution.getDistribution()
```

Figures 26 and 27 show the PDF and CDF of the truncated distributions of a Logistic($\alpha = 1.0$, $\beta = 2.0$) respectively within the ranges $[4.0, \infty[$, $[-2.0, 5.0]$ and $[-\infty, 3.0]$.

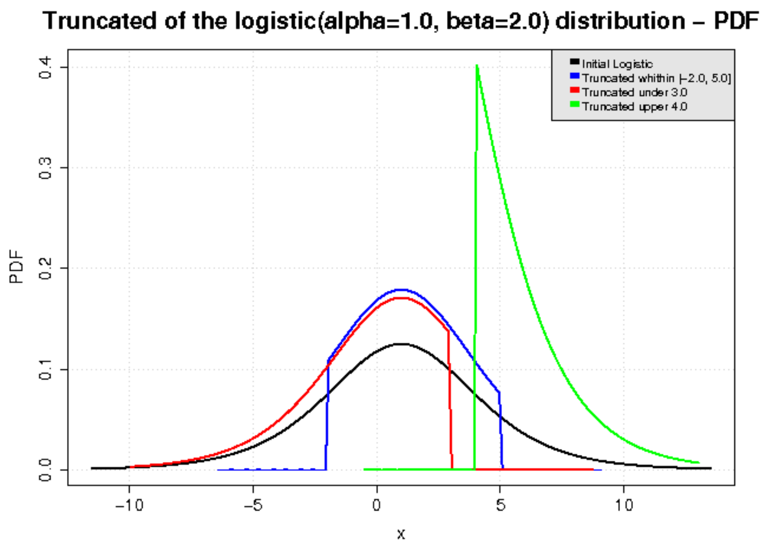


Figure 26: PDF of several truncated Logistic distributions

1.1.3 UC : Creation of a copula and a composed copula

The objective of this Use Case is to manipulate copulas of Open TURNS.

A copula may be considered as the restriction to $[0, 1]^n$ of a distribution with uniform 1D marginals on $[0, 1]$ and this copula as copula. That's why an object of type *Copula* offers the same methods as an object of type *Distribution* (see U.C. 1.1.6 to have the list of the methods).
Table. 1 gives the expression of bidimensional copulas proposed by Open TURNS.

Furthermore, Open TURNS enables to create some copula as the product of other copulas : if C_1 and C_2 are two copulas respectively of random vectors in $\mathbb{R}^{n_1}$ and $\mathbb{R}^{n_2}$, we can create the copula of a random vector of $\mathbb{R}^{n_1+n_2}$, noted C as follows :

$$C(u_1, \cdots, u_n) = C_1(u_1, \cdots, u_{n_1}) C_2(u_{n_1+1}, \cdots, u_{n_1+n_2})$$

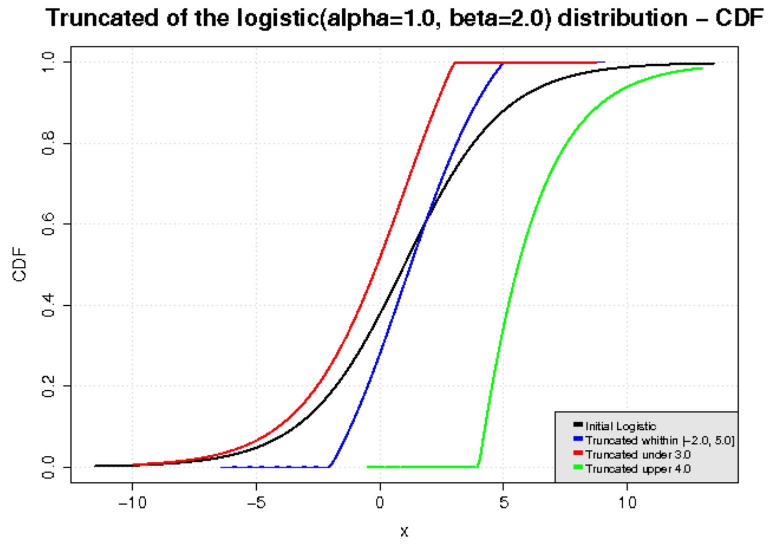


Figure 27: CDF of several truncated Logistic distributions

Name	Dimension	$C(u_1, \cdots, u_n)$	Parameters
Independent	n	$\prod_{i=1}^{i=n} u_i$	n
Normal	2	$\int_{-\infty}^{\Phi^{-1}(u_1)} \int_{-\infty}^{\Phi^{-1}(u_2)} \frac{1}{2\pi\sqrt{1-\rho^2}} \exp\left(-\frac{s^2-2\rho st+t^2}{2(1-\rho^2)}\right) \mathrm{d}s\,\mathrm{d}t$	$\underline{\underline{R}} = \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix}$ $\rho \in [-1, 1]$
Normal	n	$\int_{-\infty}^{\Phi^{-1}(u_1)} \cdots \int_{-\infty}^{\Phi^{-1}(u_n)} \frac{1}{(2\pi)^{n/2}\sqrt{\det(\underline{\underline{R}})}} \exp\left(-\frac{1}{2}\underline{x}^t \underline{\underline{R}}^{-1} \underline{x}\right) \mathrm{d}\underline{x}$	$\underline{\underline{R}}$, SDP
Frank	2	$-\frac{1}{\theta}\log\left(1+\frac{(e^{-\theta u_1}-1)(e^{-\theta u_2}-1)}{e^{-\theta}-1}\right)$	$\theta \neq 0$
Clayton	2	$\left(u_1^{-\theta}+u_2^{-\theta}-1\right)^{-1/\theta}$	$\theta \geq 0$
Gumbel	2	$\exp\left(-\left((-\log(u_1))^\theta+(-\log(u_2))^\theta\right)^{1/\theta}\right)$	$\theta \geq 1$

Table 1: Expressions of the copulas of Open TURNS.

It means that both subvectors $(u_1, \cdots, u_{n_1})$ and $(u_{n_1+1}, \cdots, u_{n_1+n_2})$ of $\mathbb{R}^{n_1}$ and $\mathbb{R}^{n_2}$ are independent.

Requirements	none
Results	• a Normal, Clayton, Gumbel, Frank and Independent copulas : *normalCopula, claytonCopula, gumbelCopula, frankCopula, independentCopula* **type** : NormalCopula, ClaytonCopula, GumbelCopula, FrankCopula, IndependentCopula • a composed copula : *finalCopula* **type** : ComposedCopula

Python script for this UseCase :

```

# INDEPENDENT copula

# Independent Copula parametered by its dimension
    # For example, dimension = 3
    dim = 3
    independentCopula = IndependentCopula(dim)


# NORMAL copula

# Case 1 :  Normal Copula parametered by its correlation matrix R

    # For example, dimension = 3 and R :
    dim = 3
    R =  CorrelationMatrix(dim)
    for i in range(dim-1) :
        R[i, i + 1] = 0.8

    # Create a normal copula from the correlation matrix R
    normalCopula = NormalCopula(R)
    normalCopula.setName("a␣normal␣copula")


# Case 2 : Create a normal copula from the Spearman rank correlation matrix S

    # For example, dimension = 3 and S :
    dim = 3
    S = CorrelationMatrix(dim)
    for i in range(1,dim) :
        S[i, i - 1] = 0.25

    # Create the correlation matrix R of the  normal copula
    # from the Spearman correlation matrix S
    R = NormalCopula.getNormalCorrelationFromSpearmanCorrelation(S)

    # Create the normal copula from the R correlation matrix
    normalCopula = NormalCopula(R)
    normalCopula.setName("another normal copula")

# Case 3 : Normal Copula parametered by its dimension

    # Correlation matrix R is equal to identity
    dim = 3
    normalCopula = NormalCopula(dim)


# CLAYTON copula

    # Only for dimension = 2
    # Clayton copula is parametered by theta whithout restriction
    # For example, theta = -2.5
    theta = -2.5
    claytonCopula = ClaytonCopula(theta)


# GUMBEL copula

    # Only for dimension = 2
    # Gumbel copula is parametered by theta whithout restriction
    # For example, theta = 2.5
    theta = 2.5
    gumbelCopula = ClaytonCopula(theta)


# FRANK copula

    # Only for dimension = 2
    # Frank copula is parametered by theta whithout restriction
    # For example, theta = 9.2
    theta = 9.2
    frankCopula = FrankCopula(theta)

# COMPOSED copula

    # For example, the GumbelCopula concatenated to a Clayton one
    # Create the collection of copulas
    copulaColl = CopulaCollection(2)
    copulaColl[0] = Copula(gumbelCopula)
    copulaColl[1] = Copula(claytonCopula)

    # Create the composed copula in R^4
    finalCopula = ComposedCopula(copulaColl)
```

We draw in Figures 28 to 32 the iso-curves of the PDF respectively of some copulas of type : independent, Normal, Clayton, Gumbel, Frank.

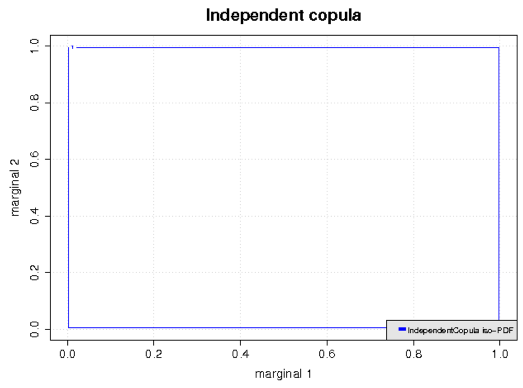


Figure 28: Iso-PDF of an independent copula.

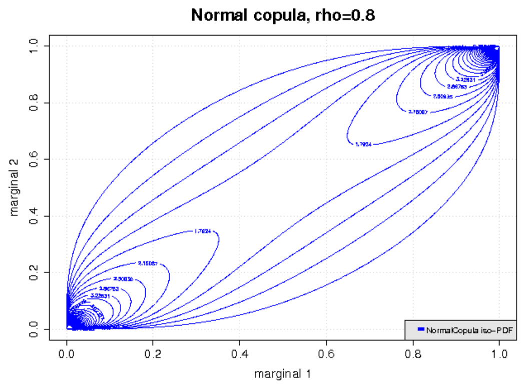


Figure 29: Iso-PDF of a Normal copula.

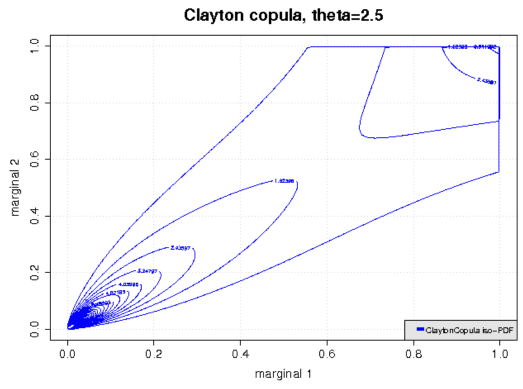


Figure 30: Iso-PDF of a Clayton copula.

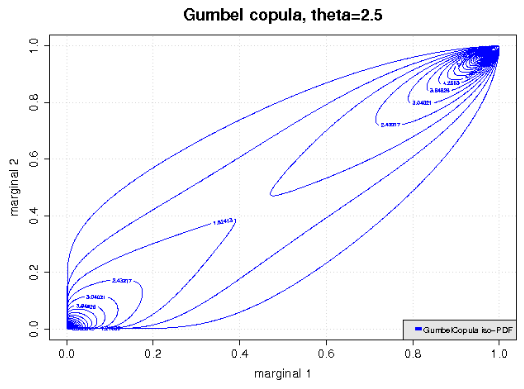


Figure 31: Iso-PDF of a Gumbel copula.

1.1.4 UC : Creation of nD distribution from (marginals, copula)

The objective of the US is to model a distribution, described by its marginal distributions and its dependence structure (a particular copula). This UC is particularly adapted to the modelisation of the distribution of the input random vector.

The example here is a distribution of dimension 3 defined by :

- Beta, Triangular and Uniform marginals,
- an independent copula.

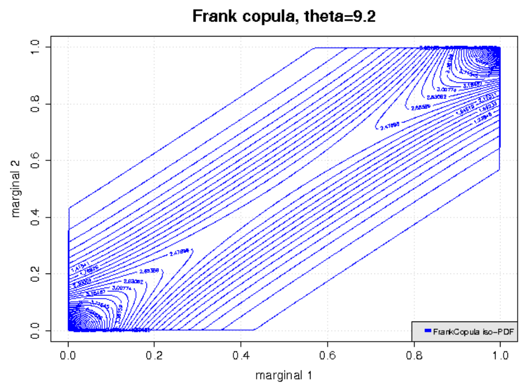


Figure 32: Iso-PDF of a Frank copula.

Requirements	none
Results	• a nD distribution : *myDistribution* **type** : Distribution which implementation is a ComposedDistribution

Python script for this UseCase :

```
# Create a collection of distribution of dimension 3
    aCollection = DistributionCollection(3)

# Create the first marginal : Weibul(mu, sigma, gamma) = Weibull(2.0, 1.0, 0.0)
    weibDist = Weibull(2.0, 1.0, 0.0, Weibull.MUSIGMA)
    weibDist.setName("First␣Marginal␣:␣Weibull")
    aCollection[0] = Distribution(weibDist)

# Create the second marginal : Triangular(a,m,b) = Triangular(1.0, 3.0, 5.0)
    triangularDist = Triangular(1.0, 3.0, 5.0)
    triangularDist.setName("Second␣Marginal␣:␣Triangular")
    aCollection[1] = Distribution(triangularDist)

# Create the third marginal : Uniform(a,b) = Uniform(2.0, 4.0)
    uniformDist = Uniform(2.0, 4.0)
    uniformDist.setName("Third␣Marginal␣:␣Uniform")
    aCollection[2] = Distribution(uniformDist)

# Create a copula : Normal copula of dimension 3 fom Spearman rank correlation
   matrix
    spearmanMatrix = CorrelationMatrix(3)
    spearmanMatrix[0,1] = 0.25
    spearmanMatrix[1,2] = 0.25
    aCopula = NormalCopula(NormalCopula.
        GetNormalCorrelationFromSpearmanCorrelation(spearmanMatrix))
    aCopula.setName("Normal copula")

# Instanciate one distribution object
    myDistribution = ComposedDistribution(aCollection, Copula(aCopula))

# Give a Description to the Distribution
    aDescription = Description(3)
    aDescription[0] = "X1 distribution"
    aDescription[1] = "X2 distribution"
    aDescription[2] = "X3 distribution"
    myDistribution.setDescription(aDescription)
```

We draw in Figures 33 to 35 the iso-curves of each 2D distribution defined by two of the three components of the distribution.

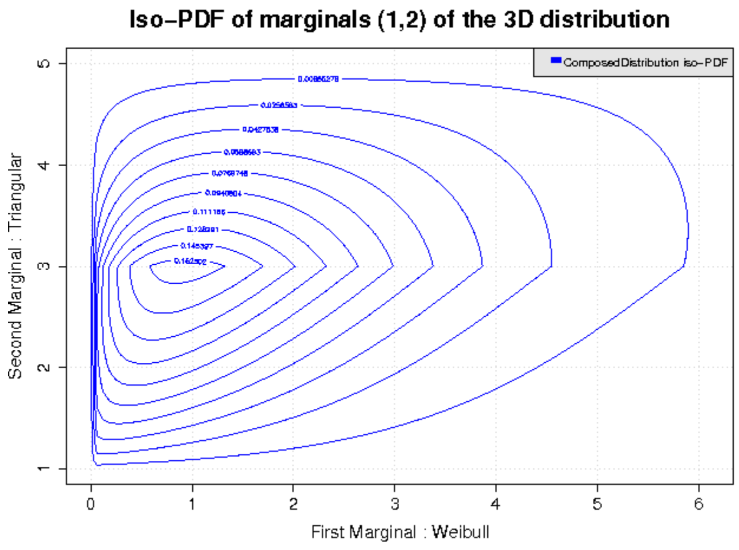


Figure 33: Iso-PDF of the distribution defined by the marginals 1 and 2.

1.1.5 UC : Creation of a nD distribution from a Mixture

In Open TURNS, a Mixture is a distribution which probability density function is a linear combination of probability density functions.

The objective of the US is to model a distribution, defined as a mixture. This UC is particularly adapted to the modelisation of the distribution of the input random vector.

The example here is a mixture of three 1D distributions Triangular(1.0, 2.0, 4.0), Normal(-1.0, 1.0) and Uniform(5.0, 6.0), with respective weights : (0.2, 0.3, 0.5).
The PDF and CDF graphs the mixture distribution are drawn in Figures 36 and 37.

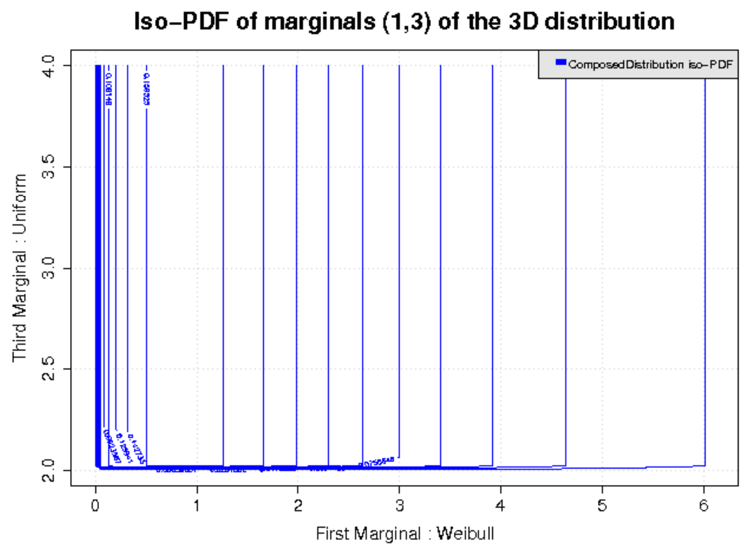


Figure 34: Iso-PDF of the distribution defined by the marginals 1 and 3.

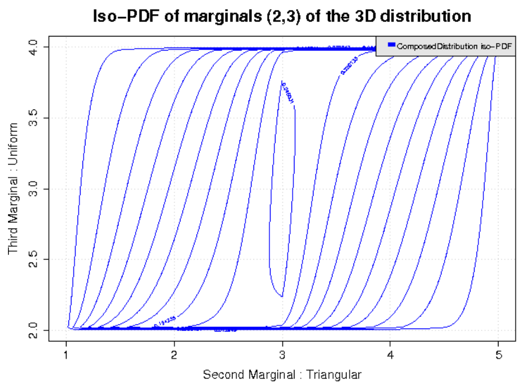


Figure 35: Iso-PDF of the distribution defined by the marginals 2 and 3.

Requirements	none
Results	• a mixture distribution : *myMixture* **type** : Mixture • a random input vector : *input* **type** : RandomVector wich implementation is a UsualRandomVector

Python script for this UseCase :

```

# Create the three distributions :
    # Triangular(1.0, 2.0, 4.0)
    triang = Triangular(1.0, 2.0, 4.0)
    # Normal(-1.0, 1.0)
    norm = Normal(-1.0, 1.0)
    # Uniform(5.0, 6.0)
    unif = Uniform(5.0,6.0)

# Create a collection of distribution
    aCollection = DistributionCollection(3)
    aCollection[0] = Distribution(triang)
    aCollection[1] = Distribution(norm)
    aCollection[2] = Distribution(unif)

# Put weight to each distribution
# CARE : these weights must be in [0,1]
# If not normalised (ie sum = 1.0), weights are modified to have sum = 1.0
    # Weight of the Triangular distribution in [0,1]
    aCollection[0].setWeight(0.20)

    # Weight of the Normal distribution in [0,1]
    aCollection[1].setWeight(0.50)

    # Weight of the Weibull distribution in [0,1]
    aCollection[2].setWeight(0.30)

# Instanciate one distribution object
    myMixture = Mixture(aCollection)

# Draw the PDF and CDF of this distribution
    # Impose a x-range
    myMixture_pdf = myMixture.drawPDF(-3.0,7.0)
    myMixture_pdf.setLegendPosition("topleft")

    myMixture_cdf = myMixture.drawCDF(-3.0,7.0)

    # Or impose a bounding box : x-range and y-range
    # boundingBox = [xmin, xmax, ymin, ymax]
    myBoundingBox = NumericalPoint(4)
    myBoundingBox[0] = xmin
    myBoundingBox[1] = xmax
    myBoundingBox[2] = ymin
    myBoundingBox[3] = xmax
    myMixture_cdf.setBoundingBox(myBoundingBox)

    # In order to see the graphs whithout creating the files .EPS, .PNG and .FIG
    Show(myMixture_pdf)
    Show(myMixture_cdf)

    # Create the files .EPS, .PNG and .FIG
    myMixture_pdf.draw("pdf_Mixture")
    myMixture_cdf.draw("cdf_Mixture")

    # Visualize the file .PNG wihthin the TUI
    ViewImage(myMixture_pdf.getBitmap())
    ViewImage(myMixture_cdf.getBitmap())
```

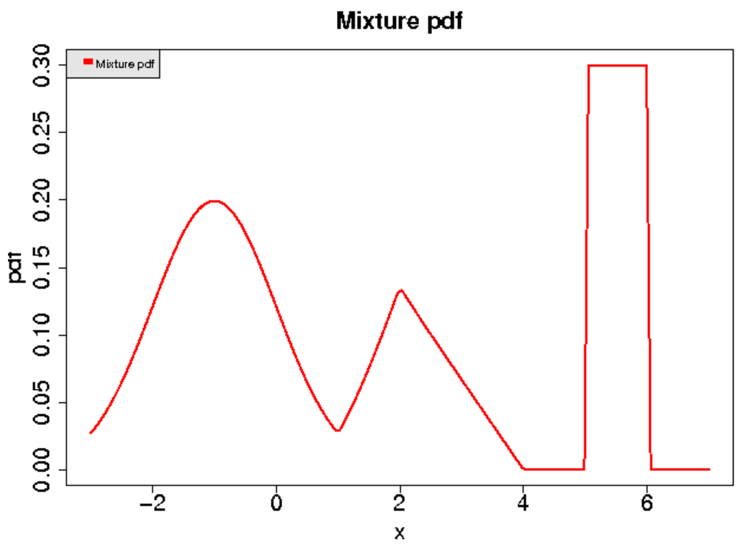


Figure 36: PDF of the Mixture distribution = 0.2*Triangular(1.0, 2.0, 4.0) + 0.5*Normal(-1.0, 1.0) + 0.3*Uniform(5.0, 6.0)

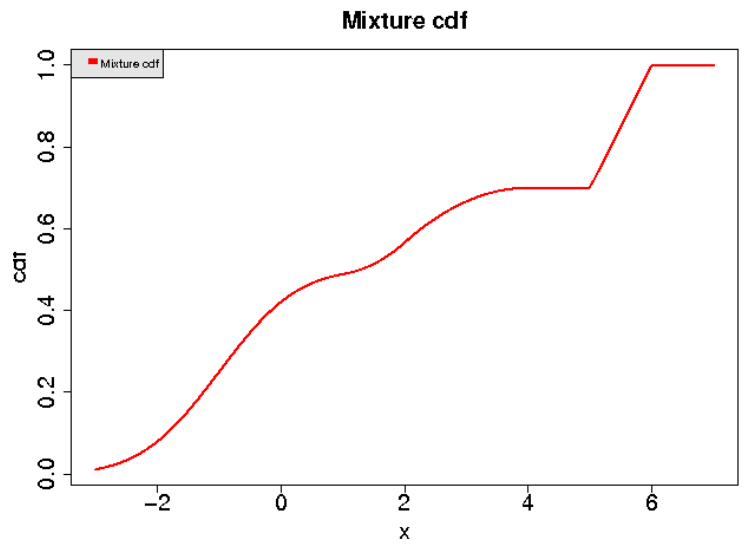


Figure 37: CDF of the Mixture distribution = 0.2*Triangular(1.0, 2.0, 4.0) + 0.5*Normal(-1.0, 1.0) + 0.3*Uniform(5.0, 6.0)

1.1.6 UC : Manipulation of a distribution

The objective of this UC is to describe the main functionalities that Open TURNS enables to manipulate a distribution of dimension $n \geq 1$.
Let's note $\underline{X} = (X_1, \cdots, X_n)$ the random vector associated to that distribution, which PDF is note p. Open TURNS enables :

- to ask for the dimension, with the method *getDimension*,
- if $n > 1$, to extract the extracted distribution of dimension $k < n$ corresponding to k 1D marginals, with the method *getMarginal*,
- to get the copula, with the method *getCopula*, only for the types UsualDistribution and ComposedDistribution (defined from the 1D marginals and a copula),
- to ask for some properties on the copula, with the method *hasIndependentCopula, hasEllipticalCopula*, only for the types Usual Distribution and ComposedDistribution (defined from the 1D marginals and a copula),
- to evaluate the mean vector (potentially of dimension 1), the covariance matrix (potentially of dimension 1×1), the standard deviation, skewness and kurtosis vectors (potentially of dimension 1), with the methods *getMean, getStandardDeviation, getCovariance, getKurtosis, getSkewness*, defined by the following

expressions :

$$\left\{\begin{array}{lcl} \underline{E}[\underline{X}] & = & (E[X_1], \cdots, E[X_n]) \\ \underline{StdDev}[\underline{X}] & = & (\sqrt{E[(X_1 - E[X_1])^2]}, \cdots, \sqrt{E[(X_n - E[X_n])^2]}) \\ \underline{\underline{Cov}}[\underline{X}] & = & (E\left[(X_i - E[X_i])(X_j - E[X_j])\right])_{i,j} \\ \underline{skewness}[\underline{X}] & = & (E\left[\left(\frac{(X_1 - E[X_1])}{\sqrt{Var[X_1]}}\right)^3\right], \cdots, E\left[\left(\frac{(X_n - E[X_n])}{\sqrt{Var[X_n]}}\right)^3\right]) \\ \underline{kurtosis}[\underline{X}] & = & (E\left[\left(\frac{(X_1 - E[X_1])}{\sqrt{Var[X_1]}}\right)^4\right], \cdots, E\left[\left(\frac{(X_n - E[X_n])}{\sqrt{Var[X_n]}}\right)^4\right]) \end{array}\right.$$

- to evaluate the roughness, with the method *getRoughness*, defined by :

$$roughness(\underline{X}) = ||p||_{\mathcal{L}^2} = \sqrt{\int_{\underline{x}} p^2(\underline{x}) d\underline{x}}$$

- to get once the distribution or simultaneously n realisations, with the method *getRealization, getNumericalSample*,
- to evaluate the Cumulative Density Function (CDF) or the Probability Density Function (PDF) at a point, with the method *computeCDF, computePDF*,
- to evaluate a quantile, with the method *computeQuantile*,
- to evaluate the derivative of the CDF or PDF with respect to the parameters of the distribution at a particular point, with the methods *computeCDFGradient, computePDFGradient*,
- to draw :
  - for a 1D distribution : the PDF and CDF curves, with the methods *drawPDF, drawCDF*,
  - for a 2D distribution : the PDF and CDF iso-curves, with the methods *drawPDF, drawCDF*, and the PDF and CDF curves of its 1D marginals, with the methods *drawMarginal1DPDF, drawMarginal1DCDF* ,
  - for a nD with $n \geq 3$ distribution : the PDF and CDF of each 1D marginal, with the methods *drawMarginal1DPDF, drawMarginal1DCDF* and the PDF and CDF iso-curves for a specified 2D marginal, with the methods *drawMarginal2DPDF, drawMarginal2DCDF.*

Let's note that it is possible to visualise a graph hithin the TUI whithout creating the .EPS, .PNG or .FIG files, thanks to the command *Show.*

Requirements	• one distribution : *dist* **type** : Distribution
Results	none

Python script for this UseCase :

```

# Get the dimension
  dim = dist.getDimension()
  print "Dimension of the distribution = ", dim

# Get the marginals
  # the i-th marginal
  # Care : the numerotation begins at 0
  marginal_i = dist.getMarginal(i)

  # the marginal of the sub-distribution defined by several components
  # Put the indices of the concerned components together
  # for example, the three first components  (if dimension >2)
  indices = Indices(3)
  indices[0] = 0
  indices[1] = 1
  indices[2] = 2
  3Dmarginal_123 = dist.getMarginal(indices)

# Get the copula
  # CARE : only for a ComposedDistribution
  copula = dist.getCopula()

# Ask some properties on the copula
 print "hasIndependentCopula", dist.hasIndependentCopula
 print "hasEllipticalCopula", dist.hasEllipticalCopula

# Get the mean vector of the distribution
  meanVector = dist.getMean()

# Get the covariance matrix of the distribution
  meanVector = dist.getCovariance()

# Get the kurtosis vector of the distribution
  kurtosisVector = dist.getKurtosis()

# Get the standard deviation vector of the distribution
  standardDeviationVector = dist.getStandardDeviation()

# Get the skewness vector of the distribution
  skewnessVector = dist.getSkewness()

# Get the roughness of the distribution
  roughness = dist.getRoughness()

# Get one realisation of the distribution
  oneRealisationVector = dist.getRealization()
```

```

# Get several realisations of the distribution
  # For example, 100 ones
  100_realisations = dist.getNumericalSample(100)

# Evaluate the CDF and PDF
  # CARE : if the dimension is 1
  # For example, at pointValue=2.3
  pointValue = 2.3
  CDF_value = dist.computeCDF(pointValue)
  PDF_value = dist.computePDF(pointValue)

  # CARE : if the dimension is >1
  # For example, with dimension 2, at pointVector=(2.3, 4.5)
  pointVector = NumeriaclPoint(2)
  pointVector[0] = 2.3
  pointVector[1] = 4.5
  CDF_vector = dist.computeCDF(pointVector)
  PDF_vector = dist.computePDF(pointVector)

# Evaluate the quantile of order p
  # For example, the quantile 90%
  quantile_Vector_90 = dist.computeQuantile(0.90)

# Evaluate the derivatives of the PDF/CDF with respect to the parameters at a
    particular point
  # For example, with dimension 2, at pointVector=(2.3, 4.5)
  derivatives_PDF_Vector = dist.computePDFGradient(pointVector)
  derivatives_CDF_Vector = dist.computeCDFGradient(pointVector)


# GRAPH 1 : Draw the PDF and CDF for a distribution of dimension 1

  # No specification of support
  PDF_1D_graph = dist.drawPDF()

  # Or Specify the support a and b (two scalars)
  # For example, a=-10.0 and b=10.0
  a=-10.0
  b=10.0
  PDF_1D_graph = dist.drawPDF(a,b)
  CDF_1D_graph = dist.drawCDF(a,b)

  # Or impose a bounding box : x-range and y-range
  # boundingBox = [xmin, xmax, ymin, ymax]
  myBoundingBox = NumericalPoint(4)
  myBoundingBox[0] = xmin
  myBoundingBox[1] = xmax
```

```
    myBoundingBox[2] = ymin
    myBoundingBox[3] = xmax
    PDF_1D_graph.setBoundingBox(myBoundingBox)

    # In order to see the graph whithout creating the associated files
    Show(PDF_1D_graph)
    Show(CDF_1D_graph)

    # Create the files corresponding to the graph
    # the files .EPS, .PNG and .FIG are created in the current python session
    PDF_1D_graph.draw("PDF_graph")
    CDF_1D_graph.draw("CDF_graph")

    # Or only the .EPS file
    # 640 and 480 are the pixels number in both axes
    PDF_1D_graph.draw("PDF_graph", 640, 480, GraphImplementation.EPS)
    CDF_1D_graph.draw("CDF_graph", 640, 480, GraphImplementation.EPS)

    # Visualize the PNG file whithin the TUI
    ViewImage(PDF_1D_graph.getBitmap())
    ViewImage(CDF_1D_graph.getBitmap())


# GRAPH 2 :Draw the PDF and CDF iso-curves for a distribution of dimension 2

    # No specification of support
    PDF_graph = dist.drawPDF()
    CDF_graph = dist.drawCDF()

    # Or Specify the support pointMin and pointMax
    # the graph will be drawned in the box with low-left corner : pointMin
    # and up-right corner : pointMax
    # For example, pointMin=(-3.0, -2.0) and pointMax=(4.0, 5.0)
    pointMin = NumericalPoint(2)
    pointMin[0] = -3.0
    pointMin[1] = -2.0
    pointMax = NumericalPoint(2)
    pointMax[0] = 4.0
    pointMax[1] = 5.0
    PDF_graph = dist.drawPDF(pointMin, pointMax)
    CDF_graph = dist.drawCDF(pointMin, pointMax)

    # Or impose a bounding box : x-range and y-range
    # boundingBox = [xmin, xmax, ymin, ymax]
    myBoundingBox = NumericalPoint(4)
    myBoundingBox[0] = xmin
    myBoundingBox[1] = xmax
    myBoundingBox[2] = ymin
```

