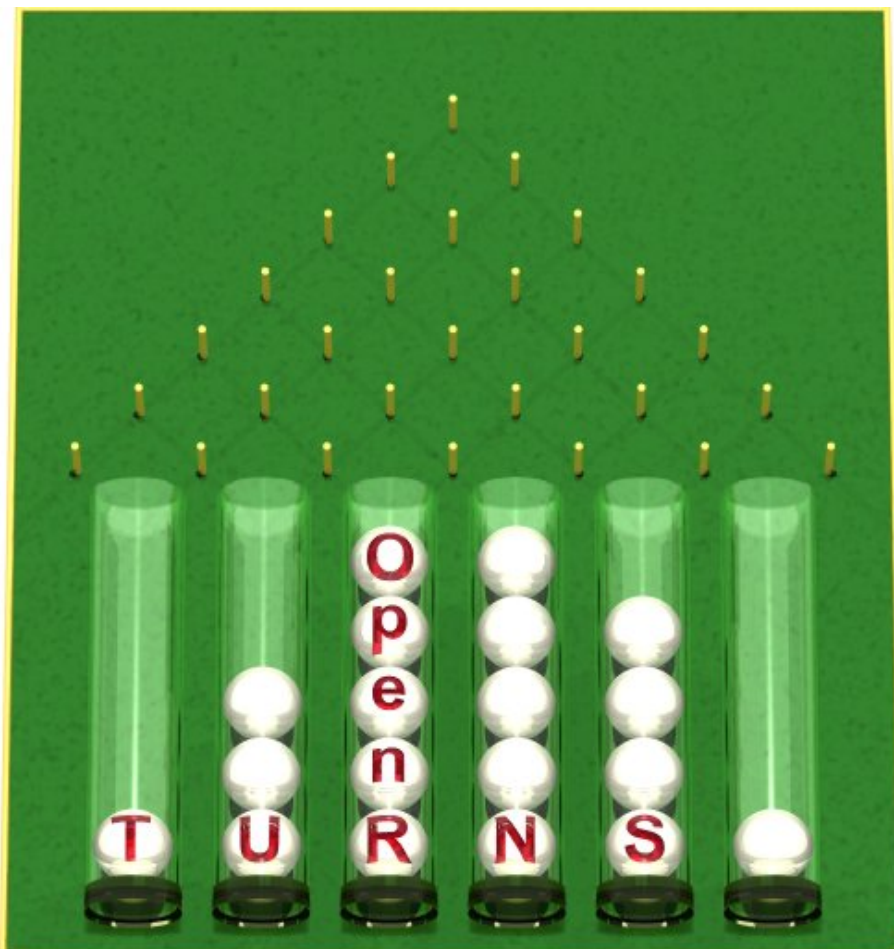


User Manual for the Textual User Interface

Open TURNS version 1.0

Documentation built from package openturns-doc-April2012

April 20, 2012



Contents

1	Base Objects	9
1.1	BoolCollection	9
1.2	Description	10
1.3	Indices	11
1.4	Domain	12
1.5	HistogramPair	13
1.6	HistogramPairCollection	14
1.7	HistoryStrategy	15
1.8	Interval	17
1.9	Matrix	20
1.9.1	Matrix	20
1.9.2	ComplexMatrix	22
1.9.3	CorrelationMatrix	24
1.9.4	HermitianMatrix	25
1.9.5	SquareMatrix	27
1.9.6	Tensor	28
1.9.7	TriangularComplexMatrix	30
1.10	NumericalPoint	32
1.11	NumericalPointCollection	34
1.12	NumericalScalarCollection	35
1.13	UserDefinedPair	36
1.14	UserDefinedPairCollection	37
2	Probabilistic modeling	38
2.1	Distribution	38
2.2	Usual Distributions	46
2.2.1	Arcsine	46
2.2.2	Bernoulli	48
2.2.3	Beta	49
2.2.4	Binomial	51
2.2.5	Burr	52
2.2.6	Chi	53
2.2.7	ChiSquare	54
2.2.8	Dirichlet	55
2.2.9	Epanechnikov	56
2.2.10	Exponential	57
2.2.11	FisherSnedecor	58
2.2.12	Gamma	59
2.2.13	Geometric	61
2.2.14	Gumbel	62
2.2.15	Histogram	64
2.2.16	InverseNormal	65
2.2.17	Laplace	66
2.2.18	Logistic	67
2.2.19	LogNormal	68
2.2.20	LogUniform	70

2.2.21	Multinomial	71
2.2.22	NegativeBinomial	72
2.2.23	NonCentralChiSquare	73
2.2.24	NonCentralStudent	74
2.2.25	Normal	76
2.2.26	Poisson	78
2.2.27	Rayleigh	79
2.2.28	Rice	80
2.2.29	Student	81
2.2.30	Trapezoidal	83
2.2.31	Triangular	85
2.2.32	TruncatedNormal	87
2.2.33	Uniform	89
2.2.34	UserDefined	90
2.2.35	Weibull	91
2.2.36	ZipfMandelbrot	93
2.3	TruncatedDistribution	94
2.4	Copulas	95
2.4.1	Copula	95
2.4.2	ClaytonCopula	96
2.4.3	FrankCopula	97
2.4.4	GumbelCopula	98
2.4.5	IndependentCopula	99
2.4.6	MinCopula	100
2.4.7	NormalCopula	101
2.4.8	SklarCopula	102
2.4.9	ComposedCopula	103
2.5	ComposedDistribution	104
2.6	Linear combination of probability density functions	105
2.6.1	Mixture	105
2.6.2	KernelMixture	106
2.6.3	KernelSmoothing	107
2.7	Affine combinations of independent univariate random variables	109
2.7.1	RandomMixture	109
2.8	Random vector	111
2.8.1	RandomVector	111
2.8.2	ConditionalRandomVector	114
3	Design of experiments	115
3.1	Experiment	115
3.2	Stratified Design of Experimentss	116
3.2.1	StratifiedExperiment	116
3.2.2	Axial	117
3.2.3	Factorial	118
3.2.4	Composite	119
3.2.5	Box	120
3.3	WeightedExperiment	121
3.4	Random Weighted Design of Experimentss	122

3.4.1	LHSExperiment	122
3.4.2	MonteCarloExperiment	123
3.4.3	ImportanceSamplingExperiment	124
3.5	Deterministic Weighted Experiments	125
3.5.1	FixedExperiment	125
3.5.2	LowDiscrepancyExperiment	126
3.5.3	GaussProductExperiment	127
3.6	Low Discrepancy Sequences	128
3.6.1	FaureSequence	128
3.6.2	HaltonSequence	129
3.6.3	ReverseHaltonSequence	130
3.6.4	HaselgroveSequence	131
3.6.5	SobolSequence	132
4	Functions	133
4.1	DualLinearCombinationEvaluationImplementation	133
4.2	DualLinearCombinationGradientImplementation	135
4.3	DualLinearCombinationHessianImplementation	136
4.4	NumericalMathFunction	137
4.5	NumericalMathEvaluationImplementation	146
4.6	NumericalMathGradientImplementation	147
4.7	NumericalMathHessianImplementation	148
4.8	CenteredFiniteDifferenceMathGradientImplementation	149
4.9	NonCenteredFiniteDifferenceMathGradientImplementation	150
4.10	CenteredFiniteDifferenceMathHessianImplementation	151
4.11	FiniteDifferenceStep	152
4.12	ConstantStep	153
4.13	BlendedStep	154
4.14	MarginalTransformationEvaluation	155
4.15	SpatialFunction	156
4.16	TemporalFunction	157
4.17	DynamicalFunction	158
5	FFT	160
5.0.1	KissFFT	161
6	Graphs	162
6.1	Graph	162
6.2	Drawable	166
6.3	BarPlot	172
6.4	Cloud	173
6.5	Contour	174
6.6	Curve	176
6.7	Staircase	177
6.8	Pie	178
6.9	Show	179

7	Optimization	180
7.1	Minimization of the distance under an equality constraint	180
7.1.1	NearestPointAlgorithm	180
7.1.2	Cobyla	182
7.1.3	CobylaSpecificParameters	183
7.1.4	AbdoRackwitz	184
7.1.5	AbdoRackwitzSpecificParameters	185
7.1.6	SQP	186
7.1.7	SQPSpecificParameters	187
7.1.8	NearestPointAlgorithmImplementationResult	188
7.2	Optimization of a function under an inequality constraint	189
7.2.1	BoundConstrainedAlgorithm	189
7.2.2	TNC (Truncated Newton Constrained)	191
7.2.3	TNCSpecificParameters	192
7.2.4	BoundConstrainedAlgorithmImplementationResult	194
8	Polynomials	195
8.1	UniVariatePolynomial	195
8.2	PolynomialCollection	197
8.3	ProductPolynomialEvaluationImplementation	198
9	Response Surface : Parametric Approximation	199
9.1	Taylor approximation	199
9.1.1	LinearTaylor	199
9.1.2	QuadraticTaylor	201
9.2	Least squares approximation	203
9.2.1	LinearLeastSquares	203
9.2.2	QuadraticLeastSquares	205
10	Response Surface : Functional Chaos Expansion	207
10.1	FunctionalChaosAlgorithm	207
10.2	FunctionalChaosResult	208
10.3	Construction of the multivariate orthogonal basis	211
10.3.1	OrthogonalBasis	211
10.3.2	OrthogonalProductPolynomialFactory	212
10.3.3	OrthogonalUniVariatePolynomialFamily	213
10.3.4	StandardDistributionPolynomialFactory	214
10.3.5	CharlierFactory	215
10.3.6	HermiteFactory	216
10.3.7	JacobiFactory	217
10.3.8	KrawtchoukFactory	218
10.3.9	LaguerreFactory	219
10.3.10	LegendreFactory	220
10.3.11	MeixnerFactory	221
10.3.12	OrthonormalizationAlgorithm	222
10.3.13	ChebychevAlgorithm	223
10.3.14	GramSchmidtAlgorithm	224
10.3.15	EnumerateFunction	225

10.3.16	LinearEnumerateFunction	227
10.3.17	HyperbolicAnisotropicEnumerateFunction	228
10.4	Construction of the truncated multivariate orthogonal basis	229
10.5	Construction of the truncated multivariate orthogonal basis	230
10.5.1	AdaptiveStrategy	230
10.5.2	FixedStrategy	231
10.5.3	SequentialStrategy	232
10.5.4	CleaningStrategy	233
10.6	Evaluation of the coefficients	234
10.6.1	ProjectionStrategy	234
10.6.2	LeastSquaresStrategy	235
10.6.3	IntegrationStrategy	236
10.6.4	ApproximationAlgorithmFactory	237
10.6.5	PenalizedLeastSquaresAlgorithmFactory	238
10.6.6	LeastSquaresMetaModelSelectionFactory	239
10.6.7	BasisSequenceFactory	240
10.6.8	LAR	240
10.6.9	FittingAlgorithm	241
10.6.10	CorrectedLeaveOneOut	241
10.6.11	KFold	241
10.7	FunctionalChaosRandomVector	242
11	Statistics on sample	244
11.1	Numerical Sample	244
11.1.1	NumericalComplexCollection	244
11.1.2	NumericalSample	245
11.2	Distribution factory	251
11.2.1	DistributionImplementationFactory	251
11.3	Correlation analysis	252
11.3.1	CorrelationAnalysis	252
11.4	Sensitivity Analysis	254
11.4.1	SensitivityAnalysis	254
11.5	Fitting test	256
11.5.1	TestResult	256
11.5.2	FittingTest	257
11.5.3	VisualTest	259
11.5.4	NormalityTest	262
11.5.5	HypothesisTest	263
11.5.6	LinearModelTest	267
11.6	Linear model	269
11.6.1	LinearModelFactory	269
11.6.2	LinearModel	270
12	Stochastic process	271
12.1	General common objects	272
12.1.1	Process	272
12.1.2	RegularGrid	274
12.1.3	TimeSeries	275

12.1.4	ProcessSample	277
12.1.5	TimeSeriesCollection	279
12.2	Temporal information	280
12.2.1	CovarianceModel	280
12.2.2	StationaryCovarianceModel	282
12.2.3	ExponentialModel	283
12.3	Spectral information	285
12.3.1	SpectralModel	285
12.3.2	CauchyModel	287
12.3.3	UserDefinedSpectralModel	289
12.3.4	SpectralModelFactory	289
12.3.5	WelchFactory	291
12.3.6	FilteringWindows	292
12.3.7	Hanning	293
12.3.8	Hamming	294
12.4	Link Temporal - Spectral information	295
12.4.1	SecondOrderModel	295
12.4.2	ExponentialCauchy	297
12.5	Normal process	299
12.5.1	NormalProcess	299
12.5.2	SpectralNormalProcess	300
12.5.3	TemporalNormalProcess	301
12.6	ARMA	302
12.6.1	ARMA	302
12.6.2	ARMACoefficients	305
12.6.3	ARMAState	307
12.6.4	BoxCoxFactory	309
12.6.5	BoxCoxTransform	310
12.6.6	InverseBoxCoxTransform	312
12.6.7	TrendFactory	314
12.6.8	TrendTransform	316
12.6.9	InverseTrendTransform	318
12.7	ARMAFactory	319
12.7.1	ARMAFactory	319
12.7.2	WhittleFactoryState	319
12.7.3	WhittleFactory	321
12.8	RandomWalk	324
12.9	WhiteNoise	325
12.10	CompositeProcess	326
12.10.1	CompositeProcess	326
13	Threshold probability : Reliability algorithms	327
13.1	Reliability Algorithms	327
13.1.1	Analytical	327
13.1.2	AnalyticalResult	329
13.1.3	Event	332
13.1.4	StandardEvent	333
13.1.5	FORM	334

13.1.6 FORMResult	335
13.1.7 SORM	336
13.1.8 SORMResult	337
13.2 The Strong Maximum Test	338
13.2.1 StrongMaximumTest	338
14 Threshold probability : Simulation algorithms	341
14.1 RandomGenerator	342
14.2 Wilks	343
14.3 Simulation	344
14.4 MonteCarlo	347
14.5 LHS	348
14.6 RandomizedLHS	349
14.7 DirectionalSampling	350
14.7.1 DirectionalSampling	350
14.7.2 RootStrategy	351
14.7.3 RiskyAndFast	352
14.7.4 MediumSafe	353
14.7.5 SafeAndSlow	354
14.7.6 SamplingStrategy	355
14.7.7 RandomDirection	356
14.7.8 OrthogonalDirection	357
14.7.9 Solver	358
14.7.10 Bisection	359
14.7.11 Brent	360
14.7.12 Secant	361
14.8 ImportanceSampling	362
14.9 PostAnalyticalSimulation	363
14.10 PostAnalyticalImportanceSampling	364
14.11 PostAnalyticalControlledImportanceSampling	365
14.12 QuasiMonteCarlo	366
14.13 RandomizedQuasiMonteCarlo	367
14.14 SimulationResult	368
15 Taylor decomposition of the limit state function	370
15.1 QuadraticCumul	370

1 Base Objects

In this section a description of general objects is given. These objects are used in the different following sections.

1.1 BoolCollection

Usage :

BoolCollection(size)

BoolCollection(size, value)

Arguments :

size : an integer, the size of the boolean values collection. It must be > 0 .

integer : an integer, the boolean value. It must be ≥ 0 .

Value : a BoolCollection

in the first usage, the BoolCollection is a list a size *size* with the value 0.

in the second usage, the BoolCollection is a list a size *size* with the value *integer*. If *integer* > 0 , it corresponds to *True*. If *integer* = 0, it corresponds to *False*.

Some methods :

add

Usage : *add(value)*

Arguments : *value* : an integer, the boolean value to be added. It must be ≥ 0 .

Value : a BoolCollection which size has been increased of 1 and which last value is *value*.

getSize

Usage : *getSize()*

Arguments : none

Value : an integer, the number of boolean values.

resize

Usage : *resize(newsize)*

Arguments : *newsize* : an integer, the new size of the collection of boolean values.

Value : a BoolCollection which size has been modified to *newsize*. If *newsize* $> size$, then the added boolean values are equal to 0. If *newsize* $< size$, the BoolCollection is restricted to its first *newsize* components.

at

Usage : *at(i)*

Arguments : *i* : an integer. Must be \leq to the BoolCollection size.

Value : an integer, the value of the component *i*.

1.2 Description

Usage :

Description(dim)

Description(dim, name)

Description(sequence)

Description(array)

Arguments :

dim : an integer, the dimension of the Description

name : a string to name the Description

sequence : a python list / tuple of strings

array : an 1-d string numpy array

Value : a Description

Some methods :

[]

Usage : *Description*[*i*]

Arguments : *i* : an integer, constraint : $0 \leq i \leq dim - 1$

Value : a string, the description of the (*i* + 1)-th element of the Description

add

Usage : *add(str)*

Arguments : *str* : a string

Value : an element is added to the Description which name is *str*

getSize

Usage : *getSize()*

Arguments : none

Value : an integer, the size of the Description (*dim*)

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the Description

setName

Usage : *setName(name)*

Arguments : *name* : a string to name the Description

Value : the Description is then named *name*

1.3 Indices

Usage :

Indices(dim)

Indices(sequence)

Arguments :

dim : an integer, the size of the collection

sequence : a python list / tuple of integers

Value : a Indices

Details :

This object represents a set of integers

Some methods :

[]

Usage : *indices[i]*

Arguments : *i* : an integer, constraint : $0 \leq i \leq dim - 1$

Value : a string, the description of the $(i + 1)$ -th element of the Indices

add

Usage : *add(index)*

Arguments : *index* : an integer

Value : an element is added to the Indices which value is *index*

getSize

Usage : *getSize()*

Arguments : none

Value : an integer, the size of the Indices (*dim*)

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the Indices

setName

Usage : *setName(name)*

Arguments : *name* : a string to name the Indices

Value : the Indices is then named *name*

1.4 Domain

Usage :

Domain()

Domain(a, b)

Arguments :

a : a NumericalPoint of dimension *d*, the "lower bound" of the domain

a : a NumericalPoint of dimension *d*, the "upper bound" of the domain

Value : a Domain.

while calling the second constructor, the domain is an Interval (similar to call *Interval(a, b)*)

Examples :

[a, b] : Domain(a, b)

[a, b] : Domain(Interval(a, b))

Some methods :

getDimension

Usage : *getDimension()*

Arguments : no argument

Value : an integer, the dimension of the Domain (returns *dim*)

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the Domain

contains

Usage : *contains(point)*

Arguments : a NumericalPoint with the same dimension as the current domain's dimension

Value : a Bool telling if the given point is inside of the domain or not.

setName

Usage : *setName(name)*

Arguments : name : a string

Value : the Domain is named *name*

1.5 HistogramPair

Usage : *HistogramPair*(h, l)

Arguments :

h : a real value, the height of each element of the Histogram

l : a real value, the width of each element of the Histogram

Value : a HistogramPair

Details :

This object is used to build a HistogramPairCollection (hence, also used to create an Histogram)

1.6 HistogramPairCollection

Usage : *HistogramPairCollection(dim)*

Arguments : *dim* : an integer, the number of elements of the HistogramPairCollection

Value : a HistogramPairCollection, to be filled after

Some methods :

[]

Usage : *HistogramPairCollection[i]*

Arguments : *i* : an integer, must be $< dim$

Value : a HistogramPair, the $(i + 1)$ -th of the HistogramPairCollection

add

Usage : *add(HistP)*

Arguments : *HistP* : a HistogramPair

Value : an HistogramPairCollection of $Size = dim + 1$ with instance of the $dim + 1$ -th element of the HistogramPairCollection

getSize

Usage : *getSize()*

Arguments : no argument

Value : an integer, the size of HistogramPairCollection (returns *dim*)

str

Usage : *str()*

Arguments : no argument

Value : a string with elements of HistogramPairCollection

1.7 HistoryStrategy

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. Open TURNS proceeds as follows :
 1. it stores the first $2N$ simulations : the size of the stored sample is $2N$,
 2. it selects only 1 out of 2 of the stored simulations : then the size of the stored sample decreases to N (this is the *compact* step),
 3. it stores the next N simulations when selecting 1 out of 2 of the next simulations : the size of the stored sample is $2N$,
 4. it selects only 1 out of 2 of the stored simulations : then the size of the stored sample decreases to N ,
 5. it stores the next N simulations when selecting 1 out of 4 of the next simulations : the size of the stored sample is $2N$,
 6. then it keeps on until reaching the stop criteria.

The stored numerical sample will have a size within N and $2N$. This strategy is proposed by the *Compact* class which requires to specify the number of points to store.

Usage : *HistoryStrategy()*

Arguments : none

Value : a HistoryStrategy that can be : Null, Compact, Full or Last.

Some methods :

getSample

Usage : *getSample()*

Arguments : none

Value : a NumericalSample which is the collection of points stored by the history strategy.

reset

Usage : *reset()*

Arguments : none

Value : none. It erases the previously stored points.

store

Usage :

store(point)
store(sample)

Arguments :

point : a NumericalPoint,
sample : a NumericalSample

Value : none. It adds the unique *point* or all the point of the *sample* in the natural order to the history.

1.8 Interval

Usage :

Interval()

Interval(dim)

Interval(lowerBound, upperBound)

Interval(lowerBound, upperBound, finiteLowerBound, finiteUpperBound)

Arguments :

dim : an integer, the dimension of the interval.

lowerBound : a scalar or a NumericalPoint, the lower bound of the interval.

upperBound : a scalar or a NumericalPoint, the upper bound of the interval.

Note: the lowerBound and the upperBound must be of the same type: both NumericalPoint must have the same dimension in case of NumericalPoint.

finiteLowerBound: a BoolCollection, flags telling for each component of the lower bound whether it is finite or not.

finiteUpperBound: a BoolCollection, flags telling for each component of the upper bound whether it is finite or not.

Note: the meaning of a flag is: if $flag_i$ is true, the corresponding component of the given bound is finite and its value is given by $bound_i$. If not, the corresponding component is infinite and its value is either $-\infty$ if $bound_i < 0$ or $+\infty$ if $bound_i \geq 0$.

Value : an Interval.

No parameter leads to the interval $[0, 1]$

The second usage leads to the finite interval $[0, 1]^{dim}$

The third usage leads to the finite interval:

$$[lowerBound_0, upperBound_0] \times \dots \times [lowerBound_{dim-1}, upperBound_{dim-1}]$$

It is allowed to have $lowerBound_i \geq upperBound_i$ for some i : it simply defines an empty interval.

The fourth usage allows to define partially infinite intervals. The value of the infinite bounds is defined according to the rule mentioned above.

Some Examples :

$[a, b]$: *Interval(a, b)*

$[a, +\infty]$:

boundLow = *NumericalPoint(1, a)*

boundUp = *NumericalPoint(1, 1)*

boolLow = *BoolCollection(1, 1)*

boolUp = *BoolCollection(1, 0)*

int = *Interval(boundLow, boundUp, boolLow, boolUp)*

$[-\infty, B]$:

```

boundLow = NumericalPoint(1, -1)
boundUp = NumericalPoint(1, b)
boolLow = BoolCollection(1, 0)
boolUp = BoolCollection(1, 1)
int = Interval(boundLow, boundUp, boolLow, boolUp)

```

Some methods :

*: right multiplication by a scalar s using *interval* * s .

+: addition of two intervals.

–: subtraction of two intervals.

contains

Usage : *contains(point)*

Arguments : *point* : a NumericalPoint with the same dimension as the current interval

Value : a Bool telling if the given point is inside the interval or not.

numericallyContains

Usage : *numericallyContains(point)*

Arguments : *point* : a NumericalPoint with the same dimension as the current interval.

