Open TURNS - Integral Compound Poisson Distribution

2 février 2011

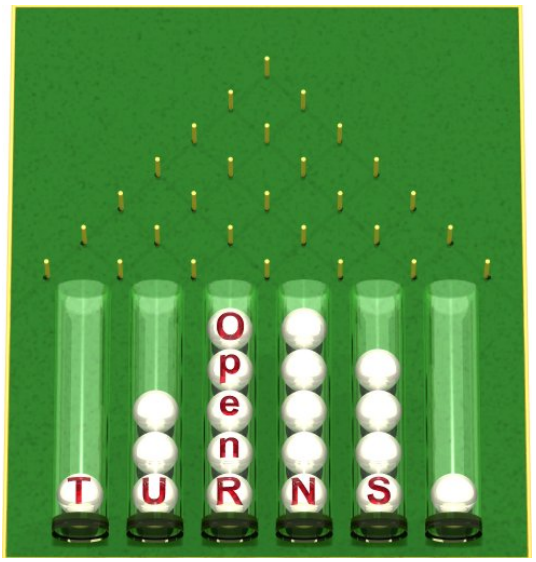

Table des matières

1 Reference guide **2**
1.1 Definition . . . 2
1.2 Distribution of the Integral Compound Poisson Distribution . . . 2
1.3 Algorithmic details . . . 2
1.3.1 Cauchy's integral formula . . . 2
1.3.2 Poisson summation formula . . . 3
1.3.3 Fast Fourier Transform (FFT) . . . 5
Bibliographie . . . 5

2 Use Cases Guide **7**
2.1 UC1 : Creation and manipulation of an *IntegralUserDefined* variable . . . 7
2.2 UC2 : Creation and manipulation of an *IntegralCompoundPoisson* variable . . . 8

3 User Manual **11**
3.1 Scripts required . . . 11
3.2 IntegralUserDefined . . . 11
3.3 IntegralUserDefinedFactory . . . 12
3.4 IntegralCompoundPoisson . . . 12
3.5 Polynomials . . . 14

1 Reference guide

1.1 Definition

In probability theory, a *compound Poisson distribution* is the sum of a Poisson-distributed number of independent identically-distributed random variables. In the simplest cases, the result can be either a continuous or a discrete distribution.
If N is a poisson distributed variable and if $(X_i)_i$ are identically distributed random variables that are mutually independent and also independent of N, then :

$$Y = \left(\sum_{i=1}^{N} X_i \right) \mathbb{1}_{N \geq 1} \tag{1}$$

is a compound Poisson distribution.

This module only focuses on variables X_i which are discrete, with finite range and integer values. We name this particular case the *Integral Compound Poisson Distribution*. It implies that the compound Poisson distribution is also discrete, with finite range and integer values. Then, it allows the use of its generatrice function in order to evaluate its probability distribution.

1.2 Distribution of the Integral Compound Poisson Distribution

Let us suppose that the variables $(X_i)_i$ are identically and independently distributed according to the X distribution and that N follows a Poisson distribution parameterised by λ.

The generatrice function of N is, for $z \in [-1, 1]$:

$$\phi_N(z) = E[z^N] = e^{-\lambda(1-z)} \tag{2}$$

We show that the generatric function of Y writes :

$$\phi_Y(z) = \phi_N \circ \phi_X(z) \tag{3}$$

Then, the probability distribution of Y is derived from its generatrice function as follows :

$$\forall n \in \mathbb{N},\ \mathbb{P}(Y = n) = \frac{1}{n!} \frac{d^{(n)} \phi_Y(z)}{dz^n} \bigg|_{z=0} \tag{4}$$

1.3 Algorithmic details

The evaluation of the integral compound Poisson distribution is based on the previous results. The references [1], [2] et [3] give more details on these results. We develop below some important points : the Cauchy's integral formula and the Poisson summation formula.

1.3.1 Cauchy's integral formula

In mathematics, Cauchy's integral formula, named after Augustin-Louis Cauchy, is a central statement in complex analysis. It expresses the fact that a holomorphic function defined on a disk is completely determined by its values on the boundary of the disk, and it provides integral formulas for all derivatives of a holomorphic function. Cauchy's celebrated formula shows that, in complex analysis, differentiation is

equivalent to integration (text extracted from Wikipedia).