```
  myBoundingBox[3] = xmax
  PDF_graph.setBoundingBox(myBoundingBox)

  # In order to see the graph whithout creating the associated files
  Show(PDF_graph)
  Show(CDF_graph)

  # Create the files corresponding to the graph
  # the files .EPS, .PNG and .FIG are created in the current python session
  PDF_graph.draw("PDF_graph")
  CDF_graph.draw("CDF_graph")

  # Or only the .EPS file
  # 640 and 480 are the pixels number in both axes
  PDF_graph.draw("PDF_isocurves_graph", 640, 480, GraphImplementation.EPS)
  CDF_graph.draw("CDF_isocurves_graph", 640, 480, GraphImplementation.EPS)

  # Visualize the PNG file in the TUI
  ViewImage(PDF_graph.getBitmap())
  ViewImage(CDF_graph.getBitmap())


# GRAPH 3 :Draw the PDF and CDF of the 1D marginals for a distribution of
    dimension >=2

  # For example, marginal i
  # Care : the numerotation begins at 0

  # Specify the support a and b (two scalars) and the number of points of the
      curve
  # For example, a=-10.0 and b=10.0
  a = -10.0
  b = 10.0
  pointnumber = 101
  PDF_graph = dist.drawMarginal1DPDF(i, a, b, pointnumber)
  CDF_graph = dist.drawMarginal1DCDF(i, a, b, pointnumber)

  # Or impose a bounding box : x-range and y-range
  # boundingBox = [xmin, xmax, ymin, ymax]
  myBoundingBox = NumericalPoint(4)
  myBoundingBox[0] = xmin
  myBoundingBox[1] = xmax
  myBoundingBox[2] = ymin
  myBoundingBox[3] = xmax
  PDF_graph.setBoundingBox(myBoundingBox)

  # In order to see the graph whithout creating the associated files
  Show(PDF_graph)
```

```
    Show(CDF_graph)

    # Create the files corresponding to the graph
    # the files .EPS, .PNG and .FIG are created in the current python session
    PDF_graph.draw("PDF_graph")
    CDF_graph.draw("CDF_graph")

    # Or only the .EPS file
    # 640 and 480 are the pixels number in both axes
    PDF_graph.draw("PDF_1DMarginals_graph", 640, 480, GraphImplementation.EPS)
    CDF_graph.draw("CDF_1DMarginals_graph", 640, 480, GraphImplementation.EPS)

    # Visualize the PNG file in the TUI
    ViewImage(PDF_graph.getBitmap())
    ViewImage(CDF_graph.getBitmap())


# GRAPH 4 :Draw the PDF and CDF iso-curves for a distribution of dimension n>2

    # For example, the marginals i and j
    # Care : the numerotation begins at 0

    # Specify the support pointMin and  pointMax, and the number of points of the
        curve (all vectors)
    # For example, pointMin=(-3.0, -2.0) and pointMax=(4.0, 5.0)
    pointMin = NumericalPoint(2)
    pointMin[0] = -3.0
    pointMin[1] = -2.0
    pointMax = NumericalPoint(2)
    pointMax[0] = 4.0
    pointMax[1] = 5.0
    pointNumber = NumericalPoint(2)
    pointNumber[0] = 101
    pointNumber[1] = 101
    PDF_graph = dist.drawMarginal2DPDF(i, j, pointMin, pointMax, pointNumber)
    CDF_graph = dist.drawMarginal2DCDF(i, j, pointMin, pointMax, pointNumber)

    # Or impose a bounding box : x-range and y-range
    # boundingBox = [xmin, xmax, ymin, ymax]
    myBoundingBox = NumericalPoint(4)
    myBoundingBox[0] = xmin
    myBoundingBox[1] = xmax
    myBoundingBox[2] = ymin
    myBoundingBox[3] = xmax
    PDF_graph.setBoundingBox(myBoundingBox)

    # In order to see the graph whithout creating the associated files
    Show(PDF_graph)
    Show(CDF_graph)

    # Create the files corresponding to the graph
    # the files .EPS, .PNG and .FIG are created in the current python session
    PDF_graph.draw("PDF_2DMarginal_ij_graph")
    CDF_graph.draw("CDF_2DMarginal_ij_graph")

    # Or only the .EPS file
    # 640 and 480 are the pixels number in both axes
    PDF_graph.draw("PDF_2DMarginal_ij_graph", 640, 480, GraphImplementation.EPS)
    CDF_graph.draw("CDF_2DMarginal_ij_graph", 640, 480, GraphImplementation.EPS)

    # Visualize the PNG file in the TUI
    ViewImage(PDF_graph.getBitmap())
    ViewImage(CDF_graph.getBitmap())
```

We draw respectively in Figures 38 and 39 the iso-curves of the PDF of the two following distributions :

- Distribution 1 : Mixture of Normal distributions of dimension 2
- Distribution 2 : Composed Distribution, with a Gumbel copula and each marginal some mixture of normals of dimension 1.

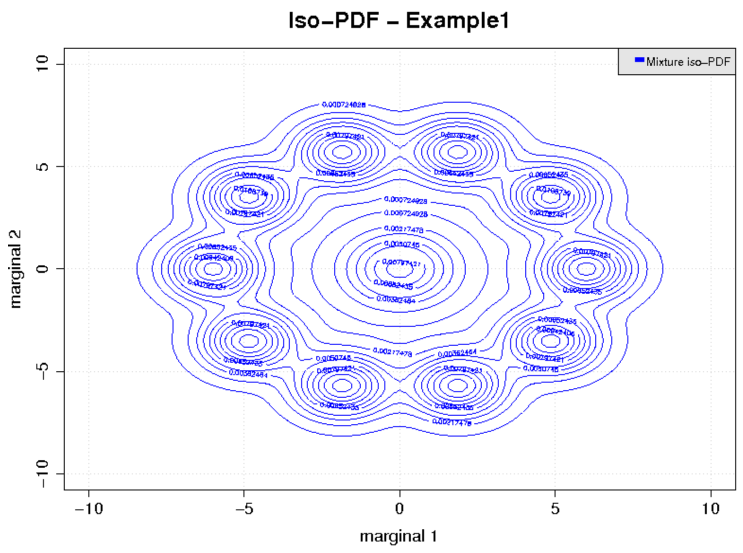


Figure 38: Iso-curves of the PDF of Distribution 1 : Mixture of Normal distributions of dimension 2.

1.1.7 UC : Creation of the random input vector from a distribution

The objective of this UC is to model a random vector described by its joint probability density function. This random vector is called a *UsualRandomvector*. This UC is particularly adapted to the input random vector.

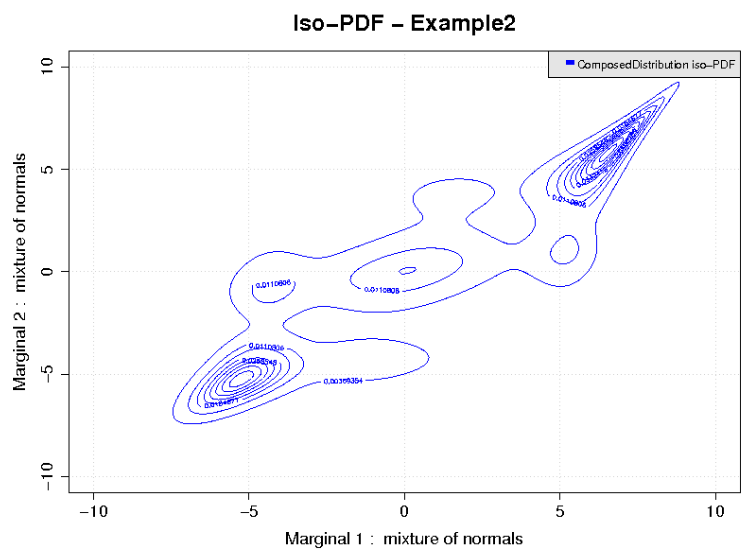


Figure 39: Iso-curves of the PDF of Distribution 2 : Composed Distribution, with a Gumbel copula and each marginal some mixture of normals of dimension 1.

Requirements	• the input distribution : *inputDistribution* **type** : Distribution
Results	• the random input vector : *inputRandomVector* **type** : RandomVector which implementation is a UsualRandomVector

Python script for this UseCase :

```

# Create the UsualRandomVector 'inputRandomVector' from
# the Distribution 'inputDistribution'
    inputRandomVector = RandomVector(inputDistribution)
```

1.2 With samples on data : manipulation on data

It is important to note that all the Use Cases described in this section are usefull to fit a distribution from a sample in order to model the random input vector. However, it is possible to apply them to fit a distribution to the output variable of interest when described by a sample.

1.2.1 UC : Import / Export data from a file at format CSV (Comma Separated Value)

The objective of this UC is to import a file at format CSV containing a list of data and to export a NumericalSample into a file at format CSV.
To be a proper sample file, the following rules must be respected :

- Data are presented in line : each line corresponds to the realisation of the aleatory vector. The number of lines is the size of the sample. The number of data on each line is the dimension of the sample.
- Data must be separated by a ";".
- No missing data must appear (it means each line must have the same number of data).
- Each data must described with a number as "3.7" or "3.e-4".

When a line presents an error, the line is ignored but all the right ones are taken into account. The number of lines which don't follow the previous rules are signaled and the reason is given.

Requirements	• a file containing data : *sampleFile.csv* **type** : a CSV format file respecting rules explicited before • or a numerical sample to be stored : *mySampleToBeStored* **type** : a NumericalSample
Results	• the sample issued from the data file *sampleFile.csv* : *aSample* **type** : a NumericalSample • a file containing*mySampleToBeStored* : *mySampleStoredFile.csv* **type** : a CSV format file respecting rules explicited before

Python script for this UseCase :

```
# IMPORT a CSV FILE

# We give in argument of the static method ImportFromCSVFile()
# the absolute adress of the file sampleFile.csv
# for example : /tmp/sampleFile.csv
# if only the name sampleFile.csv is fulfilled ,
# Open TURNS looks for the file in the current directory
    aSample = NumericalSample.ImportFromCSVFile("/tmp/sampleFile.csv")

    # We give a name to the sample loaded
    aSample.setName("first_data_sample")

# EXPORT INTO A CSV FILE

# We give in argument of the dynamic method exportToCSVFile
# the absolute adress where the storing file mySampleStoredFile.csv
# will be created
# for example : /tmp/mySampleStoredFile.csv
# if only the name mySampleStoredFile.csv is fulfilled ,
# Open TURNS creates the file in the current directory
    mySampleToBeStored.exportToCSVFile("/tmp/mySampleStoredFile.csv")
```

1.2.2 UC : Drawing Empirical CDF, Histogram, Clouds / PDF or superposition of two clouds from data

The objective of this UC is to draw :

- the empirical cumulative density function (CDF) from data : GRAPH 1,
- the histogram from data : GRAPH 2 (with imposed number of bars) and GRAPH 3 (with free number of bars) ,
- the superposition of two 2D samples where the first sample is given as sample and the second sample is evaluated from a given from a 2D distribution : GRAPH 4,
- the superposition of two 2D samples where both samples are given as samples : GRAPH 5.

To draw an histogram, it is possible :

- to fix the number of bars,
- or not to mention it : Open TURNS will determine automatically the bandwith of the histogram according to the Silverman rule (gaussian empirical rule).

Requirements	• one scalar numerical sample : *sample* • two 2D numerical samples : *sample2, sample3* **type** : NumericalSample • one 2D distribution : *dist2D* **type** : Distribution
Results	• the files containing the empirical CDF graph : *sampleCDF.png, sampleCDF.eps, sampleCDFZoom.png, sampleCDFZoom.eps* **type** : files at format PNG or EPS or FIG • the files containing the histogram graph : *sampleHist.png, sampleHist.eps, sampleHistOpt.png, sampleHistOpt.eps* **type** : files at format PNG or EPS or FIG • the files containing the superposed samples (sample 2 and issued from dist2D) : *sampleCloudPdf.png, sampleCloudPdf.eps* **type** : files at format PNG or EPS or FIG • the files containing the superposed samples (sample 2 and issued from dist2D) : *sampleClouds.png, sampleClouds.eps* **type** : files at format PNG or EPS or FIG

Python script for this UseCase :

```
# GRAPH 1 : Empirical CDF graph
    # Generate the Graph structure for the empirical CDF graph
    # graph range : min(sample) - 1, man(sample) + 1
    # CARE : sample must be of dimension 1
    sampleCDF = VisualTest.DrawEmpiricalCDF(sample, sample.getMin()[0] - 1.0,
        sample.getMax()[0] + 1.0)

    # Or impose a bounding box : x-range and y-range
    # boundingBox = [xmin, xmax, ymin, ymax]
    myBoundingBox = NumericalPoint(4)
    myBoundingBox[0] = xmin
    myBoundingBox[1] = xmax
    myBoundingBox[2] = ymin
    myBoundingBox[3] = xmax
    sampleCDF.setBoundingBox(myBoundingBox)

    # In order to see the graph whithout creating the associated files
    Show(sampleCDF)

    # Draw the graph on the file sampleCDF.png and sampleCDF.eps
    # if the file adress is not fulfilled , the file is created in the current
        directory
    sampleCDF.draw("sampleCDF")

    # View the bitmap file
    ViewImage(sampleCDF.getBitmap())

    # Check if it worked
    print "bitmap␣=␣" , sampleCDF.getBitmap()
    print "postscript␣=␣" , sampleCDF.getPostscript()

# GRAPH 2 : Histogram graph with number of bars fixed by the user
    # Generate the Graph structure for the histogram graph
    # Number of bars fixed to 10
    # CARE : sample must be of dimension 1
    sampleHist = VisualTest.DrawHistogram(sample, 10)

    # Or zoom the histogramm : impose a bounding box : x-range and y-range
    # boundingBox = [xmin, xmax, ymin, ymax]
    myBoundingBox = NumericalPoint(4)
    myBoundingBox[0] = xmin
    myBoundingBox[1] = xmax
    myBoundingBox[2] = ymin
    myBoundingBox[3] = xmax
    sampleHist.setBoundingBox(myBoundingBox)

    # In order to see the graph whithout creating the associated files
    Show(sampleHist)

    # Draw the graph on the file sampleHist.png and sampleHist.eps
    # if the file adress is not fulfilled , the file is created in the current
        directory
    sampleHist.draw("sampleHist")

    # View the bitmap file
    ViewImage(sampleHist.getBitmap())

    # Check if it worked
    print "bitmap␣=␣" , sampleHist.getBitmap()
    print "postscript␣=␣" , sampleHist.getPostscript()

# GRAPH 3 : Histogram graph with free number of bars
# (automatically determined by Open TURNS according to the Silverman rule)
    # Generate the Graph structure for the histogram graph
    # CARE : sample must be of dimension 1
    sampleHistOpt = VisualTest.DrawHistogram(sample)

    # Or zoom the histogramm : impose a bounding box : x-range and y-range
    # boundingBox = [xmin, xmax, ymin, ymax]
    myBoundingBox = NumericalPoint(4)
    myBoundingBox[0] = xmin
    myBoundingBox[1] = xmax
    myBoundingBox[2] = ymin
    myBoundingBox[3] = xmax
    sampleHistOpt.setBoundingBox(myBoundingBox)

    # Draw the graph on the file sampleHistOpt.png and sampleHist.eps
    # if the file adress is not fulfilled , the file is created in the current
        directory
    sampleHistOpt.draw("sampleHistOpt")

    # In order to see the graph whithout creating the associated files
    Show(sampleHistOpt)

    # View the bitmap file
    ViewImage(sampleHistOpt.getBitmap())

    # Check if it worked
    print "bitmap␣=␣" , sampleHistOpt.getBitmap()
    print "postscript␣=␣" , sampleHistOpt.getPostscript()


# GRAPH 4 : Superposition of two 2D samples where
# first sample is given as sample
# second sample is issued from a 2D distribution
    # CARE : sample2 must be of dimension 2
    # and dist is of dimension 2
    # the sample issued from dist2D have the same size than sample2
    cloudPdfGraph = VisualTest.DrawClouds(sample2, Distribution(dist2D))

    # Impose a bounding box : x-range and y-range
    # boundingBox = [xmin, xmax, ymin, ymax]
    myBoundingBox = NumericalPoint(4)
    myBoundingBox[0] = xmin
    myBoundingBox[1] = xmax
    myBoundingBox[2] = ymin
    myBoundingBox[3] = xmax
    cloudPdfGraph.setBoundingBox(myBoundingBox)

    # In order to see the graph whithout creating the associated files
    Show(cloudPdfGraph)

    # Draw the graph on the file sampleCloudPdf.png and sampleCloudPdf.eps
    # if the file adress is not fulfilled , the file is created in the current directory
    cloudPdfGraph.draw("sampleCloudPdf")

    # View the bitmap file
    ViewImage(cloudPdfGraph.getBitmap())

    # Check if it worked
    print "bitmap = " , cloudPdfGraph.getBitmap()
    print "postscript = " , cloudPdfGraph.getPostscript()

# GRAPH 5 : Superposition of the two 2D samples : sample2 and sample3
    # CARE : sample2 and sample3 must be of dimension 2
    cloudPdfGraph2 = VisualTest.DrawClouds(sample2, sample3)

    # Impose a bounding box : x-range and y-range
    # boundingBox = [xmin, xmax, ymin, ymax]
    myBoundingBox = NumericalPoint(4)
    myBoundingBox[0] = xmin
    myBoundingBox[1] = xmax
    myBoundingBox[2] = ymin
    myBoundingBox[3] = xmax
    cloudPdfGraph.setBoundingBox(myBoundingBox)

    # In order to see the graph whithout creating the associated files
    Show(cloudPdfGraph2)

    # Draw the graph on the file sampleCloudPdf.png and sampleCloudPdf.eps
    # if the file adress is not fulfilled , the file is created in the current directory
    cloudPdfGraph2.draw("sampleClouds")

    # View the bitmap file
    ViewImage(cloudPdfGraph2.getBitmap())

    # Check if it worked
    print "bitmap = " , cloudPdfGraph2.getBitmap()
    print "postscript = " , cloudPdfGraph2.getPostscript()
```

For example, Figure 40 contains the GRAPH3 obtained with a sample of size 1000 from a Normal(0.0, 1.0) distribution.

For example, Figure 41 contains the GRAPH4 obtained by giving :

- a sample (actually generated from a 2D Normal distribution with (2.0, 2.0) mean (1.0, 1.0) standard deviation and $\rho = -0.8$ correlation coefficient),

- a 2D Normal distribution with (2.0, 2.0) mean (1.0, 1.0) standard deviation and $\rho = +0.8$ correlation coefficient

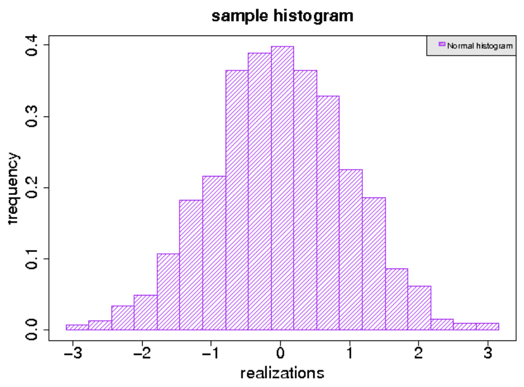


Figure 40: Histogram from a sample.

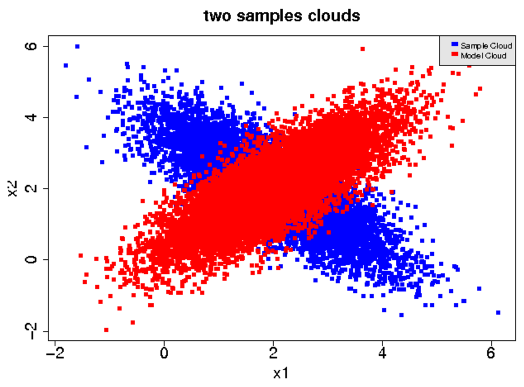


Figure 41: Superposition of two 2D clouds.

1.2.3 UC : Do two samples have the same distribution : QQ-plot visual test, Smirnov numerical test

The objective of this UC is to decide whether both samples follow the same distribution or not.
To help the decision, Open TURNS proposes one visual test and one numerical test :

- the QQ-plot visual test : Open Turns associates the empirical quantiles of each data from the both numerical samples,
- the Smirnov test : it tests if both samples (continuous ones only) follow the same distribution. If $F^*_{n_1}$ and $F^*_{n_2}$ are the empirical cumulative density functions of both samples of size n_1 and n_2, the Smirnov test evaluates the decision variable :

$$D^2 = \sqrt{\frac{n_1 n_2}{n_1 + n_2}} \sup_x |F^*_{n_1}(x) - F^*_{n_2}(x)|$$

which tends towards the Kolmogorov distribution. The hypothesis of same distribution is rejected if D^2 is too high (depending on the p-value threshold).

Requirements	• two numerical continuous samples of dimension 1 : *continuousSample1, continuousSample2* **type** : NumericalSample
Results	• the files containing the QQ-plot graph : *twoSamplesQQPlot.png, twoSamplesQQPlot.eps* **type** : files at format PNG or EPS or FIG • test result : *resultSmirnov* **type** : TestResult

Python script for this UseCase :

```
# GRAPH 1 : QQ-plot graph
    # Generate the Graph structure for the QQ-plot graph
    # number of points of the graph fixed to 100 (20 by default)
    twoSamplesQQPlot = VisualTest.DrawQQplot(continuousSample1,
        continuousSample2, 100)

    # Impose a bounding box : x-range and y-range
    # boundingBox = [xmin, xmax, ymin, ymax]
    myBoundingBox = NumericalPoint(4)
    myBoundingBox[0] = xmin
    myBoundingBox[1] = xmax
    myBoundingBox[2] = ymin
    myBoundingBox[3] = xmax
    twoSamplesQQPlot.setBoundingBox(myBoundingBox)

    # In order to see the graph whithout creating the associated files
    Show(twoSamplesQQPlot)

    # Draw the graph on the file twoSamplesQQPlot.png and twoSamplesQQPlot.eps
    # if the file adress is not fulfilled, the file is created in the current
        directory
    twoSamplesQQPlot.draw("twoSamplesQQPlot")

    # View the bitmap file
    ViewImage(twoSamplesQQPlot.getBitmap())

    # Check if it worked
    print "bitmap = " , twoSamplesQQPlot.getBitmap()
    print "postscript = " , twoSamplesQQPlot.getPostscript()

# Smirnov Test : test if two samples have a monotonous relation
    # H0 : same continuous distribution
    # Test = True <=> same continuous distribution
    # p-value threshold : probability of the H0 reject zone : 1-0.90
    # p-value : probability (test variable decision > test variable decision
        evaluated on the samples)
    # Test = True <=> p-value > p-value threshold
    resultSmirnov = HypothesisTest.Smirnov(continuousSample1, continuousSample2,
        0.90)

    # Print result of the Smirnov Test
    print "Test Succes ? ", (resultSmirnov.getBinaryQualityMeasure()==1)

    # Get the p-value of the Smirnov Test
    print "p-value of the Smirnov Test = ", resultSmirnov.getPvalue()

    # Get the p-value threshold of the  Test
    print "p-value threshold = ", resultSmirnov.getThreshold()
```

1.2.4 UC : Are two scalar samples independent : ChiSquared test, Pearson test, Spearman test

The objective of this UC is to decide whether two samples are independent or not.
To help the decision, Open TURNS proposes the following tests :

- the ChiSquared test : it tests if both scalar samples (discret ones only) are independent.
  If n_{ij} is the number of values of the sample $i = (1,2)$ in the modality $1 \leq j \leq m$, $n_{i.} = \sum_j n_{ij}$ $n_{.j} = \sum_i n_{ij}$, and the ChiSquared test evaluates the decision variable :
  $$D^2 = \sum_i \sum_j \frac{(n_{ij} - \frac{n_{i.}n_{.j}}{n})^2}{\frac{n_{i.}n_{.j}}{n}}$$
  wich tends towards the $\chi^2(m-1)$ distribution. The hypothesis of independence is rejected if D^2 is too high (depending on the p-value threshold).

- the Pearson test : it tests if there exists a linear relation between two scalar samples which form a gaussian vector (which is equivalent to have a linear correlation coefficient not equal to zero).
  If both samples are $\underline{x} = (x_i)_{1 \leq i \leq n}$ and $\underline{y} = (y_i)_{1 \leq i \leq n}$, and $\bar{x} = \frac{1}{n}\sum_i x_i$ and $\bar{y} = \frac{1}{n}\sum_i y_i$, the Pearson test evaluates the decision variable :
  $$D = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum(x_i - \bar{x})^2 \sum(y_i - \bar{y})^2}}$$
  The variable D tends towards a $\chi^2(n-2)$, under the hypothesis of normality of both samples. The hypothesis of a linear coefficient equal to 0 is rejected (which is equivalent to the independence of the samples) if D is too high (depending on the p-value threshold).

- the Spearman test : it tests if there exists a monotonous relation between two scalar samples.
  If both samples are $\underline{x} = (x_i)_{1 \leq i \leq n}$ and $\underline{y} = (y_i)_{1 \leq i \leq n,}$, the Spearman test evaluates the decision variable :
  $$D = 1 - \frac{6\sum_i (r_i - s_i)^2}{n(n^2-1)}$$
  where $r_i = rank(x_i)$ and $s_i = rank(y_i)$. D is such that $\sqrt{n-1}D$ tends towards the gaussian (0,1) distribution.

Requirements	• two continuous scalar numerical samples of dimension 1 : *continuousSample1, continuousSample2* **type** : NumericalSample • two discrete scalar numerical sample *discreteSample1, discreteSample2* **type** : NumericalSample
Results	• tests results : *resultChiSquared, resultPearson, resultSpearman* **type** : TestResult

Python script for this UseCase :

```
# ChiSquared Independance test : test if two scalar samples (of sizes not necessarily equal) are independant ?
    # Care : discrete distributions only
    # H0 = independent samples
    # p-value threshold : probability of the H0 reject zone : 1-0.90
    # p-value : probability (test variable decision > test variable decision evaluated on the samples)
    # Test = True <=> p-value > p-value threshold
    resultChiSquared = HypothesisTest.ChiSquared(discreteSample1, discreteSample2, 0.90)

    # Print result of the ChiSquared Test
    print "Test Succes ? ", (resultChiSquared.getBinaryQualityMeasure()==1)

    # Get the p-value of the  Test
    print "p-value of the  Test = ", resultChiSquared.getPvalue()

    # Get the p-value threshold of the ChiSquared Test
    print "p-value threshold = ", resultChiSquared.getThreshold()

# Pearson Test : test if two scalar samples which form a gaussian vector are independent (based on the evaluation of the linear correlation coefficient)
    # H0 : independent samples (linear correlation coefficient = 0)
    # Test = True <=> independent samples (linear correlation coefficient = 0)
    # p-value threshold : probability of the H0 reject zone : 1-0.90
    # p-value : probability (test variable decision > test variable decision evaluated on the samples)
    # Test = True <=> p-value > p-value threshold
    resultPearson = HypothesisTest.Pearson(continuousSample1, continuousSample2, 0.90)

    # Print result of the Pearson Test
    print "Test Succes ? ", (resultPearson.getBinaryQualityMeasure()==1)

    # Get the p-value of the Pearson Test
    print "p-value of the Pearson Test = ", resultPearson.getPvalue()

    # Get the p-value threshold of the  Test
    print "p-value threshold = ", resultPearson.getThreshold()

# Spearman Test : test if two scalar samples have a monotonous relation
    # H0 : no monotonous relation between both samples
    # Test = True <=> no monotonous relation
    # p-value threshold : probability of the H0 reject zone : 1-0.90
    # p-value : probability (test variable decision > test variable decision
        evaluated on the samples)
    # Test = True <=> p-value > p-value threshold
    resultSpearman = HypothesisTest.Spearman(continuousSample1,
        continuousSample2, 0.90)

    # Print result of the Spearman Test
    print "Test Succes ? ", (resultSpearman.getBinaryQualityMeasure()==1)

    # Get the p-value of the Spearman Test
    print "p-value of the Spearman Test = ", resultSpearman.getPvalue()

    # Get the p-value threshold of the  Test
    print "p-value threshold = ", resultSpearman.getThreshold()
```

1.2.5 UC : Particular manipulations of the Pearson and Spearman tests, when the first sample is of dimension superior to 1.

The objective of this UC is to decide whether two samples follow a monotonous or linear relation in the case where the first sample is of dimension > 1.
The Pearson and Spearman tests are evaluated successively between some (or all) coordinates of the first sample and the second one, which must be of dimension 1.

Requirements	• one continuous scalar numerical sample of dimension n : *continuousSample1* **type** : NumericalSample • one continuous scalar numerical sample of dimension 1 : *continuousSample2* **type** : NumericalSample
Results	• tests results : *resultPartialPearson, resultFullPearson, resultPartialSpearman, resultFullSpearman* **type** : TestResultCollection

Python script for this UseCase :

```

# Partial Pearson Test : test if two scalar samples which form a gaussian vector are independent (based on the evaluation of the linear correlation coefficient)
  # H0 : independent samples (linear correlation coefficient = 0)
  # Test = True <=> independent samples (linear correlation coefficient = 0)
  # p-value threshold : probability of the H0 reject zone : 1-0.90
  # p-value : probability (test variable decision > test variable decision evaluated on the samples)
  # Test = True <=> p-value > p-value threshold

  # selection of coordinates of continuousSample1 to be tested to continuousSample2
  # for example, coordinates 1, 2, 3, 4, 5, (suppose n>5)
  selection = Indices(5)
  for i in range(5) :
    selection[i] = i

  # Perform the Partial Pearson Test
  resultPartialPearson = HypothesisTest.PartialPearson(continuousSample1, continuousSample2, selection, 0.90)

  # Print the global result of the Pearson Test
  print "Test global result : ", resultPartialPearson

  # Print result of the Pearson Test for each coordinate tested
  for i in range(5) :
      print "Test Succes for Coordinate =  ", selection[i], "? ", (resultPartialPearson[i].getBinaryQualityMeasure()==1)

  # Get the p-value of the Pearson Test
      print "p-value of the Pearson Test = ", resultPartialPearson[i].
          getPvalue()

  # Get the p-value threshold of the  Test
      print "p-value threshold for Coordinate =  ", selection[i], " = ",
          resultPartialPearson[i].getThreshold()

# Full Pearson Test : it performs the partial Pearson test on the whole
  coordinates of the first sample

  # Perform the Full Pearson Test
  resultFullPearson = HypothesisTest.FullPearson(continuousSample1,
      continuousSample2, 0.90)

  # Same manipulations than those effected on resultPartialPearson to get the
      results

# Partial Spearman Test : test if two scalar samples have a monotonous relation
  # H0 : no monotonous relation between both samples
  # Test = True <=> no monotonous relation
  # p-value threshold : probability of the H0 reject zone : 1-0.90
  # p-value : probability (test variable decision > test variable decision
      evaluated on the samples)
  # Test = True <=> p-value > p-value threshold

  # selection of coordinates of continuousSample1 to be tested to
      continuousSample2
  # for example, coordinates 1, 2, 3, 4, 5, (suppose n>5)
  selection = Indices(5)
  for i in range(5) :
    selection[i] = i

  # Perform the Partial Spearman Test
  resultPartialSpearman = HypothesisTest.PartialSpearman(continuousSample1,
      continuousSample2, selection, 0.90)

  # Print the global result of the Spearman Test
  print "Test global result : ", resultPartialSpearman

  # Print result of the Spearman Test for each coordinate tested
  for i in range(5) :
      print "Test Succes for Coordinate =  ", selection[i], "? ", (
          resultPartialSpearman[i].getBinaryQualityMeasure()==1)

  # Get the p-value of the Spearman Test
      print "p-value of the Spearman Test = ", resultPartialSpearman[i].
          getPvalue()

  # Get the p-value threshold of the  Test
      print "p-value threshold for Coordinate =  ", selection[i], " = ",
          resultPartialSpearman[i].getThreshold()

# Full Spearman Test : it performs the partial Pearson test on the whole
  coordinates of the first sample

  # Perform the Full Spearman Test
  resultFullSpearman = HypothesisTest.FullSpearman(continuousSample1,
      continuousSample2, 0.90)

  # Same manipulations than those effected on resultPartialSpearman to get the
      results
```

1.2.6 UC : Regression test between two scalar numerical samples

The objective of this UC is to detect a linear relation between two scalar numerical samples.

Requirements	• one continuous scalar numerical sample of dimension n : *continuous-Sample1* **type** : NumericalSample • one continuous scalar numerical sample of dimension 1 : *continuous-Sample2* **type** : NumericalSample
Results	• tests results : *resultPartialRegression, resultFullRegression, resultPartialSpearman, resultFullSpearman* **type** : TestResultCollection

Python script for this UseCase :

```

# Partial Regression Test between 2 samples : firstSample of dimension n and
    secondSample of dimension 1. If firstSample[i] is the numerical sample
    extracted from firstSample (ith coordinate of each point of the numerical
    sample), PartialRegression performs the Regression test simultaneously on all
     firstSample[i] and secondSample, for i in the selection. The Regression test
     tests if the regression model between two scalar numerical samples is
    significant. It is based on the deviation analysis of the regression. The
    Fisher distribution is used.