Value : a Bool telling if the given point is inside the interval or not given the numerical truncation of any infinite interval. In case of bounded intervals, the numerical bounds coincide with the real bounds by default. It is possible to change the numerical bounds with the method *setLowerBound* and *setUpperBound*.

getDimension

Usage : *getDimension()*

Arguments : no argument

Value : an integer, the dimension of the Interval (returns *dim*)

getFiniteLowerBound

Usage : *getFiniteLowerbound()*

Arguments : no argument

Value : a BoolCollection of flags. If the i -th element is 0, the corresponding component of the lower bound is infinite and its value is given according to the rule given above. Otherwise, it is finite.

getFiniteUpperBound

Usage : *getFiniteUpperbound()*

Arguments : no argument

Value : a BoolCollection of flags. If the i -th element is 0, the corresponding component of the upper bound is infinite and its value is given according to the rule given above. Otherwise, it is finite.

getLowerBound

Usage : *getLowerbound()*

Arguments : no argument

Value : a NumericalPoint, the value of the lower bound or the sign of the component if it is infinite.

getUpperBound

Usage : *getUpperbound()*

Arguments : no argument

Value : a NumericalPoint, the value of the upper bound or the sign of the component if it is infinite.

setFiniteLowerBound

Usage : *setFiniteLowerbound(flag)*

Arguments : a BoolCollection

Value : no value returned

setFiniteUpperBound

Usage : *setFiniteUpperbound(flag)*

Arguments : a BoolCollection

Value : no value returned

setLowerBound

Usage : *setLowerbound(bound)*

Arguments : a NumericalPoint

Value : no value returned

setUpperBound

Usage : *setUpperbound(bound)*

Arguments : a NumericalPoint

Value : no value returned

intersect

Usage : *intersect(other)*

Arguments : an Interval of the same dimension

Value : an interval corresponding to the intersection of the current interval with *other*.

join

Usage : *join(other)*

Arguments : an Interval of the same dimension

Value : the smallest interval that contains both the current interval and *other*

isEmpty

Usage : *isEmpty()*

Arguments : no argument

Value : a Bool telling if the interior of the interval is empty or not.

1.9 Matrix

1.9.1 Matrix

Usage :

Matrix(n_r, n_c)

Matrix($n_r, n_c, values$)

Arguments :

n_r : an integer, the number of rows of the Matrix

n_c : an integer, the number of columns of the Matrix

$values$: a NumericalScalarCollection with $n_r \times n_c$ elements

Value : a Matrix

while using the first parameters set, the matrix is filled with 0.

while using second parameters set, the Matrix contains values of the NumericalScalarCollection. The matrix is filled by row

Some methods :

[,]

Usage : *Matrix*[i, j]

Arguments :

i : an integer, constraint : $0 \leq i \leq n_r - 1$

j : an integer, constraint : $0 \leq j \leq n_c - 1$

Value : a real value, the (i, j) element of the Matrix

getNbColumns

Usage : *getNbColumns*()

Arguments : none

Value : an integer : the number of column n_c

getNbRows

Usage : *getNbRows*()

Arguments : none

Value : an integer : the number of row n_r

transpose

Usage : *transpose*()

Arguments : none

Value : the transposed Matrix

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the Matrix

setName

Usage : *setName(name)*

Arguments : name : a string

Value : the Matrix is named *name*

solveLinearSystem

Usage : *solveLinearSystem(y)*

Arguments : *y* a NumericalPoint of dimension n_r (the number of rows of the Matrix)

Value : NumericalPoint, *x*, such that

$$\text{Matrix} * \mathbf{x} = \mathbf{y}$$

1.9.2 ComplexMatrix

Usage :

ComplexMatrix(n_r, n_c)
ComplexMatrix($n_r, n_c, values$)

Arguments :

n_r : an integer, the number of rows of the ComplexMatrix
 n_c : an integer, the number of columns of the ComplexMatrix
 $values$: a NumericalComplexCollection with $n_r \times n_c$ elements (the collection might also be a NumericalScalarCollection)

Value : a ComplexMatrix

while using the first parameters set, the matrix is filled with (0, 0).

while using second parameters set, the ComplexMatrix contains values of the NumericalComplexCollection. The complex matrix is filled by row

Some methods :

[,]

Usage : *ComplexMatrix*[i, j]

Arguments :

i : an integer, constraint : $0 \leq i \leq n_r - 1$

j : an integer, constraint : $0 \leq j \leq n_c - 1$

Value : a complex value, the (i, j) element of the ComplexMatrix

conjugate

Usage : *conjugate*()

Arguments : none

Value : the conjugated ComplexMatrix, ie the ComplexMatrix such as $\underline{\underline{M}}_{i,j} = \overline{\underline{\underline{M}}}_{i,j}$

conjugateTranspose

Usage : *conjugateTranspose*()

Arguments : none

Value : the conjugated and transposed ComplexMatrix, ie the ComplexMatrix such as $\underline{\underline{M}}_{j,i} = \overline{\underline{\underline{M}}}_{i,j}$

getName

Usage : *getName*()

Arguments : none

Value : a string, the name of the ComplexMatrix

getNbColumns

Usage : *getNbColumns*()

Arguments : none

Value : an integer : the number of column n_c

getNbRows

Usage : *getNbRows()*

Arguments : none

Value : an integer : the number of row n_r

transpose

Usage : *transpose()*

Arguments : none

Value : the transposed ComplexMatrix

setName

Usage : *setName(name)*

Arguments : name : a string

Value : the ComplexMatrix is named *name*

1.9.3 CorrelationMatrix

Usage :

CorrelationMatrix(dim)

CorrelationMatrix(dim, values)

Arguments :

dim : an integer, the dimension of the CorrelationMatrix (square matrix with *dim* rows and *dim* colons)

values : a NumericalScalarCollection of dimension dim^2 which contains values to put in the CorrelationMatrix, filled by rows. When these values are not specified, the CorrelationMatrix is initialized to the identity matrix.

Value : a CorrelationMatrix

while using the first parameters set, the correlation matrix is the identity matrix

while using second parameters set, the correlation matrix contains the specified values, filled by row

Some methods :

str

Usage : *str()*

Arguments : no argument

Value : a string giving the description of the (class, name, dimension, values)

transpose

Usage : *transpose()*

Arguments : no argument

Value : a CorrelationMatrix, the transposed CorrelationMatrix

computeDeterminant

Usage : *computeDeterminant()*

Arguments : no argument

Value : a real value giving the determinant of the CorrelationMatrix

computeEigenValues

Usage : *computeEigenValues()*

Arguments : no argument

Value : a NumericalPoint giving the eigen values of the CorrelationMatrix

Links : [see docref_B121_ChoixLoi](#)

1.9.4 HermitianMatrix

Usage :

HermitianMatrix(dim)

Arguments :

dim : an integer, the dimension of the HermitianMatrix (square matrix with *dim* rows and *dim* colons)

Value : HermitianMatrix

while using the first parameters set, the HermitianMatrix is filled with (0,0). It is not possible to fill the matrix from a collection of complex values (to be done later)

Some methods :

computeCholesky

Usage : *conjugate()*

Arguments : none

Value : the Cholesky factor \underline{G} , ie the ComplexMatrix such as the $\underline{G} * \underline{G}^*$ is the initial matrix

conjugate

Usage : *conjugate()*

Arguments : none

Value : the conjugated Hermitian, ie the matrix such as $\underline{M}_{i,j} = \overline{\underline{M}_{i,j}}$

conjugateTranspose

Usage : *conjugateTranspose()*

Arguments : none

Value : the conjugated and transposed HermitianMatrix, ie the matrix such as $\underline{M}_{j,j} = \overline{\underline{M}_{i,j}}$

getDimension

Usage : *getDimension()*

Arguments : none

Value : an integer, the dimension of the HermitianMatrix (it returns *dim*)

power

Usage : *power(n)*

Arguments : an integer

Value : the power of the matrix, ie the HermitianMatrix $\underline{\underline{M}}_n$ such as $\underline{\underline{M}}_n = \underline{\underline{M}} \times \underbrace{\underline{\underline{M}} \dots \times \underline{\underline{M}}}_{n \text{ times}}$

transpose

Usage : *transpose()*

Arguments : none

Value : the transposed HermitianMatrix

1.9.5 SquareMatrix

Usage :

SquareMatrix(dim)
SquareMatrix(dim, values)

Arguments :

dim : an integer, the dimension of the SquareMatrix (square matrix with *dim* rows and *dim* colons)
values : a NumericalScalarCollection of dimension dim^2

Value : SquareMatrix

while using the first parameters set, the SquareMatrix is filled with 0.

while using the second parameters set, the SquareMatrix contains values of the NumericalScalarCollection. SquareMatrix is filled by rows.

Some methods :

computeDeterminant

Usage : *computeDeterminant()*

Arguments : none

Value : a real value giving the determinant of the SquareMatrix

computeEigenValues

Usage : *computeEigenValues()*

Arguments : none

Value : a NumericalPoint giving the eigen values of the SquareMatrix

getDimension

Usage : *getDimension()*

Arguments : none

Value : an integer, the dimension of the SquareMatrix (it returns *dim*)

solveLinearSystem

Usage : *solveLinearSystem(y)*

Arguments : *y* : a NumericalPoint of dimension n_r (the number of row of the SquareMatrix)

Value : NumericalPoint, this NumericalPoint, *x*, is such that
 $SquareMatrix * x = y$

transpose

Usage : *transpose()*

Arguments : none

Value : the transposed SquareMatrix

1.9.6 Tensor

Usage :

Tensor(n_r, n_c, n_s)

Matrix($n_r, n_c, n_s, values$)

Arguments :

n_r : an integer, the number of rows of the Tensor

n_c : an integer, the number of columns of the Tensor

n_s : an integer, the number of sheets of the Tensor

values : NumericalScalarCollection with $n_r \times n_c \times n_s$ elements

Value : Tensor

while using the first parameters set, the matrix is filled with 0.

while using the second parameters set, the Matrix contains values of the NumericalScalarCollection. The tensor is filled by row.

Some methods :

[, ,]

Usage : *Tensor*[i, j, k]

Arguments :

i : an integer, constraint : $0 \leq i \leq n_r - 1$

j : an integer, constraint : $0 \leq j \leq n_c - 1$

k : an integer, constraint : $0 \leq k \leq n_s - 1$

Value : a real value, the (i, j, k) element of the Tensor

getNbColumns

Usage : *getNbColumns*()

Arguments : none

Value : an integer : the number of column n_c

getNbRows

Usage : *getNbRows*()

Arguments : none

Value : an integer : the number of row n_r

getNbSheets

Usage : *getNbSheets*()

Arguments : none

Value : an integer : the number of sheet n_s

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the Tensor

setName

Usage : *setName(name)*

Arguments : name : a string

Value : the Tensor is named *name*

1.9.7 TriangularComplexMatrix

Usage :

TriangularComplexMatrix(dim)
TriangularComplexMatrix(dim, isLower)

Arguments :

dim : an integer, the dimension of the TriangularComplexMatrix (square matrix with *dim* rows and *dim* colons)

isLower : a boolean flag, tells if the triangular matrix is lower (*True*) or upper (*False*). Notice that the a missing flag corresponds to *True*

Value : TriangularComplexMatrix

while using the first parameters set, the TriangularComplexMatrix is filled with (0,0). Matrix is lower triangular

while using the second parameters set, the TriangularComplexMatrix is filled with (0,0) and we fix if the matrix is lower or upper triangular.

It is not possible to fill the matrix from a collection of complex values (to be done later)

Some methods :

conjugate

Usage : *conjugate()*

Arguments : none

Value : the conjugated triangular matrix.

conjugateTranspose

Usage : *conjugateTranspose()*

Arguments : none

Value : the conjugated and transposed TriangularComplexMatrix. Notice that the result is an upper triangular matrix if the initial matrix is lower triangular.

getDimension

Usage : *getDimension()*

Arguments : none

Value : an integer, the dimension of the TriangularComplexMatrix (it returns *dim*)

isTriangularLower

Usage : *isTriangularLower()*

Arguments : none

Value : a boolean, tells if the matrix is lower or upper triangular.

transpose

Usage : *transpose()*

Arguments : none

Value : the transposed `TriangularComplexMatrix`. Notice that the transpose of a lower triangular matrix is an upper triangular matrix.

1.10 NumericalPoint

Usage :

NumericalPoint(dim)
NumericalPoint(dim, value)
NumericalPoint(list)
NumericalPoint(tuple)
NumericalPoint(array)

Arguments :

dim : an integer, the dimension of the NumericalPoint
value : a real value, the value of each component of the NumericalPoint
list : a *list* in the environment python
tuple : a *tuple* in the environment python
array: a *Numpy array* with dimension (ndim) 1

Value :

in the first usage, a NumericalPoint of dimension *dim*, which each component is equal to 0.0
 in the second usage, a NumericalPoint of dimension *dim*, which each component is equal to *value*
 in the third usage, a NumericalPoint which components are given by the *list* python. For example, *list = [1.1, 2.2, 3.3, 4.4]* and the created NumericalPoint is $(1.1, 2.2, 3.3, 4.4)^t$
 in the fourth usage, a NumericalPoint which components are given by the *tuple* python. For example, $(1.1, 2.2, 3.3, 4.4)$ and the created NumericalPoint is $(1.1, 2.2, 3.3, 4.4)^t$

Some methods :

[]

Usage : *NumericalPoint[i]*

Arguments :

i : an integer, constraint : $0 \leq i \leq dim - 1$

Value : a real value, the value of the (*i* + 1)-th element of the NumericalPoint

getDimension

Usage : *getDimension()*

Arguments : none

Value : an integer, the value of the dimension of the NumericalPoint (it returns dim)

norm

Usage : *norm()*

Arguments : none

Value : a real value, the euclidian norm of the NumericalPoint

normalize

Usage : *normalize()*

Arguments : none

Value : a NumericalPoint which is the unitary vector colinear to the initial one and pointing in the same direction. In other words, if \underline{x} is the initial vector, it returns $\frac{\underline{x}}{\|\underline{x}\|}$.

norm2

Usage : *norm2()*

Arguments : none

Value : a real value, the square of the euclidian norm of the NumericalPoint

normalize2

Usage : *normalize2()*

Arguments : none

Value : a NumericalPoint which is the vector \underline{y} defined as follows : if \underline{x} is the initial vector, then

$$\underline{y} = \left(\frac{x_i^2}{\|\underline{x}\|^2} \right)_i.$$

str

Usage : *str()*

Arguments : none

Value : a string describing the NumericalPoint

dot

Usage : *dot(x, y)*

Arguments : x, y : NumericalPoint

Value : a real value, the dot product (also known as the scalar product) of x and y

getName

Usage : *getName()*

Arguments : none

Value : a string giving the name of the NumericalPoint

setName

Usage : *setName(name)*

Arguments : $name$: a string

Value : no value, it gives a name for the considered NumericalPoint

*: right multiplication by a scalar. The NumericalPoint n can be multiplied by a scalar s using $n * s$.

1.11 NumericalPointCollection

Usage : *NumericalPointCollection(dim)*

Arguments : *dim* : an integer, the number of elements of the NumericalPointCollection

Value : a NumericalPointCollection, filled by default with 0.0

Some methods :

[]

Usage : *NumericalPointCollection[i]*

Arguments : *i* : an integer, constraint : $0 \leq i \leq dim - 1$

Value : a NumericalPoint, the $(i + 1)$ -th element of the NumericalPointCollection

add

Usage : *add(numericalPoint2)*

Arguments : *numericalPoint2* : a NumericalPoint

Value : The NumericalPointCollection of size $dim + 1$. The $dim + 1$ element of this object is then equal to *numericalPoint2*

getSize

Usage : *getSize()*

Arguments : none

Value : an integer : the size of the NumericalPointCollection

1.12 NumericalScalarCollection

Usage : *NumericalScalarCollection(dim)*

Arguments : *dim* : an integer, the number of elements of the NumericalScalarCollection

Value : a NumericalScalarCollection, filled by default with 0

Some methods :

[]

Usage : *NumericalScalarCollection[i]*

Arguments :

i : an integer, constraint : $0 \leq i \leq dim - 1$

Value : a real value, the $(i + 1)$ -th element of the NumericalScalarCollection

add

Usage : *add(val)*

Arguments : *val* : a real value

Value : The NumericalScalarCollection of size $dim + 1$. The $dim + 1$ element of this object is then equal to *val*

getSize

Usage : *getSize()*

Arguments : none

Value : an integer : the size of the NumericalScalarCollection

1.13 UserDefinedPair

Usage : *UserDefinedPair*(x,p)

Arguments :

x : a NumericalPoint,

p : a real value, constraint $0 \leq p \leq 1$ (the probability associated to the point x)

Value : a UserDefinedPair

Some methods :

getX

Usage : *getX*()

Arguments : no argument

Value : a NumericalPoint, the point of the UserDefinedPair

getP

Usage : *getP*()

Arguments : no argument

Value : a NumericalScalar, the scalar of the UserDefinedPair

Each get method is associated to a set method.

1.14 UserDefinedPairCollection

Usage : *UserDefinedPairCollection(dim)*

Arguments : *dim* : an integer, the number of elements of the UserDefinedPairCollection

Value : an UserDefinedPairCollection, to be filled after

Some methods :

[]

Usage : *UserDefinedPairCollection[i]*

Arguments : *i* : an integer, the *i*th element of UserDefinedPairCollection

Value : a UserDefinedPair, the (*i* + 1)-th element of UserDefinedPairCollection

add

Usage : *add(UseDefP)*

Arguments : *UseDefP* : an UserDefinedPair

Value : a UserDefinedPairCollection of size *dim* + 1 with instance of the (*dim* + 1) element of the UserDefinedPairCollection

getSize

Usage : *getSize()*

Arguments : no argument

Value : an integer, the size of UserDefinedPairCollection (returns *dim*)

str

Usage : *str()*

Arguments : no argument

Value : a string with elements of UserDefinedPairCollection

2 Probabilistic modeling

In this section, we describe all the objects necessary to model a random vector.

2.1 Distribution

Usage : *Distribution(dist)*

Arguments :

dist : a DistributionImplementation which is particular distribution.

Value : a Distribution

Some methods :

computeProbability

Usage :

computeProbability(interval)

Arguments :

interval : an Interval

Value :

it gives the evaluation (a scalar) of the probability for the given distribution to take values within the given interval *interval*

computeCDF

Usage :

computeCDF(scalar)

computeCDF(scalar, flag)

computeCDF(vector)

computeCDF(vector, flag)

computeCDF(sample)

computeCDF(sample, flag)

Arguments :

scalar : a NumericalScalar

vector : a NumericalPoint

sample : a NumericalSample

flag : a Bool

Value :

if flag is false (default value), the method computes the CDF, else it computes the complementary CDF, i.e. $\text{computeCDF}(x, \text{False}) = 1 - \text{computeCDF}(x)$.

using the first usage, it gives the evaluation (a scalar) of the CDF (Cumulative Distribution Function) of a distribution of dimension 1 at the given scalar value *scalar*

using the second usage, it gives the evaluation (a scalar) of the CDF (Cumulative Distribution Function) of a distribution of arbitrary dimension at the given point *vector*

using the third usage, it gives the evaluation (a NumericalSample) of the CDF (Cumulative Distribution Function) of a distribution of arbitrary dimension over the given sample *sample*

*computeCDFGradient***Usage :** *computeCDFGradient(vector)***Arguments :** *vector* : a NumericalPoint**Value :** a NumericalPoint object, the gradient of the distribution CDF, with respect to the parameters of the distribution, evaluated at point *vector**computeCharacteristicFunction***Usage :** *computeCharacteristicFunction(vector)***Arguments :** *vector* : a NumericalPoint**Value :** a NumericalComplex object, the value of the characteristic function at point *vector*. Open TURNS proposes an implementation of all its univariate distributions, continuous or discrete ones. But only some of the them have the implementation of a specific algorithm that evaluates the characteristic function : it is the case of all the discrete distributions and most of (but not all) the continuous ones. In that case, the evaluation is performant. For the remaining distributions, the generic implementation might be time consuming for high arguments.*computeDDF***Usage :***computeDDF(vector)**computeDDF(sample)***Arguments :***vector* : a NumericalPoint*sample* : a NumericalSample**Value :**while using the first usage, a NumericalPoint value, the gradient of the PDF (Probability Distribution Function) of the considered distribution at *vector* (DDF = Derivative Density Function)while using the second usage, a NumericalSample, the gradient of the PDF (Probability Distribution Function) of the considered distribution at *vector* (DDF = Derivative Density Function)*computeGeneratingFunction***Usage :***computeGeneratingFunction(value)**computeGeneratingFunction(value, logScale)***Arguments :***value* : a NumericalComplex, a numerical complex value within which module is < 1 *logScale* : a Boolean which indicates whether the generating function is computed on a logarithmic scale. By default, *logScale* = *False*.**Value :** a numerical complex value, the value of the generating function at *value**computePDF***Usage :**

computePDF(value)
computePDF(vector)
computePDF(sample)

Arguments :

vector : a NumericalPoint
sample : a NumericalSample

Value :

while using the first usage, a NumericalScalar, the PDF (Cumulative Distribution Function) of dimension 1 value of the considered distribution at *value*
 while using the second usage, a NumericalPoint, the PDF (Cumulative Distribution Function) value of the considered distribution at the vector *vector*
 while using the third usage, a NumericalSample, the PDF (Cumulative Distribution Function) values of the considered distribution at *sample*

computePDFGradient

Usage : *computePDFGradient(vector)*

Arguments : *vector* : a NumericalPoint

Value : a NumericalPoint object, the gradient of the distribution PDF, with respect to the parameters of the distribution, evaluated at point *vector*

computeQuantile

Usage :

computeQuantile(p)
computeQuantile(p, flag)

Arguments :

p : a real scalar $0 \leq p \leq 1$
flag : a Bool

Value : a NumericalPoint, the value of the p -quantile if *flag* = False, the value of the $(1 - p)$ -quantile if *flag* = True. If the distribution is of dimension $n > 1$, the p -quantile is the hyper surface in \mathbb{R}^n defined by $\{\underline{x} \in \mathbb{R}^n, F(x_1, \dots, x_n) = p\}$ where F is the CDF. Open TURNS makes the choice to return one particular point among these points : (x_1^p, \dots, x_n^p) such that $\forall i, F_i(x_i^p) = \tau$ where F_i is the marginal of component X_i and $F(x_1, \dots, x_n) = C(\tau, \dots, \tau)$ where C is the distribution copula. Thus, Open TURNS resolves the equation $C(\tau, \dots, \tau) = p$ then computes $F_i^{-1}(\tau) = x_i^p$.

drawCDF

Usage :

drawCDF()
drawCDF(min, max)
drawCDF(min, max, pointNumber)
drawCDF(vectMin, vectMax)
drawCDF(vectMin, vectMax, vectPointNumber)

Arguments :

min and *max* : real values with $min < max$, the range for the CDF curve of a distribution of dimension 1

pointNumber : an integer, the number of points to draw the CDF iso-curves of a distribution of dimension 1

vectMin and *vectMax* : two NumericalPoint of dimension 2, respectively the left-bottom and ritgh-up corners of the square for the CDF iso-curves of a distribution of dimension 2

vectPointNumber : a NumericalPoint of dimension 2, the the number of points to draw the iso-curves of a distribution of dimension 2 on each direction

Value : a Graph, containing the elements of the curve or iso-curves of the CDF, depending on the dimension of the distribution (1 or 2)

drawPDF

Usage :

drawPDF()

drawPDF(min, max)

drawPDF(min, max, pointNumber)

drawPDF(vectMin, vectMax)

drawPDF(vectMin, vectMax, vectPointNumber)

Arguments :

min and *max* : real values with $min < max$, the range for the PDF curve of a distribution of dimension 1

pointNumber : an integer, the number of points to draw the PDF iso-curves of a distribution of dimension 1

vectMin and *vectMax* : two NumericalPoint of dimension 2, respectively the left-bottom and ritgh-up corners of the square for the PDF iso-curves of a distribution of dimension 2

vectPointNumber : a NumericalPoint of dimension 2, the number of points to draw the iso-curves of a distribution of dimension 2 on each direction

Value : a Graph, containing the elements of the curve or iso-curves of the PDF, depending on the dimension of the distribution (1 or 2)

drawMarginal1DCDF

Usage :

drawMarginal1DCDF(i, min, max, pointNumber)

Arguments :

i : an integer, the marginal we want to draw (Care : numerotation begins at 0)

min and *max* : real values with $min < max$, the range for the CDF curve of a distribution of dimension >1

pointNumber : an integer, the number of points to draw the CDF iso-curves of a distribution of dimension >1

Value : a Graph, containing the elements of the curve of the CDF of the marginal *i* of the distribution of dimension >1

drawMarginal1DPDF

Usage :

drawMarginal1DPDF(i, min, max, pointNumber)

Arguments :

i : an integer, the marginal we want to draw (Care : numerotation begins at 0)

min and *max* : real values with $min < max$, the range for the PDF curve of a distribution of dimension >1

pointNumber : an integer, the number of points to draw the PDF iso-curves of a distribution of dimension >1

Value : a Graph, containing the elements of the curve of the PDF of the marginal *i* of the distribution of dimension >1

drawMarginal2DCDF

Usage :

drawMarginal2DCDF(i, j, vectMin, vectMax, vectPointNumber)

Arguments :

i and *j* : two integer, the marginal we want to draw (Care : numerotation begins at 0)

vectMin and *vectMax* : two NumericalPoint of dimension $n>2$, respectively the left-bottom and ritgh-up corners of the square for the PDF iso-curves of a distribution of dimension *n*

vectPointNumber : a NumericalPoint of dimension $n>2$, the number of points to draw the iso-curves of a distribution of dimension *n* on each direction

Value : a Graph, containing the elements of the iso-curve of the CDF of the marginals (i,j) of distribution of dimension $n>2$

drawMarginal2DPDF

Usage :

drawMarginal2DPDF(i, j, vectMin, vectMax, vectPointNumber)

Arguments :

i and *j* : two integer, the marginal we want to draw (Care : numerotation begins at 0)

vectMin and *vectMax* : two NumericalPoint of dimension $n>2$, respectively the left-bottom and ritgh-up corners of the square for the PDF iso-curves of a distribution of dimension *n*

vectPointNumber : a NumericalPoint of dimension $n>2$, the number of points to draw the iso-curves of a distribution of dimension *n* on each direction

Value : a Graph, containing the elements of the iso-curve of the PDF of the marginals (i,j) of distribution of dimension $n>2$

getCopula

Usage : *getCopula()*

Arguments : no argument

Value : a Copula, the copula of the considered distribution. If the distribution is of type Composed-Distribution, the copula is the one specified at the creation of the ComposedDistribution. If the distribution is not that sort (for example, a KernelMixture, a Mixture, a RandomMixture), the copula is computed from the Sklar theorem.