Cauchy Formula : Suppose U is an open subset of the complex plane C, $f : U \to C$ is a holomorphic function and the closed disk $D = \{z : |z - z_0| = r\}$ is completely contained in U. Let γ be the circle forming the boundary of D. Then for every a in the interior of D :

Let f be a a holomorphic function on $\mathcal{U}$ an open subset of the complex plane $\mathbb{C}$, K a compact of $\mathcal{U}$ completely contained in $\mathcal{U}$ and Γ its boundary. Then for every $z_0 \in K - \Gamma$, we have :

$$f(z_0).1_\Gamma(z_0) = \frac{1}{2i\pi} \int_\Gamma \frac{f(u)}{u - z_0} du \tag{5}$$

where $1_\Gamma(z_0) = \int_\Gamma \frac{1}{u - z_0} du$.

With $\Gamma = Circle(0, r)$ and $z_0 = 0$, then the relation (5) writes :

$$f(0) = \frac{1}{2i\pi} \int_\Gamma \frac{f(u)}{u} du \tag{6}$$

and we show that :

$$f^n(0) = \frac{n!}{2i\pi} \int_\Gamma \frac{f(u)}{u^{n+1}} du \tag{7}$$

Then, with a proper parametrisation of the circle, we have :

$$f^n(0) = \frac{n!}{2\pi r^n} \int_0^{2\pi} f(re^{i\theta}) e^{-in\theta} d\theta \tag{8}$$

The relation (8) applied to the generatrice function ϕ of Y writes :

$$p_n = \frac{1}{2\pi r^n} \int_0^{2\pi} \phi(re^{i\theta}) e^{-in\theta} d\theta \tag{9}$$

1.3.2 Poisson summation formula

In mathematics, the Poisson summation formula is an equation that allows us to relate the Fourier series coefficients of the periodic summation of a function to values of the function's continuous Fourier transform. Consequently, the periodic summation of a function is completely defined by discrete samples of the original function's Fourier transform. And conversely, the periodic summation of a function's Fourier transform is completely defined by discrete samples of the original function. The Poisson summation formula was discovered by Siméon Denis Poisson and is sometimes called Poisson resummation (text extracted from Wikipedia).

The relation (9) requires the Poisson summation formula.

Let g be a function from $\mathbb{R}$ into $\mathbb{R}$ as :

$$g(u) = \sum_{k=-\infty}^{+\infty} a_k e^{iku} \tag{10}$$

then a_n writes :

$$a_n = \frac{1}{2\pi} \int_0^{2\pi} g(u) e^{-in\theta} d\theta \tag{11}$$

If we consider the periodic suite :

$$a_n^p = \sum_{k=-\infty}^{+\infty} a_{n+km} \tag{12}$$

for a given integer m, we show that a_n^p can write as :

$$a_n^p = \frac{1}{m} \sum_{k=0}^{m-1} g(\frac{2\pi k}{m}) e^{-\frac{2i\pi kn}{m}} \tag{13}$$

whioch leads to the **Poisson summation formula :**

$$\sum_{k=0}^{m-1} g(\frac{2\pi k}{m}) e^{-\frac{2i\pi kn}{m}} = m \sum_{k=-\infty}^{+\infty} a_{n+km} \tag{14}$$

The figure 1 gives an illustration of the periodic suite a_n.

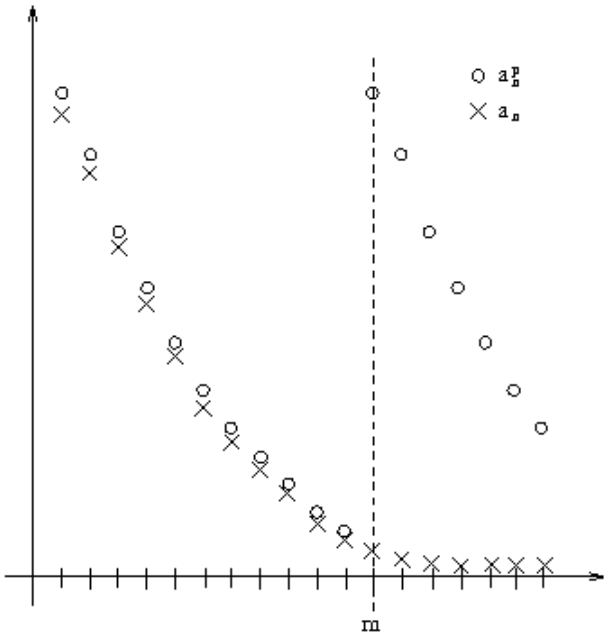

FIGURE 1 – Periodic suite a_n.