    # selection of coordinates of continuousSample1 to be tested to
        continuousSample2
    # for example, coordinates 1, 2, 3, 4, 5, (suppose n>5)
    selection = Indices(5)
    for i in range(5) :
      selection[i] = i

    # Perform the Partial Regression Test
    resultPartialRegression = HypothesisTest.PartialRegression(continuousSample1
        , continuousSample2, selection, 0.90)

    # Print the global result of the Regression Test
    print "Test global result : ", resultPartialRegression

    # Print result of the Regression Test for each coordinate tested
    for i in range(5) :
        print "Test Succes for Coordinate =  ", selection[i], "? ", (
            resultPartialRegression[i].getBinaryQualityMeasure()==1)

    # Get the p-value of the Regression Test
    print "p-value of the Regression Test = ", resultPartialRegression[i].
        getPvalue()

    # Get the p-value threshold of the  Test
    print "p-value threshold for Coordinate =  ", selection[i], " = ",
        resultPartialRegression[i].getThreshold()

# Full Regression Test : it performs the partial  Regression test on the whole
    coordinates of the first sample

    # Perform the Full Regression  Test
    resultFullRegression = HypothesisTest.FullRegression(continuousSample1,
        continuousSample2, 0.90)

    # Same manipulations than those realised on resultPartialRegression to get
        the results
```

1.2.7 UC : Distribution fitting tests, numerical and visual validation tests : ChiSquared test, Kolmogorov test, QQ-plot graph

The objective of this UC is to :

- perform some parametric fitting tests on a numerical sample in dimension 1, with the maximum likelihood principle or the moment based method,
- validate these estimations with numerical tests : the Kolmogorov test (continuous distributions) or the ChiSquared test (discrete distributions),
- validate these estimations with a visual test : the QQ-plot graph.

The QQ-plot visual validation test is used with a numerical sample (representing the data) and a distribution (representing the fitted one). For each point of the numerical sample used in the graph, Open Turns evaluates its empirical quantile and associates to it the corresponding quantile from the fitted distribution.

The example here presents :

- the fitting of a numerical sample of dimension 1 with a Beta distribution, its validation with the Kolmogorov test and the QQ-Plot graph,
- the fitting of a numerical sample of dimension 1 with a Poisson distribution, its validation with the Chi Squared test and the QQ-Plot graph.

Requirements	• a scalar numerical sample (data) : *sample* **type** : NumericalSample
Results	• a Beta fitted continuous distribution : *estimatedBetaDistribution* **type** : Distribution • a Uniform continuous fitted distribution : *estimatedUniformDistribution* **type** : Distribution • a Poisson discrete fitted distribution : *estimatedPoissonDistribution* **type** : Distribution • the files containing the QQ-plot graph : *QQPlot.png, QQPlot.eps* **type** : files at format PNG or EPS or FIG • a numerical validation by the Kolmogorov test for two continuous distributions (p-value) **type** : TestResult • a numerical validation by the ChiSquared test for discrete distribution (p-value) **type** :TestResult

Python script for this UseCase :

```
# Fit a Beta distribution to the sample
    # Create a Beta factory
    factory = BetaFactory()

    # Estimate the beta parameters
    # We estimate all the parameters of the Beta distribution from sample
    estimatedBetaDistribution = factory.buildImplementation(sample)

    # Display the resulted distribution with its parameters
    print "Estimated Beta distribution=", estimatedBetaDistribution

# Validate the Beta fitted distribution with the Kolmogorov Test
    # Test = True <=> the sample follows a Beta distribution (H0 hypothesis)
    # p-value threshold : probability of the H0 reject zone = 1-0.95
    # p-value : probability (test variable decision > test variable decision
        evaluated on the samples)
    # Test = True (=1) <=> p-value > p-value threshold
    resultKolmogorov =  FittingTest().Kolmogorov(sample,
        estimatedBetaDistribution, 0.95)

    # Print result of the Kolmogorov Test
    print "Test Succes ? ", (resultKolmogorov.getBinaryQualityMeasure()==1)

    # Get the p-value of the Kolmogorov Test
    print "p-value of the Kolmogorov Test = ", resultKolmogorov.getPvalue()

    # Get the p-value threshold of the Kolmogorov Test
    print "p-value threshold = ", resultKolmogorov.getThreshold()

# Validate the Beta fitting with a visual test : QQ-plot test
    # Generate the Graph structure for the QQ-plot graph
    # number of points of the graph fixed to 100 (20 by default)
    sampleBetaQQPlot = VisualTest.DrawQQplot(sample, Distribution(
        estimatedBetaDistribution), 100)

    # Impose a bounding box : x-range and y-range
    # boundingBox = [xmin, xmax, ymin, ymax]
    myBoundingBox = NumericalPoint(4)
    myBoundingBox[0] = xmin
    myBoundingBox[1] = xmax
    myBoundingBox[2] = ymin
    myBoundingBox[3] = xmax
    sampleBetaQQPlot.setBoundingBox(myBoundingBox)

    # In order to see the graph whithout creating the associated files
    Show(sampleBetaQQPlot)

    # Draw the graph on the file BetaQQPlot.png and twoSamplesQQPlot.eps
    # if the file adress is not fulfilled, the file is created in the current
        directory
    sampleBetaQQPlot.draw("SampleBetaQQPlot")

    # View the bitmap file
    ViewImage(sampleBetaQQPlot.getBitmap())

    # Check if it worked
    print "bitmap = ", sampleBetaQQPlot.getBitmap()
    print "postscript = ", sampleBetaQQPlot.getPostscript()

# Fit a Poisson distribution to the sample
    # Create a Poisson factory
    factory = PoissonFactory()

    # Estimate the Poisson parameters
    # We estimate all the parameters of the Poisson distribution from sample
    estimatedPoissonDistribution = factory.buildImplementation(sample)

    # Display the resulted distribution with its parameters
    print "Estimated Poisson distribution=", estimatedPoissonDistribution

# Validate the Poisson fitted distribution with the ChiSquared Test
    # Test = True <=> the sample follows a Beta distribution (H0 hypothesis)
    # p-value threshold : probability of the H0 reject zone = 1-0.95
    # p-value : probability (test variable decision > test variable decision
        evaluated on the samples)
    # Test = True (=1) <=> p-value > p-value threshold
    # Number of parameters estimated from sample : 1
    resultChiSquared =  FittingTest().ChiSquared(sample,
        estimatedPoissonDistribution, 0.95, 1)

    # Print result of the ChiSquared Test
    print "Test Succes ? ", (resultChiSquared.getBinaryQualityMeasure()==1)

    # Get the p-value threshold of the ChiSquared Test
    print "p-value threshold = ", resultChiSquared.getPvalue()

    # Get the p-value threshold (corresponding to the confidence level) of the
        ChiSquared Test
    print "p-value of the ChiSquared Test = ", resultChiSquared.getThreshold()
```

Figures 42 and 43 show a QQ-Plot graph to test the adequation of a sample coming from a Beta(r = 1.2, t = 3.4, a = 1.0, b = 2.0) to :

- the Beta(r = 1.2, t = 3.4, a = 1.0, b = 2.0) distribution : visual validation of the fitting,
- the Weibull($\mu = 1.5$, $\sigma = 1.0$, $\gamma = 1.0$) : visual invalidation of the fitting.

1.2.8 UC : Normal distribution fitting test, visual validation tests (Henry line) and numerical validation tests in extreme zones (Anderson Darling test and Cramer Von Mises test)

The objective of this UC is to fit a normal distribution to a scalar numerical sample, with the maximum likelihood principle or the moment based method, and to validate it with visual and numerical tests.

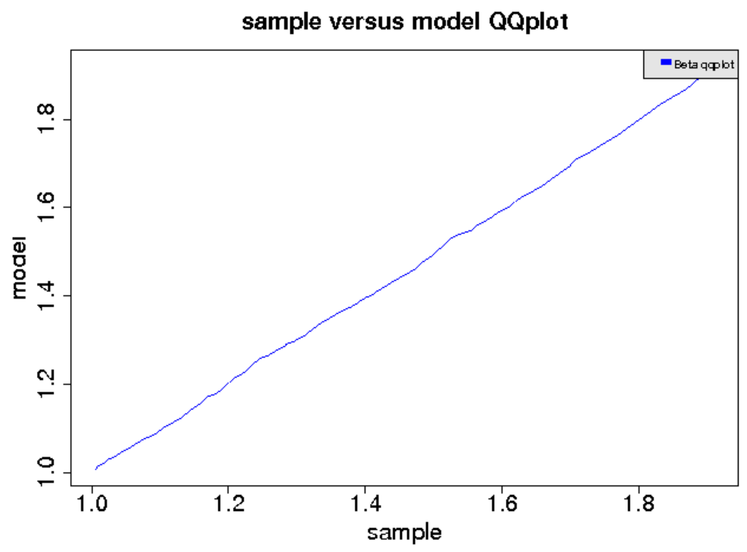


Figure 42: Fitting validation by the QQ-Plot graph : Beta fitting to a Beta-sample.

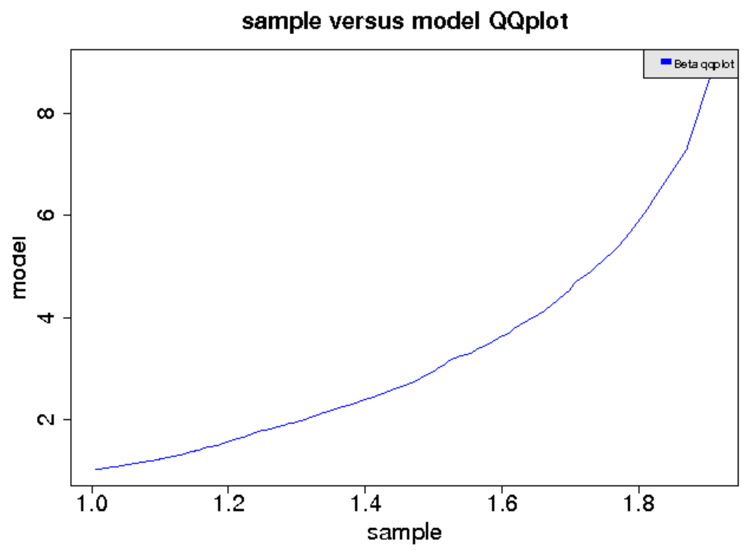


Figure 43: Fitting invalidation by the QQ-Plot graph : Weibull fitting to a Beta sample.

To help this decision, Open TURNS proposes the following tests :

- the Henry line visual test, which is the QQ-Plot graph adapted to the normal distribution,
- the Anderson Darling test : this test gives more importance to extreme values. If F_n is the empirical cumulative density function of the sample $(x_i)_{1\leq i\leq n}$ and if $(x_{(i)})_{1\leq i\leq n}$ is the ordered sample, the Anderson Darling test evaluates the decision variable :

$$\begin{aligned} AD^2 &= n\int_{\mathbb{R}} \frac{(F_n(x)-F(x))^2}{f(x)(1-F(x))}dF(x) \\ &= -n-\frac{1}{n}\sum_{i=1}^{i=n}(2i-1)[\log(f(x_{(i)}))+\log(1-F(x_{(n-i+1)}))] \end{aligned}$$

  Under the hypothesis of normality of the distribution F, the decision variable has a tabulated distribution.

- the Cramer Von Mises test : this test gives also more importance to extreme values. If F_n is the empirical cumulative density function of the sample $(x_i)_{1\leq i\leq n}$ and if $(x_{(i)})_{1\leq i\leq n}$ is the ordered sample, the Cramer Von Mises test evaluates the decision variable :

$$\begin{aligned} CM &= \int_{\mathbb{R}}(F_n(x)-F(x))^2dF(x) \\ &= \frac{1}{12n}+\sum_{i=1}^{i=n}\left[\frac{2i-1}{2n}-F(x_{(i)})\right]^2 \end{aligned}$$

  Under the hypothesis of normality of the distribution F, the decision variable has a tabulated distribution.

Requirements	• a scalar numerical sample (data) : *sample* **type** : NumericalSample
Results	• a normal fitted distribution : *estimatedNormalDistribution* **type** : Distribution • the files containing the Henry line graph : *HenryPlot.png, HenryPlot.eps* **type** : files at format PNG or EPS or FIG • a numerical validation by the Anderson Darling test for two continuous distributions (p-value) **type** : TestResult • a numerical validation by the test for Cramer Von Mises discrete distribution (p-value) **type** : TestResult

Python script for this UseCase :

```
# Henry line graph
    # Generate the Graph structure for the Henry line graph
    henryPlot = VisualTest.DrawHenryLine(sample)

    # Impose a bounding box : x-range and y-range
    # boundingBox = [xmin, xmax, ymin, ymax]
    myBoundingBox = NumericalPoint(4)
    myBoundingBox[0] = xmin
    myBoundingBox[1] = xmax
    myBoundingBox[2] = ymin
    myBoundingBox[3] = xmax
    henryPlot.setBoundingBox(myBoundingBox)

    # In order to see the graph whithout creating the associated files
    Show(henryPlot)

    # Draw the graph on the file HenryPlot.png and HenryPlot.eps
    # if the file adress is not fulfilled , the file is created in the current
        directory
    henryPlot.draw("HenryPlot")

    # View the bitmap file
    ViewImage(HenryPlot.getBitmap())

    # Check if it worked
    print "bitmap = ", HenryPlot.getBitmap()
    print "postscript = ", HenryPlot.getPostscript()

# Anderson Darling Test
    # Test = True <=> the sample follows a Normal distribution (H0 hypothesis)
    # p-value threshold : probability of the H0 reject zone = 1-0.95
    # p-value : probability (test variable decision > test variable decision
        evaluated on the samples)
    # Test = True (=1) <=> p-value > p-value threshold
    # Number of parameters estimated from sample : 4
    resultAndersonDarling =  NormalityTest.AndersonDarlingNormal(sample, 0.95)

    # Print result of the  Anderson Darling Test
    print "Test Succes ? ", (resultAndersonDarling.getBinaryQualityMeasure()==1)

    # Get the p-value of the Anderson Darling Test
    print "p-value of the Anderson Darling Test = ", resultAndersonDarling.
        getPvalue()

    # Get the p-value threshold of the Anderson Darling Test
    print "p-value threshold = ", resultAndersonDarling.getThreshold()

```

```
# Cramer Von Mises Test
    # Test = True <=> the sample follows a Normal distribution (H0 hypothesis)
    # p-value threshold : probability of the H0 reject zone = 1-0.95
    # p-value : probability (test variable decision > test variable decision
        evaluated on the samples)
    # Test = True (=1) <=> p-value > p-value threshold
    # Number of parameters estimated from sample : 4
    resultCramerVonMises =  NormalityTest.CramerVonMisesNormal(sample, 0.95)

    # Print result of the Cramer Von Mises Test
    print "Test Succes ? ", (resultCramerVonMises.getBinaryQualityMeasure()==1)

    # Get the p-value of the Cramer Von Mises Test
    print "p-value of the Cramer Von Mises Test = ", resultCramerVonMises.
        getPvalue()

    # Get the p-value threshold of the Cramer Von Mises Test
    print "p-value threshold = ", resultCramerVonMises.getThreshold()
```

Figures 44 and 45 show the Henry Line of a sample coming from a :

- Normal($\mu = 0.0$, $\sigma = 1.0$) distribution : visual validation of the normality,
- Beta(r = 0.7, t = 1.6, a = 0.0, b = 2.0) distribution : visual invalidation of the normality.

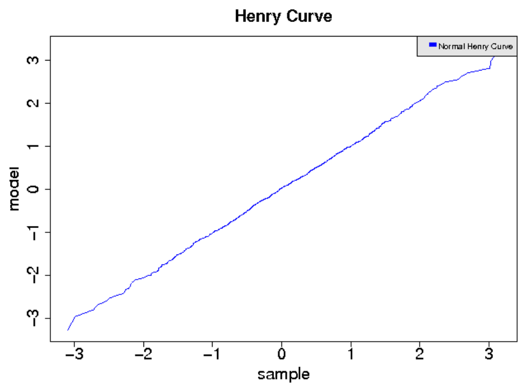


Figure 44: Validation of the hypothesis of normality by the Henry Line for a Normal-sample.

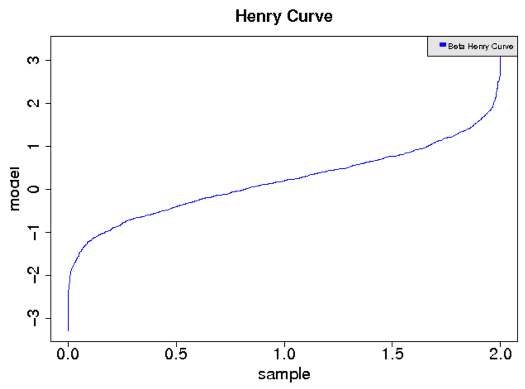


Figure 45: Invalidation of the hypothesis of normalityHenry Line for a Beta-sample.

1.2.9 UC : Making a choice between multiple fitted distributions : Kolmogorov ranking, ChiSquared ranking and BIC ranking

The objective of this UC is to help to make a choice between several distributions fitted to a numerical sample. This choice can be motivated by :

- the ranking by the Kolmogorov p-values (for continuous distributions),

- the ranking by the ChiSquared p-values (for discrete distributions),
- the ranking BIC values.

It does not necessarily require to know the parameters of the different distributions tested. It is possible to precise :

- the distribution type only : in that case, Open TURNS builds a factory for each distribution type. Open TURNS first evaluates the parameters of the distribution (through the maximum likelihood rule or the moment based one) and then ranks the distributions according to the criteria selected,
- some complete distributions with their parameters : Open TURNS will only evaluate the criteria selectd on each of them and rank them.

The example is the ranking through successively the three criteria (Kolmogorov, ChiSquared and BIC) of the following models :

- the Beta model (continuous) ,
- the Triangular model (continuous) ,
- the Poisson model (discrete) ,
- the Geometric model (discrete).

Requirements	• a numerical sample (data) : *sample* **type** : NumericalSample
Results	• a continuous distribution which ranks first by the Kolmogorov test : *bestDistributionKolmogorov* **type** : Distribution • a continuous distribution which ranks first by the BIC test : *bestDistributionBIC* **type** : Distribution • a discrete distribution which ranks first by the ChiSquared test : *bestDistributionChiSquared* **type** : Distribution

Python script for this UseCase :

```
# CASE 1 : We don't specify the parameters of the distributions tested

# Create a collection of factories for all the models we want to test
    collectionContinuousFactory = FactoryCollection(2)
    collectionContinuousFactory[0] = DistributionFactory(BetaFactory())
    collectionContinuousFactory[1] = DistributionFactory(TriangularFactory())
    collectionDiscreteFactory = FactoryCollection(2)
    collectionDiscreteFactory[0] = DistributionFactory(PoissonFactory())
    collectionDiscreteFactory[1] = DistributionFactory(GeometricFactory())

# Rank the 2 continuous models by the Kolmogorov p-values :
    bestDistributionKolmogorov = FittingTest.BestModelKolmogorov(sample,
       collectionContinuousFactory)

    # Get all information on that distribution
    print "best continuous distribution by Kolmogorov = ",
       bestDistributionKolmogorov

# Rank the 2 continuous models bythe BIC values :
    bestDistributionBIC = FittingTest.BestModelBic(sample,
       collectionContinuousFactory)

    # Get all information on that distribution
    print "best continuous distribution by BIC = ", bestDistributionBIC

# Rank the 2 discrete models by the ChiSquared p-values :
    bestDistributionChiSquared = FittingTest.BestModelChiSquared(sample,
       collectionDiscreteFactory)

    # Get all information on that distribution
    print "best continuous distribution by  = ", bestDistributionChiSquared


# CASE 2 : We specify the parameters of the distributions tested

# Create a collection of distributions we want to test
    collectionContinuousDistribution = DistributionCollection(2)
    collectionContinuousDistribution[0] = Distribution(Beta(1., 2., 3., 4.))
    collectionContinuousDistribution[1] = Distribution(Triangular(1., 2., 4.))
    collectionDiscreteDistribution = DistributionCollection(2)
    collectionDiscreteDistribution[0] = Distribution(Poisson(2))
    collectionDiscreteDistribution[1] = Distribution(Geometric(0.2))

# Rank the 2 continuous models by the Kolmogorov p-values :
    bestDistributionKolmogorov = FittingTest.BestModelKolmogorov(sample,
       collectionContinuousDistribution)

    # Get all information on that distribution
    print "best continuous distribution by Kolmogorov = ",
       bestDistributionKolmogorov

# Rank the 2 continuous models bythe BIC values :
    bestDistributionBIC = FittingTest.BestModelBic(sample,
        collectionContinuousDistribution)

    # Get all information on that distribution
    print "best continuous distribution by BIC = ", bestDistributionBIC

# Rank the 2 discrete models by the ChiSquared p-values :
    bestDistributionChiSquared = FittingTest.BestModelChiSquared(sample,
        collectionDiscreteDistribution)

    # Get all information on that distribution
    print "best continuous distribution by   = ", bestDistributionChiSquared
```

1.2.10 UC : PDF fitting by kernel smoothing and graphical validation : superposition of the empirical and kernel smoothing CDF

The objective of this UC is to model the PDF of a random vector, described by a numerical sample thanks to the kernel smoothing method and to superpose on the same graph the kernel smoothing PDF and the histogram built from the same numerical sample.

In dimension 1, the kernel smoothed PDF p_n has the following expression, where K is the kernel PDF, n the numerical sample size and $(X_1, \cdots, X_n) \in \mathbb{R}^n$ the numerical sample whith $\forall i,\ X_i \in \mathbb{R}$:

$$p_n(x) = \frac{1}{nh}\sum_{i=1}^{i=n} K(\frac{x - X_i}{h})$$

In dimension $d > 1$, the kernel of Open TURNS is the product kernel, K_d, defined by the following expression, where $\underline{x} = (x^1, \cdots, x^d) \in \mathbb{R}^d$:

$$K_d(\underline{x}) = \prod_{j=1}^{j=d} \frac{1}{h_j} K(\frac{x^j}{h^j})$$

which leads to the kernel smoothed PDF in dimension d, where $(\underline{X}_1, \cdots, \underline{X}_n)$ is the numerical sample of dimension d:

$$p_n(\underline{x}) = \frac{1}{n}\sum_{i=1}^{i=n}\prod_{j=1}^{j=d} \frac{1}{h_j} K(\frac{x^j - X_i^j}{h^j})$$

Let's note that the bandwith is the vector $\underline{h} = (h^1, \cdots, h^d)$.

The choice of the kind of the kernel is free in Open TURNS : it is possible to select any 1D distribution and to define it as a kernel. However, in order to optimise the efficiency of the kernel smoothing fitting (it means to minimise the AMISE error), it is recommended to select a **symetric distribution** for the kernel. All the distribution default constructors of Open TURNS create a symetric default distribution when possible. It is also possible to work with the Epanechnikox kernel, which is a $Beta(r = 2, t = 4, a = -1, b = 1)$.
The default kernel is a product of standard Normal distribution. The dimension of the product is automatically evaluated from the numerical sample.

The bandwith $\underline{h}$ may be fixed by the User. However, it is recommended to let Open TURNS evaluate it automatically from the numerical sample according to the Scott rule. The bandwith is evaluate for each direction according to the Scott rule :

$$h^i_{Scott} = \frac{\hat{\sigma}^n_i}{\sigma_K} n^{-1/(d+4)} \tag{2}$$

where $\hat{\sigma}^n_i$ is the standard deviation of the $i-th$ component of the sample $(\underline{X}_1, \cdots, \underline{X}_n)$, and σ_K the standard deviation of the 1D kernel K.

Note that this bandwith is a simplification of the Silverman bandwith which minimises the AMISE error when using a Normal kernel in order to fit a gaussian vector whith independent components. That's why the Scott bandwith may appear too large when the real probability density fucntion presents several maximum.

The Silverman rule proposes the following bandwith, in dimension d with a normal kernel $Normal(0.0, 1.0)$:

$$h^i_{Silv}(N) = \left(\frac{4}{d+2}\right)^{1/(d+4)} \hat{\sigma}^n_i n^{-1/(d+4)} \tag{3}$$

The Scott proposition is based on the following remarks :

- Remark 1 : the coefficient $\left(\frac{4}{d+2}\right)^{1/(d+4)}$ remains in $[0.924, 1.059]$ when the dimension d varies : Scott fixed it to 1 :

$$\left(\frac{4}{d+2}\right)^{1/(d+4)} \simeq 1 \tag{4}$$

- Remark 2 : in the case of dimension $d = 1$, the Silverman rule applied to the kernels K_1 and K_2, not necessarily normal, leads to both bandwiths h^1 and h^2 such as :

$$\frac{h^1_{Silv}(K_1)}{h^2_{Silv}(K_2)} = \frac{\sigma_{K_2}}{\sigma_{K_1}} \left[\frac{\sigma_{K_1} R(K_1)}{\sigma_{K_2} R(K_2)}\right]^{1/5} \tag{5}$$

  where $R(K) = \int K^2(z)dz$. Furthermore, the quantity $\sigma_{K_1} R(K_1)$ is quasi equal to 1 whatever the kernel K_1. Thus, relation (5) simplifies in :

$$h^2_{Silv}(K_2) \simeq h^1_{Silv}(K_1) \frac{\sigma_{K_1}}{\sigma_{K_2}} \tag{6}$$

  Scott spreads the relation (6) to any direction, whith K_1 a Normal kernel : relations (4), (3) and (6) finally lead to the Scott relation (2).

In dimension 1, the boundary effects may be taken into account in Open TURNS : the boundaries are automatically detected from the numerical sample (with the min and max functions) and the kernel smoothed PDF is corrected in the boundary areas to remain within the boundaries, according to the miroring technique :

- the Scott bandwith is evaluated frome the numerical sample : h
- two subsamples are extracted from the inital numerical sample, containing all the points within the range $[min, min + h[$ and $]max - h, max]$,
- both subsamples are transformed into their symetric samples with respect their respective boundary : its results two samples within the range $]min - h, min]$ and $[max, max + h[$,

- a kernel smoothed PDF is built from the new numerical sample composed with the initial one and the two new ones, with the previous bandwith h,
- this last kernel smoothed PDF is truncated within the inital range $[min, max]$ (conditionnal PDF).

Requirements	• a nD-sample : *sample* **type** : NumericalSample
Results	• a kernel smoothed distribution : *kernelSmoothedDist* **type** : Distribution

Python script for this UseCase :

```

# STEP 1 : Creation of the kernel

# Create the default kernel : kernel product of N(0.0, 1.0)
    kernel = KernelSmoothing()

# Create a specified kernel
    # for example, a Uniform one
    # the default construction of the Uniform
    # creates the Uniform(-1.0, 1.0)
    kernel = KernelSmoothing(Distribution(Uniform()))

# Specify totally the kernel
    # CARE : the kernel smoothing is more efficient
    # when the kernel support is symetric qith respect to 0
    myDist = Triangular(-2.0, 0.0, 2.0)
    kernel = KernelSmoothing(Distribution(myDist))


# STEP 2 : Creation of the kernel smoothed distribution
    # The dimension of the distribution is automatically
    # detected from the numerical sample

    # With no bandwith specification
    # With no boudary treatment
    kernelSmoothedDist = kernel.buildImplementation(sample)

# Check the bandwidth used
    print "kernel␣bandwidth=" , kernel.getBandwidth()

    # Specify a particular bandwith
    myBandwith = NumericalPoint(sample.getDimension(), 1.0)
    kernelSmoothedDist = kernel.buildImplementation(sample, myBandwith)

    # Add a boundary treatment
    # CARE : only in dimension 1
    kernelSmoothedDist = kernel.buildImplementation(sample, 'TRUE')
    # or
    kernelSmoothedDist = kernel.buildImplementation(sample, myBandwith, 'TRUE')


# GRAPH : In dimension 1, superposition of the kernel smoothed CDF
# and the empirical CDF
    # Create the graph containing the  kernel smoothed PDF
    kernelSmoothedCDF = kernelSmoothedDist.drawCDF()

    # Draw the empirical CDF of the sample on the same graph
    empiricalCDF = VisualTest.DrawEmpiricalCDF(sample,sample.getMin()[0],sample.
        getMax()[0])
    drawableEmpiricalCDF = empiricalCDF.getDrawable(0)

    # Add the second drawable on the first graph
    kernelSmoothedCDF.addDrawable(drawableEmpiricalCDF)

    # Impose a bounding box : x-range and y-range
    # boundingBox = [xmin, xmax, ymin, ymax]
    myBoundingBox = NumericalPoint(4)
    myBoundingBox[0] = xmin
    myBoundingBox[1] = xmax
    myBoundingBox[2] = ymin
    myBoundingBox[3] = xmax
    kernelSmoothedCDF.setBoundingBox(myBoundingBox)

    # In order to see the graph whithout creating the associated files
    Show(kernelSmoothedCDF)

    # Draw the final graph on the file smoothedCDF-EmpiricalCDF at format .eps, .
        png and .fig
    # if the adress is not fulfilled, the file is created in the current
        directory
    kernelSmoothedCDF.draw("smoothedCDF-EmpiricalCDF")