*getCovariance***Usage :** *getCovariance()***Arguments :** no argument**Value :** a CovarianceMatrix of the considered distribution (if the distribution is unidimensional, it is the variance)*getMarginal***Usage :***getMarginal(i)**getMarginal(indices)***Arguments :***i* : an integer (*i* is less or equal to the dimension of the considered distribution), with $0 \leq i$ *indices* : an Indices, which regroup all the indices considered**Value :** a Distribution, the distribution of an extracted vector of the initial distribution*getKurtosis***Usage :** *getKurtosis()***Arguments :** no argument**Value :** a NumericalPoint, the value the kurtosis of each 1D marginal of the distribution*getMean***Usage :** *getMean()***Arguments :** no argument**Value :** a NumericalPoint, the value of the considered distribution mean*getNumericalSample***Usage :** *getNumericalSample(n)***Arguments :** *n* : integer, the size of the sample**Value :** a NumericalSample representing *n* realizations of the random variable with the considered distribution*getParametersCollection***Usage :** *getParametersCollection()***Arguments :** one**Value :** a NumericalPointWithDescriptionCollection, the list of the parameters of the distribution*getRealization***Usage :** *getRealization()***Arguments :** no argument**Value :** a NumericalPoint, one realization of random variable with the considered distribution*getRoughness*

Usage : *getRoughness()*

Arguments : no argument

Value : a NumericalScalar, the value $roughness(\underline{X}) = \|p\|_{\mathcal{L}^2} = \sqrt{\int_{\underline{x}} p^2(\underline{x})d\underline{x}}$

getSupport

Usage :

getSupport(interval)

getSupport()

Arguments : *interval* : a Interval

Value :

in the first usage, a NumericalSample which gathers the different points of the discrete range.

Care : this service is implemented only for discrete 1D distribution.

in the second usage, a NumericalSample which gathers the different points of the discrete range which are inside *interval*.

getSkewness

Usage : *getSkewness()*

Arguments : no argument

Value : a NumericalPoint, the value the standard deviation of each 1D marginal of the distribution

getStandardDeviation

Usage : *getStandardDeviation()*

Arguments : no argument

Value : a NumericalPoint, the value the standard deviation of each 1D marginal of the distribution

getWeight

Usage : *getWeight()*

Arguments : no argument

Value : a NumericalScalar between 0 and 1, the weight of the considered distribution if used in a Mixture

hasEllipticalCopula

Usage : *hasEllipticalCopula()*

Arguments : no argument

Value : a boolean, it says if the considered distribution is elliptical

hasIndependentCopula

Usage : *hasIndependentCopula()*

Arguments : no argument

Value : a boolean which indicates wether the considered distribution is independent

isElliptical

Usage : *isElliptical()*

Arguments : no argument

Value : a boolean which indicates wether the considered distribution has an elliptical distribution

isIntegral

Usage : *isIntegral()*

Arguments : no argument

Value : a boolean which indicates wether the considered distribution has integer values.

str

Usage : *str()*

Arguments : no argument

Value : a string describing the object

2.2 Usual Distributions

2.2.1 Arcsine

This class inherits from the Distribution class.

Usage :

Main parameters set : $\text{Arcsine}(a, b)$

Second parameters set : $\text{Arcsine}(\mu, \sigma, 1)$

Default construction : $\text{Arcsine}()$

Arguments :

a : a real value, the lower bound

b : a real value, the upper bound, constraint: $a < b$

Value : Arcsine. In the default construction, we use the $\text{Arcsine}(a, b) = \text{Arcsine}(-1.0, 1.0)$ definition.

Some methods :

getA

Usage : *getA()*

Arguments : none

Value : a real value, the lower bound

getB

Usage : *getB()*

Arguments : none

Value : a real value, the upper bound

getMu

Usage : *getMu()*

Arguments : none

Value : a real value, the mean

getSigma

Usage : *getSigma()*

Arguments : none

Value : a real value, the standard deviation

Details :

density function :

$$\frac{1}{\pi \frac{b-a}{2} \sqrt{1 - \left(\frac{x - \frac{a+b}{2}}{\frac{b-a}{2}} \right)^2}}$$

relation between parameter sets :

$$\begin{aligned}\mu &= \frac{a+b}{2} \\ \sigma &= \frac{b-a}{2\sqrt{2}}\end{aligned}$$

where

$$\mu = \mathbb{E}[X]$$

$$\sigma = \sqrt{\text{Var}[X]}$$

Links : [see docref_B121_DistributionSelection](#)

Each *getMethod* is associated to a *setMethod*.

2.2.2 Bernoulli

This class inherits from the Distribution class.

Usage :

Main parameters set : $Bernoulli(p)$

Default construction : $Bernoulli()$

Arguments : p : a real value, constraint : $0 \leq p \leq 1$

Value : a Bernoulli. In the default construction, we use the $Bernopulli() = Bernoulli(0.5)$ definition.

Some methods :

getP

Usage : *getP()*

Arguments : none

Value : a real positive value ≤ 1 , the p parameter of the distribution.

getSupport

Usage : *getSupport(interval)*

Arguments : *interval* : a *Interval*, an interval in \mathbb{R}

Value : a *NumericalSample*, all the points (here of dimension 1) of the distribution range which are included in the interval *interval*.

Details :

probability distribution:

$$\mathbb{P}(X = 1) = p, \mathbb{P}(X = 0) = 1 - p$$

relation between parameters set :

$$\begin{aligned} \mu &= p & \text{where } \mu &= \mathbb{E}[X] \\ \sigma &= \sqrt{p(1-p)} & \text{where } \sigma &= \sqrt{\text{Var}[X]} \end{aligned}$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.3 Beta

This class inherits from the Distribution class.

Usage :

Main parameters set : $Beta(r, t, a, b)$

Second parameters set : $Beta(\mu, \sigma, a, b, Beta.MUSIGMA)$

Default construction : $Beta()$

Arguments :

r : real value, first shape parameter, constraint : $r > 0$

t : real value, second shape parameter, constraint : $t > r$

a : real value, lower bound

b : real value, upper bound, constraint : $b > a$

μ : real value, mean value

σ : real value, standard deviation, constraint : $\sigma > 0$

Value : a Beta. In the default construction, we use the $Beta(r, t, a, b) = Beta(2, 4, -1, 1)$ definition.

Some methods :

getA

Usage : *getA()*

Arguments : none

Value : a real value, the a parameter of the Beta distribution

getB

Usage : *getB()*

Arguments : none

Value : a real value, the b parameter of the Beta distribution

getMu

Usage : *getMu()*

Arguments : none

Value : a real value, the μ parameter of the distribution

getSigma

Usage : *getSigma()*

Arguments : none

Value : a real value, the σ parameter of the distribution

getR

Usage : *getR()*

Arguments : none

Value : a real value, the r parameter of the distribution

getT

Usage : *getT()*

Arguments : none

Value : a real value, the t parameter of the distribution

Details :

density probability function :

$$f(x) = \frac{(x-a)^{(r-1)}(b-x)^{(t-r-1)}}{(b-a)^{(t-1)}B(r, t-r)} \mathbf{1}_{[a,b]}(x)$$

relation between parameters sets :

$$\begin{aligned}\mu &= a + \frac{(b-a)r}{t} \\ \sigma &= \frac{(b-a)}{t} \sqrt{\frac{r(t-r)}{(t+1)}}\end{aligned}$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.4 Binomial

This class inherits from the Distribution class.

Usage :

Main parameters set : $Binomial(n, p)$

Default construction : $Binomial()$

Arguments :

n : an integer > 0 ,

p : a real value such as $0 \leq p \leq 1$.

Value : a Binomial. In the default construction, we use the $Binomial() = Binomial(1, 0.5)$ definition.

Some methods :

getP

Usage : *getP()*

Arguments : none

Value : a real positive value ≤ 1 , the p parameter of the distribution.

getN

Usage : *getN()*

Arguments : none

Value : an integer, the n parameter of the distribution.

getSupport

Usage : *getSupport(interval)*

Arguments : *interval* : a *Interval*, an interval in \mathbb{R}

Value : a *NumericalSample*, all the points (here of dimension 1) of the distribution range which are included in the interval *interval*.

Details :

probability distribution:

$$\mathbb{P}(X = k) = C_n^k p^k (1 - p)^{n-k}, \forall k \in \{0, \dots, n\}$$

relation between parameters set :

$$\begin{aligned} \mu &= p & \text{where } \mu &= \mathbb{E}[X] \\ \sigma &= \sqrt{p(1-p)} & \text{where } \sigma &= \sqrt{\text{Var}[X]} \end{aligned}$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.5 Burr

This class inherits from the Distribution class.

Usage :

Main parameters set : $Burr(c, k)$

Default construction : $Burr()$

Arguments :

c : an real > 0 ,

k : a real value > 0 .

Value : a Burr. In the default construction, we use the $Burr() = Burr(1, 1.0)$ definition.

Some methods :

getC

Usage : *getC()*

Arguments : none

Value : a real positive value, the c parameter of the distribution.

getK

Usage : *getK()*

Arguments : none

Value : an real positive value, the k parameter of the distribution.

getSupport

Usage : *getSupport(interval)*

Arguments : *interval* : a *Interval*, an interval in \mathbb{R}

Value : a *NumericalSample*, all the points (here of dimension 1) of the distribution range which are included in the interval *interval*.

Details :

probability distribution:

$$p(x; \underline{\theta}) = ck \frac{x^{(c-1)}}{(1+x^c)^{(k+1)}} \mathbf{1}_{x>0}$$

relation between parameters set :

$$\mu = kBeta(k - 1/c, 1 + 1/c) \quad \text{where} \quad \mu = \mathbb{E}[X]$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.6 Chi

This class inherits from the Distribution class.

Usage :

Main parameters set : $Chi(\nu)$

Default construction : $Chi()$

Arguments :

ν : real value, degrees of freedom, constraint : $\nu > 0$

Value : a Chi. In the default construction, we use the $Chi(\nu) = Chi(1)$ definition.

Some methods :

getNu

Usage : *getNu()*

Arguments : none

Value : a real value, the ν parameter of the Chi distribution

Details :

density probability function :

$$f(x) = x^{\nu-1} e^{-x^2/2} \frac{2^{1-\nu/2}}{\Gamma(\nu/2)} \mathbf{1}_{[0,+\infty[}(x)$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.7 ChiSquare

This class inherits from the Distribution class.

Usage :

Main parameters set : *ChiSquare*(ν)

Default construction : *ChiSquare*()

Arguments :

ν : real value, degrees of freedom, constraint : $\nu > 0$

Value : a ChiSquare. In the default construction, we use the $ChiSquare(\nu) = ChiSquare(1)$ definition.

Some methods :

getNu

Usage : *getNu*()

Arguments : none

Value : a real value, the ν parameter of the ChiSquare distribution

Details :

density probability function :

$$f(x) = \frac{2^{-\nu/2}}{\Gamma(\nu/2)} x^{(\nu/2-1)} e^{-x/2} \mathbf{1}_{[0,+\infty[}(x)$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.8 Dirichlet

This class inherits from the Distribution class.

Usage : Main parameters set : *Dirichlet(vectTheta)*

Default construction : *Dirichlet()*

Arguments :

vectTheta : a NumericalPoint : $(\theta_1, \dots, \theta_{d+1})$ for a multivariate d -dimensional distribution.

Value : an Dirichlet. In the default construction, a univariate distribution is created with $(\theta_1, \theta - 2) = (1, 1)$.

Some methods :

getTheta

Usage : *getTheta()*

Arguments : none

Value : a NumericalPoint : the $(\theta_1, \dots, \theta_{d+1})$ parameter of the distribution

Details :

density probability function :

$$f_X(\underline{x}; \underline{\theta}) = \frac{\Gamma(\sum_{j=1}^{d+1} \theta_j)}{\prod_{j=1}^{d+1} \Gamma(\theta_j)} \left[1 - \sum_{j=1}^d x_j \right]^{\theta_{d+1}-1} \prod_{j=1}^d x_j^{\theta_j-1} \mathbf{1}_{\Delta}(\underline{x})$$

with $\Delta = \{\underline{x} \in \mathbb{R}^d / \forall i, x_i \geq 0, \sum_{i=1}^d x_i \leq 1\}$.

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.9 Epanechnikov

This class inherits from the Distribution class.

Usage : Default construction : *Epanechnikov*()

Arguments :

Value : an Epanechnikov, which is a *Beta*($a = -1, b = 1, r = 2, t = 4$) distribution.

Some methods :

Details :

density probability function :

$$f(x) = \frac{3}{4}(1 - x^2)\mathbf{1}_{[-1,1]}(x)$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.10 Exponential

This class inherits from the Distribution class.

Usage : Main parameters set : $Exponential(\lambda, \gamma)$

Default construction : $Exponential()$

Arguments :

λ : real value, scale parameter, constraint : $\lambda > 0$

γ : real value.

Value : an Exponential. In the default construction, we use the $Exponential(\lambda, \gamma) = Exponential(1.0, 0)$ definition.

Some methods :

getGamma

Usage : *getGamma()*

Arguments : none

Value : a real value, the γ parameter of the distribution

getLambda

Usage : *getLambda()*

Arguments : none

Value : a real value, the λ parameter of the distribution

Details :

density probability function :

$$f(x) = \lambda e^{-\lambda(x-\gamma)} \mathbf{1}_{[\gamma, +\infty[}(x)$$

relation between parameters sets :

$$\begin{aligned} \mu &= \gamma + \frac{1}{\lambda} & \text{where } \mu &= \mathbb{E}[X] \\ \sigma &= \frac{1}{\lambda} & \text{where } \sigma &= \sqrt{\text{Var}[X]} \end{aligned}$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.11 FisherSnedecor

This class inherits from the Distribution class.

Usage : Main parameters set : $FisherSnedecor(d1, d2)$

Default construction : $FisherSnedecor()$

Arguments :

$(d1, d2)$: two real value, scale parameter, constraint : $d_i > 0$

Value : an FisherSnedecor. In the default construction, we use the $FisherSnedecor(d1, d2) = FisherSnedecor(1.0, 1)$ definition.

Some methods :

getD1

Usage : *getD1()*

Arguments : none

Value : a real value, the d_1 parameter of the distribution

getD2

Usage : *getD2()*

Arguments : none

Value : a real value, the d_2 parameter of the distribution

Details :

density probability function :

$$f(x) = \frac{1}{xB(d_1/2, d_2/2)} \left[\left(\frac{d_1 x}{d_1 x + d_2} \right)^{d_1/2} \left(1 - \frac{d_1 x}{d_1 x + d_2} \right)^{d_2/2} \right] \mathbf{1}_{x \geq 0}$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.12 Gamma

This class inherits from the Distribution class.

Usage :

Main parameters set : $Gamma(k, \lambda, \gamma)$

Second parameters set : $Gamma(\mu, \sigma, \gamma, Gamma.MUSIGMA)$

Default construction : $Gamma()$

Arguments :

k : integer value constraint : $k > 0$

λ : real value, constraint : $\lambda > 0$

γ : real value,

μ : real value, mean value

σ : real value, standard deviation, constraint : $\sigma > 0$

Value : a Gamma. In the default construction, we use the $Gamma(k, lambda, gamma) = Gamma(1.0, 1.0, 0.0)$ definition.

Some methods :

getGamma

Usage : *getGamma()*

Arguments : none

Value : a real value, the γ parameter of the distribution

getK

Usage : *getK()*

Arguments : none

Value : a real value, the k parameter of the distribution

getLambda

Usage : *getLambda()*

Arguments : none

Value : a real value, the λ parameter of the distribution

setKLambda

Usage : *setKLambda(k, lambda)*

Arguments : k : a NumericalScalar, the k parameter of the distribution

$lambda$: a NumericalScalar, the λ parameter of the distribution

Details :

density probability function :

$$f(x) = \frac{\lambda}{\Gamma(k)} (\lambda(x - \gamma))^{(k-1)} e^{-\lambda(x-\gamma)} \mathbf{1}_{[\gamma, +\infty[}(x)$$

relation between parameters sets :

$$\begin{aligned} \mu &= \frac{k}{\lambda} + \gamma \quad \text{where } \mu = \mathbb{E}[X] \\ \sigma &= \frac{\sqrt{k}}{\lambda} \quad \text{where } \sigma = \sqrt{\text{Var}[X]} \end{aligned}$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.13 Geometric

This class inherits from the Distribution class.

Usage : Main parameters set : *Geometric*(*p*)

Arguments : *p* : a real value, constraint : $0 < p < 1$

Value : Geometric

Some methods :

getP

Usage : *getP*()

Arguments : none

Value : a real positive value < 1 , the *p* parameter of the distribution

getSupport

Usage : *getSupport*(*interval*)

Arguments : *interval* : a *Interval*, an interval in \mathbb{R}

Value : a *NumericalSample*, all the points (here of dimension 1) of the distribution range which are included in the interval *interval*.

Details :

probability distribution:

$$\mathbb{P}(X = k) = (1 - p)^{k-1}p, k \in \mathbb{N}^*$$

relation between parameters set :

$$\begin{aligned} \mu &= \frac{1}{p} \quad \text{where } \mu = \mathbb{E}[X] \\ \sigma &= \sqrt{\frac{1-p}{p^2}} \quad \text{where } \sigma = \sqrt{\text{Var}[X]} \end{aligned}$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.14 Gumbel

This class inherits from the Distribution class.

Usage :

Main parameters set : $Gumbel(\alpha, \beta)$

Second parameters set : $Gumbel(\mu, \sigma, 1)$

Default construction : $Gumbel()$

Arguments :

α : a real value, the scale parameter (the inverse), constraint : $\alpha > 0$

β : a real value, location parameter

μ : a real value, the mean value

σ : a real value, standard deviation, constraint : $\sigma > 0$

Value : a Gumbel. In the default construction, we use the $Gumbel(alpha, beta) = Gumbel(1.0, 1.0)$ definition.

Some methods :

getAlpha

Usage : *getAlpha()*

Arguments : none

Value : a real value, the α of the considered distribution

getBeta

Usage : *getBeta()*

Arguments : none

Value : a real value, the β of the considered distribution

getMu

Usage : *getMu()*

Arguments : none

Value : a real value, the μ parameter of the distribution

getSigma

Usage : *getSigma()*

Arguments : none

Value : a real value, the σ parameter of the distribution

Details :

density probability function :

$$f(x) = \alpha e^{-\alpha(x-\beta)} - e^{-\alpha(x-\beta)}$$

relation between parameters set :

$$\mu = \beta + \frac{c}{\alpha} \quad \text{where } c \text{ is the Euler-Mascheroni constant} \quad (c \approx 0.5772156649)$$
$$\sigma = \frac{1}{\sqrt{6}} \frac{\pi}{\alpha}$$

where $\mu = \mathbb{E}[X]$ $\sigma = \sqrt{\text{Var}[X]}$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.15 Histogram

This class inherits from the Distribution class.

Usage :

Main parameters set : $Histogram(first, Coll)$

Arguments :

$first$: a real value, the lower bound of the distribution

$Coll$: an HistogramPairCollection, the collection of (h_i, l_i) where h_i is the height and l_i the width of each barplot of the Histogram

Value : an Histogram with normalized heights

Some methods :

getFirst

Usage : *getFirst()*

Arguments : none

Value : a real value, the *first* parameter of the considered distribution

getPairCollection

Usage : *getPairCollection()*

Arguments : none

Value : a HistogramPairCollection, the *Coll* parameter of the considered distribution

Details :

density probability function :

$$f(x) = \sum_{i=1}^n H_i \mathbf{1}_{[x_i, x_{i+1}]}(x)$$

where

$H_i = h_i/S$ is the normalized heights, with $S = \sum_{i=1}^n h_i l_i$ being the initial surface of the histogram.

$l_i = x_{i+1} - x_i$, $1 \leq i \leq n$

n is the size of the HistogramPairCollection

Each *getMethod* is associated to a *setMethod*.

2.2.16 InverseNormal

This class inherits from the Distribution class.

Usage : Main parameters set : *InverseNormal(lambda, mu)*

Default construction : *InverseNormal()*

Arguments :

(lambda, mu) : two real value, scale parameter, constraint : $\lambda > 0$ and $\mu > 0$

Value : an InverseNormal. In the default construction, we use the *InverseNormal(lambda, mu) = InverseNormal()* definition.

Some methods :

getLambda

Usage : *getLambda()*

Arguments : none

Value : a real value, the λ parameter of the distribution

getMu

Usage : *getMu()*

Arguments : none

Value : a real value, the μ parameter of the distribution

Details :

density probability function :

$$f(x) = \left(\frac{\lambda}{2\pi x^3} \right)^{1/2} e^{-\lambda(x-\mu)^2/(2\mu^2x)} \mathbf{1}_{x>0}$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.17 Laplace

This class inherits from the Distribution class.

Usage :

Main parameters set : $Laplace(\lambda, \mu)$

Default construction : $Laplace()$

Arguments :

λ : a real value, scale parameter, constraint $\lambda > 0$,

μ : a real value, mean value.

Value : Laplace. In the default construction, we use the $Laplace(\lambda, \mu) = Laplace(1.0, 0.0)$ definition.

Some methods :

getLambda

Usage : *getLambda()*

Arguments : none

Value : a real value, the λ parameter of the considered distribution

getMu

Usage : *getMu()*

Arguments : none

Value : a real value, the μ parameter of the considered distribution

Details :

density function :

$$f(x) = \frac{\lambda}{2} e^{-\lambda|x-\mu|}$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.18 Logistic

This class inherits from the Distribution class.