As the function g writes as a convergent serie (10), $|a_k| \underset{k\to+\infty}{\to} 0$. Then if we take m great enough, we obtain the following approximation :

$$a_n^p = \sum_{k=-\infty}^{+\infty} a_{n+km} \simeq a_n \tag{15}$$

In our probabilistic context, the function g is :

$$g(u) = \phi(re^{iu}) = \sum_{k=0}^{+\infty} p_k r^k e^{iku} \tag{16}$$

wich leads to the expression (10) with

$$\left\{ \begin{array}{ll} a_k = 0, & \forall k < 0 \\ a_k = p_k r^k & \forall k \geq 0 \end{array} \right. \tag{17}$$

The relation (14) writes, for $n < m$ and for the function g defined in (17) :

$$m \sum_{k=0}^{+\infty} a_{n+km} = \sum_{k=0}^{m-1} g(\frac{2\pi k}{m}) e^{-\frac{2i\pi kn}{m}} \tag{18}$$

By isolating the term a_n in (18), we have :

$$a_n = \frac{1}{m} \sum_{k=0}^{m-1} g(\frac{2\pi k}{m}) e^{-\frac{2i\pi kn}{m}} - \sum_{k=1}^{+\infty} a_{n+km} \tag{19}$$

which, within the probabilistic context of (16) and (17) leads to the expression of p_n :

$$p_n = \frac{1}{mr^n} \sum_{k=0}^{m-1} \phi(re^{\frac{2i\pi k}{m}}) e^{-\frac{2i\pi kn}{m}} - e_d \tag{20}$$

where e_d is the approximation error done if we neglige this term in (20). This error writes :

$$e_d = \sum_{k=1}^{+\infty} p_{n+km} r^{km} \tag{21}$$

and can be bounded, for $0 < r < 1$:

$$0 < e_d \leq \sum_{k=1}^{+\infty} r^{km} = \frac{r^m}{1 - r^m} \tag{22}$$

Thus, the precision of the evaluation of p_n is given by the choice of the couple (m, r).

We have the following final result :

$$\forall n < m,\ p_n \simeq \hat{p}_n = \frac{1}{mr^n} \sum_{k=0}^{m-1} \phi(re^{\frac{2i\pi k}{m}}) e^{-\frac{2i\pi kn}{m}} \tag{23}$$

where

$$|p_n - \hat{p}_n| \leq \frac{r^m}{1 - r^m} \tag{24}$$

1.3.3 Fast Fourier Transform (FFT)

In order to optimize the numerical evaluation of the $(\hat{p}_n)_n$ given in (23), it is recommended to use the Fast Fourier Transform (FFT) which evaluates simultaneously the values $(p_0, \ldots, p_{m-1})$ with an optimised cost.

The relation (23) presents $mr^n p_n$ as the result of a discrete Fast Fourier Transform of the suite $(\phi(re^{\frac{2i\pi k}{m}}))_{k \geq 0}$. If we chose m as a power of 2, we can evaluate $(p_0, \ldots, p_{m-1})$ thanks to the FFT. The algorithmic cost is no more $\mathcal{O}(m^2)$ but $\mathcal{O}(m \log m)$.

Références

[1] J. Abate, W. Whitt, *The Fourier-series method for inverting transforms of probability distributions.* Queueing Systems, feb. 1991.

[2] J. Stoer, R. Bulirsch, *Introduction to numerical analysis.* Springer, Third Edition, 2002.

[3] W. Feller, *Introduction to probability theory and its application.* Wiley, Second Edition, vol 2, 1971.

2 Use Cases Guide

The scripts which have been developped are compatible with Open TURNS 0.13.2. They require the package python *numpy* and use the FFT algorithm and its python implementation proposed by *numpy*.

2.1 UC1 : Creation and manipulation of an *IntegralUserDefined* variable

This Use Case explicitates how to create and manipulate an $IntegralUserDefined$ variable which is a discrete variable with finite range and integer values.