    # View the bitmap file
    ViewImage(kernelSmoothedCDF.getBitmap())

    # Check the adress of the bitmap and Postscript files
    print "bitmap=", kernelSmoothedCDF.getBitmap()
    print"postscript=", kernelSmoothedCDF.getPostscript()
```

Figures 46 and 47 show a 1D kernel smoothing of a distribution of type Mixture which PDF is defined by : 0.2*Triangular(1.0, 2.0, 4.0) + 0.5*Normal(-1.0, 1.0) + 0.3*Exponential(1.0, 3.0), thanks to a numerical sample of size 10^4, with a Normal kernel, a Triangular one and the Epanechnikov one.

Figures 48 and 49 show the effect of the boundary treatment in the kernel smoothing through the example of the exponential distribution $Exp(\lambda = 2.0, \gamma = 0.0)$. A Normal kernel is used.

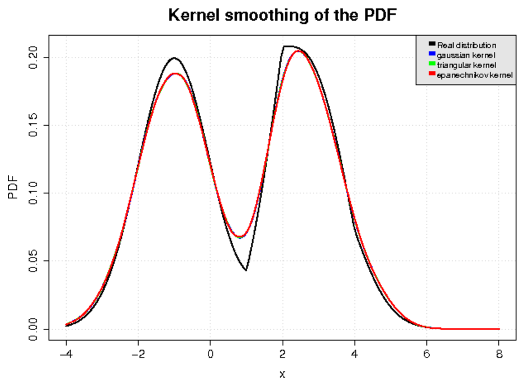


Figure 46: PDF of the kernel smoothing distributions and of the real one.

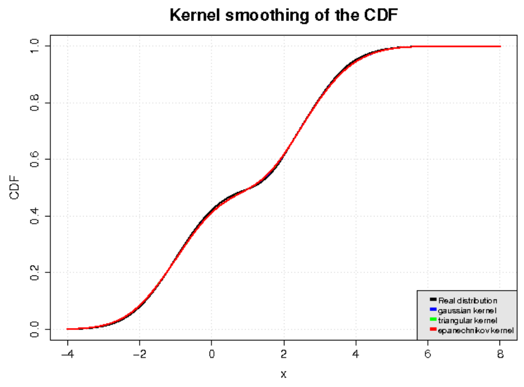


Figure 47: CDF of the kernel smoothing distributions and of the real one.

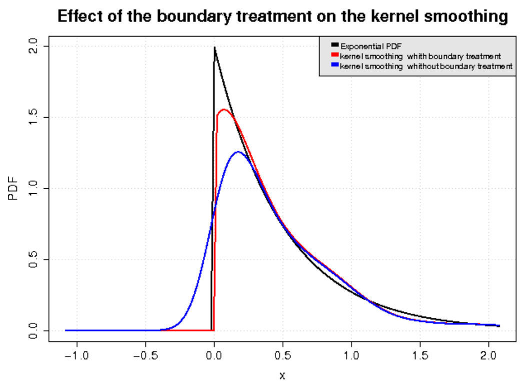


Figure 48: Effect of the boundary treatment on the kernel smoothing PDF of an exponential distribution.

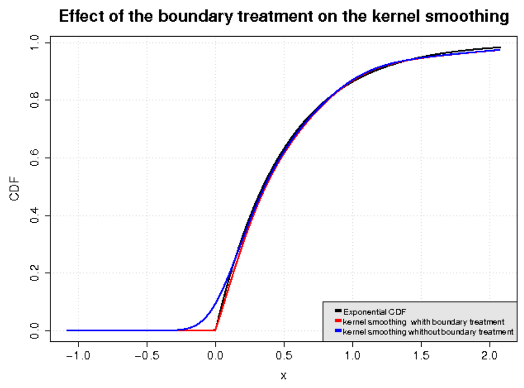


Figure 49: Effect of the boundary treatment on the kernel smoothing CDF of an exponential distribution.

1.2.11 UC : Building and validating a linear model from two samples

The objective of this UC is to build a linear regression model between a the scalar variable Y and the n-dimensionnal one $\underline{X} = (X_i)_{i \leq n}$, as follows :

$$\tilde{Y} = a_0 + \sum_i a_i X_i + \epsilon$$

where ϵ is the residual, supposed to follow the Normal(0.0, 1.0) distribution.
Each coefficient a_i is evaluated from both samples *Ysample* and *Xsample* and is accompagnied by a confidence interval and a p-value (wich tests if they are significantly different from 0.0).

The linear model may be used to evaluate predictions on particular sample of the variable X : *particularXSample.*

The linear model may be validated :

- graphically if *Xsample* is of dimension 1, by drawing on the same graph the cloud (*Xsample, Ysample*) and the regression line, with the Open TURNS method *DrawLMVisualTest*,
- numerically with the following Open TURNS tests :
  - *LMRSquared* Test which tests the quality of the linear regression model. It evaluates the indicator R^2 (regression variance analysis) and compares it to a level,
  - *LMRAdjustedSquared* which tests the quality of the linear regression model. It evaluates the indicator R^2 adjusted (regression variance analysis) and compares it to a level,
  - *LMFisher* Test which tests the nullity of the regression linear model coefficients (Fisher distribution used),
  - *LMResidual* Test which tests, under the hypothesis of a gaussian sample, if the mean of the residual is equal to zero. It is based on the Student test (equality of mean for two gaussian samples).

The hypothesis on the residuals (centered gaussian distribution) may be validated :

- graphically if *Xsample* is of dimension 1, by drawing the residual couples $(\epsilon_i, \epsilon_{i+1})$, where the residual ϵ_i is evaluated on the samples (*Xsample, Ysample*) : $\epsilon_i = Ysample_i - \tilde{Y}_i$ with $\tilde{Y}_i = a_0 + a_1 Xsample_i$. The Open TURNS method is *DrawLMResidualtest* ,
- numerically with the *LMResidualMean* Test wich tests, under the hypothesis of a gaussian sample, if the mean of the residual is equal to zero. It is based on the Student test (equality of mean for two gaussian samples).

Requirements	• a 1D-sample : *Ysample* **type** : NumericalSample • a nD-sample : *Xsample* **type** : NumericalSample • a nD-sample : *particularXSample* **type** : NumericalSample

Results	• a linear regression model : *linearRegressionModel* **type** : LinearModel • the linear coefficients $(a_i)_{0 \leq i \leq n}$: *coefValues* **type** : scalarCollection • the confidence intervals of each coefficient a_i **type** : ConfidenceIntervalCollectionf • the p-values of each coefficient a_i **type** : ConfidenceIntervalCollection • the predicted value on a particular sample : *predictedSample* **type** : NumericalSample • the sample of resual values: *residualSample* **type** : NumericalSample • the graph superposing the samples cloud and the regression line (in case of dimension 1 for X) : *linearRegressionModel.png, linear-RegressionModel.eps* **type** : files at format PNG or EPS or FIG • the graph of residual values : *residualGraph.png, residualGraph.eps* **type** : files at format PNG or EPS or FIG • LMRAdjustedSquared test result : *resultLMRAdjustedSquared* **type** : TestResult • LMRSquared test result : *resultLMRSquared* **type** : TestResult • LMFisher test result : *resultLMFisher* **type** : TestResult • LMResidualMean test result : *resultLMResidualMean* **type** : TestResult

Python script for this UseCase :

```
# Create the linear model from both sample : Ysample function of Xsample
# CARE : Xsample is of dimension n and Ysample of dimension 1
# The level confidence to evaluate the confidence interval is set to 0.90
    linearRegressionModel = LinearModelFactory().buildLM(Xsample, Ysample, 0.90)

# Get the coefficients ai
    print "coefficients of the linear regression model = " ,
       linearRegressionModel.getRegression()

# Get the confidence intervals of the ai coefficients
    print "confidence intervals of the coefficients = " , linearRegressionModel.
       getConfidenceIntervals()

# Get the p values of the (n+1) coefficients ai:
    print "p-value of each coefficient = " , linearRegressionModel.getPValues()

# Evaluate the predictions on the sample particularXSample
    print "predicted values on particularXSample = " , linearRegressionModel.
       getPredict(particularXSample)

# Get the residuals
    print "residuals values = " , linearRegressionModel.getResidual(Xsample,
       Ysample)

# GRAPH 1 : Validate the model with a visual test :
# superposition of clouds (Xsample, Ysample)
# ONLY if Xsample is a SCALAR numerical sample
# + linear regression model
    # Create the graph structure
    linearRegressionGraph =  VisualTest.DrawLMVisualTest(Xsample, Ysample,
       linearRegressionModel)

    # Impose a bounding box : x-range and y-range
    # boundingBox = [xmin, xmax, ymin, ymax]
    myBoundingBox = NumericalPoint(4)
    myBoundingBox[0] = xmin
    myBoundingBox[1] = xmax
    myBoundingBox[2] = ymin
    myBoundingBox[3] = xmax
    linearRegressionGraph.setBoundingBox(myBoundingBox)

    # In order to see the graph whithout creating the associated files
    Show(linearRegressionGraph)

    # Draw the graph on the file linearRegressionModel.png and
       linearRegressionModel.eps
    # if the file adress is not fulfilled , the file is created in the current
       directory
    linearRegressionGraph.draw("linearRegressionModel")

    # View the bitmap file
    ViewImage(linearRegressionGraph.getBitmap())

    # Check if it worked
    print "bitmap = " , linearRegressionGraph.getBitmap()
    print "postscript = " , linearRegressionGraph.getPostscript()

# GRAPH 2 : Draw the graph of the residual values
# couples (residual i, residual i+1)
# ONLY if Xsample is a SCALAR numerical sample
    # Create the graph structure
    residualValuesGraph = VisualTest.DrawLMResidualTest(Xsample, Ysample,
        linearRegressionModel)

    # Impose a bounding box : x-range and y-range
    # boundingBox = [xmin, xmax, ymin, ymax]
    myBoundingBox = NumericalPoint(4)
    myBoundingBox[0] = xmin
    myBoundingBox[1] = xmax
    myBoundingBox[2] = ymin
    myBoundingBox[3] = xmax
    linearRegressionGraph.setBoundingBox(myBoundingBox)

    # In order to see the graph whithout creating the associated files
    Show(residualValuesGraph)

    # Draw the graph on the file residualGraph.png and residualGraph.eps
    # if the file adress is not fulfilled, the file is created in the current
        directory
    residualValuesGraph.draw("residualGraph")

    # View the bitmap file
    ViewImage(residualValuesGraph.getBitmap())

    # Check if it worked
    print "bitmap = " , residualValuesGraph.getBitmap()
    print "postscript = " , residualValuesGraph.getPostscript()

# LMRSquared Test tests the quality of the linear regression model.
# It evaluates the R^2 indicator (regression variance analysis)
# and compares it to a level
    # H0 = R^2 > level
    # Test = True <=> R^2 > level
    # p-value threshold : level CARE : it is NOT a probability here!
    # p-value : R^2 CARE : it is NOT a probability here!
    # Test = True <=> p-value > p-value threshold

    # The two following tests must be equal :
```

```
    # Test 1 : We don't give the linear model wich is evaluated and then tested
    resultLMRSquared1 = LinearModelTest.LMRSquared(sampleX, sampleY, 0.90)

    # Test 2 : We give the regression linear model evaluated previously
    resultLMRSquared2 = LinearModelTest.LMRSquared(sampleX, sampleY,
        linearRegressionModel, 0.90)

    # Print result of the LMRSquared Test
    print "Test Succes ? ", (resultLMRSquared1.getBinaryQualityMeasure()==1)

    # Get the p-value of the LMRSquared Test
    # CARE : it is NOT a probability here! but the R^2 value
    print "p-value of the LMRSquared Test = ", resultLMRSquared1.getPvalue()

    # Get the p-value threshold of the LMRSquared Test
    # CARE : it is NOT a probability here! but the level=0.90 here
    print "p-value threshold = ", resultLMRSquared1.getThreshold()


# LMAdjustedRSquared Test tests the quality of the linear regression model.
# It evaluates the adjusted R^2 indicator (regression variance analysis)
# and compare it to a level
    # H0 = adjusted aR^2 > level
    # Test = True <=> adjusted R^2 > level
    # p-value threshold : level CARE : it is NOT a probability here!
    # p-value : adjusted R^2 CARE : it is NOT a probability here!
    # Test = True <=> p-value > p-value threshold

    # The two tests must be equal
    # We don't give the linear model wich is evaluated and then tested
    resultLMAdjustedRSquared1 = LinearModelTest.LMAdjustedRSquared(sampleX,
        sampleY, 0.90)

    # We give the regression linear model evaluated previously
    resultLMAdjustedRSquared2 = LinearModelTest.LMAdjustedRSquared(sampleX,
        sampleY, linearRegressionModel, 0.90)

    # Print result of the LMAdjustedRSquared Test
    print "Test Succes ? ", (resultLMAdjustedRSquared1.getBinaryQualityMeasure()
        ==1)

    # Get the p-value of the LMAdjustedRSquared Test
    # CARE : it is NOT a probability here! but the R^2 value
    print "p-value of the LMAdjustedRSquared Test = ", resultLMAdjustedRSquared1
        .getPvalue()

    # Get the p-value threshold of the LMAdjustedRSquared Test
    # CARE : it is NOT a probability here! but the level=0.90 here
    print "p-value threshold = ", resultLMAdjustedRSquared1.getThreshold()

# LMFisher Test tests the nullity of the regression linear model coefficients (
    Fisher distribution used).
    # H0 = the linear relation coefficients are those evaluated by the linear
        regresion
    # Test = True <=> the linear relation coefficients are those evaluated by
        the linear regresion
    # p-value threshold : probability of the H0 reject zone : 1-0.90
    # p-value : probability (test variable decision > test variable decision
        evaluated on the samples)
    # Test = True <=> p-value > p-value threshold

    # The two tests must be equal
    # Test 1 : We don't give the linear model wich is evaluated and then tested
    resultLMFisher1 = LinearModelTest.LMFisher(sampleX, sampleY, 0.90)

    # Test 2 : We give the regression linear model evaluated previously
    resultLMFisher2 = LinearModelTest.LMFisher(sampleX, sampleY,
        linearRegressionModel, 0.90)

    # Print result of the LMFisher Test
    print "Test Succes ? ", (resultLMFisher1.getBinaryQualityMeasure()==1)

    # Get the p-value of the  LMFisherTest
    print "p-value of the LMFisher Test = ", resultLMFisher1.getPvalue()

    # Get the p-value threshold of the LMFisher Test
    print "p-value threshold = ", resultLMFisher1.getThreshold()

# LMResidualMean Test tests, under the hypothesis of a  gaussian sample, if the
    mean of the residual is equal to zero. It is based on the Student test (
    equality of mean for two gaussian samples).
    # H0 = the residuals have a mean equal to zero
    # Test = True <=> the residuals have a mean equal to zero
    # p-value threshold : probability of the H0 reject zone : 1-0.90
    # p-value : probability (test variable decision > test variable decision
        evaluated on the samples)
    # Test = True <=> p-value > p-value threshold

    # The two tests must be equal
    # Test 1 : We don't give the linear model wich is evaluated and then tested
    resultLMResidualMean1 = LinearModelTest.LMResidualMean(sampleX, sampleY,
        0.90)

    # Test 2 : We give the regression linear model evaluated previously
    resultLMResidualMean2 = LinearModelTest.LMResidualMean(sampleX, sampleY,
        linearRegressionModel, 0.90)

    # Print result of the LMResidualMean Test
    print "Test Succes ? ", (resultLMResidualMean1.getBinaryQualityMeasure()==1)

    # Get the p-value of the  LMResidualMeanTest
    print "p-value of the LMResidualMean Test = ", resultLMResidualMean1.
        getPvalue()

    # Get the p-value threshold of the LMResidualMean Test
    print "p-value threshold = ", resultLMResidualMean1.getThreshold()
```

The following figures draw the regression model superposed on the samples cloud ($Xsample$, $Ysample$) of size 10^3 and the residuals graph in both cases :

- where the regression model seems validated : Figures 50 and 51,
- where the regression model doesn't seem to be validated (relation of kind $Y = X^2$) : Figures 52 and 53.
- where the regression model doesn't seem to be validated (relation of kind $Y = sin(X)$) : Figures 54 and 55.

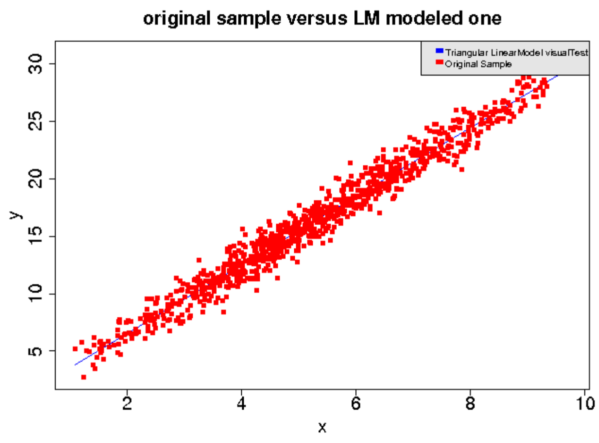


Figure 50: Visual validation of the Linear Regression Model.

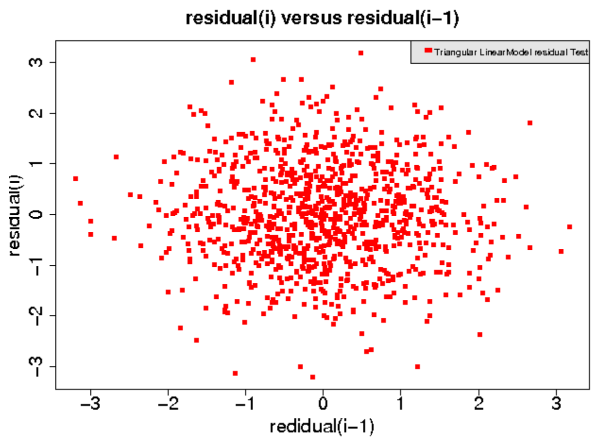


Figure 51: Visual Validation of the Linear Regression Model : residuals graph.

1.2.12 UC : Statistical manipulations on data : min, max, covariance, skewness, kurtosis, quantile, empirical CDF, Pearson, Kendall and Spearman correlation matrixes and rank/sort functionnalities

The objective of this UC is to describe the main statistical functionalities that Open TURNS enables to manipulate some data, represented by a NumericalSample.

Open TURNS enables to calculate per components :

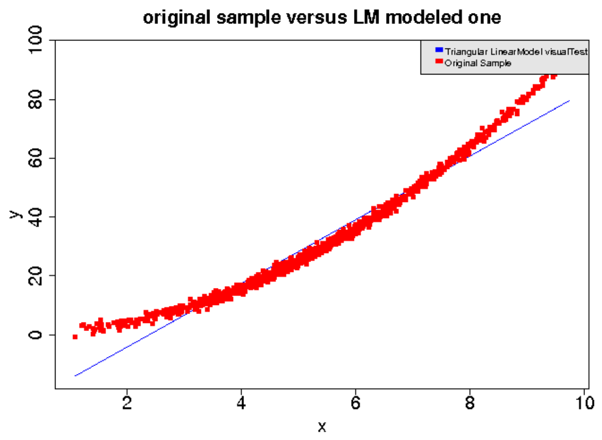


Figure 52: Visual invalidation of the Linear Regression Model.

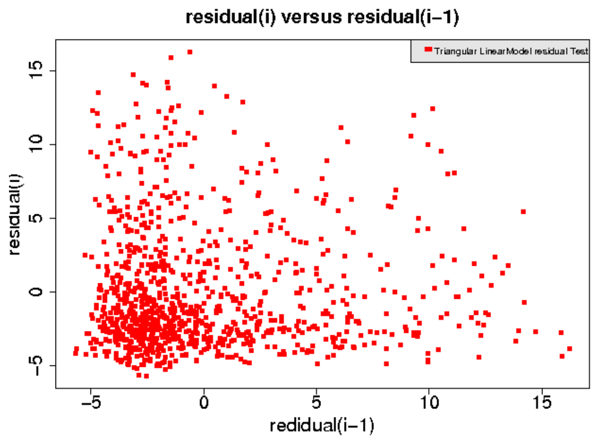


Figure 53: Visual invalidation of the Linear Regression Model : residuals graph.

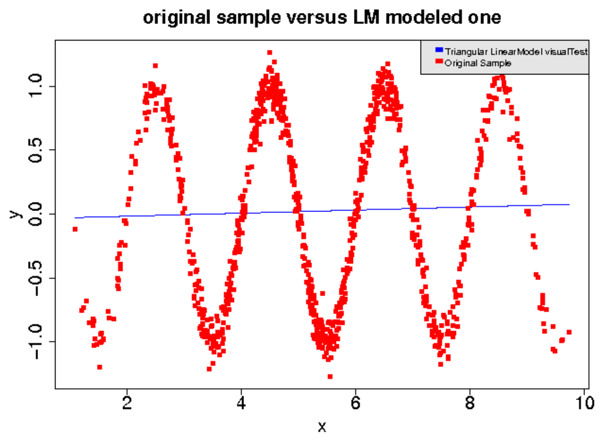


Figure 54: Visual invalidation of the Linear Regression Model.

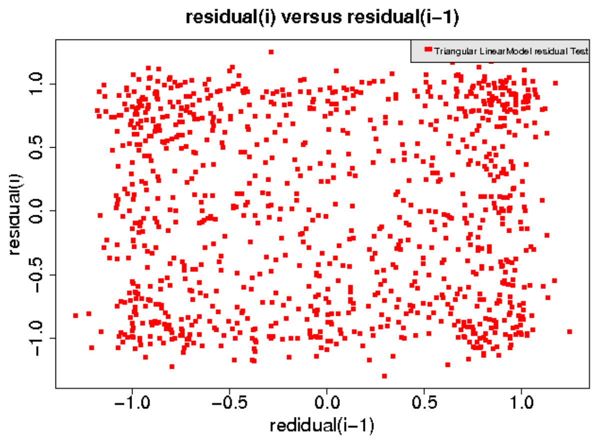


Figure 55: Visual invalidation of the Linear Regression Model : residuals graph.

- min and max per component, with the methods *getMin, getMax*
- range per component, with the method *computeRangePerComponent*
- mean, variance, standard deviation , skewness and kurtosis per component, with the methods *computeMean, computeVariancePerComponent, computeStandardDeviationPerComponent, computeSkewnessPerComponent, computeKurtosisPerComponent*
- empirical median and other quantiles per component, with the methods *computeMedianPerComponent, computeQuantilePerComponent*

Open TURNS enables some global calculs :

- covariance of the sample, with the methods *computeCovariance*
- standard deviation of the sample : the Cholesky factor of the covariance matrix, with the methods *computeStandardDeviation*
- Pearson, Kendall and Spearman correlation matrix, with the methods *computePearsonCorrelation, computeKendallTau, computeSpearmanCorrelation*
- empirical CDF evaluated on a point, with the methods *computeEmpiricalCDF*
- empirical quantiles, with the method *computeQuantile.*

At last, it is possible :

- to copy into a NumericalSample whose components are the respective ranks of the components, with the method *rank*,
- to copy into a NumericalSample whose components are all sorted in ascending order, with the method *sort* ,
- to extract the $(i + 1)$ component whose components are all sorted in ascending order, with the method *sort(i)* ,
- to copy into a NumericalSample whose NumericalPoints are reordered such that the $(i + 1)$ component is sorted in ascending order, with the method *sortAccordingAComponent(i)*,
- to keep from the Numericalsample only the i first points, with the method *split(i)*,
- to translate the points of the NumericalSample, with the method *translate* ,
- to multiply all the components of the points by a factor, with the method *scale*,
- to remove a particular point from the NumericalSample, with the method *erase*.

Requirements	• a numerical sample : *sample* **type** : NumericalSample
Results	• statistical elements listed previously **type** : NumericalPoint, SquareMatrix or CorrelationMatrix

Python script for this UseCase :

```
# Get min and max per component
    print "Min per component =" , sample.getMin()
    print "max per component =" , sample.getMax()

# Get the range per component
    print "Range per component =" , sample.computeRangePerComponent()

# Get the mean per component
    print "Mean =" , sample.computeMean()

# Get the standard deviation per component
    print "Standard deviation per component =" , sample.
        computeStandardDeviationPerComponent()

# Get the Variance per component
    print "Variance =" , sample.computeVariancePerComponent()

# Get the Skewness per component
    print "Skewness =" , sample.computeSkewnessPerComponent()

# Get the Kurtosis per component
    print "Kurtosis =" , sample.computeKurtosisPerComponent()

# Get the median per component
    print "Median per component =" , sample.computeMedianPerComponent()

# Get the empirical 0.95 quantile per component
    print "0.95 quantile per component =" , sample.computeQuantilePerComponent
        (0.95)

# Get the sample covariance
    print "Covariance =" , sample.computeCovariance()

# Get the sample standard deviation
    print "Standard deviation =" , sample.computeStandardDeviation()

# Get the sample  Pearson correlation matrix
    print "Pearson correlation =" , sample.computePearsonCorrelation()

# Get  the sample Kendall  correlation matrix
    print "Kendall correlation =" , sample.computeKendallCorrelation()

# Get  the sample Spearman  correlation matrix
    print "Spearman correlation =" , sample.computeSpearmanCorrelation()

```

```
# Get the value of the empirical CDF at point POINT
    POINT = sample.computeQuantilePerComponent(0.25)
    print "Empirical CDF at point POINT = " , sample.computeEmpiricalCDF(POINT)

# Get the empirical 0.95 quantile
    print "0.95 quantile  =" , sample.computeQuantile(0.95)
```

To illustrate each method, we give here an example in dimension 2 : consider the following NumericalSample $numSample = [(1.3, 1.2); (4.1, 1.0); (2.3, 2.7)]$. Then,
At last, it is possible :

- $new = numSample.rank()$: $new = [(0, 1); (2, 0); (1, 2)]$,
- $new = numSample.sort()$: $new = [(1.3, 1.0); (2.3, 1.2); (4.1, 2.7)]$,
- $new = numSample.sort(0))$: $new = [(1.3); (2.3); (4.1)]$,
- $new = numSample.sortAccordingAComponent(1)$: $new = [(4.1, 1.0); (1.3, 1.2); (2.3, 2.7)]$,
- $new = numSample.split(2)$: $new = [(2.3, 2.7)]$ and $numSample = [(1.3, 1.2); (4.1, 1.0)]$,
- $new = numSample.translate(NumericalPoint(2, 1.0)$: $new = [(2.3, 2.2); (4.1, 2.0); (3.3, 3.7)]$,
- $new = numSample.scale(NumericalPoint(2, 2.0)$: $new = [(2.6, 2.4); (8.2, 2.0); (4.6, 5.4)]$,
- $new = numSample.erase(1)$: $new = [(1.3, 1.2); (2.3, 2.7)]$.

1.2.13 UC : Drawing one cloud

The objective of this UC is to draw on a graph one point cloud of dimension 2.

Requirements	• one numerical sample of dimension 2 : *sample* **type** : NumericalSample
Results	• the files containing the cloud graph : *Graph_Cloud_OT.png, Graph_Cloud_OT.eps* **type** : files at format PNG or EPS or FIG

Python script for this UseCase :

```
# Create an empty graph
    myGraph = Graph("Sample", "x1", "x2", True, "topright")
    print "myGraph=" , myGraph

# Create the cloud Drawable
    # cloud : filled squares in blue
    myCloud = Cloud(sample, "blue", "fsquare","First Cloud")
    print "myCloud=" , myCloud

# Then, add it in the empty graph
    myGraph.addDrawable(Drawable(myCloud1))

# Impose a bounding box : x-range and y-range
# boundingBox = [xmin, xmax, ymin, ymax]
    myBoundingBox = NumericalPoint(4)
    myBoundingBox[0] = xmin
    myBoundingBox[1] = xmax
    myBoundingBox[2] = ymin
    myBoundingBox[3] = xmax
    myGraph.setBoundingBox(myBoundingBox)

# In order to see the graph whithout creating the associated files
    Show(myGraph)

# Draw the graph containing the cloud
    myGraph.draw("Graph_Cloud_OT")

# View the bitmap file
    ViewImage(myGraph.getBitmap())

# Check if it worked
    print "bitmap=" , myGraph.getBitmap()
    print "postscript=" , myGraph.getPostscript()
```

The following Figure (56 draw the superposition of two clouds of dimension 2 and size 1000, realisations of

- a Normal distribution with $\underline{0}$ mean, unit standard deviation and independant components,
- a Normal distribution with unit-mean, unit-standard deviation and independant components.

2 Creation of the limit state function and the output variable of interest

The objective of the section is to specify the limit state function and the output variable of interest, defined from the limit state function.
It corresponds to the step 'Step A : Specify the output variable of interest' of the global methodology.

2.1 Creation of the limit state function

2.1.1 UC : From an external wrapper with gradient and hessian implementations

The objective of this UC is to specify the limit state function, defined through an external wrapper .
The example here is the wrapper *poutre.xml* which contains the implementations of :

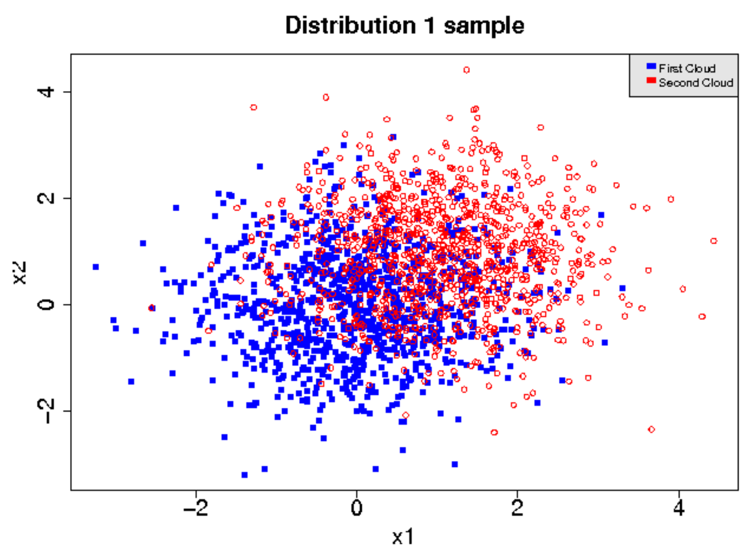


Figure 56: Superposition of two normal NumericalSample of dimension 2.

- the function *func_exec_compute_deviation*,
- its gradient *grad_exec_compute_deviation* and
- its hessian *hes_exec_compute_deviation*.

It is necessary to refer to the documentation *Open TURNS - Wrappers Guide* to have explanations on what constitues an Open TURNS wrapper.

Requirements	• wrapper of the limit state function *poutre.xml*
Results	• the limit state function : *poutre*(*) **type** : NumericalMathFunction

(*) :

$$poutre : \left| \begin{array}{lcl} \mathbb{R}^4 & \rightarrow & \mathbb{R} \\ (E,F,L,I) & \mapsto & y_0 = \dfrac{FL^3}{3EI} \end{array} \right. \quad (7)$$

Python script for this UseCase :

```
# Create the limit state function 'poutre' from the wrapper 'poutre'
    poutre = NumericalMathFunction("poutre")
```

2.1.2 UC : From an analytical formula declared in line

The objective of this UC is to specify the limit state function, defined through an analytical formula declared in line. Open TURNS automatically gives to the analytical formula an implementation for the gradient and the hessian : by default,

- the gradient evaluation method is the centered finite difference method, with the differential increment $h = 1e - 5$ for each direction,
- the hessian evaluation method is the centered finite difference method, with the differential increment $h = 1e - 4$ for each direction.

it is possible to change the evaluation method for the gradietn or the hessian. The following Use Case shows how to proceed.

The example here is the AnalyticalFunction *myAnalyticalFunction* defined by the formula :

$$myAnalyticalFunction : \left| \begin{array}{ccl} \mathbb{R}^2 & \to & \mathbb{R} \\ (x_0, x_1) & \mapsto & y_0 = -(6 + x_0^2 - x_1) \end{array} \right.$$

Requirements	none
Results	• the analytical limit state function : *myAnalyticalFunction* **type** : NumericalMathFunction

Python script for this UseCase :

```
# Describe the input vector of dimension 2
    inputFunc = Description(2)
    inputFunc[0] = "x0"
    inputFunc[1] = "x1"

# Describe the output vector of dimension 1
    outputFunc = Description(1)
    outputFunc[0] = "Output Variable of Interest 1"

# Give the formulas
    formulas = Description(outputFunc.getSize())
    formulas[0] = "-(6 - x1 + x0^2)"
    print "formulas=" , formulas

# Create the analyticalfunction 'myFunction'
    myAnalyticalFunction = NumericalMathFunction(inputFunc, outputFunc, formulas
       )

# Change the gradient evaluation method
# (some algorithms need it)
    # Type : non centered finite difference method
    myGradient = NonCenteredFiniteDifferenceGradient(NumericalPoint(2, 1.0e-7),
        myAnalyticalFunction.getEvaluationImplementation())
    print "myGradient = ", myGradient

    # Substitute the gradient
    myAnalyticalFunction.setGradientImplementation(myGradient)

# Change the hessian evaluation method
    # type : non centered finite difference method
    myHessian = CenteredFiniteDifferenceHessian(NumericalPoint(2, 1.0e-7),
        myAnalyticalFunction.getEvaluationImplementation())
    print "myHessian = ", myHessian

    # Substitute the hessian
    myAnalyticalFunction.setHessianImplementation(myHessian)

# Check if it worked
    x = NumericalPoint(myAnalyticalFunction.getInputNumericalPointDimension())
    x[0] = 1.0
    x[1] = 2.0
    print "myAnalyticalFunction(", x[0], ",", x[1], ")=", myAnalyticalFunction(x
        )
```

2.1.3 UC : Introducing some deterministic variables, using a LinearNumericalMathFunction

We suppose that the following limit state function *limitStateFunc* has been created in Open TURNS :

$$limitStateFunc : \left| \begin{array}{ccc} \mathbb{R}^n & \rightarrow & \mathbb{R}^p \\ \underline{X} & \mapsto & limitStateFunc(\underline{X}) \end{array} \right.$$

Suppose now that some of the input variables are deterministic : the random input vector is reduced to a subvector of $\underline{X}$: $\underline{X}_{prob} \in \mathbb{R}^{n_{prob}}$, with $n_{prob} \leq n$.
Let's note $\underline{X} = (\underline{X}_{prob}, \underline{X}_{det})$.

In order to create the new limit state function associated to the random input vector $\underline{X}_{prob}$, it is necessary to compose the initial limit state function *limitStateFunc* with the linear function *increase* defined by :

$$increase : \left| \begin{array}{ccc} \mathbb{R}^{n_{prob}} & \rightarrow & \mathbb{R}^n \\ \underline{X}_{prob} & \mapsto & increase(\underline{X}_{prob}) = \underline{\underline{A}}\underline{X}_{prob} + \underline{B} \end{array} \right.$$

where $\underline{\underline{A}}$ is the matrix in $\mathbb{M}_{n,n_{prob}}(\mathbb{R})$ defined by :

$$\underline{\underline{A}} = \left(\begin{array}{c} 1_{n_{prob}} \\ 0 \end{array} \right)$$

and $\underline{B}$ the vector in $\mathbb{R}^n$ defined by :

$$\underline{B} = \left(\begin{array}{c} \underline{0} \\ \underline{X}_{det} \end{array} \right)$$

Then, the new limit state function associated to the random input vector $\underline{X}_{prob}$ is

$$newLimitStateFunc = limitStateFunc \circ increase$$

defined by :

$$newLimitStateFunc : \left| \begin{array}{ccc} \mathbb{R}^{n_{prob}} & \rightarrow & \mathbb{R}^p \\ \underline{X}_{prob} & \mapsto & newLimitStateFunc(\underline{X}_{prob}) \end{array} \right.$$

The example here is the limit state function *poutre* defined in Eq.(7) and the random input vector (E, F, L, I) that is reduced to the subvector (E, F). The other variables (L, I) are fixed to $(10.0, 5.0)$.

Requirements	• the initial limit state function : *poutre* **type** : LinearNumericalMathFunction ($\mathbb{R}^4 \rightarrow \mathbb{R}$)
Results	• the *increase* function **type** : NumericalMathFunction ($\mathbb{R}^2 \rightarrow \mathbb{R}^4$) • the new limit state function : *poutreReduced = poutre* $\circ$ *increase* **type** : NumericalMathFunction ($\mathbb{R}^2 \rightarrow \mathbb{R}$)

Python script for this UseCase :