Usage :

Main parameters set : $Logistic(\alpha, \beta)$

Default construction : $Logistic()$

Arguments :

α : a real value, mean value

β : a real value, scale parameter, constraint : $\beta \geq 0$

Value : Logistic. In the default construction, we use the $Logistic(\alpha, \beta) = Logistic(0.0, 1.0)$ definition.

Some methods :

getAlpha

Usage : *getAlpha()*

Arguments : none

Value : a real value, the α parameter of the considered distribution

getBeta

Usage : *getBeta()*

Arguments : none

Value : a real value, the β parameter of the considered distribution

Details :

density function :

$$f(x) = \frac{e^{\left(\frac{x-\alpha}{\beta}\right)}}{\beta \left(1 + e^{\left(\frac{x-\alpha}{\beta}\right)}\right)^2} \mathbf{1}_{[\alpha, +\infty[}(x)$$

relation between parameters set :

$$\begin{aligned} \mu &= \alpha \\ \sigma &= \sqrt{\frac{1}{3}\pi^2\beta^2} \end{aligned}$$

$$\text{where} \quad \mu = \mathbb{E}[X] \quad \sigma = \sqrt{\text{Var}[X]}$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.19 LogNormal

This class inherits from the Distribution class.

Usage :

Main parameters set : $LogNormal(\mu_\ell, \sigma_\ell, \gamma)$

Second parameters set : $LogNormal(\mu, \sigma, \gamma, LogNormal.MUSIGMA)$

Third parameters set : $LogNormal(\mu, \sigma/\mu, \gamma, LogNormal.MUSIGMAOVERMU)$

Default construction : $LogNormal()$

Arguments :

μ_ℓ : a real value, mean value of $\log(X)$,

σ_ℓ : a real value, standard deviation of $\log(X)$, constraint : $\sigma_\ell > 0$

γ : a real value

μ : a real value, mean value, constraint : $\mu > \gamma$

σ : a real value, standard deviation, constraint : $\sigma > 0$

Value : a LogNormal . In the default construction, we use the $LogNormal(\mu_\ell, \sigma_\ell, \gamma) = LogNormal(0.0, \text{definition})$.

Some methods :

getGamma

Usage : *getGamma()*

Arguments : none

Value : a real value, the γ parameter of the LogNormal distribution

getMu

Usage : *getMu()*

Arguments : none

Value : a real value, the μ parameter of the LogNormal distribution

getMuLog

Usage : *getMuLog()*

Arguments : none

Value : a real value, the μ_ℓ parameter of the LogNormal distribution

getSigma

Usage : *getSigma()*

Arguments : none

Value : a real value, the σ parameter of the LogNormal distribution

getSigmaLog

Usage : *getSigmaLog()*

Arguments : none

Value : a real value, the σ_ℓ parameter of the LogNormal distribution

getSigmaOverMu

Usage : *getSigmaOverMu()*

Arguments : none

Value : a real value, the σ/μ parameter of the considered distribution

Details :

density probability function :

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma_\ell(x-\gamma)} e^{-\frac{1}{2}\left(\frac{\log(x-\gamma)-\mu_\ell}{\sigma_\ell}\right)^2} \mathbf{1}_{[\gamma,+\infty[}(x)$$

relation between parameters set :

$$\begin{aligned} \mu &= e^{\mu_\ell + \sigma_\ell^2/2} + \gamma \\ \sigma &= e^{\mu_\ell + \sigma_\ell^2/2} \sqrt{(e^{\sigma_\ell^2} - 1)} \end{aligned}$$

where

$$\mu = \mathbb{E}[X]$$

$$\sigma = \sqrt{\text{Var}[X]}$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.20 LogUniform

This class inherits from the Distribution class.

Usage :

Main parameters set : $LogUniform(a_\ell, b_\ell)$

Default construction : $LogUniform()$

Arguments :

a_ℓ : a real value, lower bound of $\log(X)$,

b_ℓ : a real value, upper bound of $\log(X)$, constraint : $b_\ell > a_\ell$

Value : a LogUniform : the log of the variate is $Uniform(a_\ell, b_\ell)$ distributed. In the default construction, we use the $LogUniform(a_\ell, b_\ell) = LogUniform(-1.0, 1.0)$ definition.

Some methods :

getALog

Usage : *getALog()*

Arguments : none

Value : a real value, the a_ℓ parameter of the LogUniform distribution

getBLog

Usage : *getBLog()*

Arguments : none

Value : a real value, the b_ℓ parameter of the LogUniform distribution

Details :

density probability function :

$$f(x) = \frac{1}{x(b_\ell - a_\ell)} \mathbf{1}_{[a_\ell, b_\ell]}(\log(x))$$

relation between parameters set :

$$\begin{aligned} \mu &= \frac{e^{b_\ell} - e^{a_\ell}}{b_\ell - a_\ell} \\ \sigma^2 &= \frac{1}{2} \frac{(e^{b_\ell} - e^{a_\ell}) [e^{b_\ell}(b_\ell - a_\ell - 2) + e^{a_\ell}(b_\ell - a_\ell + 2)]}{(b_\ell - a_\ell)^2} \end{aligned}$$

$$\text{where} \quad \mu = \mathbb{E}[X] \quad \sigma = \sqrt{\text{Var}[X]}$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.21 Multinomial

This class inherits from the Distribution class.

Usage : Main parameters set : $Multinomial(p, N)$

Arguments :

p : NumericalPoint of dimension n , constraint : $0 \leq p_i \leq 1$, $q = \sum_{i=1}^n p_i \leq 1$

N : an integer,

Value : a Multinomial

Some methods :

getN

Usage : *getN()*

Arguments : none

Value : a integer, the N parameter of the considered distribution

getP

Usage : *getP()*

Arguments : none

Value : a NumericalPoint, the p parameter of the considered distribution

getSupport

Usage : *getSupport(interval)*

Arguments : *interval* : a *Interval*, an interval in \mathbb{R}

Value : a *NumericalSample*, all the points (here of dimension n) of the distribution range which are included in the interval *interval*.

Details :

probability function :

$$P(\underline{X} = \underline{x}) = \frac{N!}{x_1! \dots x_n! (N-s)!} p_1^{x_1} \dots p_n^{x_n} (1-q)^{N-s}$$

with $0 \leq p_i \leq 1$, $x_i \in \mathbb{N}$, $q = \sum_{i=1}^n p_i \leq 1$, $s = \sum_{i=1}^n x_i \leq N$

relation between parameters set :

$$\begin{aligned} \mu_i &= n p_i \\ \sigma_i &= \sqrt{n p_i (1-p_i)} \\ \sigma_{i,j} &= -n p_i p_j \end{aligned}$$

$$\text{where} \quad \mu_i = \mathbb{E}[X_i] \quad \sigma_i = \sqrt{\text{Var}[X_i]} \quad \sigma_{i,j} = \text{Cov}[(X_i, X_j)]$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.22 NegativeBinomial

This class inherits from the Distribution class.

Usage :

Main parameters set : *NegativeBinomial*(r, p)

Default construction : *NegativeBinomial*()

Arguments :

r : a real value > 0 ,

p : a real value such as $0 < p < 1$.

Value : a NegativeBinomial. In the default construction, we use the *NegativeBinomial*() = *NegativeBinomial*(1,0) definition.

Some methods :

getP

Usage : *getP*()

Arguments : none

Value : a real value in $(0, 1)$, the p parameter of the distribution.

getR

Usage : *getR*()

Arguments : none

Value : a positive real value, the r parameter of the distribution.

getSupport

Usage : *getSupport*(*interval*)

Arguments : *interval* : a *Interval*, an interval in \mathbb{R}

Value : a *NumericalSample*, all the points (here of dimension 1) of the distribution range which are included in the interval *interval*.

Details :

probability distribution:

$$\mathbb{P}(X = k) = \frac{\Gamma(k + r)}{\Gamma(r)\Gamma(k + 1)} p^k (1 - p)^r, \forall k \in \mathbb{N}$$

relation between parameters set :

$$\begin{aligned} \mu &= \frac{rp}{1 - p} \quad \text{where } \mu = \mathbb{E}[X] \\ \sigma &= \frac{\sqrt{rp}}{1 - p} \quad \text{where } \sigma = \sqrt{\text{Var}[X]} \end{aligned}$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.23 NonCentralChiSquare

This class inherits from the Distribution class.

Usage :

Main parameters set : $NonCentralChiSquare(\nu, \lambda)$

Default construction : $NonCentralChiSquare()$

Arguments :

ν : a real positive value, constraint : $\nu > 0$

λ : a real value, constraint : $\lambda \geq 0$

Value : a NonCentralChiSquare distribution. In the default construction, we use the $NonCentralChiSquare(\nu, \lambda) = NonCentralChiSquare(5, 0)$ definition.

Some methods :

getNu

Usage : *getNu()*

Arguments : none

Value : a real value, the μ parameter of the NonCentralChiSquare distribution

getLambda

Usage : *getLambda()*

Arguments : none

Value : a real value, the λ parameter of the NonCentralChiSquare distribution

Details :

density function :

$$f(x) = \sum_{j=0}^{\infty} e^{-\lambda} \frac{\lambda^j}{j!} p_{\chi^2(\nu+2j)}(x)$$

where $p_{\chi^2(q)}$ is the probability density function of a $\chi^2(q)$ random variate.

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.24 NonCentralStudent

This class inherits from the Distribution class.

Usage :

Main parameters set : *NonCentralStudent*(ν, δ, γ)

Main parameters set : *NonCentralStudent*(ν, δ)

Main parameters set : *NonCentralStudent*(ν)

Default construction : *NonCentralStudent*()

Arguments :

ν : a real positive value, generalised number degree of freedom, constraint : $\nu > 0$

δ : a real value, the non-centrality parameter

γ : a real value, the shift from the origin

Value : a NonCentralStudent distribution. In the default construction, we use the *NonCentralStudent*(ν, δ, γ) = *NonCentralStudent*(0.5, 0.0, 0.0) definition, and in the alternative constructions we use *NonCentralStudent*($\nu, \delta, 0.0$) which is the classical non-central Student distribution and *NonCentralStudent*($\nu, 0.0, 0.0$) which is the classical Student distribution.

Some methods :

getNu

Usage : *getNu*()

Arguments : none

Value : a real value, the μ parameter of the NonCentralStudent distribution

getDelta

Usage : *getDelta*()

Arguments : none

Value : a real value, the δ parameter of the NonCentralStudent distribution

getGamma

Usage : *getGamma*()

Arguments : none

Value : a real value, the γ parameter of the NonCentralStudent distribution

Details :

density function :

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

relation between parameters set :

$$\sigma = \sqrt{\frac{\nu}{\nu - 2}}$$

where

$$\mu = \mathbb{E}[X]$$

$$\sigma = \sqrt{\text{Var}[X]}$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.25 Normal

Usage :

Normal(mean, standardDeviation)

Normal(dim)

Normal(μ , σ , R)

Normal(μ , Σ)

Default construction : *Normal()*

Arguments :

mean : a scalar, the mean value of the 1D normal distribution

standardDeviation : a scalar, the standard deviation value of the 1D normal distribution

dim, an integer : the dimension of the Normal distribution

μ : a NumericalPoint, the mean of the Distribution

σ : a NumericalPoint, the standard deviation of each component, constraint : $\sigma[i] > 0, \forall i$

R : a CorrelationMatrix, the linear correlation matrix of the Normal distribution

Σ : a CovarianceMatrix, the covariance matrix of the Normal distribution

value :

while using the first usage, a 1D normal distribution with *mean* as mean value, *standardDeviation* as standard deviation

while using the second usage, a normal distribution of dimension *dim*, with 0 mean value vector, 1-standard deviation vector and identity correlation matrix

while using the third usage, a nD normal distribution with μ as mean vector, σ as standard deviation vector and R as linear correlation matrix

while using the fourth usage, a nD normal distribution with μ as mean vector, Σ as covariance matrix

while using the default usage, a 1D normal distribution with 0 mean and unit variance.

Some methods :

getMu

Usage : *getMu()*

Arguments : none

Value : a NumericalPoint, the μ parameter of the distribution

getSigma

Usage : *getSigma()*

Arguments : none

Value : a NumericalPoint, the σ parameter of the distribution

getCorrelationMatrix

Usage : *getCorrelationMatrix()*

Arguments : none

Value : a CorrelationMatrix, the R parameter of the distribution

Details : probability density function :

$$\frac{1}{(2\pi)^{\frac{n}{2}} (\det \Sigma)^{\frac{1}{2}}} e^{-\frac{1}{2}(x-\mu)^t \Sigma^{-1}(x-\mu)}$$

with $\Sigma = \Lambda(\sigma)R\Lambda(\sigma)$, $\Lambda(\sigma) = \text{diag}(\sigma)$, R symmetric, definite and positive, $\sigma_i > 0$.

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.26 Poisson

This class inherits from the Distribution class.

Usage : Main parameters set : *Poisson*(λ)

Arguments : λ : real value, mean and variance value, constraint : $\lambda > 0$

Value : Poisson

Some methods :

getLambda

Usage : *getLambda*()

Arguments : none

Value : a real positive value, the λ parameter of the considered distribution

Details :

probability function :

$$\mathbb{P}(X = k) = \frac{\lambda^k}{k!} e^{-\lambda}, k \in \mathbb{N}$$

relation between parameters set :

$$\begin{aligned}\mu &= \lambda \\ \sigma &= \sqrt{\lambda}\end{aligned}$$

where

$$\mu = \mathbb{E}[X]$$

$$\sigma = \sqrt{\text{Var}[X]}$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.27 Rayleigh

This class inherits from the Distribution class.

Usage : Main parameters set : $Rayleigh(\sigma, \gamma)$

Arguments :

σ : a real positive value, constraint : $\sigma > 0$

γ : a real value

Value : Rayleigh. In the default construction, we use the $Rayleigh(\sigma, \gamma) = Rayleigh(1.0, 0.0)$ definition.

Some methods :

getSigma

Usage : *getSigma()*

Arguments : none

Value : a real positive value, the σ parameter of the considered distribution

getGamma

Usage : *getGamma()*

Arguments : none

Value : a real value, the γ parameter of the considered distribution

Details :

probability function :

$$f(x) = \frac{(x - \gamma)}{\sigma^2} e^{-\frac{(x-\gamma)^2}{2\sigma^2}} \mathbf{1}_{[\gamma, +\infty[}(x)$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.28 Rice

This class inherits from the Distribution class.

Usage : Main parameters set : $Rice(\sigma, \nu)$

Arguments :

σ : a real positive value, constraint : $\sigma > 0$

ν : a real value, constraint : $\nu \geq 0$

Value : Rice. In the default construction, we use the $Rice(\sigma, \nu) = Rice(1, 0)$ definition.

Some methods :

getSigma

Usage : *getSigma()*

Arguments : none

Value : a real positive value, the σ parameter of the considered distribution

getNu

Usage : *getNu()*

Arguments : none

Value : a real value, the ν parameter of the considered distribution

Details :

probability function :

$$f(x) = 2 \frac{x}{\sigma^2} p_{\chi^2(2, \frac{\nu^2}{\sigma^2})} \left(\frac{x^2}{\sigma^2} \right)$$

where $p_{\chi^2(\nu, \lambda)}$ is the probability density function of a Non Central Chi Square distribution.

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.29 Student

This class inherits from the Distribution class.

Usage :

$Student(\nu, \underline{\mu}, \underline{\sigma}, \underline{R})$

$Student(\nu)$

$Student(\nu, \mu)$

$Student(\nu, \mu, \sigma)$

$Student(\nu, d)$

Default construction : $Student()$

Arguments :

ν : a real positive value, generalised number degree of freedom, constraint : $\nu > 0$.

$\underline{\mu}$ (μ) : a NumericalPoint (a real value), the mean value of the distribution if $\nu > 1$, its location parameter for $\nu \leq 1$.

$\underline{\sigma}$ (σ) : a NumericalPoint (a real value), the scale parameter of the distribution, constraint : $\sigma_i > 0$.

\underline{R} : a CorrelationMatrix, the correlation matrix of the distribution if $\nu > 2$, its generalized correlation matrix for $\nu \leq 2$.

d : an integer value, the dimension of the distribution, constraint: $d \geq 1$.

Value : a Student distribution. In the simplified constructions where the dimension d is not specified (usages number 2 to 5) and in the default construction, we use $d = 1$, $\nu = 3$, $\mu = 0$, $\sigma = 1$. In the last construction where the dimension dd is specified, we use $\underline{\mu} = (1, \dots, 1) \in \mathbb{R}^d$, $\underline{\sigma} = (1, \dots, 1) \in \mathbb{R}^d$ and $\underline{R} = \underline{Id}(d) \in \mathbb{R}^d \times \mathbb{R}^d$.

Some methods :

getMu

Usage : *getMu()*

Arguments : none

Value : a real value, the μ parameter of the Student distribution. Only defined when the dimension is 1 (else, use the *getMEan()* method inherited from the EllipticalDistribution class).

getNu

Usage : *getNu()*

Arguments : none

Value : a real value, the generalized number of degrees of freedom of the Student distribution;

Details :

density function :

$$f(\underline{x}) = \frac{\Gamma\left(\frac{\nu+d}{2}\right)}{(\pi d)^{\frac{d}{2}} \Gamma\left(\frac{\nu}{2}\right)} \frac{|\det(\underline{R})|^{-1/2}}{\prod_{k=1}^d \sigma_k} \left(1 + \frac{\underline{z}^t \underline{R}^{-1} \underline{z}}{\nu}\right)^{-\frac{\nu+d}{2}}$$

where $\underline{z} = \underline{\Delta}^{-1}(\underline{x} - \underline{\mu})$ with $\underline{\Delta} = \underline{\underline{diag}}(\underline{\sigma})$.

relation between parameters and moments :

$$\begin{aligned}\mathbb{E}[X] &= \underline{\mu} \text{ if } \nu > 1 \\ \underline{\underline{\text{Cov}}}[X] &= \underline{\underline{\Delta}}^t \underline{\underline{R}} \underline{\underline{\Delta}} \text{ if } \nu > 2\end{aligned}$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.30 Trapezoidal

This class inherits from the Distribution class.

Usage :

Main parameters set : $Trapezoidal(a, b, c, d)$

Default construction : $Trapezoidal()$

Arguments :

a : a real value, the lower bound

b : a real value, the level start

c : a real value, the level end

d : a real value, the upper bound, constraints: $a \leq b < c \leq d$

Value : Trapezoidal. In the default construction, we use the $Trapezoidal(a, b, c, d) = Trapezoidal(-2.0, -1.0, 1.0, 2.0)$ definition.

Some methods :

getA

Usage : *getA()*

Arguments : none

Value : a real value, the a parameter of the Trapezoidal distribution

getB

Usage : *getB()*

Arguments : none

Value : a real value, the b parameter of the Trapezoidal distribution

getC

Usage : *getC()*

Arguments : none

Value : a real value, the c parameter of the Trapezoidal distribution

getD

Usage : *getD()*

Arguments : none

Value : a real value, the d parameter of the Trapezoidal distribution

Details :

density function :

$$f_X(x; \underline{\theta}) = \begin{cases} h \frac{x-a}{b-a} & \text{if } a \leq x < b \\ h & \text{if } b \leq x < c \\ h \frac{d-x}{d-c} & \text{if } c \leq x < d \\ 0 & \text{otherwise} \end{cases}$$

with: $h = \frac{2}{d+c-a-b}$

relation between parameter sets :

$$\begin{aligned}\mu &= \frac{h}{6}(d^2 + cd + c^2 - b^2 - ab - a^2) \\ \sigma^2 &= \frac{h^2}{72}(d^4 + 2cd^3 - 3bd^3 - 3ad^3 - 3bcd^2 - 3acd^2 + \dots \\ &\quad \dots + 4b^2d^2 + 4abd^2 + 4a^2d^2 + 2c^3d - 3bc^2d - 3ac^2d \\ &\quad + 4b^2cd + 4abcd + 4a^2cd - 3b^3d - 3ab^2d - \dots \\ &\quad \dots 3a^2bd - 3a^3d + c^4 - 3bc^3 - 3ac^3 + 4b^2c^2 + 4abc^2 \\ &\quad + 4a^2c^2 - 3b^3c - 3ab^2c - 3a^2bc - \dots \\ &\quad \dots 3a^3c + b^4 + 2ab^3 + 2a^3b + a^4)\end{aligned}$$

$$\text{where} \quad \mu = \mathbb{E}[X] \quad \sigma^2 = \text{Var}[X]$$

Links : [see docref_B121_DistributionSelection](#)

Each *getMethod* is associated to a *setMethod*.

2.2.31 Triangular

This class inherits from the Distribution class.

Usage :

Main parameters set : $Triangular(a, m, b)$

Default construction : $Triangular()$

Arguments :

a : a real value, the lower bound

b : a real value, the upper bound, constraint : $b \geq a$

m : a real value, the mode, constant, $b \geq m \geq a$

Value : Triangular. In the default construction, we use the $Triangular(a, m, b) = Triangular(-1.0, 0.0, 1.0)$ definition.

Some methods :

getA

Usage : *getA()*

Arguments : none

Value : a real value, the a parameter of the Triangular distribution

getB

Usage : *getB()*

Arguments : none

Value : a real value, the b parameter of the Triangular distribution

getM

Usage : *getM()*

Arguments : none

Value : a real value, the m parameter of the Triangular distribution

Details :

density function :

$$f(x) = \begin{cases} \frac{2(x-a)}{(m-a)(b-a)} & a \leq x \leq m \\ \frac{2(b-x)}{(b-m)(b-a)} & m \leq x \leq b \\ 0 & \text{elsewhere} \end{cases}$$

relation between parameters set :

$$\begin{aligned}\mu &= \frac{1}{3}(a + m + b) \\ \sigma &= \sqrt{\frac{1}{18}(a^2 + b^2 + m^2 - ab - am - bm)}\end{aligned}$$

where $\mu = \mathbb{E}[X]$ $\sigma = \sqrt{\text{Var}[X]}$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.32 TruncatedNormal

This class inherits from the Distribution class.

Usage :

Main parameters set : $TruncatedNormal(\mu_n, \sigma_n, a, b)$

Default construction : $TruncatedNormal()$

Arguments :

μ_n : a real value which corresponds to the mean of the associated non truncated normal

σ_n : a real value which corresponds to the standard deviation of the associated non truncated normal

a : a real value, the lower bound

b : a real value, the upper bound, constraint : $b \geq a$

Value : TruncatedNormal . In the default construction, we use the

$TruncatedNormal(\mu_n, \sigma_n, a, b) = TruncatedNormal(0.0, 1.0, -1.0, 1.0)$ definition.

Some methods :

getA

Usage : *getA()*

Arguments : none

Value : a real value, the a parameter of the TruncatedNormal distribution

getB

Usage : *getB()*

Arguments : none

Value : a real value, the b parameter of the TruncatedNormal distribution

getMu

Usage : *getMu()*

Arguments : none

Value : a real value, the μ_n parameter of the TruncatedNormal distribution

getSigma

Usage : *getSigma()*

Arguments : none

Value : a real value, the σ_n parameter of the TruncatedNormal distribution

Details :

probability density function :

$$f(x) = \frac{\frac{1}{\sigma_n} \phi\left(\frac{x-\mu_n}{\sigma_n}\right)}{\Phi\left(\frac{b-\mu_n}{\sigma_n}\right) - \Phi\left(\frac{a-\mu_n}{\sigma_n}\right)} \mathbf{1}_{[a,b]}(x)$$

(where ϕ and Φ are, respectively, the probability density distribution function and the cumulative distribution function of a standard normal distribution)

relation between parameters set :

$$\mu = \mu_n - \sigma_n \frac{\phi(b_{red}) - \phi(a_{red})}{\Phi(b_{red}) - \Phi(a_{red})}$$

$$\sigma = \sigma_n \left\{ 1 - \frac{b_{red} \phi(b_{red}) - a_{red} \phi(a_{red})}{\Phi(b_{red}) - \Phi(a_{red})} - \left[\frac{\phi(b_{red}) - \phi(a_{red})}{\Phi(b_{red}) - \Phi(a_{red})} \right]^2 \right\}^{1/2}$$

where

$$a_{red} = \frac{a - \mu_n}{\sigma_n} \quad b_{red} = \frac{b - \mu_n}{\sigma_n}$$

and

$$\mu = \mathbb{E}[X] \quad \sigma = \sqrt{\text{Var}[X]}$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.33 Uniform

This class inherits from the Distribution class.