Requirements	• values of the range and associated probabilities : *range* and *weights* **type :** *range* a python list of integers, *weights* a list of reals, with unit sum or not • or *sample* a numerical sample of the discrete variable X **type :** a $NumericalSample$
Results	• a discrete variable with finite range and integer values : *myX* **type :** IntegralUserDefined

Python script for this Use Case :

```
##############################################################################
# CASE 1 : Creation from the range values and the associated probabilities
##############################################################################

# Creation of the python list of the range (integers only)
# for example : 5, 7, 32
range = [5,7,32]

# Creation of the associated probabilities list
poidweightss = [0.1, 0.2, 0.3, 0.4]

# Creation of the IntegralUserDefined variable
myX = IntegralUserDefined(range, weights)

# Other signature
myX = IntegralUserDefined(UnsignedLongCollection(range), ...
      ... NumericalPoint(weights))


##################################################################
# CASE 2 : Creation form a numerical sample
##################################################################

# Creation of the IntegralUserDefined variable
myX = IntegralUserDefinedFactory.buildImplementation(sample)
```

```
##############################################
# Manipulation of the IntegralUserDefined
##############################################

# Get the range
print "range=", myd_Sc.getRange()

# Get thes weights
print "weights=", myX.getWeights()

# Generate a realisation
print "realization=", myX.getRealization()

# Generate a numerical samplee
size = 10
print "numerical sample=", myX.getNumericalSample(size)

# Compute the probability of a particular state
print "weight of the value ", 5, "=", myX.computePDF(5)

# Compute the CDF of a particular state
print "CDF at ", 5, "=", myX.computeCDF(5)
```

2.2 UC2 : Creation and manipulation of an *IntegralCompoundPoisson* variable

This Use Case shows how to create and manipulate an *IntegralCompoundPoisson* variable, defined as :

$$Y = \left(\sum_{i=1}^{N} X_i \right) \mathbb{1}_{N \geq 1} \tag{25}$$

where N is a Poisson distributed variable parameterised by $\theta > 0$ and $(X_i)_i$ identically distributed random variables, with finite range and integer values. They are also mutually independent and also independent of N.

When a variable Y of type *IntegralCompoundPoisson* is created, all the values $p_k = \mathcal{P}(Y = k)$ are authomatically evaluated for $k \in [0, m-1]$ with $m = 2^{param}$. If the User asks for the evaluation of p_n for $n \geq m$, then Open TURNS authomatically re-evaluates all the values p_k for $k \in [0, N]$ with $N = 2^{E[log_2 n]+1}$ (ie the first power of 2 which is strictly $> n$).

The figures Fig. 2 et Fig.3 draw the probability distribution and its cumulated probability distribution of an intergal compound distribution defined by :

- $(X_i)_i$ are distributed according to the discrete distribution of range $[1, 2, 4, 7]$ associated to the respective probabilities $[0.1, 0.2, 0.3, 0.4]$,
- N is a Poisson distribution which parameter is $\theta = 20$.

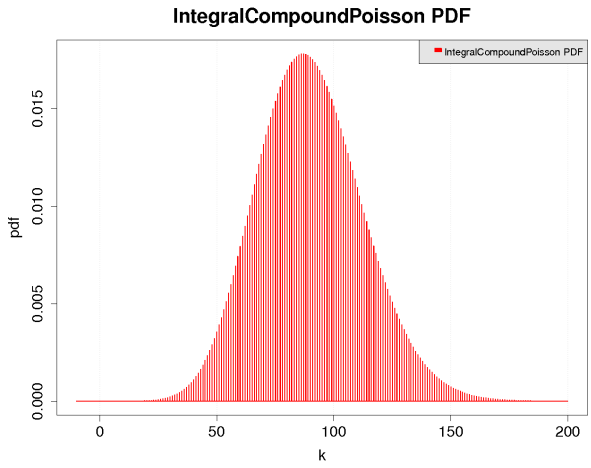

FIGURE 2 – Probability Distribution of an Integral Compound Poisson distribution.

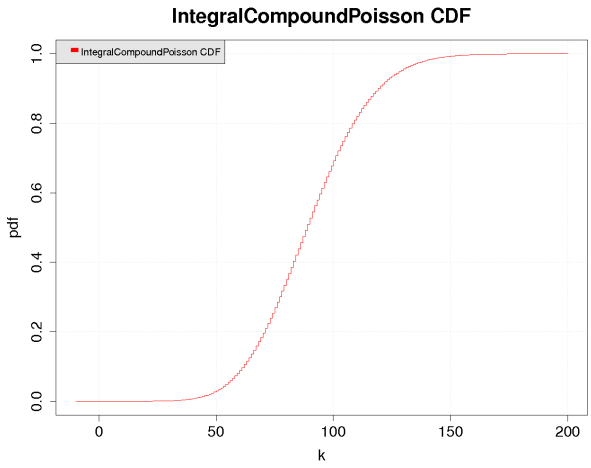

FIGURE 3 – Cumulated Probability Distribution of an Integral Compound Poisson distribution.