```
# Dimension of the random input vector
    stochasticDimension = 2

# Dimension of the deterministic input vector
    deterministicDimension = 2

# Dimension of the input vector of the limit state function 'poutre'
    inputDim = poutre.getInputNumericalPointDimension()

# Fixe deterministic values for the two last variables
# of the input vecteor (E,F,L,I)
    # L
    X2 = 10.0
    # I
    X3 = 5.0

# Create the 'increase' linear function
    # a LinearNumericalMathFunction expression is :
    # linear * (X- center) + constant
    # center = null
    center = NumericalPoint(stochasticDimension)

    # constant term = (0.0, 0.0, X2, X3)^t
    constant = NumericalPoint(inputDim)
    constant[0] =  0.0
    constant[1] =  0.0
    constant[2] =  X2
    constant[3] =  X3

    # Linear term (lines number, columns number)
    linear = Matrix(inputDim,  stochasticDimension)
    linear[0,0] = 1.0
    linear[0,1] = 0.0
    linear[1,0] = 0.0
    linear[1,1] = 1.0
    linear[2,0] = 0.0
    linear[2,1] = 0.0
    linear[3,0] = 0.0
    linear[3,1] = 0.0

    # 'increase' = linear * (X- center) + constant
    increase = LinearNumericalMathFunction(center, constant, linear, "increase")

# Create the new limit state function :
# 'poutreReduced = poutre o increase'
    poutreReduced = NumericalMathFunction(poutre, increase)

# Check if it worked
    x = NumericalPoint(increase.getInputNumericalPointDimension()
    x[0] = 50.0
    x[1] = 1.0
    print "poutreReduced(x)=", poutreReduced(x)
    xRef = NumericalPoint(inputDim)
    xRef[0] = x[0]
    xRef[1] = x[1]
    xRef[2] = X2
    xRef[3] = X3
    print "ref=", externalCode(xRef)
```

2.1.4 UC : Introducing some deterministic variables, optimising memory and CPU time

Let's have the same context than in the UC2.1.3. The idea here is to avoid the introduction of the potentially huge matrix $\underline{\underline{A}}$ and the gradient matrix and hessian tensor of the functions *increase* and *poutre*. For that last problem, it is sufficient to define the gradient matrix and hessian tensor to the final function *poutreReduced* from a finite difference technique.

The function *increase* is defined as follows :

$$increase : \left| \begin{array}{lcl} \mathbb{R}^{n_{prob}} & \rightarrow & \mathbb{R}^{n} \\ \underline{X}_{prob} = \left| \begin{array}{l} "inputProb1" \\ \cdots \\ "inputProbNprob" \end{array} \right. & \mapsto & increase(\underline{X}_{prob}) = \left| \begin{array}{l} "inputProb1" \\ \cdots \\ "inputProbNprob" \\ valDet1 \\ \cdots \\ valDetNdet \end{array} \right. \end{array} \right.$$

where all the $(valDet1, ..., valDetNdet)$ are the n_{det} values of the determinist components of $\underline{X}$.

The same example is re-written in the folloing Use Case.

Requirements	• the initial limit state function : *poutre* **type** : LinearNumericalMathFunction ($\mathbb{R}^4 \rightarrow \mathbb{R}$)
Results	• the *increase* function **type** : NumericalMathFunction ($\mathbb{R}^2 \rightarrow \mathbb{R}^4$) • the new limit state function : *poutreReduced = poutre ∘ increase* **type** : NumericalMathFunction ($\mathbb{R}^2 \rightarrow \mathbb{R}$)

Python script for this UseCase :

```
# Dimension of the random input vector
    stochasticDimension = 2

# Dimension of the deterministic input vector
    deterministicDimension = 2

# Dimension of the input vector of the limit state function 'poutre'
    inputDim = poutre.getInputNumericalPointDimension()

# Fixe deterministic values for the two last variables
# of the input vecteor (E,F,L,I)
    # L
    X2 = 10.0
    # I
    X3 = 5.0

# Create the 'increase' function
  # Describe the input vector of dimension 2
    inputIncrease = Description(2)
    inputIncrease[0] = "E"
    inputIncrease[1] = "F"

# Describe the output vector of dimension 1
    outputIncrease = Description(4)
    outputIncrease[0] = "E"
    outputIncrease[1] = "F"
    outputIncrease[2] = "L"
    outputIncrease[3] = "I"

# Give the formulas
    formulas = Description(4)
    formulas[0] = "E"
    formulas[1] = "F"
    formulas[2] = X2
    formulas[3] = X3
    print "formulas=" , formulas

# Create the analyticalfunction 'increase'
    increase = NumericalMathFunction(inputIncrease, outputIncrease, formulas)

# Create the new limit state function :
# 'poutreReduced = poutre o increase'
    poutreReduced = NumericalMathFunction(poutre, increase)

# Give directly to the 'poutreReduced' function a gradient evaluation method
# thanks to the finite difference technique
    # For example, radient technique : non centered finite difference method
    myGradient = NonCenteredFiniteDifferenceGradient(NumericalPoint(2, 1.0e-7),
        poutreReduced.getEvaluationImplementation())
    print "myGradient = ", myGradient

    # Substitute the gradient
    poutreReduced.setGradientImplementation(myGradient)

# Give directly to the 'poutreReduced' function a hessian evaluation method
# thanks to the finite difference technique
    # type : non centered finite difference method
    myHessian = CenteredFiniteDifferenceHessian(NumericalPoint(2, 1.0e-7),
        poutreReduced.getEvaluationImplementation())
    print "myHessian = ", myHessian

    # Substitute the hessian
    poutreReduced.setHessianImplementation(myHessian)
```

2.1.5 UC : Manipulation of a NumericalMathFunction

The objective of this UC is to describe the main functionalities that Open TURNS enables to manipulate a numerical function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^p$.

Open TURNS enables :

- to ask the dimension of its input and output vectors, with the methods *getInputDimension, getOutputDimension*,
- to evaluate itself, its gradient and hessian, with the methods *gradient, hessian.* The evaluation of the function is a vector of dimension p, the gradient is a matrix with p rows and n columns, the hessian is a tensor of order 3 with p rows, n columns and n sheets,
- to evaluate the number of times the function or its gradient or its hessian have been evaluated **since the beginning of the python session**, with the methods *getEvaluationCallsNumber, getGradientCallsNumber, getHessianCallsNumber*,
- to ask the description of its input and output vectors, with the methods *getInputDescription, getOutputDescription*,
- to extract its components if $p > 1$, wich are functions $f_i : \mathbb{R}^n \longrightarrow \mathbb{R}$, with the method *getMarginal*,
- to ask for its parameters with the method *getParameters*,
- to define its parameters, with the method *setParameters*,
- to compose two functions,
- to ask for the valid operators in Open TURNS, the valid constants and functions, with the methods *GetValidOperators, GetValidConstants, GetValidFunctions.*

Requirements	
Results	• a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^p$: *myFunction* **type** : NumericalMathFunction

Python script for this UseCase :

```

# Ask for the dimension of the input and output vectors
    print myFunction.getInputDimension()
    print myFunction.getOutputDimension()

# Evaluate the function at a particular point
    point = NumericalPoint(myFunction.getInputDimension())
    functinovector = myFunction(point)

# Evaluate the gradient of the function at a particular point
    gradientMatrix = myFunction.gradient(point)

# Evaluate the hessian of the function at a particular point
    hessianMatrix = myFunction.hessian(point)

```

```
# Get the number of times the function has been evaluated
    callsNumber = myFunction.getEvaluationCallsNumber()

# Get the number of times the gradient has been evaluated
    callsNumber = myFunction.getGradientCallsNumber()

# Get the number of times the hessian has been evaluated
    callsNumber = myFunction.getHessianCallsNumber()

# Get the description of its input and output vectors
    print myFunction.getInputDescription()
    print myFunction.getOutputDescription()

# Get the component i
# Care : the numerotation begins at 0
    i=3
    component = myFunction.getMarginal(i)

# Get the parameters of the function
    paremeters =  myFunction.getParameters()

# Set the parameters of the function
    myFunction.setParameters()

# Compose the two NumericalMathFunction : h=fog
   g=NumericalMathFunction(f,g)

# Get the valid operators in Open TURNS
   print NumericalMathFunction.GetValidOperators()

# Get the valid functions in Open TURNS
   print NumericalMathFunction.GetValidFunctions()

# Get the valid constants in Open TURNS
   print NumericalMathFunction.GetValidConstants()
```

2.2 Creation of the output variable of interest from the limit state function and the random input vector

The objective of the section is to determine the output variable of interest directly from a limit state function and a random input vector declared previously.

2.2.1 UC : Creation of the ouput random vector

We suppose in that section that the random input vector is exactly the entry vector of the limit state function.

Requirements	• the limit state function : *myFunction* **type** : NumericalMathFunction • the random input vector : *inputVector* **type** : RandomVector which implementation is a UsualRandomVector
Results	• the output variable of interest *output = myFunction(input)* **type** : RandomVector which implementation is a CompositeRandomVector

Python script for this UseCase :

```
# Create the output variable of interest 'output = poutre(input)'
    output = RandomVector(myFunction, input)

# Name the output variable of interest
# for example, it is of dimension 1
    outputDescription = Description(dim)
    outputDescription[0] = "Output_Variable_Of_Interest_1"
    output.setDescription(outputDescription)
```

2.2.2 UC : Extraction of a random subvector from a random vector

The objective of this UC is to extract a subvector from a random vector which has been defined as well as a UsualRandomvector (it means thanks to a distribution, see UC. 1.1.7) than as a CompositeRandomVector (as the image through a limit state function of an input random vector, see UC. 2.2.1).

Let's note $\underline{Y} = (Y_1, \cdots, Y_n)$ a random vector and $I \subset [1, n]$ a set of indices :

- In the first case, the subvector is defined by $\tilde{\underline{Y}} = (Y_i)_{i \in I}$,
- In the second case, where $\underline{Y} = f(\underline{X})$ with $f = (f_1, \cdots, f_n)$, f_i some scalar functions, the sub vector is $\tilde{\underline{Y}} = (f_i(\underline{X}))_{i \in I}$.

Requirements	• the random vector : *myRandomVector* **type** : RandomVector wich implementation is a UsualRandomVector or CompositeRandomVector
Results	• the extracted random vector : *myExtractedRandomVector* **type** : RandomVector which implementation is a UsualRandomVector or CompositeRandomVector

Python script for this UseCase :

```

# CASE 1 : Get the marginal of the random vector
# Corresponding to the component i

 # Care : numerotation begins at 0
 myExtractedRandomVector = myRandomVector.getMarginal(i)


# CASE 2 : Get the marginals of the random vector
# Corresponding to several components
# decribed in the myIndice table
 # For example, components 0, 1, and 5
 myIndices = Indices(3)
 myIndices[0] = 0
 myIndices[1] = 1
 myIndices[2] = 5

 myExtractedRandomVector = myRandomVector.getMarginal(myIndices)
```

3 Uncertainty propagation and Uncertainty sources ranking

The objective of this section is to manipulate all the functionalities to propagate uncertainties from the random input vector through the limit state function until the output variable of interest.
It corresponds to the step 'Step C : Propagate the uncertainties' of the global methodology.

3.1 Deterministic approach : Min/Max study

In this section, we focus on the deterministic approach which consists of researching the variation range of the output variable of interest.

3.1.1 UC : Creation of a deterministic experiment plane

Open TURNS enables to define four types of deterministic experiment planes : axial, composite, factorial and box. In order to define an experiment plane, follow the 3 steps, whatever the type of the experiment plane, where n is the dimension of the space and n_{level} the number of levels (the same for each direction) :

- Step 1 : Define a reduced and centered grid structure, centered on $\underline{0} \in \mathbb{R}^n$, by specifying the levels which will be consider on each direction,
- Step 2 : Scale each direction with a specific scale factor for each direction, in order to give a unit effect on each direction,
- Step 3 : Translate the scaled grid structure onto a specified center point.

Each experiment plane has a specific method to define its reduced and centered grid structure :

- **Axial** : the points grid is obtained by discretizing each direction according to the specified levels, symmetrically with respect to 0. The number of points generated is $1 + 2n * n_{level}$.

- **Factorial** : the points grid is obtained by discretizing each principal diagonal according to the specified levels, symmetrically with respect to 0. The number of points generated is $1 + 2^n * n_{level}$.
- **Composite** : the points grid is obtained as the union between an axial and a factorial experiment plane. The number of points generated is $1 + 2n * n_{level} + 2^n * n_{level}$.
- **Box** : the points grid is obtained by discretizing the unit pavement $[-0.5, 0.5]^n$, regularly with the number of intermediate points specified for each direction. The number of points generated is $\prod_{i=1}^{n}(2 + n_{level}(direction\ i))$.

In order to scale each direction according to a specified factor or/and to translate the points grid until a specified center, the methods *scale* and *translate* must be used.

The following example works in $\mathbb{R}^2$.

Requirements	• none
Results	• a centered and reducted grid structure : *myCenteredReductedPlane* **type** : an ExperimentPlane, which type is Axial, Composite, Factorial or Box • the numerical sample associated to the centered and reducted grid structurethen scaled then translated grid structrue : *myExperimentPlane* **type** : a NumericalSample

Python script for this UseCase :

```

# Define a scale factor for each direction
  scaledVector = NumericalPoint(2)
  scaledVector[0] = 1.5
  scaledVector[1] = 2.5

# Define the translation until the final center of the experiment plane
  translationVector = NumericalPoint(2)
  translationVector[0] = 2
  translationVector[1] = 3

# Define the different levels of the grid structure
# CARE : for the axial, composite and factorial experiment planes,
# these levels are all applied along each direction
# Here : 3 levels on each direction
  levels = NumericalPoint(3)
  levels[0] = 1
  levels[1] = 1.5
  levels[2] = 3.

# For the box experiment plane, levels specifies the number of
# intermediate points on each direction (one per direction)
# Here : direction 1 will be discretised with 2 intermediate points
# and direction 2 with 4 intermediate points
  levelsBox = NumericalPoint(2)
  levels[0] = 2
  levels[1] = 4


# STEP 1 : Define a reduced and centered grid structure

  # AXIAL structure
  myCenteredReductedGrid = Axial(2,levels)

  # COMPOSITE structure
  myCenteredReductedGrid = Composite(2,levels)

  # FACTORIAL structure
  myCenteredReductedGrid = Factorial(2,levels)

  # BOX structure
  myCenteredReductedGrid = Box(levelsBox)

  # Generate the centered and reduxted grid structure
  myExperimentPlane = myCenteredReductedGrid.generate()

  # Get the number of points of the grid structure
  # a NumericalSample is created
  pointsNumber = myExperimentPlane.getSize()


# STEP 2 : Scale each direction with a specific scale factor

  # The NumericalSample is transformed
  myExperimentPlane.scale(scaledVector)


# STEP 3 : Translate the scaled grid structure onto a specified center point

  # The NumericalSample is transformed
  myExperimentPlane.translate(translationVector)
```

Figures 57 to 68 draw the different grid structures obtained after the *scale* or *translate* methods.

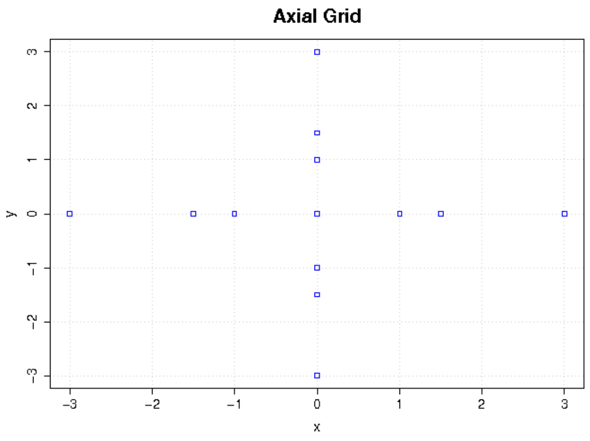


Figure 57: Axial Experiment Plane : initial grid.

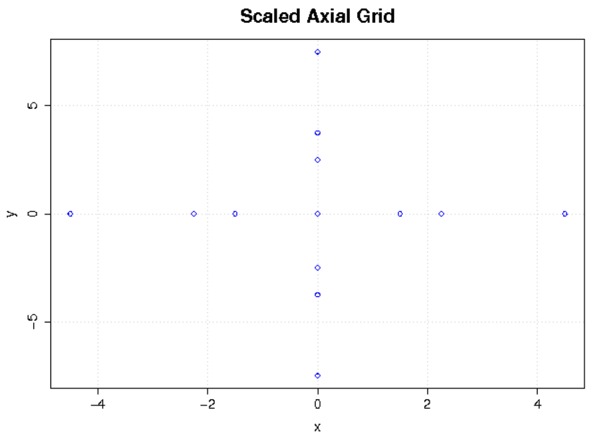


Figure 58: Axial Experiment Plane : after scaling.

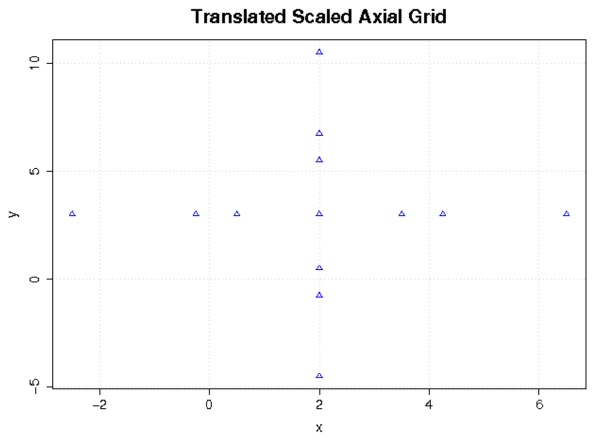


Figure 59: Axial Experiment Plane : after scaling and translation.

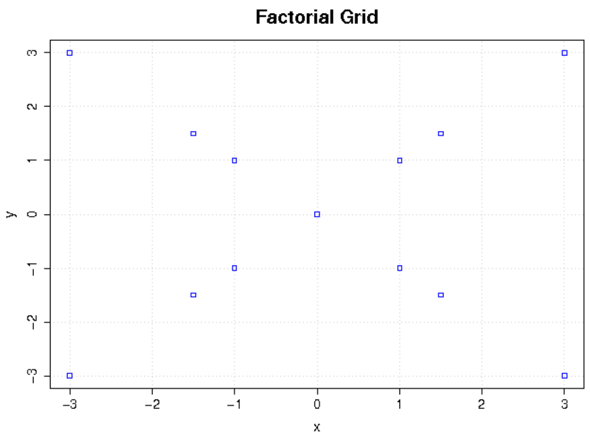


Figure 60: Factorial Experiment Plane : initial grid.

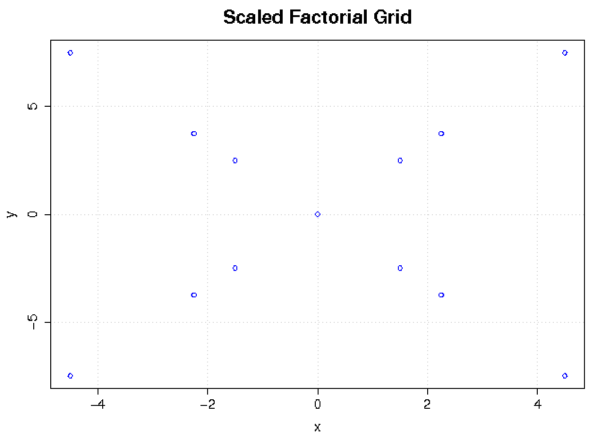


Figure 61: Factorial Experiment Plane : after scaling.

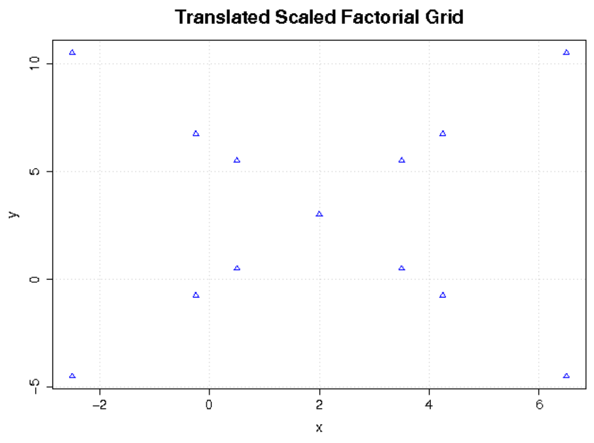


Figure 62: Factorial Experiment Plane : after scaling and translation.

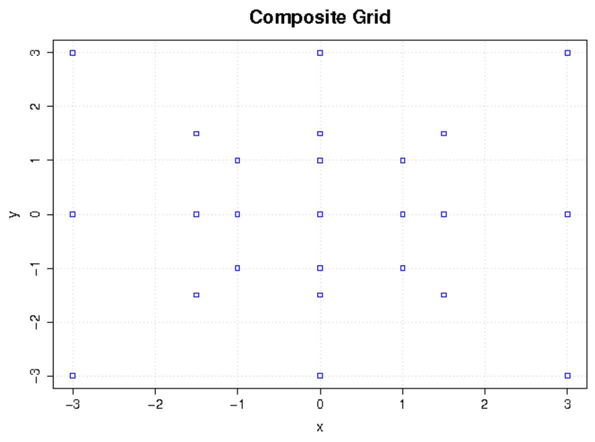


Figure 63: Composite Experiment Plane : initial grid.

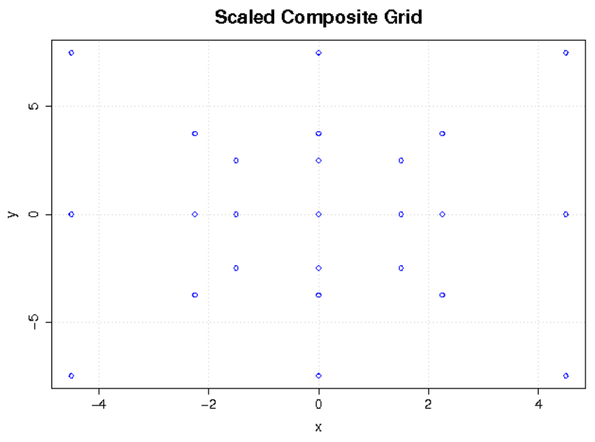


Figure 64: Composite Experiment Plane : after scaling.

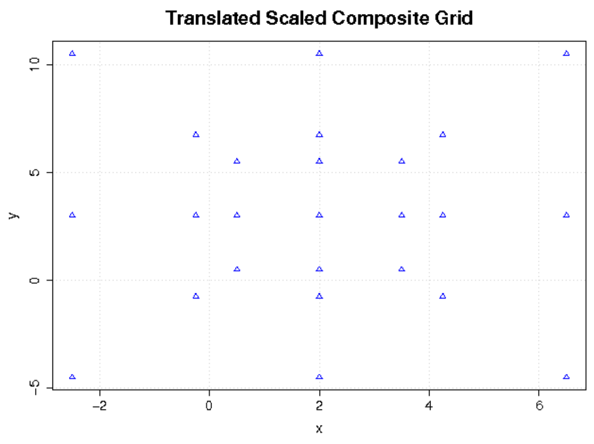


Figure 65: Composite Experiment Plane : after scaling and translation.

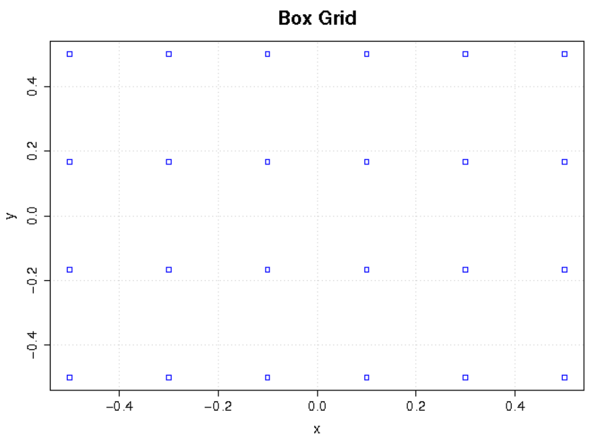


Figure 66: Box Experiment Plane : initial grid.

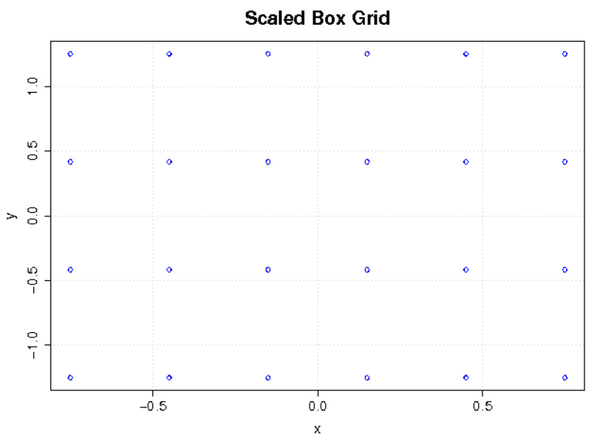


Figure 67: Box Experiment Plane : after scaling.

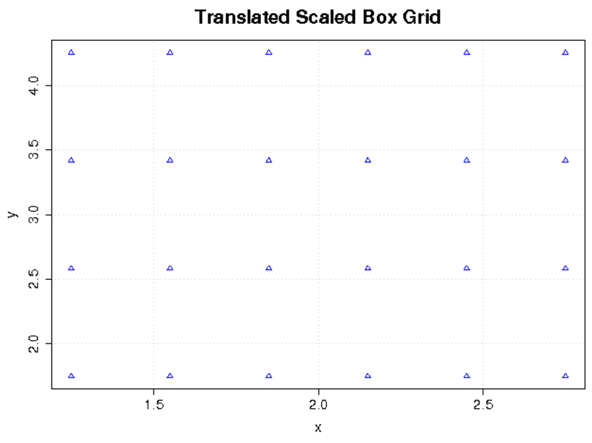


Figure 68: Box Experiment Plane : after scaling and translation.

3.1.2 UC : Drawing an experiment plane in dimension 2

This UseCase draws an experiment plane in dimension 2.

Requirements	• the points of an experiment plane : *mySample* **type** : a NumericalSample
Results	• the files containing the graph, in format .EPS, .FIG, .PNG : *experimentPlane* **type** : -

Python script for this UseCase :

```
# Draw it
  mySampleDrawable = Cloud(mySample, "blue", "square", "My experiment Plane")
  graph = Graph("My experiment Plane", "x", "y", True)
  graph.addDrawable(mySampleDrawable)
  graph.draw("experimentPlane")
  ViewImage(graph.getBitmap())

# In order to see the drawable whithout creating the associated files
  # CARE : it requires to have created the graph structure before
  Show(mySampleDrawable)
  # or to see the graph whithout creating the associated files
  Show(graph)
```

3.1.3 UC : Creation of a deterministic experiment plane in the physical space (type : Axial) where levels are proportionnal to the standard deviation of each component of the random input vector, and centered on the mean vector of the random input vector

In this Use Case, the objective is to determine the variation range of the output variable of interest from a deterministic experiment plane on the random input vector.

The example here is an axial experiment plane where levels are proportionnal to the standard deviation of each component of the random input vector, and centered on the mean vector of the random input vector.
There are three levels : +/-4, +/-8, +/-16 around a center fixed equal to the center point ($\underline{0}$).
The dilatation vector is composed of the standard deviation of each component of the random input vector.

Requirements	• the input vector : *input* **type** : RandomVector
Results	• an experiment plane : *myPlane* **type** : Axial • a sample of *input* according to *myPlane* : *inputSample* **type** : NumericalSample

Python script for this UseCase :

```
# In order to use the 'sqrt' function
    from math import *

# Dimension of the use case : 4
    dim = 4

# Give the levels of the experiment plane
    # here,  3 levels : +/-4, +/-8, +/-16
    levelsNumber = 3
    levels = NumericalPoint(levelsNumber, 0.0, "Levels")
    levels[0] = 4
    levels[1] = 8
    levels[2] = 16

# Create the axial plane centered on the vector (0)
# and with the levels 'levels'
    myPlane = Axial(dim, levels)

# Generate the points according to the structure
# of the experiment plane (in a reduced centered space)
    inputSample = myPlane.generate()

# Scale the structure of the experiment plane
# proportionnally to the standard deviation of each component
# of 'input' in case of a RandomVector
    # Scaling vector for each dimension of the levels of the structure
    # to take into account the dimension of each component
    scaling = NumericalPoint(dim, 0)
    scaling[0] = sqrt(input.getCovariance()[0,0])
    scaling[1] = sqrt(input.getCovariance()[1,1])
    scaling[2] = sqrt(input.getCovariance()[2,2])
    scaling[3] = sqrt(input.getCovariance()[3,3])
    inputSample.scale(scaling)
```

```

# Translate the nonReducedSample onto the center of the experiment plane
    # Translation vector for each dimension
    center = input.getMean()
    inputSample.translate(center)
```

3.1.4 UC : Creation of a random experiment plane

We determine the variation range of the output variable of interest from a random experiment plane on the random input vector.

The example here is the generation of a sample of size 10^2, according to the random distribution of the input vector *input*.

Before any simulation, we initialise the state of the random generator.

Requirements	• the input vector : *input* **type** : RandomVector
Results	• sample *inputSample* generated according to the distribution of *input* **type** : NumericalSample

Python script for this UseCase :

```
# Initialise the state of the random generator
  # thanks to the fonctionality SetSeed(n) where n is an UnsignedLong in [0,
      2^(32)-1]
  # which enables an easy initialisation for the user
  RandomGenerator.SetSeed(77)

  # or by specifying a complete state of the random generator : particularState
  # coming from a previous particularState = RandomGenerator.GetState() :
  # RandomGenerator.SetState(particularState)

# Get the complete state of the random generator before simulation
  randomGeneratorStateBeforeRandomExperiment = RandomGenerator.GetState()

# Generate a random sample of size 100 according to the distribution
# of the input vector 'input
  size = 100
  inputSample = input.getNumericalSample(size)
  print "inputSample = ", inputSample
```

3.1.5 UC : Min/Max research of the output variable of interest from an experiment plane in the physical space (deterministic or random) of the input random vector and deterministic sensitivity of the output variable to the input vector at a particular point

The objective of this UC is to evaluate the min and max values of the output variable of interest from an experiment plane in the physical space and to evaluate the gradient of the limit state function defining the output variable of interest at a particular point.

The example here is the limit state function *poutre* defined in Eq.(7) with the random input vector (E, F, L, I).

Requirements	• the sampled generated according to the (deterministic or random one) experiment plane of the random input vector *input* : *inputSample* **type** : NumericalSample • the limit state function : *poutre* **type** : NumericalMathFunction
Results	• the sample of the output variable of interest *output = poutre(input)* corresponding to *inputSample* : *outputSample* **type** : NumericalSample • the min and max of the output variable of interest *output* **type** : NumericalPoint • the deterministic gradient of *output* with respect to *input* at a particular point $input_0$ **type** : Matrix

Python script for this UseCase :

```
# Dimension of the use case : 4
    dim = 4

# Generate the  values of the output variable of interest
# 'output = poutre(input)' corresponding to 'inputSample'
    outputSample = poutre(inputSample)
    print "outputSample = ", outputSample

# Get the min and the max of the output variable, component by component
    min = outputSample.getMin()
    max = outputSample.getMax()
    print  "max =  ", max
    print  "min =  ", min

```