Usage :

Main parameters set : $Uniform(a, b)$

Default construction : $Uniform()$

Arguments :

a : a real value, the lower bound

b : a real value, the upper bound, constraint : $b \geq a$

Value : Uniform. In the default construction, we use the $Uniform(a, b) = Uniform(-1.0, 1.0)$ definition.

Some methods :

getA

Usage : *getA()*

Arguments : none

Value : a real value, the a parameter of the Uniform distribution

getB

Usage : *getB()*

Arguments : none

Value : a real value, the b parameter of the Uniform distribution

Details :

density function :

$$f(x) = \begin{cases} \frac{1}{(b-a)} & a \leq x \leq b \\ 0 & \text{elsewhere} \end{cases}$$

relation between parameters set :

$$\begin{aligned} \mu &= \frac{a+b}{2} \\ \sigma &= \frac{b-a}{2\sqrt{3}} \end{aligned}$$

where

$$\mu = \mathbb{E}[X]$$

$$\sigma = \sqrt{\text{Var}[X]}$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.34 UserDefined

This class inherits from the Distribution class.

Usage : *UserDefined(Coll)*

Arguments : *Coll* : a UserDefinedPairCollection. The collection of UserDefinedPair of the UserDefinedPairCollection does not need such that $\sum_1^n p_i = 1.0$. If not the case, the weights are normalized.

Value : a UserDefined

Some methods :

getPairCollection

Usage : *getPairCollection()*

Arguments : none

Value : a UserDefinedPairCollection, the *Coll* parameter of the considered distribution where the weights are normalized.

getSupport

Usage : *getSupport(interval)*

Arguments : *interval* : a *Interval*, an interval in \mathbb{R}

Value : a *NumericalSample*, all the points (here of dimension *n*) of the distribution range which are included in the interval *interval*.

isIntegral

Usage : *isIntegral()*

Arguments : no argument

Value : a boolean which indicates whether the considered distribution has integer values.

Details :

probability function :

$$\mathbb{P}(X = x_i) = p_i, \quad i = 1, \dots, n$$

where

(x_i, p_i) and $i = 1, \dots, n$ are respectively a *NumericalPoint* and its associated probability
n is the size of the UserDefinedPairCollection

One must have

$$\sum_{i=1}^n p_i = 1$$

Each *getMethod* is associated to a *setMethod*.

2.2.35 Weibull

This class inherits from the Distribution class.

Usage :

Main parameters set : $Weibull(\alpha, \beta, \gamma)$

Second parameter set : $Weibull(\mu, \sigma, \gamma, 1)$

Default construction : $Weibull()$

Arguments :

α : a real value, the shape parameter, constraint : $\alpha > 0$

β : a real value, the scale parameter, constraint : $\beta > 0$

γ : a real value, the location parameter

μ : a real value, the mean value,

σ : a real value, the standard deviation value, constraint : $\sigma > 0$

Value : a Weibull. In the default construction, we use the $Weibull(\alpha, \beta, \gamma) = Weibull(1.0, 1.0, 0.0)$ definition.

Some methods :

getAlpha

Usage : *getAlpha()*

Arguments : none

Value : a real value, the α of the considered distribution

getBeta

Usage : *getBeta()*

Arguments : none

Value : a real value, the β of the considered distribution

getGamma

Usage : *getGamma()*

Arguments : none

Value : a real value, the γ parameter of the considered distribution

getMu

Usage : *getMu()*

Arguments : none

Value : a real value, the μ parameter of the considered distribution

getSigma

Usage : *getSigma()*

Arguments : none

Value : a real value, the σ parameter of the considered distribution

Details :

density function :

$$f(x) = \frac{\beta}{\alpha} \left(\frac{x - \gamma}{\alpha} \right)^{\beta-1} e^{-\left(\frac{x-\gamma}{\alpha}\right)^\beta} \mathbf{1}_{[\gamma, +\infty[}(x)$$

relation between parameters set :

$$\begin{aligned} \mu &= \alpha \Gamma \left(1 + \frac{1}{\beta} \right) + \gamma \\ \sigma &= \alpha \sqrt{\Gamma \left(1 + \frac{2}{\beta} \right) - \Gamma^2 \left(1 + \frac{1}{\beta} \right)} \end{aligned}$$

where Γ is the Γ -function and

$$\mu = \mathbb{E}[X] \qquad \sigma = \sqrt{\text{Var}[X]}$$

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.2.36 ZipfMandelbrot

This class inherits from the Distribution class.

Usage :

Main parameters set : $ZipfMandelbrot(N, q, s)$

Default construction : $ZipfMandelbrot()$

Arguments :

N : a integer, $N \geq 1$

q : a real value, $q \geq 0$

s : a real value, $s > 0$

Value : ZipfMandelbrot. In the default construction, we use the $ZipfMandelbrot(N, q, s) = ZipfMandelbrot(1, 0, 1)$ definition.

Some methods :

getN

Usage : *getN()*

Arguments : none

Value : an integer, the N parameter of the ZipfMandelbrot distribution

getQ

Usage : *getQ()*

Arguments : none

Value : a real value ≥ 0 , the q parameter of the ZipfMandelbrot distribution

getS

Usage : *getS()*

Arguments : none

Value : a real value > 0 , the s parameter of the ZipfMandelbrot distribution

Details :

distribution :

$$\forall k \in [1, N], P(X = k) = \frac{1}{(k + q)^s} \frac{1}{H(N, q, s)}$$

where $H(N, q, s)$ is the Generalized Harmonic Number : $H(N, q, s) = \sum_{i=1}^N \frac{1}{(i + q)^s}$.

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.3 TruncatedDistribution

This class enables to truncate any distribution within a specified range which one of the bounds may be infinite. It offers the methods of the Distribution class.

Usage :

TruncatedDistribution(*distribution*, *lowerBound*, *upperBound*)

TruncatedDistribution(*distribution*, *bound*, *TruncatedDistribution.UPPER*)

TruncatedDistribution(*distribution*, *bound*, *TruncatedDistribution.LOWER*)

Arguments :

distribution : a Distribution

lowerBound : a real, the new lower bound of the distribution : the distribution range is [*lowerBound*, ∞ [or [*lowerBound*, *max*[if the distribution is already bounded by *max*

upperBound : a real, the new upper bound of the distribution : the distribution range is [$-\infty$, *upperBound*] or [*min*, *upperBound*] if the distribution is already bounded by *min*

Value : a Distribution

Some methods :

isContinuous

Usage : *isContinuous*()

Arguments : no argument

Value : a boolean which indicates wether the considered distribution is continuous.

isIntegral

Usage : *isIntegral*()

Arguments : no argument

Value : a boolean which indicates wether the considered distribution has integer values.

getSupport

Usage : *getSupport*()

Arguments : none

Value : a NumericalSample which gathers the different points of the discrete range. Care : this service is implemented only for discrete 1D distribution.

Links : [see docref_B122_Copules_en](#)

Each *getMethod* is associated to a *setMethod*.

2.4 Copulas

2.4.1 Copula

This class inherits from the Distribution class.

Usage : *Copula(copulaImplementation)*

Arguments : *copulaImplementation* : a CopulaImplementation, i.e a distribution which must verify the properties of a copula. This distribution can be any of the IndependentCopula, NormalCopula, ClaytonCopula, GumbelCopula, FrankCopula, MinCopula or SklarCopula built upon a multivariate distribution such as the Normal, Student, Dirichlet, Multinomial, KernelMixture, Mixture ones.

Value : a Copula

Links : [see docref_B122_Copules_en](#)

Each *getMethod* is associated to a *setMethod*.

2.4.2 ClaytonCopula

This class inherits from the CopulaImplementation class.

Usage :

ClaytonCopula()

ClaytonCopula(theta)

Arguments : *theta* : a real, the only parameter of the Clayton copula, which PDF is : $\left(u_1^{-\theta} + u_2^{-\theta} - 1\right)^{-1/\theta}$,
for $u_i \in [0, 1]$

Value :

In the first usage, a ClaytonCopula of dimension 2 with $\theta = 2.0$,

In the second usage, a ClaytonCopula of dimension 2 with the θ specified.

Links : [see docref_B122_Copules_en](#)

2.4.3 FrankCopula

This class inherits from the CopulaImplementation class.

Usage :

FrankCopula()

FrankCopula(theta)

Arguments : *theta* : a real, the only parameter of the Gumbel copula, which PDF is :

$$-\frac{1}{\theta} \log \left(1 + \frac{(e^{-\theta u_1} - 1)(e^{-\theta u_2} - 1)}{e^{-\theta} - 1} \right)$$

, for $u_i \in [0, 1]$

Value :

In the first usage, a FrankCopula of dimension 2 with $\theta = 2.0$,

In the second usage, a FrankCopula of dimension 2 with the θ specified.

Links : [see docref_B122_Copules_en](#)

2.4.4 GumbelCopula

This class inherits from the CopulaImplementation class.

Usage :

GumbelCopula()

GumbelCopula(theta)

Arguments : *theta* : a real, the only parameter of the Gumbel copula, which PDF is :

$$e^{-\left(-\log(u_1)\right)^\theta + \left(-\log(u_2)\right)^\theta}^{1/\theta}$$

, for $u_i \in [0, 1]$

Value :

In the first usage, a GumbelCopula of dimension 2 with $\theta = 2.0$,

In the second usage, a GumbelCopula of dimension 2 with the θ specified.

Links : [see docref_B122_Copules_en](#)

2.4.5 IndependentCopula

This class inherits from the CopulaImplementation class.

Usage :

IndependentCopula()

IndependentCopula(dim)

Arguments : *dim* : an integer, the dimension of the copula

Value :

In the first usage, a IndependentCopula of dimension 1,

In the second usage, a IndependentCopula of the dimension *dim* specified.

Links : [see docref_B122_Copules_en](#)

2.4.6 MinCopula

This class inherits from the CopulaImplementation class.

Usage :

IndependentCopula()

IndependentCopula(dim)

Arguments : *dim* : an integer, the dimension of the copula

Value :

In the first usage, a MinCopula of dimension 1,

In the second usage, a MinCopula of the dimension *dim* specified.

Links : [see docref_B122_Copules_en](#)

2.4.7 NormalCopula

This class inherits from the CopulaImplementation class.

Usage :

NormalCopula()

NormalCopula(R)

Arguments : *R* : a CorrelationMatrix which is not the Kendall nor the Spearman rank correlation matrix of the distribution. The *R* matrix can be evaluated from the Spearman or Kendall correlation matrix.

Value :

In the first usage, a NormalCopula of dimension 1

In the second usage, a NormalCopula with the correlation matrix *R* specified.

Some methods :

GetCorrelationFromKendallCorrelation

Usage : *NormalCopula.GetCorrelationFromKendallCorrelation(K)*

Arguments : *K* : a CorrelationMatrix, it must be the Kendall correlation matrix of the considered random vector

Value : a CorrelationMatrix, the correlation matrix of the normal copula evaluated from the Kendall correlation matrix *K*

GetCorrelationFromSpearmanCorrelation

Usage : *NormalCopula.GetCorrelationFromSpearmanCorrelation(S)*

Arguments : *S* : a CorrelationMatrix, it must be the Spearman correlation matrix of the considered random vector

Value : a CorrelationMatrix, the correlation matrix of the normal copula evaluated from the Spearman correlation matrix *S*

Links : [see docref_B122_Copules_en](#)

2.4.8 SklarCopula

This class inherits from the CopulaImplementation class.

Usage : *SklarCopula(distribution)*

Arguments : *distribution* : a Distribution, whatever its type (UsualDistribution, ComposedDistribution, KernelMixture, Mixture, RandomMixture, Copula, ...)

Value : a SklarCopula with the same dimension as the *distribution*

Some methods : no specific method.

Links : [see docref_B122_Copules_en](#)

2.4.9 ComposedCopula

This class inherits from the CopulaImplementation class.

Usage : *ComposedCopula(copulaCollection)*

Arguments : *copulaCollection* : a CopulaCollection

Value : a ComposedCopula, defined as the product of the initial copulas. For example, if C_1 and C_2 are two copulas respectively of \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 :

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

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

Some methods :

getCopulaCollection

Usage : *getCopulaCollection()*

Arguments : none

Value : a CopulaCollection, the collection of copulas from which the ComposedCopula is built

Links : [see docref_B_JoinedCDF_en](#)

2.5 ComposedDistribution

This class inherits from the Distribution class.

Usage :

ComposedDistribution(distributionCollection, copula)

ComposedDistribution(distributionCollection)

Arguments :

distributionCollection : a DistributionCollection, the collection of the marginals of the distribution

copula : a Copula, the copula of the distribution.

Value : a ComposedDistribution,

in the first usage : which marginals and copula are specified,

in the second usage : which marginals are specified, and which copula is the independent one.

Some methods :

getDistributionCollection

Usage : *getDistributionCollection()*

Arguments : none

Value : a DistributionCollection, the collection of distributions from which the ComposedDistribution is built

Links : [see docref_B_JoinedCDF_en](#)

Each *getMethod* is associated to a *setMethod*.

2.6 Linear combination of probability density functions

2.6.1 Mixture

A Mixture is a distribution such that its probability density function is a linear combination of probability density functions, with the linear combination coefficients greater or equal to zero such that their sum is equal to 1.

It is important to note that the linear combination coefficients are given through the *weight* attribute of each component of the *DistributionCollection*, thanks to the command *DistributionCollection[i].setWeight(coefficient)*.

Usage :

Mixture(distributionCollection)

Mixture(distributionCollection, weights)

Arguments :

distributionCollection : a *DistributionCollection*, the collection of the distributions which compose the linear combination

weights : a *NumericalPoint*, which contains the weights of the distributions in the mixture. These weights will be used instead of the individual weights of each distribution.

Value : a Mixture

Some methods :

getDistributionCollection

Usage : *getDistributionCollection()*

Arguments : none

Value : a *DistributionCollection* the collection of distribution from which the Mixture is built

Details :

probability density function :

$$f(x) = \sum_{i=1}^n \alpha_i p_i(x)$$

with $\alpha_i \geq 0$. The null weights are automatically removed from the sum, and the coefficients are automatically normalized such that $\sum_{i=1}^n \alpha_i = 1$.

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.6.2 KernelMixture

A KernelMixture is a distribution built from a NumericalSample, such that its probability density function is a linear combination of the kernel specified by the User, centered on each point of the NumericalSample, which standard deviation is the bandwidth specified by the User. It is important to note that the linear combination coefficients are all equal.

Usage : *KernelMixture(kernel, bandwidth, sample)*

Arguments : *distributionCollection* : a DistributionCollection, the collection of the distributions which compose the linear combination

Value : a KernelMixture

Some methods :

getBandwidth

Usage : *getBandwidth()*

Arguments : none

Value : a NumericalPoint, the bandwidth of the kernel mixture, (see equation below for dimension 1). The bandwidth is the same at each point of the NumericalSample

getKernel

Usage : *getKernel()*

Arguments : none

Value : a Distribution, the kernel K of the mixture, (see equation below for dimension 1)

getSample

Usage : *getSample()*

Arguments : none

Value : a NumericalSample, the NumericalSample of the mixture, (see equation below for dimension 1)

Details :

Probability density function in dimension 1 :

$$f(x) = \sum_{i=1}^n \frac{1}{nh} K\left(\frac{X^i - x}{h}\right), x \in \mathbb{R}$$

where (X^1, \dots, X^n) is a NumericalSample

Links : [see Reference Guide - Standard parametric models](#)

Each *getMethod* is associated to a *setMethod*.

2.6.3 KernelSmoothing

The class KernelSmoothing enables to build some kernels used to fit a distribution to a numerical sample.

Usage :

```
KernelSmoothing()
KernelSmoothing(Distribution(myDistribution))
KernelSmoothing(DistributionImplementation())
```

Arguments :

myDistribution : a 1D Distribution of any kind
DistributionImplementation() : default constructor of the 1D UsualDistribution. For example, *Uniform()*, *Triangular()*, ...

Value : a Distribution

In the first usage, the kernel is the kernel product of 1D Normal(1.0, 0.0). The dimension of the product is detected from the numerical sample.

In the second usage, the kernel is the kernel product of 1D distributions specified by *myDistribution*. Care : the kernel smoothing method is all the more efficient than the kernel is symmetric with respect to 0.0. The dimension of the product is detected from the numerical sample.

In the third usage, the kernel is the kernel product of the default constructions of the 1D UsualDistributions. Note that the default constructor of a UsualDistribution builds a distribution which is symmetric with respect to 0.0 when it is possible. The dimension of the product is detected from the numerical sample.

Some methods :

build

Usage :

```
build(sample)
build(sample, boundaryCorrection)
build(sample, bandwidth)
build(sample, bandwidth, boundaryCorrection)
```

Arguments :

sample : a NumericalSample, the numerical sample from which the kernel mixture is built
boundaryCorrection : a Bool which indicates if it is necessary to make a boundary treatment (according to the mirroring technique)
bandwidth : a NumericalPoint, the bandwidth of the kernel product. The dimension is detected from the numerical sample *sample* and evaluated according to the Scott rule.

Value : a Distribution. When the bandwidth is not specified, Open TURNS proceeds as follows : the plug-in method on the entire numerical sample if its size is inferior to 250; the mixed method in the other case. Refer to the Reference Guide in order to have details on these methods.

computeSilvermanBandwidth

Usage : *computeSilvermanBandwidth(sample)*

Arguments : *sample* : a NumericalSample, the numerical sample from which the kernel mixture is built

Value : a NumericalPoint, the bandwidth automatically evaluated by Open TURNS from the numerical sample according to the Silverman rule (see Reference Guide)

computePluginBandwidth

Usage : *computePluginBandwidth(sample)*

Arguments : *sample* : a NumericalSample, the numerical sample from which the kernel mixture is built

Value : a NumericalPoint, the bandwidth automatically evaluated by Open TURNS from the numerical sample according to the plug-in method (see Reference Guide)

computeMixedBandwidth

Usage : *computeMixedBandwidth(sample)*

Arguments : *sample* : a NumericalSample, the numerical sample from which the kernel mixture is built

Value : a NumericalPoint, the bandwidth automatically evaluated by Open TURNS from the numerical sample according to the mixed method (see Reference Guide)

getBandwidth

Usage : *getBandwidth()*

Arguments : none

Value : a NumericalPoint, the bandwidth of the kernel mixture, (see equation below for dimension 1). The bandwidth is the same at each point of the NumericalSample

getKernel

Usage : *getKernel()*

Arguments : none

Value : a Distribution, the kernel adopted for the kernel smoothing

Details :

Probability density function in dimension 1 :

$$p_n(x) = \sum_{i=1}^n \frac{1}{nh} K\left(\frac{x - X^i}{h}\right), x \in \mathbb{R}$$

where (X^1, \dots, X^n) is a NumericalSample and K the kernel PDF,

Probability density function in dimension N :

$$p_n(x) = \frac{1}{n} \sum_{i=1}^n \prod_{j=1}^N \frac{1}{h_j} K\left(\frac{x_j - X_j^i}{h_j}\right)$$

where $\prod_{j=1}^N K(x^j)$ is the kernel product and $\underline{h} = (h^1, \dots, h^N)$ the vector of bandwidth.

Links : [see Reference Guide - Standard parametric models](#)

2.7 Affine combinations of independent univariate random variables

2.7.1 RandomMixture

A RandomMixture Y is a univariate random variable defined as an affine combination of independent univariate random variable, as follows :

$$Y = a_0 + \sum_{k=1}^n a_k X_k$$

where $(a_i)_{0 \leq k \leq n} \in \mathbb{R}$ and $(X_k)_{1 \leq k \leq n}$ are some independent univariate distributions.

Usage :

RandomMixture(collDist)
RandomMixture(collDist, constant)
RandomMixture(collDist, weights)
RandomMixture(collDist, weights, constant)

Arguments :

collDist : a DistributionCollection, the collection of the univariate independent distributions distributions which compose the affine combination,
constant : a scalar, the constant term a_0 of the affine combination,
weights : a NumericalPoint, which contains the weights of the affine combination : (a_1, \dots, a_n) .

Value : a RandomMixture such that :

in the first usage : the weights a_i have been affected in the distributions of the univariate random variables X_i , thanks to the method *setWeight(a_i)*, before the creation of *collDist*. If not specified, the weight by default is 1.0. The constant term $a_0 = 0$.
in the second usage : the weights are defined as in the first usage and the constant term a_0 is equal to *constant*.
in the third usage : the weights are directly specified at the creation of the RandomMixture. The distribution weights are modified to be equal to those specified values. The constant term $a_0 = 0$.
in the fourth usage : the weights are specified as in the third usage and the constant term a_0 is equal to *constant*.

Some methods : As a *RandomMixture* is a *Distribution*, it is possible to ask it any request compatible with a *Distribution* object : moments, quantiles, PDF and CDF evaluations, ...

project

Usage :

project(factoryCollection, testResult)
project(factoryCollection, testResult, size)

Arguments :

factoryCollection : a FactoryCollection, the collection of models from which one want to find the best approximation of the RandomMixture.

testResult : a `TestResult`, the test result associated with the best model found within the collection of `DistributionFactory` upon which one projects the `RandomMixture`.

size : an integer. The `RandomMixture` is regularly sampled over a grid of $2 \textit{size} + 1$ points that correspond to quantiles of regularly spaced levels, then one uses the Kolmogorov test to identify the best candidate amongst the several factories. If not given, *size* defaults to 50, which means a regular sample of size 101.

Value : a `Distribution`, the best candidates found according to the Kolmogorov test within the given `DistributionFactoryCollection` using the regular sample of size $2 \textit{size} + 1$.

Links : [see Reference Guide - Standard parametric models](#)

2.8 Random vector

2.8.1 RandomVector

Usage :

RandomVector(distribution)
RandomVector(function, antecedent)
RandomVector(functionalChaosResult)
RandomVector(constant)

Arguments :

distribution : a Distribution
function : a NumericalMathFunction
antecedent : a RandomVector of type Usual (see farther)
functionalChaosResult : a FunctionalChaosResult which contains the results of a Chaos Polynomial approximation
constant : a NumericalPoint

Value : a RandomVector, which is of type :

Usual : if created thanks to the first usage. In that case, the RandomVector has for distribution the one specified through *distribution*.
Composite : if created thanks to the second usage. In that case, the RandomVector is defined as the image through the function *function* of the Usual RandomVector *antecedent* : $Y = fuction(antecedent)$.
FunctionalChaosRandomVector : if created thanks to the third usage. In that case, the RandomVector is the image through a functional chaos approximation model of the assoiated Usual RandomVector.
Constant : if created thanks to the fourth usage. In that case, the RandomVector is a constant vector equal to the NumericalPoint specified *constant*.

Some methods :

getAntecedent

Usage : *getAntecedent()*

Arguments : no argument

Value : a RandomVector, only in the case of Composite RandomVector : the RandomVector X such that $Y = function(X)$.

getCovariance

Usage : *getCovariance()*

Arguments : no argument

Value : a CovarianceMatrix, only in the case of Usual or FunctionalChaos RandomVector : the covariance of the considered RandomVector

getDistribution

Usage : *getDistribution()*

Arguments : no argument

Value : a Distribution, only in the case of Usual RandomVector : the distribution of the RandomVector

getDescription

Usage : *getDescription()*

Arguments : no argument

Value : a Description, the description of the Randomvector

getDimension

Usage : *getDimension()*

Arguments : no argument

Value : an integer, the dimension of the RandomVector

getFunctionalChaosResult

Usage : *getFunctionalChaosResult()*

Arguments : no argument

Value : a FunctionalChaosResult, only in the case of FunctionalChaos RandomVector : the result structure of the Chaos Polynomial study.

getMarginal

Usage :

getMarginal(i)

getMarginal(indices)

Arguments :

i : an integer which indicates the component concerned

indices : an Indices which regroupes all the components concerned

Value : a RandomVector restricted to the concerned components.