Requirements	• a discrete variable with finite range and integer values : *myX* **type :** a *IntegralUserDefined* • the parameter of the Poisson distribution : *myTheta* **type :** a strictly positive real
Results	• a Integral Compound Poisson variable : *myY* **type :** a *IntegralCompoundPoisson*

Python script for this UseCase :

```
#####################################################
# Creation of the IntegralCompoundPoisson
#####################################################

# Creation of the IntegralCompoundPoisson variable
myY = IntegralCompoundPoisson(myX, myTheta)


#############################################################
# Manipulation of the IntegralCompoundPoisson
#############################################################

# Get the finite discrete distribution myX of type IntegralUserDefined
print "Inside IntegralUserDefined=", myYc.getAtomDistributionD()

# Get the Poisson parameter
print " Poisson parameter=", myY.getTheta()

# Get the param m
```

```
print "parameter m=", myY.getM()

# Get the param parameter
print "parameter param=", myY.getLog2cache()

# Generate a realisation of the distribution
print "realisation=", myY.getRealization()

# Compute the PDF at one particular point of the range
print "PDF at ", point, "=", myY.computePCDF(point5)

# Compute the CDF at one particular point of the range
print "CDF at ", point, "=", myY.computeCDF(point5)

# Compute the mean value
print "mean value=", myY.getMean()

# Compute the variance
print "variance=", myY.getCovariance()

# Compute the standard deviation
print "standard deviation=", myY.getStandardDeviation()

# Compute the quantile of order q
print "quantile order", q, "=", myY.computeQuantile(q)
```

3 User Manual

This document gives details on the new methods proposed by the python scripts.

3.1 Scripts required

We describe here the python scripts required by the module :

- *IntegralUserDefined.py*, which implements the IntegralUserDefined class. This class is the UserDefined already contained in Open TURNS, specialised for integral variables.
- *IntegralUserDefinedFactory.py*, which gives the functionnalities to adjust a IntegralUserDefined variable to a set of data.
- *IntegralCompoundPoisson.py* which implements the integral compound Poisson distribution.
- *Polynomial.py* which gives functionnalities to manipulate sparse polynomials.
- *t_IntegralUserDefined_std.py* which is the test file associated to the IntegralUserDefined class.
- *t_IntegralUserDefinedFactory_std.py* which is the test file associated to the IntegralUserDefinedFactory class.
- *t_IntegralCompoundPoisson_validation.py* which is the test file associated to the IntegralCompoundPoisson class.

3.2 IntegralUserDefined

Usage : $IntegralUserDefined(range, weight)$

Arguments :

range : python list of integers or integer collection of type *UnsignedLongCollection*. For example, $range = [5, 7, 32]$ or $range = UnsignedLongCollection([5, 7, 32])$

weight : python list of the associated weights or a *NumericalPoint*. If the weights are not normalised, they are authomatically normalised by Open TURNS. For example, $weight = [0.1, 0.2, 0.3, 0.4]$ or $weight = NumericalPoint([0.1, 0.2, 0.3, 0.4])$.

Value : a *IntegralUserDefined*, which is a variable which range (*range*) is finite with integer values, associated to the weights given in *weight*.

Some methods :

getRange

Usage : $getRange()$

Arguments : none

Value : a *UnsignedLongCollection*, which is a python list of integers which are the positive integer values of the range.

getWeights

Usage : $getWeights()$

Arguments : none

Value : a *NumericalPoint*, the python list of the associated weights.

getNormalizedWeights

Usage : $getNormalizedWeights()$

Arguments : none

Value : a *NumericalPoint*, the python list of the associated normalised weights.

3.3 IntegralUserDefinedFactory

Usage : *IntegralUserDefinedFactory* is a static class which is used through its unique method *buildImplementat*

One method : *buildImplementation*

Usage : *buildImplementation*(*echantillon*)

Arguments : *sample* : un *NumericalSample* which represents a numerical sample of the random variable. Its values must be integers.

Value : a *IntegralUserDefined*, a random variable which range is defined by the values f the *sample*. The probability of k is estimated by $p_k = \frac{N_k}{n}$ where n is the size of *sample* and N_k the number of values of the *sample* equal to k.

3.4 IntegralCompoundPoisson

Usage : strut

IntegralCompoundPoisson(*atomDistribution*, *theta*)

IntegralCompoundPoisson(*atomDistribution*, *theta*, *param*)

Arguments :

atomDistribution : a *IntegralUserDefined* distribution

theta : a strictly positive real

param : an integer. By default, $param = 10$. This parameter is such that $m = 2^{param}$ where m is defined in the relation (23) with the relation $m = 2^{param}$.