```
# Get the gradient of 'poutre' with respect to 'input'
# at a particular point 'input_0'
  input0 = NumericalPoint(dim)
  input0[0] = 50
  input0[1] = 1
  input0[2] = 10
  input0[3] = 5
  sensitivity = poutre.gradient(input0)
  print "sensitivity at point input0 = ", sensitivity
```

3.2 Random approach : central uncertainty

In this section, we focus on the random approach which aims at evaluating the central tendance of the output variable of interest.

In order to evaluate the central tendance of the output variable of interest described by a numerical sample, it is possible to use all the functionalities described in the Use Case 1.2.12.

The Use Case 3.2.1 describes the correlation analysis we can perform between the random input vector, described by a numerical sample, and the output variable of interest described by a numerical sample too.

3.2.1 UC : Correlation analysis on samples : Pearson and Spearman coefficients, PCC, PRCC, SRC, SRRC coefficients

Requirements	• a first numerical sample : *inputSample*, may be of dimension ¿1 **type** : NumericalSample • a second numerical sample : *outputSample*, must be of dimension =1 **type** : NumericalSample
Results	• the different correlation coefficients : *PCCcoefficient, PRCCcoefficient, SRCcoefficient, SRRCcoefficient, pearsonCorrelation, spearmanCorrelation* **type** : NumericalPoint

Python script for this UseCase :

```
# PCC coefficients evaluated between the outputSample and each coordinate of
    inputSample
  PCCcoefficient = CorrelationAnalysis.PCC(inputSample, outputSample)

# PRCC evaluated between the outputSample and each coordinate of inputSample (
    based on the rank values)
  PRCCcoefficient = CorrelationAnalysis.PRCC(inputSample, outputSample)
```

```

# SRC evaluated between the outputSample and each coordinate of inputSample
    SRCcoefficient = CorrelationAnalysis.SRC(inputSample, outputSample)

# SRRC evaluated between the outputSample and each coordinate of inputSample (
    based on the rank values)
    SRRCcoefficient = CorrelationAnalysis.SRRC(inputSample, outputSample)

# Pearson Correlation Coefficient
# CARE : inputSample must be of dimension 1
    pearsonCorrelation = CorrelationAnalysis.PearsonCorrelation(inputSample,
        outputSample)

# Spearman Correlation Coefficient
# CARE : inputSample must be of dimension 1
     spearmanCorrelation = CorrelationAnalysis.SpearmanCorrelation(inputSample,
         outputSample)
```

3.2.2 UC : Moments evaluation from the Taylor variance decomposition method and evaluation of the importance factors associated

The objective of this UC is to evaluate the mean and standard deviation of the output variable of interest thanks to the Taylor variance decomposition method of order one or two.

Requirements	• the random input vector : *input* **type** : RandomVector which implementation is a UsualRandomVector • the output variable of interest : *output* **type** : RandomVector which implementation is a CompositeRandomVector
Results	• Moments (order 1, 2, 3) of the variable of interest and its components **type** : NumericalPoint, Matrix • Importance factors from quadratical cumul method only for *output* of dimension 1 **type** : NumericalPoint

Python script for this UseCase :

```
# Create a quadraticCumul algorithm
     myQuadraticCumul = QuadraticCumul(output)

# Stream out the result
```

```
    print "myQuadraticCumul=", myQuadraticCumul

# Compute the several elements provided by the quadratic cumul algorithm
    # First order mean
    print "First order mean=", myQuadraticCumul.getMeanFirstOrder()
    # Second order mean
    print "Second order mean=", myQuadraticCumul.getMeanSecondOrder()
    # Covariance Matrix
    print "Covariance=", myQuadraticCumul.getCovariance()
    # Importance factors
    # CARE : for this calculus only, the output variable of interest must be of
        dimension 1
    print "Importance factors=", myQuadraticCumul.getImportanceFactors()

# Graph 1 : Importance Factors graph
    importanceFactorsGraph = myQuadraticCumul.drawImportanceFactors()

    # In order to see the graph whithout creating the associated files
    Show(importanceFactorsGraph)

    # Create the .PNG, .EPS and .FIG files
    importanceFactorsGraph.draw("ImportanceFactorsDrawingQuadraticCumul")

    # View the bitmap file
    ViewImage(importanceFactorsGraph.getBitmap())

    # Check if it worked
    print "bitmap=" , importanceFactorsGraph.getBitmap()
    print "postscript=", importanceFactorsGraph.getPostscript()
```

Figure 69 shows an importance factors pie evaluated from the quadratic cumul method, in the beam example described here before, where :

- E follows the Beta($r = 0.94$, $t = 3.19$, $a = 2.78e7$, $b = 4.83e7$) distribution,
- F follows the LogNormal($\mu = 3e5$, $\sigma = 9e3$, $\gamma = 1.5e4$) distribution,
- L follows the Uniform($a = 250$, $b = 260$) distribution,
- I follows the Beta($r = 2.5$, $t = 4.0$, $a = 3.1e2$, $b = 4.5e2$) distribution,
- the four components are independent.

3.2.3 UC : Quantile estimations : Wilks and empirical estimators

The objective of this UC is to evaluate a particular quantile, with the empirical estimator or the Wilks one, from a numerical sample of the random variable. Each estimation is associated to a confidence interval, which

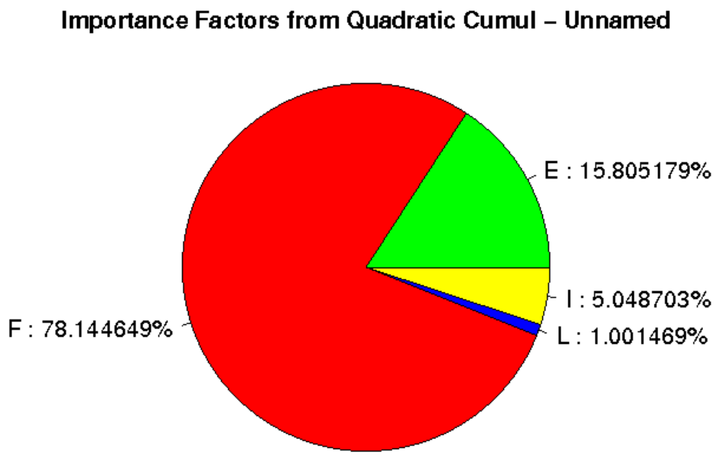


Figure 69: Importance Factors from the Taylor variance decomposition method in the beam example.

level is specified.

Let's suppose we want to estimate the quantile q_α of order α of the variable $Y : Proba(Y \leq q_\alpha) = \alpha$, from the numerical sample $(Y_1, ..., Y_n)$ of size n, with a confidence level equal to β. We note $(Y^{(}1), ..., Y^{(}n))$ the numerical sample where the values are sorted in ascending order.
The empirical estimator, noted q_α^{emp}, and its confidence intervall, is defined by the expressions :

$$\left\{\begin{array}{lcl} q_\alpha^{emp} & = & Y^{(E[n\alpha])} \\ P(q_\alpha \in [Y^{(i_n)}, Y^{(j_n)}]) & = & \beta \\ i_n & = & E[n\alpha - a_\alpha\sqrt{n\alpha(1-\alpha)}] \\ i_n & = & E[n\alpha + a_\alpha\sqrt{n\alpha(1-\alpha)}] \end{array}\right.$$

The Wilks estimator, noted $q_{\alpha,\beta}^{Wilks}$, and its confidence intervall, is defined by the expressions :

$$\left\{\begin{array}{lcl} q_{\alpha,\beta}^{Wilks} & = & Y^{(n-i)} \\ P(q_\alpha \leq q_{\alpha,\beta}^{Wilks}) & \geq & \beta \\ i \geq 0 \ / \ n \geq N_{Wilks}(\alpha,\beta,i) & & \end{array}\right.$$

Once the order i has been chosen, the Wilks number $N_{Wilks}(\alpha,\beta,i)$ is evaluated by Open TURNS, thanks to the static method $ComputeSampleSize(\alpha,\beta,i)$ of the $Wilks$ object.

In the example, we want to evaluate a quantile $\alpha = 95\%$, with a confidence level of $\beta = 90\%$ thanks to the 4th maximum of the ordered sample (associated to the order $i = 3$).
Care : $i = 0$ signifies that the Wilks estimator is the maximum of the numerical sample : it corresponds to the first maximum of the numerical sample.

Before any simulation, we initialise the state of the random generator.

The method *computeQuantile* evaluates the empirical quantile from a numerical sample in the case of dimension $n \geq 1$. However, the evaluation of the confidence interval is given only in the case of dimension 1.
Furter more, the Wilks estimator and its confidence interval is evaluated in the case of dimension 1 only.

Requirements	• the output variable of interest of dimension 1 : *output* **type** : RandomVector
Results	• the quantile estimators **type** : NumericalSaclar • Confidence Interval length **type** : NumericalScalar

Python script for this UseCase :

```
# Initialise the state of the random generator
  # thanks to the fonctionality SetSeed(n) where n is an UnsignedLong in [0,
      2^(32)-1]
  # which enables an easy initialisation for the user
  RandomGenerator.SetSeed(77)

  # or by specifying a complete state of the random generator : particularState
  # coming from a previous particularState = RandomGenerator.GetState() :
  # RandomGenerator.SetState(particularState)

# Get the complete state of the random generator before simulation
  randomGeneratorStateBeforeMonteCarlo = RandomGenerator.GetState()

# Order of the quantile to estimate
  alpha = 0.95

# Confidence level of the estimation
  beta = 0.90



# Empirical Quantile Estimator

  # Get the numerical sample of the variable
  N = 10^4
  outputNumericalSample = output.getNumericalSample(N)

  # Get the empirical estimation
  empiricalQuantile = outputNumericalSample.computeQuantile(alpha)

```

```
# Confidence interval of the Empirical Quantile Estimator
  # Get the indices of the confidence interval bounds
  aAlpha = Normal(1).computeQuantile((1-beta)/2)
  min = int(N*alpha - aAlpha*sqrt{n*alpha*(1-alpha)})
  max = int(N*alpha + aAlpha*sqrt{n*alpha*(1-alpha)})

  # Get the sorted numerical sample
  sortedOutputNumericalSample = outputNumericalSample.sort()

  # Get the Confidence interval [infQuantile, supQuantile]
  infQuantile =  sortedOutputNumericalSample[min][0]
  infQuantile =  sortedOutputNumericalSample[max][0]


# Wilks Quantile Estimator

  # Get the Wilks number : the minimal number of realisations to perform
  # in order to garantee that the empirical quantile alpha be greater than
  # the theoretical one with a probability of beta,
  # when the empirical quantile is evaluated with the (n-i)th maximum of the
      sample.
  # For the example, we consider alpha=0.95, beta=0.90 and i=3
  # By default, i=0 (empirical quantile = maximum of the sample)
  wilksNumber = Wilks.ComputeSampleSize(0.95, 0.90, 3)

  # Get the numerical sample of the variable
  outputNumericalSample = output.getNumericalSample(wilksNumber)

  # Get the sorted numerical sample
  sortedOutputNumericalSample = outputNumericalSample.sort()

  # Calculate the indice of the Wilks quantile
  indice = wilksNumber-i

  # Get the empirical estimation
  wilksQuantile = sortedOutputNumericalSample[indice][0]
```

3.3 Random approach : threshold exceedance

In this section, we focus on the random approach which aims at evaluating the probability of an event, defined as a threshold exceedance.

3.3.1 UC : Creation of an event in the physical and the standard spaces

This section gives elements to create events in the physical space *Event* and in the standard space *StandardEvent*.

The example here is the output variable *output* defined from the limit state function *poutre* defined in Eq.(7)

and the random input vector (E, F, L, I). The event considered is :

$$myEvent = \{(E, F, L, I) \in \mathbb{R}^4 / poutre(E, F, L, I) \leq -1.5\}.$$

Requirements	• the random input vector : *input* **type** : RandomVector which implementation is a UsualRandomVector • the output variable of interest : *output* of dimension 1 **type** : RandomVector which implementation is a CompositeRandomVector
Results	• the events in the physical and standard spaces : *myEvent*, *myStandardEvent* **type** : Event and StandardEvent

Python script for this UseCase :

```
# Create an event in the physical space
# from the variable of interest 'output'
    myEvent = Event(output, ComparisonOperator(Less()), -1.5, "Event 1")

# Create an standard event in the standard space
    # 1 : from the variable of interest 'output'
    myStandardEvent = StandardEvent(output,ComparisonOperator(Less()), 1.0)

    # 2 : Build a standard event based on an event
    myStandardEvent2 = StandardEvent(myEvent)
```

3.3.2 UC : Manipulation of a StandardEvent

This section gives elements to manipulate an *StandardEvent* in Open TURNS .

The example here is an output variable *output* defined from the limit state function *f* and the random input vector *input*. The event considered is :

$$myEvent = \{output = f(input) \leq -1.5\}.$$

Requirements	• an event expressed in the physical space : *myEvent* **type** : Event • the associated event in the standard space : *myStandardEvent* **type** : StandardEvent
Results	none

Python script for this UseCase :

```
# myEvent : E = (output=f(input), operator : <, threshold : -1,5)

# Realization of 'input' as antecedent of 'output'
    print "myStandardEvent (as a RandomVector) antecedent realization =" ,
       RandomVector(myStandardEvent).getImplementation().getAntecedent().
       getRealization()

# Realization of 'myEvent' as a Bernoulli
    print "myStandardEvent realization=" , myStandardEvent.getRealization()

# Sample of 10 realizations of 'myEvent  as a Bernoulli
    print "myStandardEvent sample=" , myStandardEvent.getNumericalSample(10)

# Realization of 'input' as antecedent of 'myEvent'
    print "myStandardEvent antecedent realization=" , myStandardEvent.
       getImplementation().getAntecedent().getRealization()
```

3.3.3 UC : Probability evaluation from FORM method and results associated : importance factors, reliability indexes, sensitivity on the FORM event probability and Hasofer-Lind reliability index

The objective of this UC is to evaluate the event probability from the FORM method and all the reliability indicators associated to the FORM method.
The constraints algorithms presnt in open TURNS are :

- Abdo-Rackwitz,
- Cobyla, which doesn't require the gradient evaluation of the limit state function,
- SQP.

Requirements	• the random input vector : *input* **type** : RandomVector which implementation is a UsualRandomVector • the output variable of interest : *output* of dimension 1 **type** : RandomVector which implementation is a CompositeRandomVector • the limit state function *limitStateFunction* such as : *output = limitStateFunction(input)* **type** : NumericalMathFunction • the event in physical space *myEvent* **type** : Event
Results	• FORM Event probability **type** : NumericalScalar • Reliability Index **type** : NumericalScalar • Importance factors **type** : NumericalPoint • Reliability index Sensitivity factors **type** : AnalyticalSensitivity • Event probability Sensitivity factors **type** : AnalyticalSensitivity • sensitivity graphs **type** Graph

Python script for this UseCase :

```
# Create a NearestPoint algorithm with Cobyla
    myCobyla = Cobyla()
    # Give default specific parameters to the Cobyla algoithm
    myCobyla.setSpecificParameters(CobylaSpecificParameters())
    print "Specific Parameters of Cobyla = ", myCobyla.getSpecificParameters()

# We could have created a NearestPoint algorithm with AbdoRackwitz
    # myAbdoRackwitz = AbdoRackwitz()
    # myAbdoRackwitz.setSpecificParameters(AbdoRackwitzSpecificParameters())
    # print "Specific Parameters of AbdoRackwitz = ", myAbdoRackwitz.
        getSpecificParameters()

# We could have created a NearestPoint algorithm with SQP
    # mySQP = SQP()
    # mySQP.setSpecificParameters(SQPSpecificParameters())
    # print "Specific Parameters of SQP = ", mySQP.getSpecificParameters()

# Change the parameters of the algorithm
    myCobyla.setMaximumIterationsNumber(100)
    myCobyla.setMaximumAbsoluteError(1.0e-10)
    myCobyla.setMaximumRelativeError(1.0e-10)
    myCobyla.setMaximumResidualError(1.0e-10)
    myCobyla.setMaximumConstraintError(1.0e-10)
    print "myCobyla=", myCobyla

# Create a FORM algorithm :
    # The first parameter is a NearestPointAlgorithm
    # The second parameter is an Event in the physical space
    # The third parameter is a starting point for the design point research
    mean = input.getMean()
    myAlgo = FORM(NearestPointAlgorithm(myCobyla), myEvent, mean)
    # or:
    # myAlgo = FORM(NearestPointAlgorithm(myAbdoRackwitz), myEvent, mean)
    # myAlgo = FORM(NearestPointAlgorithm(mySQP), myEvent, mean)
    print "FORM=" , myAlgo

# Save the number of calls to the limit state fucntion, its gradient and hessian
    already done
    limitStateFunctionCallNumberBefore = limitStateFunction.
        getEvaluationCallsNumber()
    limitStateFunctionGradientCallNumberBefore = limitStateFunction.
        getGradientCallsNumber()
    limitStateFunctionHessianCallNumberBefore = limitStateFunction.
        getHessianCallsNumber()

# Perform the simulation
    myAlgo.run()

# Save the number of calls to the limit state fucntion, its gradient and hessian
    already done
    limitStateFunctionCallNumberAfter = limitStateFunction.
        getEvaluationCallsNumber()
    limitStateFunctionGradientCallNumberAfter = limitStateFunction.
        getGradientCallsNumber()
    limitStateFunctionHessianCallNumberAfter = limitStateFunction.
        getHessianCallsNumber()
```

```

# Stream out the result
    result = myAlgo.getResult()

# Generalized and Hasofer reliability index
    print "generalized reliability index=", result.getGeneralisedReliabilityIndex()
    print "Hasofer reliability index=", result.getHasoferReliabilityIndex()

# Give the design point in the standard and physical spaces
    print "standard space design point=", result.getStandardSpaceDesignPoint()
    print "physical space design point=", result.getPhysicalSpaceDesignPoint()

# Is the standard point origin in failure space?
    print "is standard point origin in failure space? ", result.getIsStandardPointOriginInFailureSpace()

# Give the FORM probability of the event 'myEvent'
    print "event probability=", result.getEventProbability()

# Importance factors : numerical results
    print "importance factors=", result.getImportanceFactors()

# Hasofer Reliability Index Sensitivity : numerical results
    hasoferReliabilityIndexSensitivity = result.getHasoferReliabilityIndexSensitivity()
    print "hasoferReliabilityIndexSensitivity = ", hasoferReliabilityIndexSensitivity

# FORM Event Probability Sensitivity : numerical results
    eventProbabilitySensitivity = result.getEventProbabilitySensitivity()
    print "eventProbabilitySensitivity = ", eventProbabilitySensitivity

# Graph 1 : Importance Factors graph
    importanceFactorsGraph = result.drawImportanceFactors()
    importanceFactorsGraph.draw("ImportanceFactorsDrawingFORM")

    # View the bitmap file
    ViewImage(importanceFactorsGraph.getBitmap())

    # Check that the correct files have been generated
    # by computing their checksum
    print "bitmap=", importanceFactorsGraph.getBitmap()
    print "postscript=", importanceFactorsGraph.getPostscript()

    # In order to see the graph whithout creating the associated files
    Show(importanceFactorsGraph)

```

```
# Graph 2 : Hasofer Reliability Index Sensitivity Graphs graph
    reliabilityIndexSensitivityGraphs = result.
       drawHasoferReliabilityIndexSensitivity()

    # Sensitivity to parameters of the marginals of
    # the input random vector
    graph2a = reliabilityIndexSensitivityGraphs[0]
    graph2a.draw("HasoferReliabilityIndexMarginalSensitivityDrawing")

    # View the bitmap file
    ViewImage(graph2a.getBitmap())

    # Check that the correct files have been generated
    # by computing their checksum
    print "bitmap=" , graph2a.getBitmap()
    print "postscript=" , graph2a.getPostscript()

    # In order to see the graph whithout creating the associated files
    Show(graph2a)

    # Sensitivity to the other parameters (dependance)
    graph2b = reliabilityIndexSensitivityGraphs[1]
    graph2b.draw("HasoferReliabilityIndexOtherSensitivityDrawing")

    # or in order to quickly draw it : with default options
    # default options : 640, 480 and the files are on the current repertory
    importanceFactorsGraph.draw("ImportanceFactorsDrawingFORM")
    # View the bitmap file
    ViewImage(graph2b.getBitmap())

    # Check that the correct files have been generated
    # by computing their checksum
    print "bitmap=" , graph2b.getBitmap()
    print "postscript=" , graph2b.getPostscript()

    # In order to see the graph whithout creating the associated files
    Show(graph2b)

# Graph 3 : FORM Event Probability Sensitivity Graphs graph
    eventProbabilitySensitivityGraphs = result.drawEventProbabilitySensitivity()

    # Sensitivity to parameters of the marginals of the input random vector
    graph3a = eventProbabilitySensitivityGraphs[0]
    graph3a.draw("EventProbabilityIndexMarginalSensitivityDrawing")

    # View the bitmap file
    ViewImage(graph3a.getBitmap())

    # Check that the correct files have been generated
    # by computing their checksum
    print "bitmap=" , graph3a.getBitmap()
    print "postscript=" , graph3a.getPostscript()

    # In order to see the graph whithout creating the associated files
    Show(graph3a)

    # Sensitivity to the other parameters (dependance)
    graph3b = eventProbabilitySensitivityGraphs[1]
    graph3b.draw("EventProbabilityIndexOtherSensitivityDrawing")

    # View the bitmap file
    ViewImage(graph3b.getBitmap())

    # Check that the correct files have been generated
    # by computing their checksum
    print "bitmap=" , graph3b.getBitmap()
    print "postscript=" , graph3b.getPostscript()

    # In order to see the graph whithout creating the associated files
    Show(graph3b)
```

Figure 70 shows an importance factors pie evaluated from the FORM method, in the beam example described here before, where :

- E follows the Beta($r = 0.94$, $t = 3.19$, $a = 2.78e7$, $b = 4.83e7$) distribution,
- F follows the LogNormal($\mu = 3e5$, $\sigma = 9e3$, $\gamma = 1.5e4$) distribution,
- L follows the Uniform($a = 250$, $b = 260$) distribution,
- I follows the Beta($r = 2.5$, $t = 4.0$, $a = 3.1e2$, $b = 4.5e2$) distribution,
- the four components are independant.

The output is expressed in centimeters.
The event considered is :

$$myEvent = \{output = f(input) \leq -30\}.$$

3.3.4 UC : Probability evaluations from SORM methods and results associated : importance factors, reliability indexes, sensitivity on the Hasofer-Lind reliability index

The objective of this UC is to evaluate the event probability from the SORM method and all the reliability indicators associated to the SORM method.

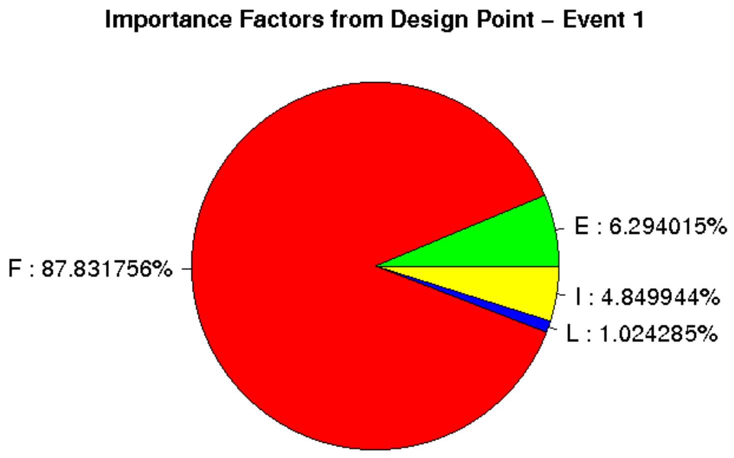


Figure 70: Importance factors from the FORM method.

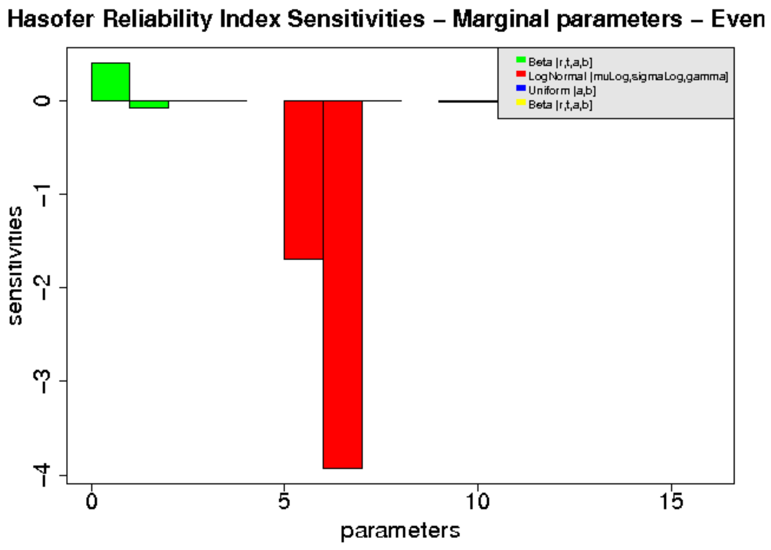


Figure 71: Hasofer Reliability Index sensitivities with respect to the marginal parameters.

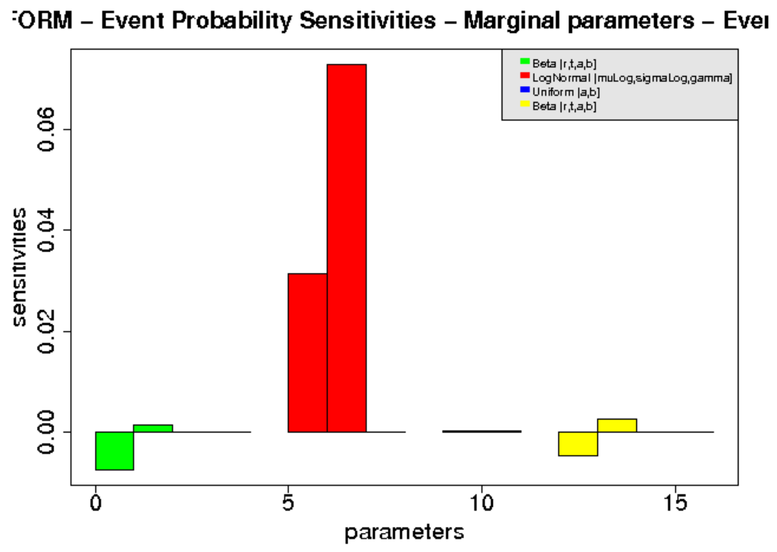


Figure 72: FORM probability sensitivities with respect to the marginal parameters.

Requirements	• the random input vector : *input* **type** : RandomVector which implementation is a UsualRandomVector • the output variable of interest of dimension 1 : *output* **type** : RandomVector which implementation is a CompositeRandomVector • the limit state function *limitStateFunction* such as : *output = limitStateFunction(input)* **type** : NumericalMathFunction • the event in physical space *myEvent* **type** : Event
Results	• SORM Event probabilities (Breitung, HohenBichler and Tvedt approximations) **type** : NumericalScalar • Reliability Index **type** : NumericalScalar • Importance factors **type** : NumericalPoint • Reliability index Sensitivity factors **type** : AnalyticalSensitivity • graphs

Python script for this UseCase :

```
# Create a NearestPoint algorithm with Cobyla
    myCobyla = Cobyla()
    # Give default specific parameters to the Cobyla algoithm
    myCobyla.setSpecificParameters(CobylaSpecificParameters())
    print "Specific Parameters of Cobyla = ", myCobyla.getSpecificParameters()

# We could have created a NearestPoint algorithm with AbdoRackwitz
    # myAbdoRackwitz = AbdoRackwitz()
    # myAbdoRackwitz.setSpecificParameters(AbdoRackwitzSpecificParameters())

# Change the parameters of the algorithm
    myCobyla.setMaximumIterationsNumber(100)
    myCobyla.setMaximumAbsoluteError(1.0e-10)
    myCobyla.setMaximumRelativeError(1.0e-10)
    myCobyla.setMaximumResidualError(1.0e-10)
    myCobyla.setMaximumConstraintError(1.0e-10)
    print "myCobyla=", myCobyla

# Create a SORM algorithm
    # The first parameter is a NearestPointAlgorithm
    # The second parameter is an event
    # The third parameter is a starting point for the design point research
    mean = input.getMean()
    myAlgo = SORM(NearestPointAlgorithm(myCobyla), myEvent, mean)
    print "SORM=" , myAlgo

# Save the number of calls to the limit state fucntion, its gradient and hessian
    already done
    limitStateFunctionCallNumberBefore = limitStateFunction.
       getEvaluationCallsNumber()
    limitStateFunctionGradientCallNumberBefore = limitStateFunction.
       getGradientCallsNumber()
    limitStateFunctionHessianCallNumberBefore = limitStateFunction.
       getHessianCallsNumber()

# Perform the simulation
    myAlgo.run()

# Save the number of calls to the limit state fucntion, its gradient and hessian
    already done
    limitStateFunctionCallNumberAfter = limitStateFunction.
       getEvaluationCallsNumber()
    limitStateFunctionGradientCallNumberAfter = limitStateFunction.
       getGradientCallsNumber()
    limitStateFunctionHessianCallNumberAfter = limitStateFunction.
```

```
        getHessianCallsNumber()

# Stream out the result
    result = myAlgo.getResult()

# Give the design point in the standard and physical spaces
    print "standard space design point=" ,result.getStandardSpaceDesignPoint()
    print "physical space design point=" , result.getPhysicalSpaceDesignPoint()

# Is the standard point origin in failure space?
    print "is standard point origin in failure space? ", result.
        getIsStandardPointOriginInFailureSpace()

# Importance factors : numerical results
    print "importance factors=" , result.getImportanceFactors()

# Give the SORM probability of the event myEvent
    # with Breitung approximation
    print "Breitung event probability=", result.getEventProbabilityBreitung()

    # with  HohenBichler approximation
    print "HohenBichler event probability=", result.
        getEventProbabilityHohenBichler()

    # with Tvedt approximation
    print "Tvedt event probability=", result.getEventProbabilityTvedt()

# Hasofer Reliability Index : numerical results
    print "Hasofer reliability index=", result.getHasoferReliabilityIndex()

# Generalised Reliability Indexes
    # with Breitung approximation
    print "Breitung generalized reliability index=", result.
        getGeneralisedReliabilityIndexBreitung()

    # with  HohenBichler approximation
    print "HohenBichler generalized reliability index=", result.
        getGeneralisedReliabilityIndexHohenBichler()

    # with Tvedt approximation
    print "Tvedt generalized reliability index=", result.
        getGeneralisedReliabilityIndexTvedt()

# Hasofer Reliability Index Sensitivity : numerical results
    hasoferReliabilityIndexSensitivity = result.
        getHasoferReliabilityIndexSensitivity()
    print "hasoferReliabilityIndexSensitivity = ",
        hasoferReliabilityIndexSensitivity
```

```


# Graph 1 : Importance Factors graph
    importanceFactorsGraph = result.drawImportanceFactors()
    importanceFactorsGraph.draw("ImportanceFactorsDrawingFORM")

    # View the bitmap file
    ViewImage(importanceFactorsGraph.getBitmap())

    # Check that the correct files have been generated
    # by computing their checksum
    print "bitmap=" , importanceFactorsGraph.getBitmap()
    print "postscript=" , importanceFactorsGraph.getPostscript()

    # In order to see the graph whithout creating the associated files
    Show(importanceFactorsGraph)

# Graph 2 : Hasofer Reliability Index Sensitivity Graphs
    reliabilityIndexSensitivityGraphs = result.
        drawHasoferReliabilityIndexSensitivity()

    # Sensitivity to parameters of the marginals of
    # the input random vector
    graph2a = reliabilityIndexSensitivityGraphs[0]
    graph2a.draw("HasoferReliabilityIndexMarginalSensitivityDrawing")

    # View the bitmap file
    ViewImage(graph2a.getBitmap())

    # Check that the correct files have been generated
    # by computing their checksum
    print "bitmap=" , graph2a.getBitmap()
    print "postscript=" , graph2a.getPostscript()

    # In order to see the graph whithout creating the associated files
    Show(graph2a)

    # Sensitivity to the other parameters (dependence)
    graph2b = reliabilityIndexSensitivityGraphs[1]
    graph2b.draw("HasoferReliabilityIndexOtherSensitivityDrawing")

    # Check that the correct files have been generated
    # by computing their checksum
    print "bitmap=" , graph2b.getBitmap()
    print "postscript=" , graph2b.getPostscript()