Details : Let's note $\underline{Y} = (Y_1, \dots, Y_n)^t$ a random vector and $I \in [1, n]$ a set of indices. If \underline{Y} is a UsualRandomvector, the subvector is defined by $\tilde{\underline{Y}} = (Y_i)_{i \in I}^t$. If \underline{Y} is a CompositeRandomVector, defined by $\underline{Y} = f(\underline{X})$ with $f = (f_1, \dots, f_n)$, f_i some scalar functions, the sub vector is $\tilde{\underline{Y}} = (f_i(\underline{X}))_{i \in I}$.

getMean

Usage : ()

Arguments : no argument

Value : a NumericalPoint, only in the case of Usual or FunctionalChaos RandomVector : the mean vector of the associated distribution

getName

Usage : *getName()*

Arguments : no argument

Value : a string, the name of the RandomVector

getNumericalSample

Usage : *getNumericalSample()*

Arguments : no argument

Value : a NumericalSample a sample of the random vector

isComposite

Usage : *isComposite()*

Arguments : no argument

Value : a boolean which indicates if the RandomVector is of type Composite or Usual.

Each *getMethod* is associated to a *setMethod*.

2.8.2 ConditionalRandomVector

Usage : *ConditionalRandomVector(conditionalDist, randomParameters)*

Arguments :

conditionalDist : a Distribution

randomParameters : a RandomVector

Value : a ConditionalRandomVector, \underline{Y} which distribution $\mathcal{L}(\underline{\Theta})$ has random parameters $\underline{\Theta}$ distributed according to the distribution $\mathcal{D}_{\underline{\Theta}}$. The random vector $\underline{\Theta}$ can be :

- a *UsualRandomVector* which means described by a given distribution $\mathcal{D}_{\underline{\Theta}}$,
- or a *CompositeRandomVector* which means the output vector of a function f evaluated on the random vector \underline{X} : $\underline{\Theta} = f(\underline{X})$. In that case, the distribution $\mathcal{D}_{\underline{\Theta}}$ is not explicitly known.

Some methods :

getAntecedent

Usage : *getRealization()*

Arguments : no argument

Value : a NumericalPoint, a realization of the final distribution of \underline{Y} . Open TURNS proceeds as follows : it first generates a realization $\underline{\theta}$ of the random vector $\underline{\Theta}$ according to $\mathcal{D}_{\underline{\Theta}}$ then a realization of the distribution $\mathcal{L}(\underline{\theta})$ where the random vector $\underline{\Theta}$ is fixed to the previous realization $\underline{\theta}$.

getRandomParameters

Usage : *getRandomParameters()*

Arguments : no argument

Value : a RandomVector, the random vector $\underline{\Theta}$.

getDistribution

Usage : *getDistribution()*

Arguments : no argument

Value : a Distribution, the conditional distribution $\mathcal{L}(\underline{\Theta})$.

getDimension

Usage : *getDimension()*

Arguments : no argument

Value : an integer, the dimension of the distribution $\mathcal{L}(\underline{\Theta})$.

3 Design of experiments

3.1 Experiment

Usage : *Experiment(expPlaneImplentation)*

Arguments : *expPlaneImplentation* : an ExperimentImplementation, which is a WeightedExperiment or a StratifiedExperiment.

Value : an Experiment

Links [see docref_C11_MinMaxPlanExp](#)

Each *getMethod* is associated to a *setMethod*.

3.2 Stratified Design of Experimentss

3.2.1 StratifiedExperiment

This class inherits from the ExperimentImplementation class.

Usage : *StratifiedExperiment(center, levels)*

Arguments :

center : a NumericalPoint, which has different meanings according to the nature of the stratified experiment : Axial, Composite, Factorial or Box (see corresponding documentation)

levels : a NumericalPoint, which has different meanings according to the nature of the stratified experiment : Axial, Composite, Factorial or Box (see corresponding documentation)

Value : an StratifiedExperiment

Some methods :

generate

Usage : *generate()*

Arguments : none

Value : a NumericalSample, the points which constitute the stratified design of experiments , according to its nature : Axial, Composite, Factorial or Box (see corresponding documentation)

getLevels

Usage : *getLevels()*

Arguments : none

Value : a NumericalPoint which corresponds to the levels of the stratified experiment, according to its nature : Axial, Composite, Factorial or Box (see corresponding documentation)

getCenter

Usage : *getCenter()*

Arguments : none

Value : a NumericalPoint which corresponds to the cenetr of the stratified experiment, according to its nature : Axial, Composite, Factorial or Box (see corresponding documentation)

.str()

Usage : *str()*

Arguments : no argument

Value : a string which describes the Experiment

3.2.2 Axial

This class inherits from the StratifiedExperiment class.

Usage :

Axial(center, levels)

Axial(dimension, levels)

Arguments :

center : a NumericalPoint, the center of the design of experiments

levels : a NumericalPoint, the discretisation of directions (the same for each one), without any consideration of unit

dimension : an integer, the dimension of the space where the design of experiments is created

Value : a Axial

In the first usage, the design of experiments is centered on *center*.

In the second usage, the design of experiments is centered on the *center = 0*. It is recommended to use that usage, then to scale the NumericalSample generated in order to take into consideration the unit of each direction, then to translate it onto the center point.

Details :

Number of points generated : $1 + 2 * levels.getDimension() * dimension$

The axial plane generates a NumericalSample where :

the first point is the vector (*center*),

the following points are : each coordinate one at a time is equal to +/- levels[i], for each direction so on, until the last level.

Links

3.2.3 Factorial

This class inherits from the StratifiedExperiment class.

Usage :

Factorial(center, level)

Factorial(dimension, level)

Arguments :

center : a NumericalPoint, the center of the design of experiments

level : a NumericalPoint, the discretisation of directions (the same for each one), without any consideration of unit

dimension : an integer, the dimension of the space where the design of experiments is created

Value : a Factorial

In the first usage, the design of experiments is centered on *center*.

In the second usage, the design of experiments is centered on the *center = 0*. It is recommended to use that usage, then to scale the NumericalSample generated in order to take into consideration the unit of each direction, then to translate it onto the proper center.

Details :

Number of points generated : $1 + levels.getDimension() * 2^{dimension}$

The factorial plane generates a NumericalSample where :

the first point is the vector (*center*),

the following points are : all coordinates are equal to +/- levels[i] for each direction

Links

3.2.4 Composite

This class inherits from the StratifiedExperiment class.

Usage :

Composite(center, level)

Composite(dimension, level)

Arguments :

center : a NumericalPoint, the center of the design of experiments

level : a NumericalPoint, the discretisation of directions (the same for each one), without any consideration of unit

dimension : an integer, the dimension of the space where the design of experiments is created

Value : a Composite

if defined with the first usage, the design of experiments is centered on *center*

if defined with the second usage, the design of experiments is centered on the *center* = 0. It is recommended to use that usage, then to scale the NumericalSample generated in order to take into consideration the unit of each direction, then to translate it onto the proper center.

Details :

A composite plane is the union of an axial and a factorial one

Number of points generated : $1 + levels.getDimension() * (2 * dimension + 2^{dimension})$

The composite plane generates a NumericalSample where :

In the first usage, the design of experiments is centered on *center*.

Links

3.2.5 Box

This class inherits from the StratifiedExperiment class.

Usage :

Box(levels)

Arguments :

levels : a NumericalPoint, which specifies the number of intermediate points on each direction which regularly discretize $[0, 1]$. In direction i , the points number is $levels[i] + 2$: the extreme bounds 0 and 1 are always taken.

Value : a Box, which regularly discretizes the unit pavement $[0, 1]^n$ with the specified number of intermediate points for each direction

Details :

Box discretizes the unit pavement $[0, 1]^n$ where $n = levels.getDimension()$.

Number of points generated : $\prod_{i=1}^n (2 + levels[i])$.

The box plane generates a NumericalSample which discretizes the unit pavement $[0, 1]^n$ where $n = levels.getDimension()$: each direction i contains the extreme bounds 0 and 1 and the $levels[i]$ intermediate points regularly positioned between these extreme bounds.

It is recommended to use the *scale*, *translate* methods of the NumericalSample in order to scale each direction and translate the grid structure onto the proper center.

Links [see docref_C11_MinMaxPlanExp](#)

3.3 WeightedExperiment

This class inherits from the ExperimentImplementation class.

It is used to define the approximation of the the esperance E_μ :

$$E_\mu [f(\underline{Z})] \simeq \sum_{i \in I} \omega_i f(\Xi_i) \quad (1)$$

where f is a function $L_1(\mu)$.

Usage : *WeightedExperiment(distribution, size)*

Arguments :

distribution : a Distribution, the distribution μ of (1).

size : an integer, the number of points that will be generated in the experiment.

Value : a WeightedExperiment

Some methods :

generate

Usage : *generate()*

Arguments : none

Value : a NumericalSample, the points $(\Xi_i)_{i \in I}$ which constitute the design of experiments with $cardI = size$. The sampling method is defined by the nature of the weighted experiment.

getDistribution

Usage : *getDistribution()*

Arguments : none

Value : the Distribution μ of (1). .

getSize

Usage : *getSize()*

Arguments : none

Value : an integer, the number $cardI$ of points of the design of experiments .

getWeight

Usage : *getWeight()*

Arguments : none

Value : a NumericalPoint of dimension 1 and size $cardI$ that contains all the weights : $(\omega_i)_{i \in I}$.

.str()

Usage : *str()*

Arguments : no argument

Value : a string which describes the WeightedExperiment.

Links

3.4 Random Weighted Design of Experimentss

3.4.1 LHSExperiment

This class inherits from the WeightedExperiment class.

Usage :

LHSExperiment(distribution, size)

Arguments :

distribution : the Distribution μ of (1) according to which the points of the design of experiments will be generate with the LHS technique. CARE : that distribution must have an independent copula to make the method valid;

size : an integer, the number *cardI* of points that will be generated.

Value : a LHSExperiment. The weights asosciated to the points are all equal to $\frac{1}{cardI}$.

Details : The method *generate* generates a NumericalSample of points $(\Xi_i)_{i \in I}$ which points are generated according to μ with the LHS technique : some cellulars are determined, with the same probabilistic content according to *distribution*, then cellulars are selected verifying the constraint «one by line and column», then points are selected among these selected cellulars. When the method *generate* is recalled, the NumericalSample generated changes : the point selection within the cellulars changes but not the cellulars selection. To change the cellular selection, it is necessary to create a new LHS Experiment.

Links

3.4.2 MonteCarloExperiment

This class inherits from the WeightedExperiment class.

Usage :

MonteCarloExperiment(distribution, size)

Arguments :

distribution : a Distribution, the distribution μ of (1) according to which the points of the design of experiments will be generate with the Monte Carlo technique. CARE : that distribution must have an independent copula to make the method valid;

size : an integer, the number of points *cardI* that will be generated.

Value : a MonteCarloExperiment. The weights asosciated to the points are all equal to $\frac{1}{cardI}$.

Details : The method *generate* generates a NumericalSample of points $(\Xi_i)_{i \in I}$ which points are independently generated from μ . When the method *generate* is recalled, the sample generated changes.

Links

3.4.3 ImportanceSamplingExperiment

This class inherits from the WeightedExperiment class.

Usage :

ImportanceSamplingExperiment(distribution, importanceDistribution, size)

Arguments :

distribution : a Distribution, the distribution μ bution, the distribution μ of (1). CARE : that distribution must have an independent copula to make the method valid.

importanceDistribution : a Distribution, the distribution p according to which the points of the design of experiments will be generated with the Importance Sampling technique. CARE : that distribution must have an independent copula to make the method valid.

size : an integer, the number *cardI* of points that will be generated.

Value : a ImportanceSamplingExperiment. The weights asociated to the points are all equal to $\frac{1}{cardI} \frac{\mu(\Xi_i)}{p(\Xi_i)}$.

Details : The method *generate* generates a NumericalSample of points $(\Xi_i)_{i \in I}$ which points are independently generated from *distribution*. When the method *generate* is recalled, the sample generated changes.

Links

3.5 Deterministic Weighted Experiments

3.5.1 FixedExperiment

This class inherits from the `WeightedExperiment` class.

Usage :

FixedExperiment(partNumSamp)

FixedExperiment(partNumSamp, weight)

Arguments :

partNumSamp : a `NumericalSample`, a sample that already exists.

weight : a `NumericalPoint`, the weight of each point of *partNumSamp*

Value : a `FixedExperiment`. The weights associated to the points are all equal to $\frac{1}{\text{card}I}$ when not specified. Then, the sample *partNumSamp* is considered as generated from the limit distribution $\lim_{\text{card}I \rightarrow \infty} \sum_{i \in I} \omega_i \delta_{\underline{X}_i} = \mu$.

Details : The method *generate* always gives the same numerical sample, the *partNumSamp*, even if it is recalled. It enables to take into account a random sample which has been obtained outside the Open TURNS study or at another step of the Open TURNS study.

The method *setDistribution* has no side effect, as the distribution is fixed by the initial sample.

Links

3.5.2 LowDiscrepancyExperiment

This class inherits from the WeightedExperiment class.

Usage :

LowDiscrepancyExperiment(sequence, size)

LowDiscrepancyExperiment(sequence, size, measure)

Arguments :

sequence : a LowDiscrepancySequence wich is a sequence of points $(\underline{u}_1, \dots, \underline{u}_N)$ with low discrepancy (see different models of such sequences in Open TURNS).

size : the integer N , the number of points of the sequence.

measure : the Distribution μ of (1) with independent copula and dimension n . The low discrepancy sequence (x_1, \dots, x_N) is uniforly distributed over $[0, 1]^n$. We use the marginal transformation $\xi_i = F_i^{-1}(u_i)$ to generate points $(\xi_i)_{i \in I}$ according to the distribution μ . The weights are all equal to $\frac{1}{N}$.

Value : a LowDiscrepancyExperiment.

Details : The method *generate* generates a NumericalSample of points $(\xi_i)_{i \in I}$ and which points are independently generated from *distribution*. When the method *generate* is recalled, the sample generated changes.

Links

3.5.3 GaussProductExperiment

This class inherits from the WeightedExperiment class.

Usage :

GaussProductExperiment(*measure*, *marginalDegrees*)

Arguments :

sequence : a GaussProductExperiment which contains the Gauss quadrature points and their associated weights in dimension n .

marginalDegrees : a Indices that fixes the number of points N_i for each direction.

measure : the Distribution μ of (1) of dimension n with independent copula.

Value : a GaussProductExperiment which contains $cardI = \prod_{i=1}^n N_i$.

Details : If the number of points bfor each direction doesn't change, the methode *generate* always gives the same sample.

Links

3.6 Low Discrepancy Sequences

3.6.1 FaureSequence

Usage :

FaureSequence(*dimension*)

FaureSequence()

Arguments :

dimension : an integer which is the dimension of the points.

Value : a FaureSequence. When not mentioned, *dimension* = 1.

Details : The method *generate*(*size*) generates a NumericalSample composed by the *size* first points of the sequence.

Links [see Reference Guide - Low Discrepancy Sequences](#)

3.6.2 HaltonSequence

Usage :

HaltonSequence(dimension)

HaltonSequence()

Arguments :

dimension : an integer which is the dimension of the points.

Value : a HaltonSequence. When not mentioned, *dimension* = 1.

Details : The method *generate(size)* generates a NumericalSample composed by the *size* first points of the sequence.

Links [see Reference Guide - Low Discrepancy Sequences](#)

3.6.3 ReverseHaltonSequence

Usage :

ReverseHaltonSequence(dimension)

ReverseHaltonSequence()

Arguments :

dimension : an integer which is the dimension of the points.

Value : a ReverseHaltonSequence. When not mentioned, *dimension* = 1.

Details : The method *generate(size)* generates a NumericalSample composed by the *size* first points of the sequence.

Links [see Reference Guide - Low Discrepancy Sequences](#)

3.6.4 HaselgroveSequence

Usage :

HaselgroveSequence(base)

HaselgroveSequence(dimension)

HaselgroveSequence()

Arguments :

base : a NumericalPoint of positive real values linearly independent over the integer ring, i.e no linear combination with integer coefficients of these values can be zero excepted if all the coefficients are zero. The dimension of the sequence is given by the dimension of the base.

dimension : an integer which is the dimension of the points.

Value : a HaselgroveSequence. When not mentioned, *dimension* = 1.

Details : The method *generate(size)* generates a NumericalSample composed by the *size* first points of the sequence.

Links [see Reference Guide - Low Discrepancy Sequences](#)

3.6.5 SobolSequence

Usage :

SobolSequence(dimension)

SobolSequence()

Arguments :

dimension : an integer which is the dimension of the points.

Value : a SobolSequence. When not mentioned, *dimension* = 1.

Details : The method *generate(size)* generates a NumericalSample composed by the *size* first points of the sequence.

Links [see Reference Guide - Low Discrepancy Sequences](#)

4 Functions

4.1 DualLinearCombinationEvaluationImplementation

DualLinearCombinationEvaluationImplementation inherits from NumericalMathEvaluationImplementation.

Usage :

DualLinearCombinationEvaluationImplementation(scalarFctColl, vectCoefColl)

Arguments :

scalarFctColl : a NumericalMathFunctionCollection, collection of several scalar NumericalMathFunction.

vectCoefColl : a NumericalSample.

Value : a NumericalMathFunction which is the linear combination *vectLinComb* of the scalar functions defined in *scalarFctColl* with vectorial weights defined in *vectCoefColl*.

If

$$scalarFctColl = (f_1, \dots, f_N)$$

where

$$\forall 1 \leq i \leq N, f_i : \mathbb{R}^n \longrightarrow \mathbb{R}$$

and

$$vectCoefColl = (\underline{c}_1, \dots, \underline{c}_N)$$

where

$$\forall 1 \leq i \leq N, \underline{c}_i \in \mathbb{R}^P$$

then

$$vectLinComb : \begin{cases} \mathbb{R}^n & \longrightarrow & \mathbb{R}^P \\ \underline{X} & \longrightarrow & \sum_{i=1}^N \underline{c}_i f_i(\underline{X}) \end{cases}$$

Some methods :

getCoefficients

Usage : *getCoefficients()*

Arguments : none

Value : a NumericalSample, the vectorial coefficients $(\underline{c}_1, \dots, \underline{c}_N)$ which define the linear combination *vectLinComb*.

getFunctionsCollection

Usage : *getFunctionsCollection()*

Arguments : none

Value : a NumericalMathFunctionCollection, the collection of scalar functions (f_1, \dots, f_N) which defines the linear combination *vectLinComb*.

setFunctionsCollectionAndCoefficients

Usage :

setFunctionsCollectionAndCoefficients(*scalarFctColl*, *vectCoefColl*)

Arguments :

scalarFctColl : a NumericalMathFunctionCollection, collection of several scalar NumericalMathFunction.

vectCoefColl : a NumericalSample.

Value : a NumericalMathFunction *vectLinComb* defined from *scalarFctColl* and *vectCoefColl* as mentioned above.

4.2 DualLinearCombinationGradientImplementation

DualLinearCombinationGradientImplementation inherits from NumericalMathGradientImplementation.

Usage :

DualLinearCombinationGradientImplementation(myDLCEI)

Arguments :

myDLCEI : a DualLinearCombinationEvaluationImplementation.

Value : a DualLinearCombinationGradientImplementation which the implementation of the gradient. By default, the analytical gradient is implemented.

4.3 DualLinearCombinationHessianImplementation

DualLinearCombinationHessianImplementation inherits from NumericalMathHessianImplementation.

Usage :

DualLinearCombinationHessianImplementation(myDLCEI)

Arguments :

myDLCEI : a DualLinearCombinationEvaluationImplementation.

Value : a DualLinearCombinationHessianImplementation which the implementation of the gradient. By default, the analytical gradient is implemented.

4.4 NumericalMathFunction

Usage :

1. *NumericalMathFunction(fileName)*
2. *NumericalMathFunction(modelPython)*
3. *NumericalMathFunction(input, output, formula)*
4. *NumericalMathFunction(inputString, outputString, formulaString)*
5. *NumericalMathFunction(f, g)*
6. *NumericalMathFunction(functionCollection)*
7. *NumericalMathFunction(functionCollection, scalarCoefficientColl)*
8. *NumericalMathFunction(scalarFunctionCollection, vectorCoefficientColl)*
9. *NumericalMathFunction(function, comparisonOperator, threshold)*

Arguments :

fileName : a string to name the XML file (without the extension ".xml") which contains the implementation of the considered function

modelPython : a NumericalMathFunctionImplementation, the implementation in the script python of a function. This function *modelPython* must derive from the class *OpenTURNPythonFunction* (see the Use Cases Guide to have an example of implementation) of the considered function

input : a Description which describes the input of the NumericalMathFuction

output : a Description which describes the output of the NumericalMathFuction

formula : a Description, the analytical formula of the NumericalMathFuction

inputString : a String which describes the input of the NumericalMathFuction

outputString : a String which describes the output of the NumericalMathFuction

formulaString : a String, the analytical formula of the NumericalMathFuction

f, g : two NumericalMathFunction

functionCollection : a NumericalMathFunctionCollection, collection of several NumericalMathFunction

scalarFunctionCollection : a NumericalMathFunctionCollection, collection of several scalar NumericalMathFunction

scalarCoefficientColl : a NumericalPoint,

vectorCoefficientColl : a NumericalSample,

function : a NumericalMathFunction

comparisonOperator : a ComparisonOperator

threshold : a NumericalScalar

Value :

usage 1 : a NumericalMathFunction which is implemented in the wrapper file 'fileName',

usage 2 : a NumericalMathFunction which is implemented in a python class *OpenTURNPythonFunction*,

usage 3 : a NumericalMathFunction which is defined by $output = fomula(input)$

usage 4 : a NumericalMathFunction which is defined by $outputString = fomulaString(inputString)$

usage 5 : a NumericalMathFunction which is the composition function $f \circ g$

usage 6 : a NumericalMathFunction which is the agregated function $agregFct$ defined as follows :
if

$$functionCollection = (f_1, \dots, f_N)$$

where

$$f_i : \mathbb{R}^n \longrightarrow \mathbb{R}^{p_i}$$

then the agregated function is :

$$agregFct : \begin{cases} \mathbb{R}^n & \longrightarrow \mathbb{R}^p \\ \underline{X} & \longrightarrow (f_1(\underline{X}), \dots, f_N(\underline{X}))^t \end{cases}$$

with

$$p = \sum_i p_i$$

usage 7 : a NumericalMathFunction which is the linear combination $linComb$ of the functions defined in $functionCollection$ with scalar weights defined in $scalarCoefficientColl$. If

$$functionCollection = (f_1, \dots, f_N)$$

where

$$\forall 1 \leq i \leq N, f_i : \mathbb{R}^n \longrightarrow \mathbb{R}^p$$

and

$$scalarCoefficientColl = (c_1, \dots, c_N) \in \mathbb{R}^N$$

then the linear combination is :

$$linComb : \begin{cases} \mathbb{R}^n & \longrightarrow \mathbb{R}^p \\ \underline{X} & \longrightarrow \sum_i c_i f_i(\underline{X}) \end{cases}$$

usage 8 : a NumericalMathFunction which is the linear combination $vectLinComb$ of the scalar functions defined in $scalarFunctionCollection$ with vectorial weights defined in $vectorCoefficientColl$.