Value : a *IntegralCompoundPoisson*, which the variable Y defined as :

$$Y = \left(\sum_{i=1}^{N} X_i \right) \mathbb{1}_{N \geq 1} \tag{26}$$

where N is a poisson distributed variable, the variables X_i are discrete, with finite range and integer values, identically distributed, mutually independent and also independent of N.

When a variable Y of type *IntegralCompoundPoisson* is created, all the values $p_k = \mathcal{P}(Y = k)$ are authomatically evaluated for $k \in [0, m-1]$ with $m = 2^{param}$. If the User asks for the evaluation of p_n for $n \geq m$, then Open TURNS authomatically re-evaluates all the values p_k for $k \in [0, N]$ with $N = 2^{E[log_2 n]+1}$ (ie the first power of 2 which is strictly $> n$).

Some methods :

getAtomDistribution

Usage : *getAtomDistribution*()

Arguments : none.

Value : the finite discrete distribution of the variables X_i of type *IntegralUserDefined*

getTheta

Usage : *getTheta*()

Arguments : none.

Value : un positive real, the Poisson distribution parameter.

getLog2Cache

Usage : $getLog2Cache()$

Arguments : none.

Value : an integer, the parameter $param$. By default, $param = 10$. This parameter is such that $m = 2^{param}$ where m is defined in the relation (23) with the relation $m = 2^{param}$.

getM

Usage : $getM()$

Arguments : none.

Value : the integer m such that $m = 2^{param}$. By default, $m = 2^{10}$. m is defined in the relation (23).

getRealization

Usage : $getRealization()$

Arguments : none.

Value : an integer inside the of Y.

getNumericalSample

Usage : $getNumericalSample(size)$

Arguments : $size$: an integer, the number of realistaions to generate a numerical sample of Y.

getMean

Usage : $getMean()$

Arguments : none.

Value : the mean of the random variable.

getCovariance

Usage : $getCovariance()$

Arguments : none.

Value : the variance of the random variable.

getStandardDeviation

Usage : $getStandardDeviation()$

Arguments : none.

Value : the standard deviation of the random variable.

computeCDF

Usage :

$computeCDF(value)$

$computeCDF(value, True)$

Arguments : *value* : an integer.

True : a boolean wich is False when not mentioned.

Value : a real between 0 and 1. When the boolean is FALSE, the function computes the CDF at *value*. In the other case, it computes the tail of the CDF at *value*.

computePDF

Usage : $computePDF(value)$

Arguments : *value* : an integer.

Value : a real which is the PDF value at point *value*.

computeQuantile

Usage : $computeQuantile(q)$

Arguments : q : a real between 0 and 1.

Value : the quantile of order q.

drawPDF

Usage : $drawPDF(xMin, xMax)$

Arguments : $(xMin, xMax)$: the min and max values of the graph.

Value : a *Graph* which contains the PDF curve.

drawCDF

Usage : $drawCDF(xMin, xMax)$

Arguments : $(xMin, xMax)$: the min and max values of the graph.

Value : a *Graph* which contains the CDF curve.

3.5 Polynomials

The file *Polynomials.py* contains a set of functions linked to the manipulation of polynomials.

denseToSparse

Usage : $denseToSparse(polynomial)$

Arguments : *polynomial* : a $UniVariatePolynomial$ which is the polynomials class in Open TURNS.

Value : $degrees, coefficients$: respectively a $UnsignedLongCollection$ and $NumericalPoint$ which are the degrees of the monoms which coefficient is not nul and the associated coefficient.

buildUniVariatePolynomial

Usage : $buildUniVariatePolynomial(degrees, coefficients)$

Arguments : $degrees, coefficients$: respectively a $UnsignedLongCollection$ and $NumericalPoint$ which gives the degrees of the monoms which coefficient is not nul and the associated coefficient.

Value : *polynomial* : a $UniVariatePolynomial$ which is the polynomials class in Open TURNS, with a full representation : the coefficients of all the monoms are stocked, some eventually equal to 0.

truncateUniVariatePolynomial

Usage : $truncateUniVariatePolynomial(polynomial, truncation)$

Arguments :

$polynomial$: a $UniVariatePolynomial$ which is the polynomials class in Open TURNS.

$truncation$: an integer.

Value : a $UniVariatePolynomial$: this is the initial polynomials truncated to the degree $truncation+1$. The degree of the truncated polynomials is $truncation$.