    # In order to see the graph whithout creating the associated files
    Show(graph2b)
```

3.3.5 UC : Probability evaluation from the Monte Carlo simulation method, determination of the confidence interval of the probability and drawing of the convergence curve with the confidence curves

The objective of this UC is to evaluate the event probability from the Monte Carlo simulation method and its confidence interval.

The probability P is evaluated with a simulation methods by the estimator P_n as defined :

$$P_n = \frac{1}{n}\sum_{i=1}^{i=n} X_i,$$

where

$$X_i = \frac{1}{p}\sum_{k=1}^{k=p} Y_k^i.$$

The random variable Y_k^i is adapted to the simulation used :

- with the Monte Carlo method, $Y_k^i = 1_{event}$,
- with the Directional Simulation, Y_k^i is the sum on one set of directions given by the Sampling strategy of the contribution of each direction to the event probability, this contribution being evaluated by the Root Strategy. With the RandomDirection Sampling Strategy, one set of directions is made of 2 directions. With the OrthogonalDirection Sampling Strategy parametered by the integer q, one set of directions is made of $C_n^q 2^q$ directions.

The parameter n is called the *OuterSamling* and the parameter p the *BlockSize*.

In the Monte Carlo method, the limit state function is evaluated $n * p$ times. In the Directional Simulation method, the limit state function is evaluated in average $n{*}p{*}$(mean number of evaluations of the limit state function on o

For the Directional Simulation method, it is recommended to fix $BlockSize = 1$.

Open TURNS enables to :

- store the numerical sample of the input random vector and the associated one of the output random vector which have been used to evaluate the Monte Carlo probability estimator. Points are stored according to a particular *HistoryStrategy* that we specify thanks to the method *setInputOutputStrategy* proposed by the *MonteCarlo* class.
- draw the convergence graph of the probability estimator with the confidence curves associated to a specified level. Values of P_n and σ_n^2 (empirical variance of the estimator P_n) are stored according to a particular *HistoryStrategy* that we specify thanks to the method *setConvergenceStrategy* proposed by the *MonteCarlo* class.

In order to prevent a memory problem, the User has the possibility to choose the storage strategy used to save the numerical samples. Four strategies are proposed :

- the *Null strategy* where nothing is stored. This strategy is proposed by the *Null* class which requires to specify no argument.
- the*Full strategy* where every point is stored. Be careful! The memory will be exhausted for huge samples. This strategy is proposed by the *Full* class which requires to specify no argument.
- the *Last strategy* where only the N last points are stored, where N is specified by the User. This strategy is proposed by the *Last* class which requires to specify the number of points to store.
- the *Compact strategy* where a regularly spaced sub-sample is stored. The minimum size N of the stored numerical sample is specified by the User. The stored numerical sample will have a size whithin N and $2N$. This strategy is proposed by the *Compact* class which requires to specify the number of points to store.

Before any simulation, we initialise the state of the random generator.

Requirements	• the event we want to evaluate the pobability : *myEvent* **type** : Event or StandardEvent
Results	• MonteCarlo Event probability **type** : NumericalScalar • Confidence Interval length **type** : NumericalScalar • Variance of the MonteCarlo probability estimator **type** : NumericalScalar

Python script for this UseCase :

```
# Initialise the state of the random generator
    # thanks to the fonctionality SetSeed(n) where n is an UnsignedLong in [0,
        2^(32)-1]
    # which enables an easy initialisation for the user
    RandomGenerator.SetSeed(77)

    # or by specifying a complete state of the random generator :
        particularState
    # coming from a previous particularState = RandomGenerator.GetState() :
    # RandomGenerator.SetState(particularState)

# Get the complete state of the random generator before simulation
    randomGeneratorStateBeforeMonteCarlo = RandomGenerator.GetState()

# Create a Monte Carlo algorithm
    myAlgo = MonteCarlo(myEvent)

    # Maximum number of extern iterations :
    myAlgo.setMaximumOuterSampling(N)

    # The simulation sampling is subsampled in samples of
    # BlockSize size (distribution service)
    # for MonteCarlo, LHS and Importance Sampling methods, we recommend
    # to use BlockSize = number of available CPU if the limit state function is
       low CPU,
    # else it is recommanded to fix BlockSize to a high value (Care : the less
       OuterSampling
    # iterations, the less points in the convergence graph!).
    myAlgo.setBlockSize(1)

    # The maximum number of evaluations of the limit state function
    # defining 'myEvent' is : MaximumOuterSampling * BlockSize

    # Maximum Coefficient of variation of the simulated sample
    myAlgo.setMaximumCoefficientOfVariation(0.1)
    print "Monte Carlo=" , myAlgo

    # Define the HistoryStrategy to store the numerical samples generated
     both for the input random vector and the output random vector
    # Null strategy
    myAlgo.setInputOutputStrategy(HistoryStrategy(Null()))
    # Full strategy
    myAlgo.setInputOutputStrategy(HistoryStrategy(Full()))
    # Compact strategy : N points
    myAlgo.setInputOutputStrategy(HistoryStrategy(Compact(N)))
    # Last strategy : N points
    myAlgo.setInputOutputStrategy(HistoryStrategy(last(N)))

    # Define the HistoryStrategy to store the values of $P_n$ and $\sigma_n$
    # used ot draw the convergence graph
    # Null strategy
    myAlgo.setConvergenceStrategy(HistoryStrategy(Null()))
    # Full strategy
    myAlgo.setConvergenceStrategy(HistoryStrategy(Full()))
    # Compact strategy : N points
    myAlgo.setConvergenceStrategy(HistoryStrategy(Compact(N)))
    # Last strategy : N points
    myAlgo.setConvergenceStrategy(HistoryStrategy(Last(N)))

# Perform the simulation
    myAlgo.run()

# Stream out the result
    result = myAlgo.getResult()
    print  "Monte Carlo result=" , result

# Display the number of iterations executed and the number of
# evaluations of the limite state function
   print "number of executed iterations = " , result.getOuterSampling()
   print "number of  evaluations of the limit state function defining myEvent = 
      ", result.getOuterSampling()*result.getBlockSize()

# Display the Monte Carlo probability of 'myEvent'
    probability = result.getProbabilityEstimate()
    print "Monte Carlo probability estimation = ", probability

# Display the variance of the Monte Carlo probability estimator
    print "Variance of the Monte Carlo probability estimator = ", result.
        getVarianceEstimate()

# Display the confidence interval length centered around the
# MonteCarlo probability MCProb
    # IC = [MCProb - 0.5*length, MCProb + 0.5*length]
    # level 0.95
    length95 = result.getConfidenceLength(0.95)
    print "0.95 Confidence Interval length = ", length95
    print "IC at 0.95 = [", probability - 0.5*length95, "; ", probability + 0.5*
        length95, "]"

    # level 0.90
    length90 = result.getConfidenceLength(0.90)
    print "0.90 Confidence Interval length = ", length90
    print "IC at 0.90 = [", probability - 0.5*length90, "; ", probability + 0.5*
        length90, "]"

# Draw the convergence graph and the confidence intervalle of level alpha
    # By default, alpha = 0.95
    alpha = 0.95
    convergenceGraph = myAlgo.drawProbabilityConvergence(0.90)

    # Impose a bounding box : x-range and y-range
    # boundingBox = [xmin, xmax, ymin, ymax]
    myBoundingBox = NumericalPoint(4)
    myBoundingBox[0] = xmin
    myBoundingBox[1] = xmax
    myBoundingBox[2] = ymin
    myBoundingBox[3] = xmax
    convergenceGraph.setBoundingBox(myBoundingBox)

    # In order to see the graph whithout creating the associated files
    Show(convergenceGraph)

    # Create the files .EPS, .PNG and .FIG
    convergenceGraph.draw("convergenceGraphe")

    # View the PNG file whithin the TUI
    ViewImage(convergenceGraph.getBitmap())

# Get the numerical samples of the input and output random vectors
# stored according to the History Strategy specified
# and used to evaluate the probability estimator and its variance
    inputSampleStored = myAlgo.getInputStrategy().getSample()
    outputSampleStored = mmyAlgo.getOutputStrategy().getSample()

# Get the values of the estimator and its variance
# stored according to the History Strategy specified
# and used to draw the convergence graph
    myAlgo.getConvergenceStrategy().getSample()
```

The following example illustrates the different storage strategy :

```
Initial Sample =
1 2 3 4 5 6 7 8 9 10 11 12

Null strategy sample =


Full strategy sample =
1 2 3 4 5 6 7 8 9 10 11 12

Last strategy sample (large storage :  36  last points) =
1 2 3 4 5 6 7 8 9 10 11 12

Last strategy sample (small storage :  4  last points) =
9 10 11 12

Compact strategy sample (large storage :  36  points) =
1 2 3 4 5 6 7 8 9 10 11 12

Compact strategy sample (small storage :  4  points) =
2 4 6 8 10 12
```

3.3.6 UC : Probability evaluation from Directional Sampling method, determination of the confidence interval and drawing of the convergence curve with the confidence curves

The Directional Sampling simulation operates in the standard space and define the maximum distant point of the standard space equal to 8 by default. This value may be changed (method *setMaximumDistance()*).

The Directional Sampling simulation method is defined from :

- an event,
- a Root Strategy :
  - RiskyAndFast : for each direction, we check whether there is a sign change of the standard limit state function between the maximum distant point (at distance *MaximumDistance* from the center of the standard space) and the center of the standard space.
    In case of sign change, we research one root in the segment [origine, maximum distant point] with the selectionned non linear solver.
    As soon as founded, the segment [root, infinity point] is considered within the failure space.
  - MediumSafe : for each direction, we go along the direction by step of length *stepSize* from the origin to the maximum distant point (at distance *MaximumDistance* from the center of the standard space) and we check whether there is a sign change on each segment so formed.
    At the first sign change, we research one root in the concerned segment with the selectionned non linear solver. Then, the segment [root, maximum distant point] is considered within the failure space.
    If *stepSize* is small enough, this strategy garantees us to find the root which is the nearest from the origine.
  - SafeAndSlow : for each direction, we go along the direction by step of length *stepSize* from the origine to the maximum distant point(at distance *MaximumDistance* from the center of the standard space) and we check whether there is a sign change on each segment so formed.
    We go until the maximum distant point. Then, for all the segments where we detected the presence of a root, we research the root with the selectionned non linear solver. We evaluate the contribution to the failure probability of each segment.
    If *stepSize* is small enough, this strategy garantees us to find all the roots in the direction and the contribution of this direction to the failure probability is precisely evaluated.
- a Non Linear Solver :
  - Bisection : bisection algorithm,
  - Secant : based on the evaluation of a segment between the two last iterated points,
  - Brent : mix of Bisection, Secant and inverse quadratic interpolation.
- and a Sampling Strategy :
  - RandomDirection : we generate one point on the sphere unity according to the uniform distribution and we consider both opposite directions so formed. So one set of direction is composed of 2 directions.
  - OrthogonalDirection : this strategy is parametered by $k \in \mathbb{N}$. We generate one direct orthonormalised base $(e_1, \ldots, e_n)$ within the set of orthonormalised bases. We consider all the renormalised linear combinations of k vectors within the n vectors of the base, where the coefficients of the linear combinations are equal to $+1, -1$. There are $C_n^k 2^k$ new vectors v_i. We consider each direction defined by each vector v_i. So one set of direction is composed of $C_n^k 2^k$ directions.
    If $k = 1$, we consider all the axes of the standard space.

Open TURNS enables to store the numerical samples of the input and output random vectors used to evaluate the Monte Carlo probability estimator and also the values of P_n and σ_n^2 (empirical variance of the estimator P_n) used to draw the convergence graph of the probability estimator. In order to have more information of the different storage strategies, see UC.3.3.5.

Before any simulation, we initialise the state of the random generator.

The example here is a Directional Sampling simulation method defined by :

- its parameters by default (TEST 1) : Root Strategy by default : Slow and Safe, Non Linear Solver : Brent algorithm, Sampling Strategy : Random Direction,
- some other parameters (TEST 2) : Root Strategy by default : Risky And Fast, Non Linear Solver : Brent algorithm, Sampling Strategy : Orthogonal Direction.

Requirements	• the output variable of interest *output* of dimension 1 **type** : RandomVector which implementation is a CompositeRandomVector • the limit state function *limitStateFunction* such as : *output = limitStateFunction(input)* **type** : NumericalMathFunction • the event in physical space : *myEvent* **type** : Event
Results	• Directional Sampling Event probability **type** : NumericalScalar • Confidence Interval length **type** : NumericalScalar • Variance of the Directional Sampling probability estimator **type** : NumericalScalar

Python script for this UseCase :

```

# Initialise the state of the random generator
  # thanks to the fonctionality SetSeed(n) where n is an UnsignedLong in [0,
      2^(32)-1]
  # which enables an easy initialisation for the user
  RandomGenerator.SetSeed(77)

  # or by specifying a complete state of the random generator : particularState
  # coming from a previous particularState = RandomGenerator.GetState() :
  # RandomGenerator.SetState(particularState)

# TEST 1 : Directional Sampling from an event
# Root Strategy by default : Safe And Slow
# Non Linear Solver : Brent algorithm
# Sampling Strategy : Random Direction

# Get the complete state of the random generator before simulation
  stateBeforeDirectionalSimulationTest1 = RandomGenerator.GetState()

# Create a Directional Sampling algorithm
    myAlgo = DirectionalSampling(myEvent)

    # Maximum number of external iterations
    myAlgo.setMaximumOuterSampling(250)

    # The simulation sampling is subsampled in samples of
    # BlockSize size (distribution service)
    # for the Directional Sampling method, we recommend
    # to use BlockSize = 1
    myAlgo.setBlockSize(1)

    #  Maximum Coefficient of variation of the simulated sample
    myAlgo.setMaximumCoefficientOfVariation(0.1)
    print "DirectionalSampling=", myAlgo

# Save the number of calls to the limit state fucntion, its gradient and hessian
    already done
    limitStateFunctionCallNumberBefore = limitStateFunction.
       getEvaluationCallsNumber()
    limitStateFunctionGradientCallNumberBefore = limitStateFunction.
       getGradientCallsNumber()
    limitStateFunctionHessianCallNumberBefore = limitStateFunction.
       getHessianCallsNumber()

# Perform the simulation
    myAlgo.run()

# Save the number of calls to the limit state fucntion, its gradient and hessian
    already done
    limitStateFunctionCallNumberAfter = limitStateFunction.
       getEvaluationCallsNumber()
    limitStateFunctionGradientCallNumberAfter = limitStateFunction.
       getGradientCallsNumber()
    limitStateFunctionHessianCallNumberAfter = limitStateFunction.
       getHessianCallsNumber()

# Stream out the result
    result = myAlgo.getResult()
    print "DirectionalSampling result=", result

# Display the number of iterations executed and
# the number of evaluations of the limit state function
    print "number of executed external iterations = " , result.getOuterSampling
        ()
    print "number of  evaluations of the limit state function defining myEvent =
        ",
limitStateFunctionCallNumberAfter - limitStateFunctionCallNumberBefore

# Display the Directional Sampling probability of 'myEvent'
    probability = result.getProbabilityEstimate()
    print "DirectionalSampling probability estimation = ", probability

# Display the variance of the Directional Sampling probability estimator
    print "Variance of the Directional Sampling probability estimator = ",
        result.getVarianceEstimate()

# Display the confidence interval length centered around the
# Directional Sampling probability DSProb
    # IC = [DSProb - 0.5*length, DSProb + 0.5*length]
    # level 0.95
    length95 = result.getConfidenceLength(0.95)
    print "0.95 Confidence Interval length = ", length95
    print "IC at 0.95 = [", probability - 0.5*length95, "; ", probability + 0.5*
        length95, "]"

# Draw the convergence graph and the confidence intervalle of level alpha
   # By default, alpha = 0.95
   alpha = 0.95
   convergenceGraph = myAlgo.drawProbabilityConvergence(0.90)

  # Impose a bounding box : x-range and y-range
  # boundingBox = [xmin, xmax, ymin, ymax]
   myBoundingBox = NumericalPoint(4)
   myBoundingBox[0] = xmin
   myBoundingBox[1] = xmax
   myBoundingBox[2] = ymin
   myBoundingBox[3] = xmax
   convergenceGraph.setBoundingBox(myBoundingBox)

  # In order to see the graph whithout creating the associated files
   Show(convergenceGraph)

  # Create the files .EPS; .PNG and .FIG
   convergenceGraph.draw("convergenceGraphe")

  # View the bitmap file
```

```
    ViewImage(convergenceGraph.getBitmap())


# TEST 2 : Directional Sampling from an event, a root strategy
# and a directional sampling strategy
# Root Strategy by default : MediumSafe
# Non Linear Solver : Brent algorithm
# Sampling Strategy : Orthogonal Direction

# Get the complete state of the random generator before simulation
    stateBeforeDirectionalSimulationTest2 = RandomGenerator.GetState()

# Create a Directional Sampling algorithm
    myAlgo2 = DirectionalSampling(myEvent, RootStrategy(MediumSafe()),
        SamplingStrategy(OrthogonalDirection(output.getDimension(),2)))

    # Maximum  number of extern iterations
    myAlgo2.setMaximumOuterSampling(250)

    # The simulation sampling is subsampled in samples
    # of BlockSize size (distribution service)
    # for the Directional Sampling method, we recommend
    # to use BlockSize = 1
    myAlgo2.setBlockSize(1)

    #  Maximum Coefficient of variation of the simulated sample
    myAlgo2.setMaximumCoefficientOfVariation(0.1)
    print "DirectionalSampling=", myAlgo2

# Save the number of calls to the limit state fucntion, its gradient and hessian
    already done
    limitStateFunctionCallNumberBefore = limitStateFunction.
        getEvaluationCallsNumber()
    limitStateFunctionGradientCallNumberBefore = limitStateFunction.
        getGradientCallsNumber()
    limitStateFunctionHessianCallNumberBefore = limitStateFunction.
        getHessianCallsNumber()

# Perform the simulation
    myAlgo2.run()

# Save the number of calls to the limit state fucntion, its gradient and hessian
    already done
    limitStateFunctionCallNumberAfter = limitStateFunction.
        getEvaluationCallsNumber()
    limitStateFunctionGradientCallNumberAfter = limitStateFunction.
        getGradientCallsNumber()
    limitStateFunctionHessianCallNumberAfter = limitStateFunction.
```

```
        getHessianCallsNumber()

# Stream out the result
     result2 = myAlgo2.getResult()
     print "DirectionalSampling result=", result2

# Display the number of iterations executed and the number of
# evaluations of the limit state function
     print "number of executed external iterations = ", result2.getOuterSampling
        ()
     print "number of  evaluations of the limit state function defining myEvent =
        ", limitStateFunctionCallNumberAfter -
        limitStateFunctionCallNumberBefore

# Display the Directional Sampling probability of 'myEvent'
     probability2 = result2.getProbabilityEstimate()
     print "DirectionalSampling probability estimation = ", probability2

# Display the variance of the Directional Sampling probability estimator
     print "Variance of the Directional Sampling probability estimator = ",
        result2.getVarianceEstimate()

# Display the confidence interval length centered around
# the Directional Sampling probability DSProb
     # IC = [DSProb - 0.5*length, DSProb + 0.5*length]
     # level 0.95
     length95 = result2.getConfidenceLength(0.95)
     print "0.95 Confidence Interval length = ", length95
     print "IC at 0.95 = [", probability2 - 0.5*length95, "; ", probability2 +
        0.5*length95, "]"

# Draw the convergence graph and the confidence intervalle of level alpha
    # By default, alpha = 0.95
    alpha = 0.95
    convergenceGraph2 = myAlgo2.drawProbabilityConvergence(0.90)

    # Impose a bounding box : x-range and y-range
    # boundingBox = [xmin, xmax, ymin, ymax]
    myBoundingBox = NumericalPoint(4)
    myBoundingBox[0] = xmin
    myBoundingBox[1] = xmax
    myBoundingBox[2] = ymin
    myBoundingBox[3] = xmax
    convergenceGraph2.setBoundingBox(myBoundingBox)

    # Create the files .EPS; .PNG and .FIG
    convergenceGraph2.draw("convergenceGraphe2")

    # In order to see the graph whithout creating the associated files
    Show(convergenceGraph2)

    # View the bitmap file whithin the TUI
    ViewImage(convergenceGraph2.getBitmap())
```

3.3.7 UC : Probability evaluation from Importance Sampling method centered on the design point issued from the FORM method, determination of the confidence interval and drawing of the convergence curve with the confidence curves

The objective of this UC is to evaluate the event probability from the Importance Sampling simulation method centered on the design point issued from the FORM method and its confidence intervaland its confidence interval. Open TURNS enables to draw the convergence graph of the probability estimator with the confidence curves associated to a specified level.

The importance density may be declared either in the physical space (TEST1) or in the standard space (TEST2).

The example here is a Normal distribution of importance :

- centered on the pysical design point with a specified correlation matrix,
- centered on the standard design point with a correlation matrix equal to Identity.

Open TURNS enables to store the numerical samples of the input and output random vectors used to evaluate the Monte Carlo probability estimator and also the values of P_n and σ_n (empirical variance of the estimator P_n) used to draw the convergence graph of the probability estimator. In order to have more information of the different storage strategies, see UC.3.3.5.

Before any simulation, we initialise the state of the random generator.

Requirements	• the design point evaluated after the FORM method in the physical and standard spaces : *physicalDesignPoint, standardDesignPoint* **type** : NumericalPoint • the correlation matrix and the deviation vector of the importance distribution in the physical space : *sigma, R* **type** : NumericalPoint and CorrelationMatrix • the event in physical and standard spaces : *myEvent*, *myStandardEvent* **type** : Event, StandardEvent
Results	• Directional Sampling Event probability **type** : NumericalScalar • Confidence Interval length **type** : NumericalScalar • Variance of the Directional Sampling probability estimator **type** : Matrix

Python script for this UseCase :

```

# Initialise the state of the random generator
  # thanks to the fonctionality SetSeed(n) where n is an UnsignedLong in [0,
      2^(32)-1]
  # which enables an easy initialisation for the user
  RandomGenerator.SetSeed(77)

  # or by specifying a complete state of the random generator : particularState
  # coming from a previous particularState = RandomGenerator.GetState() :
  # RandomGenerator.SetState(particularState)

# TEST 1 = Create an importance sampling algorithm in the physical space
# around the design point

# Get the complete state of the random generator before simulation
  randomGeneratorStateBeforeImportanceSamplingTest1 = RandomGenerator.GetState()

# Distribution of importance in the physical space : Normal(mean, sigma, R)
    myImportance = Normal(physicalDesignPoint, sigma, R)

```

```
     myAlgo1 = ImportanceSampling(myEvent, Distribution(myImportance))

     # Maximum  number of extern iterations
     myAlgo1.setMaximumOuterSampling(250)

     # The maximum number of evaluations of the limit state function
     # defining 'myEvent' is : MaximumOuterSampling * BlockSize
     myAlgo1.setBlockSize(4)

     #  Maximum Coefficient of variation of the simulated sample
     myAlgo1.setMaximumCoefficientOfVariation(0.1)

     print  "ImportanceSampling=" , myAlgo1

# Perform the simulation
     myAlgo1.run()

# Stream out the result
     result1 = myAlgo1.getResult()
     print  "Importance Sampling result=" , result1

# Display the number of iterations executed and the number of
# evaluations of the limite state function
     print "number of executed external iterations = " , result1.getOuterSampling
        ()
     print "number of  evaluations of the limit state function defining myEvent =
        ", result1.getOuterSampling()*result1.getBlockSize()

# Display the Importance Sampling probability of 'myEvent'
     probability1 = result1.getProbabilityEstimate()
     print "Importance Sampling  probability estimation = ", probability1

# Display the variance of the Importance Sampling probability estimator
     print "Variance of the Importance Sampling probability estimator = ",
        result1.getVarianceEstimate()

# Display the confidence interval length centered around the
# Importance Sampling probability ISProb
     # IC = [ISProb - 0.5*length, ISProb + 0.5*length]
     # level 0.95
     length95 = result1.getConfidenceLength(0.95)
     print "0.95 Confidence Interval length = ", length95
     print "IC at 0.95 = [", probability1 - 0.5*length95, "; ", probability2 +
        0.5*length95, "]"

# Draw the convergence graph and the confidence intervalle of level alpha
    # By default, alpha = 0.95
    alpha = 0.95
    convergenceGraph = myAlgo1.drawProbabilityConvergence(0.90)

    # Impose a bounding box : x-range and y-range
    # boundingBox = [xmin, xmax, ymin, ymax]
    myBoundingBox = NumericalPoint(4)
    myBoundingBox[0] = xmin
    myBoundingBox[1] = xmax
    myBoundingBox[2] = ymin
    myBoundingBox[3] = xmax
    convergenceGraph.setBoundingBox(myBoundingBox)

    # In order to see the graph whithout creating the associated files
    Show(convergenceGraph)

    # Create the files .EPS, .PNG and .FIG
    convergenceGraph.draw("convergenceGraphe")

    # View the PNG file whithin the TUI
    ViewImage(convergenceGraph.getBitmap())


# TEST 2 : Create an importance sampling  algorithm in the standard space
# around the design point

# Get the complete state of the random generator before simulation
  randomGeneratorStateBeforeImportanceSamplingTest2 = RandomGenerator.GetState()

# Distribution of importance in the standard space : 'myImportance' considered
    in the standard space
     sigma = NumericalPoint(standardDesignPoint.getDimension(), 1.0)
     R = CorrelationMatrix(standardDesignPoint.getDimension())
     myImportance = Normal(standardDesignPoint, sigma, R)
     myAlgo2 = ImportanceSampling(myStandardEvent, Distribution(myImportance))

     # Maximum  number of extern iterations
     myAlgo2.setMaximumOuterSampling(250)

     # The maximum number of evaluations of the limit state function
     # defining 'myEvent' is : MaximumOuterSampling * BlockSize
     myAlgo2.setBlockSize(4)

     #  Maximum Coefficient of variation of the simulated sample
     myAlgo2.setMaximumCoefficientOfVariation(0.1)

     print  "Importance Sampling=" , myAlgo2

# Perform the simulation
     myAlgo2.run()
```

```

# Stream out the result
    result2 = myAlgo2.getResult()
    print "Importance Sampling result=" , result2

# Display the number of iterations executed and the number of
# evaluations of the limite state function
    print "number of executed external iterations = " , result2.getOuterSampling
        ()
    print "number of  evaluations of the limit state function defining myEvent =
        ", result2.getOuterSampling()*result2.getBlockSize()

# Display the Importance Sampling probability of {\itshape myEvent}
    probability2 = result2.getProbabilityEstimate()
    print "Importance Sampling probability estimation = ", probability2

# Display the variance of the Importance Sampling probability estimator
    print "Variance of the Importance Sampling probability estimator = ",
        result2.getVarianceEstimate()

# Display the confidence interval length centered around
# the Importance Sampling probability ISProb
    # IC = [ISProb - 0.5*length, ISProb + 0.5*length]
    # level 0.95
    length95 = result2.getConfidenceLength(0.95)
    print "0.95 Confidence Interval length = ", length95
    print "IC at 0.95 = [", probability2 - 0.5*length95, "; ", probability2 +
        0.5*length95, "]"

# Draw the convergence graph and the confidence intervalle of level alpha
   # By default, alpha = 0.95
   alpha = 0.95
   convergenceGraph = myAlgo2.drawProbabilityConvergence(0.90)

   # Impose a bounding box : x-range and y-range
   # boundingBox = [xmin, xmax, ymin, ymax]
   myBoundingBox = NumericalPoint(4)
   myBoundingBox[0] = xmin
   myBoundingBox[1] = xmax
   myBoundingBox[2] = ymin
   myBoundingBox[3] = xmax
   convergenceGraph.setBoundingBox(myBoundingBox)

   # In order to see the graph whithout creating the associated files
   Show(convergenceGraph)

   # Create the files .EPS, .PNG and .FIG
   convergenceGraph.draw("convergenceGraphe")
```

```

    # View the PNG file whtithin the TUI
    ViewImage(convergenceGraph.getBitmap())
```

4 Construction of a response surface

The objective of this UC is to build a response surface from a function. This response surface may be built from :

- the linear or quadratic Taylor approximations of the function at a particular point,
- or a linear approximation by least squares method from a sample of the input vector and the function,
- or a linear approximation by least squares method from a sample of the input vector and one of the output variable.

4.1 UC : Linear and Quadratic Taylor approximations

This section details the first method to construct a response surface : from the linear or quadratic Taylor approximations of the function at a particular point.

Requirements	• a function : *myFunc* **type** : NumericalMathFunction
Results	• the linear Taylor approximation *myLinearTaylor* **type** : LinearTaylor • the quadratic Taylor approximation *myQuadraticTaylor* **type** : QuadraticTaylor

Python script for this UseCase :

```
# Taylor approximations at point 'center'
    center = NumericalPoint(myFunc.getInputNumericalPointDimension())
    for i in range(center.getDimension()) :
        center[i] = 1.0+i

# Create the linear Taylor approximation
    myLinearTaylor = LinearTaylor(center, myFunc)

# Create the quadratic Taylor approximation
   myQuadraticTaylor = QuadraticTaylor(center, myFunc)

```

```
# Perform the approximations
    # linear Taylor approximation
    myLinearTaylor.run()
    print "my linear Taylor =" , myLinearTaylor

    # quadratic Taylor approximation
    myQuadraticTaylor.run()
    print "my quadratic Taylor =" , myQuadraticTaylor

# Stream out the result
    # linear Taylor approximation
    linearResponseSurface = myLinearTaylor.getResponseSurface()
    print "responseSurface =" , linearResponseSurface

    # quadratic Taylor approximation
    quadraticResponseSurface = myQuadraticTaylor.getResponseSurface()
    print "quadraticResponseSurface =" , quadraticResponseSurface

# Compare the approximations and the function at a particluar point
    # point 'center'
    print "myFunc(" , center , ")=" , myFunc(center)
    print "linearResponseSurface(" , center , ")=" , linearResponseSurface(
        center)
    print "quadraticResponseSurface(" , center , ")=" , quadraticResponseSurface
        (center)
```

4.2 UC : Linear approximation response surface by least squares method from a sample of the input vector and the real function

This section details the second method to construct a response surface : by least squares method from a sample of the input vector and the real function. The output sample is obtained by evaluating the function on the input sample.

Requirements	• the limit state function : *myFunc* **type** : NumericalMathFunction • a sample of the input vector : *inputSample* **type** : NumericalSample
Results	• linear approximation by least squares method *responseSurface* **type** : NumericalMathFunction • the coefficients of the linear approximation $myFunc(\underline{X}) = \underline{\underline{A}}\underline{X} + \underline{B}$ **type** : Matrix ($\underline{\underline{A}}$) , NumericalPoint ($\underline{B}$)

Python script for this UseCase :

```
# Create the LinearLeastSquares algorithm
   myLeastSquares = LinearLeastSquares(inputSample, myFunc)

# Perform the algorithm
  myLeastSquares.run()
  print "myLeastSquares=", myLeastSquares

# Stream out the results :
  # get the matrix A :
  linear = myLeastSquares.getLinear()
  print "A = ", linear

  # Get the constant term B :
  constant = myLeastSquares.getConstant()
  print "B = ", constant

  # Get the linear response surface
  responseSurface = myLeastSquares.getResponseSurface()
  print "responseSurface=", responseSurface

# Compare the two models at a particular point 'inPoint'
  print "(myFunc", inPoint, ")=", myFunc(inPoint)
  print "responseSurface(", inPoint, ")=", responseSurface(inPoint)
```

4.3 UC : Linear approximation response surface by least squares method from a sample of the input vector and a sample of the output vector

This section details the second method to construct a response surface : by least squares method from a sample of the input vector and the associated sample of the output variable.

Requirements	• a sample of the input vector : *inputSample* **type** : NumericalSample • the associated sample of the output vector : *outputSample* **type** : NumericalSample
Results	• linear approximation by least squares method *responseSurface* **type** : NumericalMathFunction • the coefficients of the linear approximation $\underline{\underline{A}}\underline{X} + \underline{B}$ **type** : Matrix ($\underline{\underline{A}}$) , NumericalPoint ($\underline{B}$)

Python script for this UseCase :

```
# Create the LinearLeastSquares algorithm
   myLeastSquares = LinearLeastSquares(inputSample, outputSample)

# Perform the algorithm
  myLeastSquares.run()
  print "myLeastSquares=", myLeastSquares

# Stream out the results :
  # get the matrix A :
  linear = myLeastSquares.getLinear()
  print "A = ", linear

  # Get the constant term B :
  constant = myLeastSquares.getConstant()
  print "B = ", constant

  # Get the linear response surface
  responseSurface = myLeastSquares.getResponseSurface()
  print "responseSurface=", responseSurface
```

5 How to save and load a study ?

The objective of this UC is to describe how to save some results obtained within a study and how to load a study performed previously, with some results.

The mechanism of Open TURNS is detailed through the two following use cases. We give an example on a NumericalPoint and a NumericalMathFunction but it can be used for most objects.

The principle is the following one : all along the study, we create a list of objects we want to save, thanks to the command .add(). Then, at the end of the study, we save the list with the command .save(). Only at that time, all the study is saved.

5.1 UC : How to save a study ?

Requirements	none
Results	• an object containing all the objects saved : *myStudy* **type** : Study • a file .XML containing all the objects saved : *myXMLFile.XML* **type** : file .XML

Python script for this UseCase :