If

$$scalarFunctionCollection = (f_1, \dots, f_N)$$

where

$$\forall 1 \leq i \leq N, f_i : \mathbb{R}^n \longrightarrow \mathbb{R}$$

and

$$vectorCoefficientColl = (\underline{c}_1, \dots, \underline{c}_N)$$

where

$$\forall 1 \leq i \leq N, \underline{c}_i \in \mathbb{R}^p$$

then

$$vectLinComb : \begin{cases} \mathbb{R}^n & \longrightarrow \mathbb{R}^p \\ \underline{X} & \longrightarrow \sum_i \underline{c}_i f_i(\underline{X}) \end{cases}$$

usage 9 : a NumericalMathFunction which is the indicator function *indFactor* of the event defined by *function* , *comparisonOperator* and *threshold*. For example, if *comparisonOperator* is $>$, then

$$indFactor = 1_{\{function > threshold\}}$$

Some methods :

()

Usage :

NumericalMathFunction(*x*)
NumericalMathFunction(*sample*)

Arguments :

x : a NumericalPoint
sample : a NumericalSample

Value :

while using the first usage, a NumericalPoint, the NumericalMathFunction value at point *x*
 while using the second usage, a NumericalSample, the NumericalMathFunction value on the sample *sample*

getDescription

Usage : *getDescription*()

Arguments : none

Value : a Description which describes the inputs and the outputs of the NumericalMathFunction
 (use `print NumericalMathFunction.getDescription()` command to visualize it)

getEvaluationCallsNumber

Usage : *getEvaluationCallsNumber*()

Arguments : none

Value : an integer that counts the number of times the NumericalMathFunction has been called since its creation

getInputVariablesName

Usage : *getInputVariablesName*()

Arguments : none

Value : a Description, the description of the input variables

getFormula

Usage : *getFormula*()

Arguments : none

Value : a String, the formula between the input and the output in the case where the function is :
 $\mathbb{R} \rightarrow \mathbb{R}$

getFormulas

Usage : *getFormulas()*

Arguments : none

Value : a Description, the description of the formulas between the inputs and the outputs

getGradientCallsNumber

Usage : *getGradientCallsNumber()*

Arguments : none

Value : an integer that counts the number of times the gradient of the NumericalMathFunction has been called since its creation. Note that if the gradient is implemented by a finite difference method, the gradient calls numbers is equal to 0 and the different calls are counted in the evaluation calls number

getGradientImplementation

Usage : *getGradientImplementation()*

Arguments : none

Value : a NumericalMathGradientImplementation, the gradient function

getHessianCallsNumber

Usage : *getHessianCallsNumber()*

Arguments : none

Value : an integer that counts the number of times the gradient of the NumericalMathFunction has been called since its creation. Note that if the hessian is implemented by a finite difference method, the hessian calls numbers is equal to 0 and the different calls are counted in the evaluation calls number

getHessianImplementation

Usage : *getHessianImplementation()*

Arguments : none

Value : a NumericalMathHessianImplementation, the hessian function

getInputDescription

Usage : *getInputDescription()*

Arguments : none

Value : a Description which describes the inputs of the NumericalMathFunction

getInputNumericalPointDimension or *getInputDimension*

Usage : *getInputNumericalPointDimension()*, *getInputDimension()*

Arguments : none

Value : an integer, the dimension of the input space

getEvaluationImplementation

Usage : *getEvaluationImplementation()*

Arguments : none

Value : a NumericalMathEvaluationImplementation, the evaluation function

getMarginal

Usage :

getMarginal(i)

getMarginal(indices)

Arguments :

i : an integer corresponding to the marginal (Care : the numbering starts at 0)

indices : an Indices, the set of indices for which the marginal is extracted

Value : a NumericalMathFunction, corresponding to either f_i or $(f_i)_{i \in \text{indices}}$, with $f : \mathbb{R}^n \rightarrow \mathbb{R}^p$ and $f = (f_0, \dots, f_{p-1})$.

getOutputDescription

Usage : *getOutputDescription()*

Arguments : none

Value : a Description which describes the outputs of the NumericalMathFunction object

getOutputNumericalPointDimension or *getOutputDimension*

Usage : *getOutputNumericalPointDimension()*, *getOutputDimension()*

Arguments : none

Value : an integer, the dimension of the output space

getOutputHistory

Usage : *getOutputHistory()*

Arguments : none

Value : a *HistoryStrategy* which stores all the numerical points which are the values of the function associated to the InputStrategy.

getInputHistory

Usage : *getInputHistory()*

Arguments : none

Value : a *HistoryStrategy* which stores all the numerical points over which the function has been evaluated

disableHistory

Usage : *disableHistory()*

Arguments : None.

Value : None. Deactivate the history mechanism.

enableHistory

Usage : *enableHistory()*

Arguments : None.

Value : None. Activate the history mechanism.

isHistoryEnabled

Usage : *isHistoryEnabled()*

Arguments : None.

Value : a logical value. True if the history mechanism is activated. It is activated by default.

resetHistory

Usage : *resetHistory()*

Arguments : None.

Value : None. Clear the history.

disableCache

Usage : *disableCache()*

Arguments : None.

Value : None. Deactivate the history mechanism.

enableCache

Usage : *enableCache()*

Arguments : None.

Value : None. Activate the history mechanism.

isCacheEnabled

Usage : *isCacheEnabled()*

Arguments : None.

Value : a logical value. True if the history mechanism is activated. It is activated by default.

getCacheHits

Usage : *getCacheHits()*

Arguments : None.

Value : an integer that count the number of computatins savec by the cache mecanism.

getCacheInput

Usage : *getCacheInput()*

Arguments : None.

Value : a NumericalSample that gives all the numerical points stored ion the cache mecanism.

addCacheContent

Usage : *addCacheContent(inSample, outSample)*

Arguments : *inputSample, outputSample* : NumericalSample.

Value : none. It adds the previous samples in the cache mecanism.

getOutputVariablesName

Usage : *getOutputVariablesName()*

Arguments : none

Value : a Description, the description of the output variables

getParameters

Usage : *getParameters()*

Arguments : none

Value : a NumericalPoint, the NumericalPoint corresponding to parameters of the NumericalMathFunction

GetValidOperators

Usage : *GetValidOperators()*

Arguments : none

Value : a Description, containing the list of the operators we can use within Open TURNS

GetValidFunctions

Usage : *GetValidFunctions()*

Arguments : none

Value : a Description, containing the list of the functions we can use within Open TURNS

GetValidConstants

Usage : *GetValidConstants()*

Arguments : none

Value : a Description, containing the list of the constants we can use within Open TURNS

gradient

Usage : *gradient(x)*

Arguments : *x* : a NumericalPoint (which has the same dimension as the inputs)

Value : a Matrix, the gradient (with respect to the inputs) of the NumericalMathFunction

hessian

Usage : *hessian(x)*

Arguments : *x* : a NumericalPoint (which has the same dimension as the inputs)

Value : a SymmetricTensor, the hessian (with respect to the inputs) of the NumericalMathFunction

setName

Usage : *setName(name)*

Arguments : *name* : a string (between quotations marks)

Value : it gives a name to the NumericalMathFunction

Here is the list of constants proposed by Open TURNS :

- $_e$: Euler's constant (2.71828...),
- $_pi$: Pi constant (3.14159...)

Here is the list of functions proposed by Open TURNS :

- $sin(arg)$: sine function,
- $cos(arg)$: cosine function,
- $tan(arg)$: tangent function,
- $asin(arg)$: inverse sine function,
- $acos(arg)$: inverse cosine function,
- $atan(arg)$: inverse tangent function,
- $sinh(arg)$: hyperbolic sine function,
- $cosh(arg)$: hyperbolic cosine function,
- $tanh(arg)$: hyperbolic tangens function,
- $asinh(arg)$: inverse hyperbolic sine function,
- $acosh(arg)$: inverse hyperbolic cosine function,
- $atanh(arg)$: inverse hyperbolic tangent function,
- $log2(arg)$: logarithm in base 2, $log10(arg)$: logarithm in base 10, $log(arg)$: logarithm in base e (2.71828...), $ln(arg)$: alias for log function,
- $lngamma(arg)$: log of the gamma function,
- $gamma(arg)$: gamma function,
- $exp(arg)$: exponential function,
- $erf(arg)$: error function,
- $erfc(arg)$: complementary error function,
- $sqrt(arg)$: square root function,
- $cbrt(arg)$: cubic root function,
- $besselJ0(arg)$: 1st kind Bessel function with parameter 0,
- $besselJ1(arg)$: 1st kind Bessel function with parameter 1,
- $besselY0(arg)$: 2nd kind Bessel function with parameter 0,
- $besselY1(arg)$: 2nd kind Bessel function with parameter 1,
- $sign(arg)$: sign function -1 if $x < 0$; 1 if $x > 0$,

- $\text{rint}(arg)$: round to nearest integer function,
- $\text{abs}(arg)$: absolute value function,
- $\text{if}(arg1, arg2, arg3)$: if $arg1$ then $arg2$ else $arg3$,
- $\text{min}(arg1, \dots, argn)$: min of all arguments,
- $\text{max}(arg1, \dots, argn)$: max of all arguments,
- $\text{sum}(arg1, \dots, argn)$: sum of all arguments,
- $\text{avg}(arg1, \dots, argn)$: mean value of all arguments .

Here is the list of operators proposed by Open TURNS :

- $=$: assignement, can only be applied to variable names (priority -1),
- and : logical and (priority 1),
- or : logical or (priority 1),
- xor : logical xor (priority 1),
- \leq : less or equal (priority 2),
- \geq : greater or equal (priority 2),
- \neq : not equal (priority 2),
- $==$: equal (priority 2),
- $>$: greater than (priority 2),
- $<$: less than (priority 2),
- $+$: addition (priority 3),
- $-$: subtraction (priority 3),
- $*$: multiplication (priority 4),
- $/$: division (priority 4),
- \neg : logical negation (priority 4),
- not : alias for \neg (priority 4),
- $-$: sign change (priority 4),
- $^$: raise x to the power of y (priority 5).

4.5 NumericalMathEvaluationImplementation

Usage : this object is the result of the method *getEvaluationImplementation()* of a NumericalMathFunction.

Some methods :

getMarginal

Usage :

getMarginal(i)

getMarginal(indices)

Arguments :

i : an integer corresponding to the marginal (Care : the numbering starts at 0)

indices : a Indices, the set of indices for which the marginal is extracted

Value : a NumericalMathFunction, the restriction of the Evaluation function to its components functions which indices are *i* or in *indices*

4.6 NumericalMathGradientImplementation

Usage : this object is the result of the method *getGradientImplementation()* of a NumericalMathFunction.

Some methods :

getMarginal

Usage :

getMarginal(i)

getMarginal(indices)

Arguments :

i : an integer corresponding to the marginal (Care : the numbering starts at 0)

indices : a Indices, the set of indices for which the marginal is extracted

Value : a NumericalMathFunction, the restriction of the Gradient function to its components functions which indices are *i* or in *indices*

4.7 NumericalMathHessianImplementation

Usage : this object is the result of the method *getHessianImplementation()* of a NumericalMathFunction.

Some methods :

getMarginal

Usage :

getMarginal(i)

getMarginal(indices)

Arguments :

i : an integer corresponding to the marginal (Care : the numbering starts at 0)

indices : a Indices, the set of indices for which the marginal is extracted

Value : a NumericalMathFunction, the restriction of the Hessian function to its components functions which indices are *i* or in *indices*.

4.8 CenteredFiniteDifferenceMathGradientImplementation

CenteredFiniteDifferenceMathGradientImplementation inherits from NumericalMathGradientImplementation

Usage :

CenteredFiniteDifferenceMathGradientImplementation(epsilon, evalImpl)

CenteredFiniteDifferenceMathGradientImplementation(step, evalImpl)

Arguments :

evalImpl a NumericalMathFunctionEvaluationImplementation, the implementation of the evaluation of a function

epsilon : a NumericalScalar, the finite difference step

epsilon : a NumericalPoint, the finite difference steps for each dimension

step : a FiniteDifferenceStep, the object that defines how finite difference steps values are computed

Value :

a CenteredFiniteDifferenceMathGradientImplementation

4.9 NonCenteredFiniteDifferenceMathGradientImplementation

NonCenteredFiniteDifferenceMathGradientImplementation inherits from NumericalMathGradientImplementation

Usage :

NonCenteredFiniteDifferenceMathGradientImplementation(epsilon, evalImpl)

NonCenteredFiniteDifferenceMathGradientImplementation(step, evalImpl)

Arguments :

evalImpl a NumericalMathFunctionEvaluationImplementation, the implementation of the evaluation of a function

epsilon : a NumericalScalar, the finite difference step

epsilon : a NumericalPoint, the finite difference steps for each dimension

step : a FiniteDifferenceStep, the object that defines how finite difference steps values are computed

Value :

a NonCenteredFiniteDifferenceMathGradientImplementation

4.10 CenteredFiniteDifferenceMathHessianImplementation

CenteredFiniteDifferenceMathHessianImplementation inherits from NumericalMathHessianImplementation

Usage :

CenteredFiniteDifferenceMathHessianImplementation(epsilon, evalImpl)

CenteredFiniteDifferenceMathHessianImplementation(step, evalImpl)

Arguments :

evalImpl a NumericalMathFunctionEvaluationImplementation, the implementation of the evaluation of a function

epsilon : a NumericalScalar, the finite difference step

epsilon : a NumericalPoint, the finite difference steps for each dimension

step : a FiniteDifferenceStep, the object that defines how finite difference steps values are computed

Value :

a CenteredFiniteDifferenceMathHessianImplementation

4.11 FiniteDifferenceStep

Usage :

FiniteDifferenceStep(finiteDifferenceStepImplementation)

Arguments :

finiteDifferenceStepImplementation a FiniteDifferenceStepImplementation, the implementation of the finite difference step

Value :

a FiniteDifferenceStep

Details : Defines how finite difference steps are computed

4.12 ConstantStep

ConstantStep inherits from FiniteDifferenceStepImplementation

Usage :

ConstantStep(epsilon)

Arguments :

epsilon a NumericalPoint, the finite difference steps for each dimension

Value :

a ConstantStep

Details : The finite difference step is constant and equal to epsilon

4.13 BlendedStep

BlendedStep inherits from FiniteDifferenceStepImplementation

Usage :

BlendedStep(epsilon)

BlendedStep(epsilon, eta)

BlendedStep(epsilon, eta)

Arguments :

epsilon a NumericalPoint, the finite difference step factor for each dimension

eta a NumericalScalar, the finite difference step offset for every dimension, must be positive

eta a NumericalPoint, the finite difference step offset for each dimension, must be positive

Value :

a BlendedStep,

Details : The finite difference step is $epsilon \cdot (|x| + eta)$

4.14 MarginalTransformationEvaluation

Usage :

MarginalTransformationEvaluation(distCol, direction)

MarginalTransformationEvaluation(distCol)

MarginalTransformationEvaluation(distCol, outputDistCol)

Arguments :

distCol : a DistributionCollection,

direction : an integer or *MarginalTransformationEvaluation.FROM* (associated to the integer 0) or *MarginalTransformationEvaluation.TO* (associated to the integer 1). If not specified, *direction* = 1.

outputDistCol : a DistributionCollection

Value : NumericalMathEvaluationImplementation : this class contains NumericalMathFunction which can be evaluated in one points but which proposes no gradient nor hessian implementation.

In the first usage, if *direction* = *MarginalTransformationEvaluation.FROM*, the created operator transforms a NumericalPoint into its rank according to the marginal distributions described in *distributionCollection* : if we note $(F_{X_1}, \dots, F_{X_n})$ the CDF of the distributions contained in *distributionCollection*, then the created operator works as follows :

$$(x_1, \dots, x_n) \longrightarrow (F_{X_1}(x_1), \dots, F_{X_n}(x_n))$$

If *direction* = *MarginalTransformationEvaluation.TO*, the created operator works in the opposite direction :

$$(x_1, \dots, x_n) \longrightarrow (F_{X_1}^{-1}(x_1), \dots, F_{X_n}^{-1}(x_n))$$

In that case, it requires that all the values x_i be in $[0, 1]$.

In the second usage, the created operator transforms a NumericalPoint into the following one, where *outputDistributionCollection* contains the $(F_{Y_1}, \dots, F_{Y_n})$ distributions:

$$(x_1, \dots, x_n) \longrightarrow (F_{Y_1}^{-1} \circ F_{X_1}(x_1), \dots, F_{Y_n}^{-1} \circ F_{X_n}(x_n))$$

Some methods :

()

Usage : *MarginalTransformationEvaluation(point)*

Arguments : *point* : a NumericalPoint

Value : a NumericalPoint

4.15 SpatialFunction

SpatialFunction inherits from DynamicalFunctionImplementation.

Usage :

SpatialFunction(function)
SpatialFunction(evaluation)

Arguments :

function : a NumericalMathFunction, the function that act on the spatial part of time series.

evaluation : a NumericalMathEvaluationImplementation, the evaluation part of a NumericalMathFunction that act on the spatial part of time series.

Value : a SpatialFunction which acts on time series only on their spatial part. If f is the spatial function, with $f : \mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R} \times \mathbb{R}^p$, and g is the associated evaluation $g : \mathbb{R}^n \rightarrow \mathbb{R}^p$, we have:

$$\forall (t, \underline{x}) \in \mathbb{R} \times \mathbb{R}^n, \quad f(t, \underline{x}) = (t, g(\underline{x}))$$

The time grid of the resulting time series is the same as the time grid of the input time series.

Some methods :

getEvaluation

Usage : *getEvaluation()*

Arguments : none

Value : a NumericalMathEvaluationImplementation, the evaluation g describing the action of the SpatialFunction on time series

4.16 TemporalFunction

TemporalFunction inherits from DynamicalFunctionImplementation.

Usage :

TemporalFunction(function)
TemporalFunction(evaluation)

Arguments :

function : a NumericalMathFunction, the function that act on the couples (t, \underline{x}) of a time series.

evaluation : a NumericalMathEvaluationImplementation, the evaluation part of a NumericalMathFunction that act on the couples (t, \underline{x}) of time series.

Value : a TemporalFunction which acts pointwise on time series. If f is the spatial function, with $f : \mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R} \times \mathbb{R}^p$, and g is the associated evaluation $g : \mathbb{R}^{n+1} \rightarrow \mathbb{R}^p$, we have:

$$\forall (t, \underline{x}) \in \mathbb{R} \times \mathbb{R}^n, \quad f(t, \underline{x}) = (t, g(\underline{x}'))$$

where $\underline{x}' = (t, \underline{x}^t)^t$. The time grid of the resulting time series is the same as the time grid of the input time series.

Some methods :

getEvaluation

Usage : *getEvaluation()*

Arguments : none

Value : a NumericalMathEvaluationImplementation, the evaluation g describing the action of the TemporalFunction on time series

4.17 DynamicalFunction

Usage :

1. *NumericalMathFunction(implementation)*

Arguments :

implementation : a DynamicalFunctionImplementation object, which can be a TemporalFunction or a SpatialFunction for example.

Value : a DynamicalFunction object, it means a function that acts globally on time series to produce time series. The resulting time series can have a time grid different from the time grid of the input time series.

Some methods :

()

Usage :

DynamicalFunction(timeSeries)

DynamicalFunction(processSample)

Arguments :

timeSeries : a TimeSeries

processSample : a ProcessSample

Value :

while using the first usage, a TimeSeries, the value of the DynamicalFunction on the input time series *timeSeries*

while using the second usage, a ProcessSample, the value of the DynamicalFunction on the input process sample *processSample*

getInputDescription

Usage : *getInputDescription()*

Arguments : none

Value : a Description which describes the inputs of the DynamicalFunction

getInputDimension

Usage : *getInputDimension()*

Arguments : none

Value : an integer, the input dimension *n* of the dynamical function.

getOutputDescription

Usage : *getOutputDescription()*

Arguments : none

Value : a Description which describes the outputs of the DynamicalFunction

getOutputDimension

Usage : *getOutputDimension()*

Arguments : none

Value : an integer, the output dimension p of the dynamical function.

getCallsNumber

Usage : *getCallsNumber()*

Arguments : none

Value : an integer that counts the number of times the DynamicalFunction has been called since its creation

getMarginal

Usage :

getMarginal(i)

getMarginal(indices)

Arguments :

i : an integer corresponding to the marginal (Care : the numbering starts at 0)

indices : an Indices, the set of indices for which the marginal is extracted

Value : a DynamicalFunction, corresponding to either f_i or $(f_i)_{i \in \text{indices}}$, with $f : \mathbb{R} \times \mathbb{R}^n \longrightarrow \mathbb{R} \times \mathbb{R}^p$ and $f = (f_0, \dots, f_{p-1})$.

5 FFT

The objective of this class is to implement the Fourier transformations (direct and inverse) using external tools such as KissFFT. We describe here the methods of this class.

Usage :

FFT()

Value : FFT

This instantiates the Fast Fourier Transform (FFT) class

Some methods :

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the FFT

inverseTransform

Usage : *inverseTransform(collection)*

Arguments : NumericalComplexCollection

Value : a NumericalComplexCollection. This computes the Fourier inverse transformation of the values contained in collection.

setName

Usage : *setName(name)*

Arguments : name : a string

Value : the FFT is named *name*

transform

Usage : *transform(collection)*

Arguments : NumericalComplexCollection

Value : a NumericalComplexCollection. This computes the Fourier transformation of the values contained in collection. Notice also that the collection might be a NumericalScalarCollection

5.0.1 KissFFT

The KissFFT class inherits from the FFT class. The methods are the same as the FFT class (there is no additional method). This class interacts with the *kissfft* implemented and return results as OpenTurns objects (NumericalComplexCollection).

6 Graphs

6.1 Graph

The class Graph is the structure which contains :

- the drawable elements (may be several drawables elements) : class Drawable
- the graphical context : the potential axes and labels, the bounding box, the global title, the global legend and its position

Usage :

Graph(title, xTitle, yTitle, showAxes)

Graph(title, xTitle, yTitle, showAxes, legendPosition)

Graph(title, xTitle, yTitle, showAxes, legendPosition, legendFontSize, logScale)

Arguments :

title : a String, the title of the graph

xTitle : a String, the legend of the X axe

yTitle : a String, the legend of the Y axe

showAxes : a boolean which indicates if the axes are drawn (yes = 1, no = 0)

legendPosition : a String which indicates the position of the legend. If *legendPosition* is not specified, the Graph has no legend.

legendFontSize : an interger, the font size of the legend. If not specified, the default width will be used.

logScale : a LogScale that indicates whether the logarithmic scale is used either for one or both axes :

- *GraphImplementation.NONE* : no log scale is used,
- *GraphImplementation.LOGX* : log scale is used only for horizontal data,
- *GraphImplementation.LOGY* : log scale is used only for vertical data,
- *GraphImplementation.LOGXY* : log scale is used for both data.

Some methods :

add

Usage : *add(aDrawable)*

Arguments : *aDrawable* : a Drawable, a drawable element we want to add on the graph

Value : none, it adds the new graph on the first one, with its legend. It keeps the graphical context of the first graph

addDrawables

Usage : *addDrawable(aDrawableCollection)*

Arguments : *aDrawableCollection* : a DrawableCollection, a collection of drawables we want to add on the graph.

Value : none, it adds the drawable collection to the graph. It keeps the graphical context of the first graph.

draw

Usage :

draw(path, file, width, height)
draw(path, file, width, height, format)
draw(file, width, height)
draw(file, width, height, format)
draw(file)

Arguments :

path : a String which indicates the adress where the created file will be put. When not specified, the files is created in the current repertory.

file : a String which indicates the name of the created file (without the suffixe). The files created will be *file.png* and *file.ps*.

width, height : two real positive values, number of pixels fixing the width and the height of the graph. When not specified, the couple (640,480) is taken into account.

format = *GraphImplementation.EPS*, *GraphImplementation.PNG*, *GraphImplementation.FIG* or *GraphImplementation.PDF*. When not precised, by default, *format* = *ALL*.