```
# Create the name of the file .XML which will be created at the adress /tmp
# if the adress is not precised, the file .XML is created in the current
    repertory
    fileName = "/tmp/myXMLFile"

# Create a Study Object which will contain all the objects saved
     myStudy = Study()

     # Associate it to the file .XML just created
     myStudy.setStorageManager(XMLStorageManager(fileName))

# Perform here the study
# for example, a NumericalPoint is created we want to save
     p1 = NumericalPoint(3, 0., "Good")
     p1[0] = 10.
     p1[1] = 11.
     p1[2] = 12.
     desc = p1.getDescription()
     desc[0] = "x"
     desc[1] = "y"
     desc[2] = "z"
     p1.setDescription(desc)

# Add the NumericalPoint to the list of the objects saved
     myStudy.add(p1)

# Create an analytical NumericalMathFunction
     input = Description(2)
     input[0] = "a"
     input[1] = "b"
     output = Description(3)
     output[0] = "sum"
     output[1] = "prod"
     output[2] = "mean"
     formulas = Description(output.getSize())
     formulas[0] = "a+b"
     formulas[1] = "a*b"
     formulas[2] = "(a+b)/2"
     analytical = NumericalMathFunction(input, output, formulas, "analytical")

# Add the NumericalMathFunction to the list of the objects saved
     myStudy.add(analytical)

# Check the list of objects that will be saved
     print "Study = ", myStudy

# Remove the NumericalMathFunction to the list of the objects saved
     myStudy.remove(analytical)

# CARE!! At this point, no object has been saved : only the list have been
    written!
# SAVE NOW the objects listed
     myStudy.save()
```

5.2 UC : How to load a study ?

The principle is the following one. In order to be able to manipulate the objects contained in myStudy, it is necessary to :

- create the same empty structure in the new study,
- fill this new empty structure with the content of the loaded structure, identified with its name or its id.

Each object is identified whether with :

- its name : that's why it is usefull to give names to the objects we want to save (thanks to the command setName()). If no name has been given by the User, we can use the name by default given by Open TURNS. The name of each object saved can be checked in the file.XML created or by printing the study in the python interface (with the command print).
- or its id number : this id number is unique for each object. It is usefull to separate two objects of same type which names are identical, equal to the default name given by Open TURNS (for example, two NumericalPoint the User has not named explicitely, both called 'Unnamed' by Open TURNS). This id number may be checked by printing the study loaded in the python interface (with the command print) : be carefull, this print operation must be performed after having loaded the study (the id number may be different from the one indicated in the file.XML associated to the study).

In this use case, we load the file saved in the previous use case.

Requirements	• a file .XML containing all the objects saved previously: *myXMLFile.XML* **type** : file .XML
Results	• all the objects of the file myXMLFile.XML loaded in the new study **type** : -

Python script for this UseCase :

```
# Give the name and the adress of the file .XML that will be loaded
    fileName = "/tmp/myXMLFile"

# Create a Study Object
    myStudy = Study()

    # Associate it to the file myXMLFile.XML
    myStudy.setStorageManager(XMLStorageManager(fileName))

# Load the file and all its objects
    myStudy.load()

# Check the content of the myStudy
    print "Study = " , myStudy

# In order to be able to manipulate the objects contained in myStudy, it is
    necessary to :
# 1. create the same empty structure in the new study
# 2. fill this new empty structure with the content of the loaded structure

# Create a NumericalPoint from the one stored in myStudy
    pointLoaded = NumericalPoint()

    # Fill the new structure point with the content of the NumericalPoint saved
    # this NumericalPoint is identified with its name 'point'
    myStudy.fillObject(pointLoaded , "point")

    # Check if it worked : the NumericalPoint 'pointLoaded' has been loaded
    # and we can manipulate it
    print "pointLoaded = " , pointLoaded

# Create an analytical NumericalMathFunction from the one stored in myStudy
    analyticalLoaded = NumericalMathFunction()

    # Fill the new structure point with the content of the NumericalMathFunction saved
    # this NumericalMathFunction is identified with its id
    # to read the right id, print myStudy which has already been loaded
    print "Study = " , myStudy
    # Fill the new structure with its id : for example, 32
    myStudy.fillObject(analyticalLoaded , 32)

    # Check if it worked : the NumericalMathFunction 'analytical' has been loaded
    print "analyticalLoaded = " , analyticalLoaded
```

6 Annexe 1 : One example of a complete study

6.1 Presentation of the study case

This Annexe presents several Use Cases described previoulsy in order to show one example of a complete study. This example has been presented in the ESREL 2007 conference in the paper : *Open TURNS, an Open source initiative to Treat Uncertainties, Risks'N Statistics in a structured industrial approach*, from A. Dutfoy(EDF R&D), I. Dutka-Malen(EDF R&D), R. Lebrun (EADS innovation Works) & all.

Let's consider the following analytical example of a cantilever beam, of Young's modulus E, length L, section modulus I. One end is built in a wall and we apply a concentrated bending load at the other end of the beam. The deviation (vertical displacement) y of the free end is equal to :

$$y(E, F, L, I) = \frac{FL^3}{3EI}$$

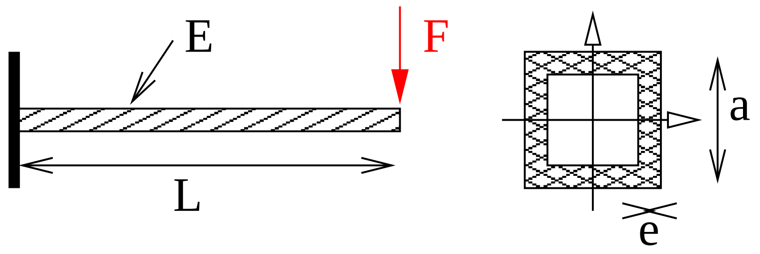


Figure 73: cantilever beam under a ponctual bending load.

The objective of this UC is to evaluate the influence of uncertainties on the input data (E, F, L, I) on the deviation y.
We consider a steel beam with a hollow square section of length $a = 2.e - 2m$ and of thickness $t = 1.e - 3m$. Thus, the flexion section inertie of the beam is equal to $I = 2.47e - 9m^4$. The beam length is $L = 1m$. The Young's modulus E is $E = 2.1e11kg.m^{-1}.s^{-2}$. The charge applied is $F = 10kg.m.s^{-2}$.
The random modelisation of the input data is the following one : we consider for each input data a gaussian distribution, which mean μ is the deterministic value given above and which standard deviation is a percentile of the mean :

- E = Gaussian(μ_E, 5% * μ_E)
- F = Gaussian(μ_F, 10% * μ_F)
- L = Gaussian(μ_L, 1% * μ_L)
- I = Gaussian(μ_I, 1% * μ_I)

This example treats the following points of the methodology :

- Deterministic Study : Min/Max study
  - with a deterministic experiment plane,
  - with a random experiment plane,

- Random Study : central tendance of the output variable of interest
  - Taylor variance decomposition,
  - Random sampling,
- Random Study : threshold exceedance: deviation ¡-1cm
  - FORM,
  - SORM,,
  - Monte Carlo simulation method,
  - Directional Sampling method,
  - Latin HyperCube Sampling method,
  - Importance Sampling method
  - Kernel Smoothing Fitting.

6.2 The TUI File

```
#! /usr/bin/env python

from openturns import *

from math import *

from openturns_viewer import ViewImage

# This function enables a pretty print of a NumericalPoint
def printNumericalPoint(point, digits) :
  oss = "["
  eps = pow(0.1, digits)
  for i in range(point.getDimension()) :
    if i == 0 :
      sep = ""
    else :
      sep = ","
    if fabs(point[i]) < eps :
      oss += sep + str(fabs(point[i]))
    else :
      oss += sep + str(point[i])
    sep = ","
  oss += "]"
  return oss


##########################################
### Fonction 'poutre'
##########################################

```

```
# We create a numerical math function
myFunction = NumericalMathFunction("poutre")


#############################################
### Random input vector
#############################################



dim = myFunction.getInputNumericalPointDimension()

# We create a normal distribution point of dimension 4
mean = NumericalPoint(dim, 0.0)
# E : steel : 210000 MPa
mean[0] = 2.1e11
# F : 1kg : 10N
mean[1] =  10.0
# L : 1 m
mean[2] = 1.0
# I : square hollow section of width 2 cm and thickness 1mm : 2.47325 e-9
mean[3] =  2.47325e-9
sigma = NumericalPoint(dim, 1.0)
# E : 5% * mean
sigma[0] = 0.05 *  mean[0]
# F : 10% * mean
sigma[1] = 0.1 * mean[1]
# L : 1% * mean
sigma[2] = 0.01 * mean[2]
# I : 1% * mean
sigma[3] = 0.01 * mean[3]

R = IdentityMatrix(dim)
myDistribution = Normal(mean, sigma, R)


input = RandomVector(myDistribution)

output =  RandomVector(myFunction, input)


#############################################
### Deterministic Study
#############################################


print "####################"
print " Deterministic Study"
print "####################"

print "deterministic evaluation at the mean point : "
print "deviation(mean point) = ", myFunction(mean)


#####################################################
# Min/Max study with deterministic experiment plane
#####################################################

print "######################################################"
print " Min/Max study with deterministic experiment plane"
print "######################################################"


# Creation of the structure of the experiment plane : type Axial

# On each direction separately, several levels are evaluated
# here,  3 levels : +/-1, +/-3, +/-5 from the center
levelsNumber = 3
levels = NumericalPoint(levelsNumber, 0.0, "Levels")
levels[0] = 1
levels[1] = 3
levels[2] = 5
# Creation of the axial plane
myPlane = Axial(dim, levels)
print "myPlane = " , myPlane

# Generation of points according to the structure of the experiment plane
# (in a reduced centered space)
inputSample = myPlane.generate()

# Scaling of the structure of the experiment plane
# scaling vector for each dimension of the levels of the structure
# to take into account the dimension of each component
# for example : the standard deviation of each component of 'input'
# in case of a RandomVector
scaling = NumericalPoint(dim)
scaling[0] = sqrt(input.getCovariance()[0,0])
scaling[1] = sqrt(input.getCovariance()[1,1])
scaling[2] = sqrt(input.getCovariance()[2,2])
scaling[3] = sqrt(input.getCovariance()[3,3])
print "sigma = ", scaling
inputSample.scale(scaling)
print "centered Sample = ", inputSample

# Translation of the nonReducedSample onto the center of the experiment plane
# center = mean point of the input distribution
```

```
center = input.getMean()
inputSample.translate(center)
print "inputSample = ", inputSample
pointNumber = inputSample.getSize()
print "points number = ", pointNumber

outputSample = myFunction(inputSample)

minValue = outputSample.getMin()
maxValue = outputSample.getMax()

print " From an axial  experiment plane of size = ", pointNumber
print "levels = ", levels
print "min Value = ", minValue[0]
print "max Value = ", maxValue[0]
print ""

#############################################################
# Min/Max study with random experiment plane
#############################################################



print "#####################################################"
print " Min/Max study with random experiment plane"
print "#####################################################"

pointNumber = 100
print " From a stochastic experiment place of size = ", pointNumber
outputSample2 = output.getNumericalSample(pointNumber)

minValue2 = outputSample2.getMin()
maxValue2 = outputSample2.getMax()

print "min Value = ", minValue2[0]
print "max Value = ", maxValue2[0]
print ""

#################################################
### Random Study : central tendance of
### the output variable of interest
#################################################



print "###########################################"
print " Random Study : central tendance of"
print " the output variable of interest"
print "##############################################"

########################################
# Taylor variance decomposition
########################################

print "################################"
print "Taylor variance decomposition"
print "################################"

# We create a quadraticCumul algorithm
myQuadraticCumul = QuadraticCumul(output)

# We test the attributs here
print "myQuadraticCumul=", myQuadraticCumul

# We compute the several elements provided by the quadratic cumul algorithm
print "First order mean=", myQuadraticCumul.getMeanFirstOrder()[0]
print "Second order mean=", myQuadraticCumul.getMeanSecondOrder()[0]
print "Standard deviation=", sqrt(myQuadraticCumul.getCovariance()[0,0])

################################
# Random sampling
################################

print "########################"
print "Random sampling"
print "########################"

size1 = 10000
output_Sample1 = output.getNumericalSample(size1)
outputMean = output_Sample1.computeMean()
outputCovariance = output_Sample1.computeCovariance()

print "sample size = ", size1
print "mean from sample = ", outputMean[0]
print "standard deviation from sample = ", sqrt(outputCovariance[0,0])

######################################################################
### Probabilistic Study : threshold exceedance: deviation <-1cm
######################################################################

print "#################################################################"
print "Probabilistic Study : threshold exceedance: deviation <-1cm"
print "#################################################################"

######
# FORM
```

```
######


print "#####"
print "FORM"
print "#####"

# We create an Event from this RandomVector
threshold = -0.01
myEvent = Event(output, ComparisonOperator(Less()), threshold)

# We create a NearestPoint algorithm
myCobyla = Cobyla()
myCobyla.setSpecificParameters(CobylaSpecificParameters())
myCobyla.setMaximumIterationsNumber(1000)
myCobyla.setMaximumAbsoluteError(1.0e-10)
myCobyla.setMaximumRelativeError(1.0e-10)
myCobyla.setMaximumResidualError(1.0e-10)
myCobyla.setMaximumConstraintError(1.0e-10)
#print "myCobyla=", myCobyla

# We create a FORM algorithm
# The first parameter is a NearestPointAlgorithm
# The second parameter is an event
# The third parameter is a starting point for the design point research
myAlgoFORM = FORM(NearestPointAlgorithm(myCobyla), myEvent, mean)

#print "FORM=", myAlgo

# Perform the simulation
myAlgoFORM.run()

# Stream out the result
resultFORM = myAlgoFORM.getResult()
digits = 5
print "FORM event probability=", resultFORM.getEventProbability()
print "generalized reliability index=", resultFORM.
    getGeneralisedReliabilityIndex()
print "standard space design point=", printNumericalPoint(resultFORM.
    getStandardSpaceDesignPoint(), digits)
print "physical space design point=", printNumericalPoint(resultFORM.
    getPhysicalSpaceDesignPoint(), digits)

print "importance factors=", printNumericalPoint(resultFORM.
    getImportanceFactors(), digits)
print "Hasofer reliability index=", resultFORM.getHasoferReliabilityIndex()

# Graph 1 : Importance Factors graph */
```

```
importanceFactorsGraph = resultFORM.drawImportanceFactors()
importanceFactorsGraph.draw("ImportanceFactorsDrawingFORM")

# View the bitmap file
ViewImage(importanceFactorsGraph.getBitmap())

# In order to see the graph whithout creating the associated files
Show(importanceFactorsGraph)

# Graph 2 : Hasofer Reliability Index Sensitivity Graphs graph */
reliabilityIndexSensitivityGraphs = resultFORM.
    drawHasoferReliabilityIndexSensitivity()
reliabilityIndexSensitivityGraphs[0].draw("
    HasoferReliabilityIndexMarginalSensitivityDrawing")

# View the bitmap file
ViewImage(reliabilityIndexSensitivityGraphs[0].getBitmap())

# In order to see the graph whithout creating the associated files
Show(reliabilityIndexSensitivityGraphs[0])

# Graph 3 : FORM Event Probability Sensitivity Graphs graph */
eventProbabilitySensitivityGraphs = resultFORM.drawEventProbabilitySensitivity()
eventProbabilitySensitivityGraphs[0].draw("
    EventProbabilityIndexMarginalSensitivityDrawing")

# View the bitmap file
ViewImage(eventProbabilitySensitivityGraphs[0].getBitmap())

# In order to see the graph whithout creating the associated files
Show(eventProbabilitySensitivityGraphs[0])


######
# SORM
######


print "#####"
print "SORM"
print "#####"

# We create a SORM algorithm
myAlgoSORM = SORM(NearestPointAlgorithm(myCobyla), myEvent, mean)

# Perform the simulation
myAlgoSORM.run()

```

```
# Stream out the result
resultSORM = myAlgoSORM.getResult()
digits = 5
print "Breitung event probability=" , resultSORM.getEventProbabilityBreitung()
print "Breitung generalized reliability index=" , resultSORM.
    getGeneralisedReliabilityIndexBreitung()
print "HohenBichler event probability=" , resultSORM.
    getEventProbabilityHohenBichler()
print "HohenBichler generalized reliability index=" , resultSORM.
    getGeneralisedReliabilityIndexHohenBichler()
print "Tvedt event probability=" , resultSORM.getEventProbabilityTvedt()
print "Tvedt generalized reliability index=" , resultSORM.
    getGeneralisedReliabilityIndex()

######
# MC
######

print "############"
print "Monte Carlo"
print "############"


maximumOuterSampling = 400
blockSize = 100000
coefficientOfVariation = 0.10

# We create a Monte Carlo algorithm
myAlgoMonteCarlo = MonteCarlo(myEvent)
myAlgoMonteCarlo.setMaximumOuterSampling(maximumOuterSampling)
myAlgoMonteCarlo.setBlockSize(blockSize)
myAlgoMonteCarlo.setMaximumCoefficientOfVariation(coefficientOfVariation)

print "MonteCarlo=" , myAlgoMonteCarlo

# Perform the simulation
myAlgoMonteCarlo.run()

# Stream out the result
print "MonteCarlo result=" , myAlgoMonteCarlo.getResult()

# Display number of iterations and number of evaluations
# of the limit state function
print "external iteration numbers = " , myAlgoMonteCarlo.getResult().
    getOuterSampling()
print "number of evaluations of the limit state function = ", myAlgoMonteCarlo.
    getResult().getOuterSampling()* myAlgoMonteCarlo.getResult().getBlockSize()

```

```
# Display the Monte Carlo probability of 'myEvent'
print "Monte Carlo probability estimation = ", myAlgoMonteCarlo.getResult().
    getProbabilityEstimate()

# Display the variance of the Monte Carlo probability estimator
print "Variance of the Monte Carlo probability estimator = ", myAlgoMonteCarlo.
    getResult().getVarianceEstimate()

# Display the confidence interval length centered around
# the MonteCarlo probability MCProb
# IC = [MCProb - 0.5*length, MCProb + 0.5*length]
# level 0.95
print "0.95 Confidence Interval length = ", myAlgoMonteCarlo.getResult().
    getConfidenceLength(0.95)
#
print "0.95 Confidence Interval = [", myAlgoMonteCarlo.getResult().
    getProbabilityEstimate() - 0.5*myAlgoMonteCarlo.getResult().
    getConfidenceLength(0.95), ", ", myAlgoMonteCarlo.getResult().
    getProbabilityEstimate() + 0.5*myAlgoMonteCarlo.getResult().
    getConfidenceLength(0.95), "]"

########################
# Directional Sampling
########################

print "########################"
print "Directional Sampling"
print "########################"

# Directional sampling from an event (slow and safe strategy by default)

# We create a Directional Sampling algorithm */
myAlgoDirectionalSim = DirectionalSampling(myEvent)
myAlgoDirectionalSim.setMaximumOuterSampling(maximumOuterSampling * blockSize)
myAlgoDirectionalSim.setBlockSize(1)
myAlgoDirectionalSim.setMaximumCoefficientOfVariation(coefficientOfVariation)

print "DirectionalSampling=", myAlgoDirectionalSim



# Save the number of calls to the limit state fucntion, its gradient and hessian
    already done
limitStateFunctionCallNumberBefore = limitStateFunction.getEvaluationCallsNumber
    ()
limitStateFunctionGradientCallNumberBefore = limitStateFunction.
    getGradientCallsNumber()
```

```
limitStateFunctionHessianCallNumberBefore = limitStateFunction.
    getHessianCallsNumber()

# Perform the simulation */
myAlgoDirectionalSim.run()

# Save the number of calls to the limit state fucntion, its gradient and hessian
     already done
limitStateFunctionCallNumberAfter = limitStateFunction.getEvaluationCallsNumber
    ()
limitStateFunctionGradientCallNumberAfter = limitStateFunction.
    getGradientCallsNumber()
limitStateFunctionHessianCallNumberAfter = limitStateFunction.
    getHessianCallsNumber()

# Stream out the result */
print "Directional Sampling result=", myAlgoDirectionalSim.getResult()

# Display number of iterations and number of evaluations
# of the limit state function
print "external iteration numbers = " , myAlgoDirectionalSim.getResult().
    getOuterSampling()
print "number of evaluations of the limit state function = ",
    limitStateFunctionCallNumberAfter - limitStateFunctionCallNumberBefore

# Display the Directional Simumation probability of 'myEvent'
print "Directional Sampling probability estimation = ", myAlgoDirectionalSim.
    getResult().getProbabilityEstimate()

# Display the variance of the Directional Simumation probability estimator
print "Variance of the Directional Sampling probability estimator = ",
    myAlgoDirectionalSim.getResult().getVarianceEstimate()

# Display the confidence interval length centered around
# the Directional Simumation probability DSProb
# IC = [DSProb - 0.5*length, DSProb + 0.5*length]
# level 0.95
print "0.95 Confidence Interval length = ", myAlgoDirectionalSim.getResult().
    getConfidenceLength(0.95)
print "0.95 Confidence Interval = [", myAlgoDirectionalSim.getResult().
    getProbabilityEstimate() - 0.5*myAlgoDirectionalSim.getResult().
    getConfidenceLength(0.95), ", ", myAlgoDirectionalSim.getResult().
    getProbabilityEstimate() + 0.5*myAlgoDirectionalSim.getResult().
    getConfidenceLength(0.95), "]"

##########################
# Latin HyperCube Sampling
###########################
```

```

print "##############################"
print "Latin HyperCube Sampling"
print "##############################"
\index{Threshold Probability!LHS}

# We create a LHS algorithm
myAlgoLHS = LHS(myEvent)
myAlgoLHS.setMaximumOuterSampling(maximumOuterSampling)
myAlgoLHS.setBlockSize(blockSize)
myAlgoLHS.setMaximumCoefficientOfVariation(coefficientOfVariation)

print "LHS=" , myAlgoLHS

# Perform the simulation
myAlgoLHS.run()

# Stream out the result
print "LHS result=" , myAlgoLHS.getResult()

# Display number of iterations and number of evaluations
# of the limit state function
print "external iteration numbers = " , myAlgoLHS.getResult().getOuterSampling()
print "number of evaluations of the limit state function = ", myAlgoLHS.
    getResult().getOuterSampling()*myAlgoLHS.getResult().getBlockSize()

# Display the LHS probability of {\itshape myEvent}
print "LHS probability estimation = ", myAlgoLHS.getResult().
    getProbabilityEstimate()

# Display the variance of the LHS probability estimator
print "Variance of the LHS probability estimator = ", myAlgoLHS.getResult().
    getVarianceEstimate()

# Display the confidence interval length centered aroung the LHS probability
    LHSProb
# IC = [LHSProb - 0.5*length, LHSProb + 0.5*length]
# level 0.95
print "0.95 Confidence Interval length = ", myAlgoLHS.getResult().
    getConfidenceLength(0.95)
print "0.95 Confidence Interval = [", myAlgoLHS.getResult().
    getProbabilityEstimate() - 0.5*myAlgoLHS.getResult().getConfidenceLength
    (0.95), ", ", myAlgoLHS.getResult().getProbabilityEstimate() + 0.5*myAlgoLHS.
    getResult().getConfidenceLength(0.95), "]"

######################
# Importance Sampling
######################
```

```


print "####################"
print "Importance Sampling"
print "####################"

standardSpaceDesignPoint = resultFORM.getStandardSpaceDesignPoint()
mean = standardSpaceDesignPoint
sigma = NumericalPoint(4, 1.0)
importanceDistribution = Normal(mean, sigma, CorrelationMatrix(4))

myStandardEvent = StandardEvent(myEvent)

myAlgoImportanceSampling = ImportanceSampling(myStandardEvent, Distribution(
    importanceDistribution))
myAlgoImportanceSampling.setMaximumOuterSampling(maximumOuterSampling)
myAlgoImportanceSampling.setBlockSize(blockSize)
myAlgoImportanceSampling.setMaximumCoefficientOfVariation(coefficientOfVariation
    )

print "Importance Sampling=" , myAlgoImportanceSampling

# Perform the simulation
myAlgoImportanceSampling.run()

# Stream out the result
print "Importance Sampling result=" , myAlgoImportanceSampling.getResult()

# Display number of iterations and number of evaluations
# of the limit state function
print "external iteration numbers = " , myAlgoImportanceSampling.getResult().
    getOuterSampling()
print "number of evaluations of the limit state function = ",
    myAlgoImportanceSampling.getResult().getOuterSampling()*
    myAlgoImportanceSampling.getResult().getBlockSize()

# Display the Importance Sampling probability of 'myEvent'
print "Importance Sampling probability estimation = ", myAlgoImportanceSampling.
    getResult().getProbabilityEstimate()

# Display the variance of the Importance Sampling probability estimator
print "Variance of the Importance Sampling probability estimator = ",
    myAlgoImportanceSampling.getResult().getVarianceEstimate()

# Display the confidence interval length centered around
# the ImportanceSampling probability ISProb
# IC = [ISProb - 0.5*length, ISProb + 0.5*length]
# level 0.95
```

```
print "0.95 Confidence Interval length = ", myAlgoImportanceSampling.getResult()
    .getConfidenceLength(0.95)
print "0.95 Confidence Interval = [", myAlgoImportanceSampling.getResult().
    getProbabilityEstimate() - 0.5*myAlgoImportanceSampling.getResult().
    getConfidenceLength(0.95), ", ", myAlgoImportanceSampling.getResult().
    getProbabilityEstimate() + 0.5*myAlgoImportanceSampling.getResult().
    getConfidenceLength(0.95), "]"

##########################
# Kernel Smoothing Fitting
##########################


print "##########################"
print "# Kernel Smoothing Fitting"
print "##########################"

# We generate a sample of the output variable
size = 1000
output_sample = output.getNumericalSample(size)

# We build the kernel smoothing distribution
kernel = KernelSmoothing()
smoothed = kernel.buildImplementation(output_sample)
print  "kernel bandwidth=" , kernel.getBandwidth()

# We draw the pdf and cdf from kernel smoothing
mean_sample = output_sample.computeMean()[0]
standardDeviation_sample = sqrt(output_sample.computeCovariance()[0,0])
xmin = mean_sample - 4*standardDeviation_sample
xmax = mean_sample + 4*standardDeviation_sample

smoothedPDF = smoothed.drawPDF(xmin, xmax, 251)
smoothedPDF.draw("smoothedPDF")

smoothedCDF = smoothed.drawCDF(xmin, xmax, 251)
smoothedCDF.draw("smoothedCDF")

# In order to see the graph whithout creating the associated files
Show(smoothedCDF)
Show(smoothedPDF)

# Probability of myEvent : 1-smoothedCDF(threshold)
print "probability of the event after kernel smoothing = ", 1.0 - smoothed.
    computeCDF(NumericalPoint(1,threshold))

# Superposition of the kernel smoothing pdf and the gaussian one
# which mean and standard deviation are those of the output_sample
```

```
meanSample = output_sample.computeMean()
standardDeviationSample = NumericalPoint(1, sqrt(output_sample.computeCovariance
    ()[0,0]))
gaussianDist = Normal(meanSample,standardDeviationSample, CorrelationMatrix(1))

gaussianDistPDF = gaussianDist.drawPDF(xmin, xmax, 251)
gaussianDistPDFDrawable = gaussianDistPDF.getDrawable(0)
gaussianDistPDFDrawable.setColor('red')
smoothedPDF.addDrawables(gaussianDistPDFDrawable)
smoothedPDF.draw("smoothedPDF_and_GaussianPDF")

# In order to see the graph whithout creating the associated files
Show(smoothedPDF)
```

Index

Comparison of distribution test
  Smirnov, 40
Composed Distribution, 19
Copula
  Clayton, 15
  Composed copula, 15
  Frank, 15
  Gumbel, 15
  Independent, 15, 19
  Normal, 15, 19
Correlation
  Correlation matrix of the Normal copula, 15, 19
  Partial Pearson correlation coefficient (PCC), 95
  Partial rank correlation coefficient (PRCC), 95
  Pearson correlation coefficient, 95
  Spearman correlation coefficient, 95
  Spearman rank correlation matrix, 15, 19
  Standard rank regression coefficient (SRRC), 95
  Standard regression coefficient (SRC), 95
CSV file, 34

Distribution
  Manipulation, 25
  Marginals and copula, 19
  Usual distribution, 3

Event
  Physical space, 100, 102, 107, 113, 117, 124
  Standard space, 100, 101, 124
Experiment Plane
  Axial, 85, 91
  Box, 85
  Composite, 85
  Drawing, 91
  Factorial, 85
  Random experiment plane, 93
  Scaling, 85, 91
  Translation, 85, 91

Fitting Distribution
  Kernel smoothing, 58
  Parametric method, 48
Fitting Test
  Anderson Darling, 51
  ChiSquared, 48
  Cramer Von Mises, 51
  Henry line, 51
  Kolmogorov, 48
  QQ-plot, 40, 48

Graph
  Clouds of points, 73
  Empirical cumulative density function, 35, 58
  FORM importance factors, 102, 107
  FORM probability sensitivity, 102
  Hasofer reliability index sensitivity, 102, 107
  Henry line, 51
  Histogram, 35
  PDF-CDF curves, 21, 25, 58
  PDF-CDF isocurves, 25
  QQ-plot, 40, 48, 51
  Quadratic Cumul importance factors, 96
  Regression linear model, 62
  Residual Regression linear model, 62
  Specifying the file format, 25
  Superposition empirical - kernel smoothed CDF, 58
  Superposition of graphs, 73
  Superposition two points clouds, 35
Graph Manipulation
  Bounding box, 21, 25, 35, 48, 51, 58, 62, 113, 117, 124
  Show, 21, 25, 35, 48, 51, 58, 62, 91, 96, 102, 107, 113, 117, 124
  ViewImage, 21, 25, 35, 48, 51, 58, 62, 91, 96, 102, 107, 113, 117, 124

Independence Test
  ChiSquared test, 42, 44
  Pearson test, 42, 44
  Spearman test, 42, 44

Limit State Function
  Analytical formula declared in line, 76
  External wrapper, 74
  Gradient, 76
  Hessian, 76
  LinearNumericalMathFunction, 77
  Reducing the initial limit state function, 77, 79

Mixture Distribution
  Mixture, 21, 25

Numerical Math Function Manipulation
  Composition, 81
  Evaluation, 81
  Gradient, 81
  Hessian, 81
  Input - Output dimension, 81
  Marginal, 81

Optimisation Algorithm
  AbdoRacwitz, 102
  Cobyla, 102, 107
  SQP, 102

Quadratic Cumul, 96
Quantile
  Distribution evaluation, 25
  Empirical estimation, 69
  Empirical estimation , 97
  Wilks estimation, 97

Random Generator, 93, 113, 117, 124, 130, 131
Random Vector
  Extracting a sub vector, 84
  Input random vector, 32
  Output random vector, 79, 83
Ranking test
  BIC, 55
  ChiSquared, 55
  Kolmogorov, 55
Regression Linear Model
  Adjusted R^2 test, 62
  Cloud sample - Line graph, 62
  Factory, 62
  Fisher test, 62
  Residual graph, 62
  Residual test, 62
  R^2 test, 47, 62
ResponseSurface
  Linear least squares approximation, 129–131
  Linear Taylor approximation, 129
  Quadratic Taylor approximation, 129

Sample Statistics
  Rank - Sort functionnalities, 69
  Cholesky factor, 69
  Covariance, 69
  Empirical cumulative density function, 69
  Empirical quantile, 69
  Kendall's tau, 69
  Kurtosis, 69
  Min - Max, 69
  Pearson correlation coefficient, 69
  Skewness, 69
  Spearman correlation coefficient, 69
Sensitivity Analysis
  FORM probability, 102
  Min-Max, 94
  SORM probability, 107
Study
  Load, 134
  Save, 133

Threshold Probability
  Directional sampling, 117
  FORM, 102
  Importance sampling, 124
  LHS, 113
  Monte Carlo, 113
  SORM Breitung, Tvedt, Hohenbichler, 107

Usual Distribution
  Beta, 3
  Exponential, 3
  Gamma, 3
  Geometric, 3
  Gumbel, 3
  Histogram, 3
  Logistic, 3
  LogNormal, 3
  MultiNomial, 3
  Non Central Student, 3
  Normal, 3
  Poisson, 3
  Student, 3
  Triangular, 3
  Truncated distribution, 14
  TruncatedNormal, 3
  Uniform, 3
  User defined, 3
  Weibull, 3

View Image, 73

Wilks, 97