Value : It generates the files *file.png*, *file.eps*, *file.fig* and *file.pdf* or only the format specified in *format*.

getAxes

Usage : *getAxes()*

Arguments : none

Value : a boolean which indicates if the axes are drawn (yes = 1, no = 0)

getBitmap

Usage : *getBitmap()*

Arguments : none

Value : a String, the adress of the file *file.png* created by the method *draw*

getBoundingBox

Usage : *getBoundingBox()*

Arguments : none

Value : a *NumericalPoint* of dimension 4, the bounding box of the drawable element, which is a rectangle determined by its range along X and its range along Y. The *BoundingBox* is $(x_{min}, x_{max}, y_{min}, y_{max})$.

getDrawables

Usage : *getDrawables()*

Arguments : none

Value : a *DrawableCollection*, the collection of the *Drawables* included in the graph

getFileName

Usage : *getFileName()*

Arguments : none

Value : a String, the name of the files containing the graph

getLegendFontSize

Usage : *getLegendFontSize()*

Arguments : none

Value : a positive real, the legend font size

getLegendPosition

Usage : *getLegendPosition()*

Arguments : none

Value : a String, the position of the legend.

getPath

Usage : *getPath()*

Arguments : none

Value : a String, the address where the files file.png and file.ps are put

getPostscript

Usage : *getPostscript()*

Arguments : none

Value : a String, the adress of the file file.ps created by the method .draw()

getTitle

Usage : *getTitle()*

Arguments : none

Value : a String, the title of the graph

getXTitle

Usage : *getXTitle()*

Arguments : none

Value : a String, the title of the X axe

getYTitle

Usage : *getYTitle()*

Arguments : none

Value : a String, the title of the Y axe

setBoundingBox

Usage :

setBoundingBox(myBoundingBox)

setBoundingBox(myInterval)

Arguments :

myBoundingBox : a BoundingBox, which is a Numericalpoint(4) composed by $[x_{min}, x_{max}, y_{min}, y_{max}]$ if we want to impose the x-range to $[x_{min}, x_{max}]$ and the y-range to $[y_{min}, y_{max}]$.

myInterval : a Interval, which is created as follows *myInterval*(BottomLeftPoint, UpRightPoint) where BottomLeftPoint is a NumericalPoint(2) representing the corner (x_{min}, y_{min}) and UpRightPoint the corner (x_{max}, y_{max}) .

Value : none

The methods *getAxes*, *getDrawables*, *getLegendPosition*, *getTitle*, *getXTitle*, *getYTitle* have their corresponding *setMethod*.

Here is the list of legend positions accepted by Open TURNS : "bottomright", "bottom", "bottomleft", "left", "topleft", "topright", "right", "center".

6.2 Drawable

A Drawable is a drawable element described by :

- its data,
- their attributes : color, line style, point style, fill style
- the specific legend of the drawable element.

Usage : *Drawable(drawableImplementation)*

Arguments : *drawableImplementation* : a DrawableImplementation, the implementation of Drawable, which is *Curve*, *Cloud*, *BarPlot*, *Staircase*, *Pie* or *Contour*

Some methods :

getBoundingBox

Usage : *getBoundingBox()*

Arguments : none

Value : a NumericalPoint of dimension 4, the bounding box of the drawable element, which is a rectangle determined by its range along X and its range along Y. The BoundingBox is $(x_{min}, x_{max}, y_{min}, y_{max})$.

getColor

Usage : *getColor()*

Arguments : none

Value : a String which describes the color of the lines within the drawable element. It can be either the name of a color (e.g. "red") or an hexadecimal code corresponding to the RGB (Red, Green, Blue) components of the color (e.g. #A1B2C3) or the RGBA (Red, Green, Blue, Alpha) components of the color (e.g. #A1B2C3D4). The alpha channel is taken into account only by the PDF and PNG formats, for the other format the color is fully transparent as soon as its alpha channel is less than 255 (or 1.0).

getData

Usage : *getData()*

Arguments : none

Value : a NumericalSample, from which the Drawable is built

getFillStyle

Usage : *getFillStyle()*

Arguments : none

Value : a String which describes the fill style of the surfaces within the drawable element

getLabels

Usage : *getLabels()*

Arguments : none

Value : a Description, the labels of both axes

getLegendName

Usage : *getLegendName()*

Arguments : none

Value : a String which is the legend of the drawable element

getLineStyle

Usage : *getLineStyle()*

Arguments : none

Value : a String which describes the style of the lines within the drawable element

getLineWidth

Usage : *getLineWidth()*

Arguments : none

Value : an integer, the width of the line included in the Drawable (if such the case)

getPointCode

Usage : *getPointCode()*

Arguments : none

Value : an integer which describes the style of the points within the drawable element

getPointStyle

Usage : *getPointStyle()*

Arguments : none

Value : a string which describes the style of the points within the drawable element

ConvertFromRGB

Usage : *ConvertFromRGB(red, green, blue)*

Arguments : *red*, *green* and *blue* are either three nonnegative integer (UnsignedLong) values or three nonnegative real (NumericalScalar) values (which will be scaled and converted to integer values). These values are the red, green and blue components of a color, a value of 0 (or 0.0) meaning that the component is absent in the color, a value of 255 (or 1.0) meaning that the component is fully saturated.

Value : a String giving the associated hexadecimal color code.

ConvertFromRGBA

Usage : *ConvertFromRGBA(red, green, blue, alpha)*

Arguments : *red*, *green*, *blue* and *alpha* are either four nonnegative integer (UnsignedLong) values or four nonnegative real (NumericalScalar) values (which will be scaled and converted to integer values). These values are the red, green and blue components of a color, a value of 0 (or 0.0) meaning that the component is absent in the color, a value of 255 (or 1.0) meaning that the component is fully saturated. The parameter alpha gives the level of transparency of the color, 0 (or 0.0) meaning that the color is fully transparent and 255 (or 1.0) meaning that the color is fully opaque. The alpha channel is only supported by a few devices, namely the PDF and PNG formats, for the other format the color is fully transparent as soon as its alpha channel is less than 255 (or 1.0).

Value : a String giving the associated hexadecimal color code.

GetValidColors

Usage : *GetValidColors()*

Arguments : none

Value : a Description which indicates the list of the valid colors of a Drawable

GetValidFillStyles

Usage : *GetValidFillStyles()*

Arguments : none

Value : a Description which indicates the list of the valid fill styles of a Drawable

GetValidLineStyle

Usage : *GetValidLineStyle()*

Arguments : none

Value : a Description which indicates the list of the valid line styles of a Drawable

GetValidPointStyles

Usage : *GetValidPointStyles()*

Arguments : none

Value : a Description which indicates the list of the valid point styles of a Drawable

All the methods *getColor*, *getFillStyle*, *getLineStyle*, *getPointCode* and *getPointStyle* have their corresponding *setMethod*.

Here is the list of codes, styles and width accepted by Open TURNS :

- map matching keys with R codes for point symbols :

Point Style	Point Code
square	0
circle	1
triangleup	2
plus	3
times	4
diamond	5
triangledown	6
star	8
fsquare	15
fcircle	16
ftriangleup	17
fdiamond	18
bullet	20
dot	127

- authorized colors :

- All the codes of the form #RRGGBB or #RRGGBBAA, where R, G, B, A are hexadecimal digits (0, ..., 9, A or a, ..., F or f). Examples: #03A21F, #3b2E43ff.
- All the names in the list: "green", "red", "blue", "yellow", "darkblue", "orange", "lightgreen", "darkcyan", "cyan", "magenta", "darkgreen", "violet", "brown", "darkred", "pink", "ivory", "gold", "darkgrey", "grey", "white", "aliceblue", "antiquewhite", "antiquewhite1", "antiquewhite2", "antiquewhite3", "antiquewhite4", "aquamarine", "aquamarine1", "aquamarine2", "aquamarine3", "aquamarine4", "azure", "azure1", "azure2", "azure3", "azure4", "beige", "bisque", "bisque1", "bisque2", "bisque3", "bisque4", "black", "blanchedalmond", "blue1", "blue2", "blue3", "blue4", "blueviolet", "brown1", "brown2", "brown3", "brown4", "burlywood", "burlywood1", "burlywood2", "burlywood3", "burlywood4", "cadetblue", "cadetblue1", "cadetblue2", "cadetblue3", "cadetblue4", "chartreuse", "chartreuse1", "chartreuse2", "chartreuse3", "chartreuse4", "chocolate", "chocolate1", "chocolate2", "chocolate3", "chocolate4", "coral", "coral1", "coral2", "coral3", "coral4", "cornflowerblue", "cornsilk", "cornsilk1", "cornsilk2", "cornsilk3", "cornsilk4", "cyan1", "cyan2", "cyan3", "cyan4", "darkgoldenrod", "darkgoldenrod1", "darkgoldenrod2", "darkgoldenrod3", "darkgoldenrod4", "darkgray", "darkkhaki", "darkmagenta", "darkolivegreen", "darkolivegreen1", "darkolivegreen2", "darkolivegreen3", "darkolivegreen4", "darkorange", "darkorange1", "darkorange2", "darkorange3", "darkorange4", "darkorchid", "darkorchid1", "darkorchid2", "darkorchid3", "darkorchid4", "darksalmon", "darkseagreen", "darkseagreen1", "darkseagreen2", "darkseagreen3", "darkseagreen4", "darkslateblue", "darkslategray", "darkslategray1", "darkslategray2", "darkslategray3", "darkslategray4", "darkslategrey", "darkturquoise", "darkviolet", "deeppink", "deeppink1", "deeppink2", "deeppink3", "deeppink4", "deepskyblue", "deepskyblue1", "deepskyblue2", "deepskyblue3", "deepskyblue4", "dimgray", "dimgrey", "dodgerblue", "dodgerblue1", "dodgerblue2", "dodgerblue3", "dodgerblue4", "firebrick", "firebrick1", "firebrick2", "firebrick3", "firebrick4", "floralwhite", "forestgreen", "gainsboro", "ghostwhite", "gold1", "gold2", "gold3", "gold4", "goldenrod", "goldenrod1", "goldenrod2", "goldenrod3", "goldenrod4", "gray", "gray0", "gray1", "gray2", "gray3", "gray4", "gray5", "gray6", "gray7", "gray8", "gray9", "gray10", "gray11", "gray12", "gray13", "gray14", "gray15", "gray16", "gray17", "gray18", "gray19", "gray20", "gray21", "gray22", "gray23", "gray24", "gray25", "gray26", "gray27", "gray28", "gray29", "gray30", "gray31", "gray32", "gray33", "gray34", "gray35", "gray36", "gray37", "gray38", "gray39", "gray40", "gray41", "gray42", "gray43", "gray44", "gray45", "gray46", "gray47", "gray48", "gray49", "gray50", "gray51", "gray52", "gray53", "gray54", "gray55", "gray56", "gray57",

"gray58", "gray59", "gray60", "gray61", "gray62", "gray63", "gray64", "gray65", "gray66", "gray67",
 "gray68", "gray69", "gray70", "gray71", "gray72", "gray73", "gray74", "gray75", "gray76", "gray77",
 "gray78", "gray79", "gray80", "gray81", "gray82", "gray83", "gray84", "gray85", "gray86", "gray87",
 "gray88", "gray89", "gray90", "gray91", "gray92", "gray93", "gray94", "gray95", "gray96", "gray97",
 "gray98", "gray99", "gray100", "green1", "green2", "green3", "green4", "greenyellow", "grey0",
 "grey1", "grey2", "grey3", "grey4", "grey5", "grey6", "grey7", "grey8", "grey9", "grey10", "grey11",
 "grey12", "grey13", "grey14", "grey15", "grey16", "grey17", "grey18", "grey19", "grey20", "grey21",
 "grey22", "grey23", "grey24", "grey25", "grey26", "grey27", "grey28", "grey29", "grey30", "grey31",
 "grey32", "grey33", "grey34", "grey35", "grey36", "grey37", "grey38", "grey39", "grey40", "grey41",
 "grey42", "grey43", "grey44", "grey45", "grey46", "grey47", "grey48", "grey49", "grey50", "grey51",
 "grey52", "grey53", "grey54", "grey55", "grey56", "grey57", "grey58", "grey59", "grey60", "grey61",
 "grey62", "grey63", "grey64", "grey65", "grey66", "grey67", "grey68", "grey69", "grey70", "grey71",
 "grey72", "grey73", "grey74", "grey75", "grey76", "grey77", "grey78", "grey79", "grey80", "grey81",
 "grey82", "grey83", "grey84", "grey85", "grey86", "grey87", "grey88", "grey89", "grey90", "grey91",
 "grey92", "grey93", "grey94", "grey95", "grey96", "grey97", "grey98", "grey99", "grey100", "honey-
 dew", "honeydew1", "honeydew2", "honeydew3", "honeydew4", "hotpink", "hotpink1", "hotpink2",
 "hotpink3", "hotpink4", "indianred", "indianred1", "indianred2", "indianred3", "indianred4", "ivory1",
 "ivory2", "ivory3", "ivory4", "khaki", "khaki1", "khaki2", "khaki3", "khaki4", "lavender", "laven-
 derblush", "lavenderblush1", "lavenderblush2", "lavenderblush3", "lavenderblush4", "lawngreen",
 "lemonchiffon", "lemonchiffon1", "lemonchiffon2", "lemonchiffon3", "lemonchiffon4", "lightblue",
 "lightblue1", "lightblue2", "lightblue3", "lightblue4", "lightcoral", "lightcyan", "lightcyan1", "light-
 cyan2", "lightcyan3", "lightcyan4", "lightgoldenrod", "lightgoldenrod1", "lightgoldenrod2", "light-
 goldenrod3", "lightgoldenrod4", "lightgoldenrodyellow", "lightgray", "lightgrey", "lightpink", "light-
 pink1", "lightpink2", "lightpink3", "lightpink4", "lightsalmon", "lightsalmon1", "lightsalmon2",
 "lightsalmon3", "lightsalmon4", "lightseagreen", "lightskyblue", "lightskyblue1", "lightskyblue2",
 "lightskyblue3", "lightskyblue4", "lightslateblue", "lightslategray", "lightslategrey", "lightsteelblue",
 "lightsteelblue1", "lightsteelblue2", "lightsteelblue3", "lightsteelblue4", "lightyellow", "lightyellow1",
 "lightyellow2", "lightyellow3", "lightyellow4", "limegreen", "linen", "magenta1", "magenta2", "ma-
 genta3", "magenta4", "maroon", "maroon1", "maroon2", "maroon3", "maroon4", "mediumaqua-
 marine", "mediumblue", "mediumorchid", "mediumorchid1", "mediumorchid2", "mediumorchid3",
 "mediumorchid4", "mediumpurple", "mediumpurple1", "mediumpurple2", "mediumpurple3", "medi-
 umpurple4", "mediumseagreen", "mediumslateblue", "mediumspringgreen", "mediumturquoise", "medi-
 umvioletred", "midnightblue", "mintcream", "mistyrose", "mistyrose1", "mistyrose2", "mistyrose3",
 "mistyrose4", "moccasin", "navajowhite", "navajowhite1", "navajowhite2", "navajowhite3", "nava-
 jowhite4", "navy", "navyblue", "oldlace", "olivedrab", "olivedrab1", "olivedrab2", "olivedrab3",
 "olivedrab4", "orange1", "orange2", "orange3", "orange4", "orangered", "orangered1", "orangered2",
 "orangered3", "orangered4", "orchid", "orchid1", "orchid2", "orchid3", "orchid4", "palegoldenrod",
 "palegreen", "palegreen1", "palegreen2", "palegreen3", "palegreen4", "paleturquoise", "paleturquoise1",
 "paleturquoise2", "paleturquoise3", "paleturquoise4", "palevioletred", "palevioletred1", "paleviolet-
 red2", "palevioletred3", "palevioletred4", "papayawhip", "peachpuff", "peachpuff1", "peachpuff2",
 "peachpuff3", "peachpuff4", "peru", "pink1", "pink2", "pink3", "pink4", "plum", "plum1", "plum2",
 "plum3", "plum4", "powderblue", "purple", "purple1", "purple2", "purple3", "purple4", "red1",
 "red2", "red3", "red4", "rosybrown", "rosybrown1", "rosybrown2", "rosybrown3", "rosybrown4",
 "royalblue", "royalblue1", "royalblue2", "royalblue3", "royalblue4", "saddlebrown", "salmon", "salmon1",
 "salmon2", "salmon3", "salmon4", "sandybrown", "seagreen", "seagreen1", "seagreen2", "seagreen3",
 "seagreen4", "seashell", "seashell1", "seashell2", "seashell3", "seashell4", "sienna", "sienna1", "si-
 enna2", "sienna3", "sienna4", "skyblue", "skyblue1", "skyblue2", "skyblue3", "skyblue4", "slate-
 blue", "slateblue1", "slateblue2", "slateblue3", "slateblue4", "slategray", "slategray1", "slategray2",

"slategray3", "slategray4", "slategrey", "snow", "snow1", "snow2", "snow3", "snow4", "springgreen", "springgreen1", "springgreen2", "springgreen3", "springgreen4", "steelblue", "steelblue1", "steelblue2", "steelblue3", "steelblue4", "tan", "tan1", "tan2", "tan3", "tan4", "thistle", "thistle1", "thistle2", "thistle3", "thistle4", "tomato", "tomato1", "tomato2", "tomato3", "tomato4", "turquoise", "turquoise1", "turquoise2", "turquoise3", "turquoise4", "violetred", "violetred1", "violetred2", "violetred3", "violetred4", "wheat", "wheat1", "wheat2", "wheat3", "wheat4", "whitesmoke", "yellow1", "yellow2", "yellow3", "yellow4", "yellowgreen"

- authorized line styles : "blank", "solid", "dashed", "dotted", "dotdash", "longdash", "twodash"
- authorized fill styles : "solid", "shaded"

The default values are the following ones :

- *Color* = "blue"
- *SurfaceColor* = "white"
- *FillStyle* = "solid"
- *PointStyle* = "plus"
- *LineWidth* = 1
- *LineStyle* = "solid"
- *Pattern* = "s"

6.3 BarPlot

It inherits from the methods of the Drawable class.

Usage :

BarPlot(data, origin, legend)

BarPlot(data, origin, color, fillStyle, lineStyle, legend)

BarPlot(data, origin, color, fillStyle, lineWidth, lineStyle, legend)

Arguments :

data : a NumericalSample, the data from which the BarPlot is built, must be of dimension 2 : the discontinuous points and their corresponding height

origin : a real value which is where the BarPlot begins

legend : a String, the legend

color : a String, the color of the curve . If not specified, by default equal to "blue"

lineStyle : a String, the style of the curve. If not specified, by default equal to "solid"

lineWidth : an integer, the width of the curve. If not specified, by default equal to 1

fillStyle : a String, the fill style of the surfaces. If not specified, by default equal to "solid"

Some methods :

getData

Usage : *getData()*

Arguments : none

Value : a NumericalSample, of dimension 2, giving the discontinuous points and their corresponding height

getOrigin

Usage : *getOrigin()*

Arguments : none

Value : a real value which is where the BarPlot begins

isConformData

Usage : *isConformData(data)*

Arguments : *data* : a NumericalSample

Value : a boolean which indicates if the type of data is conform to the type of the drawable (here a BarPlot) : a NumericalSample, of dimension 2

All the methods *getColor*, *getLegendName*, *getOrigin*, *getFillStyle* and *getLineStyle* have their corresponding *setMethod*.

6.4 Cloud

It inherits from the methods of the Drawable class.

Usage :

Cloud(data, legend)

Cloud(data, color, pointStyle, legend)

Cloud(dataComplex)

Arguments :

data : a NumericalSample, the points from which the cloud is built, must be of dimension 2

legend : a String, the legend

color : a String, the color of the curve . If not specified, by default equal to "blue"

pointStyle : a String, the style of the points. If not specified, by default equal to "plus"

dataComplex : a NumericalComplexCollection, collection of complex points

Some methods :

isConformData

Usage : *isConformData(data)*

Arguments : *data* : a NumericalSample

Value : a boolean which indicates if the type of data is conform to the type of the drawable : a NumericalSample of dimension 2

getData

Usage : *getData()*

Arguments : none

Value : a NumericalSample of dimension 2, the data from which the cloud is built

All the methods *getColor*, *getLegendName*, *getPointCode* and *getPointStyle* have their corresponding *setMethod*.

6.5 Contour

It inherits from the methods of the Drawable class.

Usage :

Contour(dimX, dimY, data)

Contour(dimX, dimY, data, legend)

Contour(sampleX, sampleY, sampleValues, levels, labels)

Arguments :

dimX : an integer,

dimY : an integer,

data : a NumericalSample, of dimension 1 and of size $dimX * dimY$. These values are those of a function $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ on each point of the grid with $dimX$ points along the X -direction and $dimY$ points along the Y -direction. The (X, Y) -values are stocked by lign.

legend : a string which gives the legend.

levels : a NumericalPoint, the levels where the contour will be drawn. If 2 points of the grid have values bracketing the *level*, a linear interpolation is made in order to find the point associated to the *level* considered.

labels : a *Description*, the labels of each curve associated to one *level*. By default, the *labels* are the values of the *levels*.

Some methods :

buildDefaultLevels

Usage : *buildDefaultLevels(n)*

Arguments : *n* : an integer, the number of levels. If not specified, the default value is taken in the ResourceMap and equal to $n = 10$.

Value : it builds n level values and the associated labels which are the level values. The level values are the empirical quantiles of the data to be sliced at orders q_k regularly distributed over $]0, 1[$:

$$q_k = \frac{k + \frac{1}{2}}{n} \text{ for } 0 \leq k \leq n - 1.$$

setLevels

Usage : *setLevels(levels)*

Arguments : *levels* : a NumericalPoint, the different levels where the iso-curves will be drawn

Value : none

setDrawLabels

Usage : *setDrawLabels(bool)*

Arguments : *bool* : a Boolean, *True* if the labels of the iso-curves must be explicited, *False* otherwise.

Value : none

setLabels

Usage : *setDrawLabels(labels)*

Arguments : *labels* : a *Description*, the list of the labels

Value : none

All the methods *setMethod* have their corresponding *getMethod*.

6.6 Curve

It inherits from the methods of the Drawable class.

Usage :

Curve(data, legend)

Curve(data, color, lineStyle, lineWidth, legend, showPoints)

Arguments :

data : a NumericalSample, the points from which the curve is built, must be of dimension 2

legend : a String, the legend

color : a String, the color of the curve

lineStyle : a String, the style of the curve

lineWidth : an integer, the line width of the curve

showPoints : a boolean which indicates whether the points that define the curve are drawn or not.

Some methods :

isConformData

Usage : *isConformData(data)*

Arguments : *data* : a NumericalSample

Value : a boolean which indicates if the type of data is conform to the type of the drawable (here a Curve) : a NumericalSample of dimension 2

getData

Usage : *getData()*

Arguments : none

Value : a NumericalSample of dimension 2, the data from which the curve is built

getLineWidth

Usage : *getLineWidth()*

Arguments : none

Value : an integer, the line width of the curve

All the methods *getColor*, *getLegendName*, *getLineStyle* and *getLineWidth* have their corresponding *setMethod*.

6.7 Staircase

It inherits from the methods of the Drawable class.

Usage :

StairCase(*data*, *legend*)

StairCase(*data*, *color*, *lineStyle*, *pattern*, *legend*)

StairCase(*data*, *color*, *lineStyle*, *lineWidth*, *pattern*, *legend*)

Arguments :

data : a NumericalSample, the points from which the Staircase is built, must be of dimension 2 : the discontinuous points and their corresponding height

legend : a String, the legend

color : a String, the color of the curve If not specified, by default equal to "blue"

lineStyle : a String, the style of the curve. If not specified, by default equal to "solid"

lineWidth : an integer, the width of the curve. If not specified, by default equal to 1

pattern : a String, the pattern of the staircase which is "S" or "s". If not specified, by default equal to "s". Going from (x_1, y_1) to (x_2, y_2) with $x_1 < x_2$, *pattern* = "s" moves first horizontal then vertical, whereas *pattern* = "S" moves the other way around.

Some methods :

getData

Usage : *getData*()

Arguments : none

Value : a NumericalSample, of dimension 2, giving the discontinuous points and their corresponding height

isConformData

Usage : *isConformData*(*data*)

Arguments : *data* : a NumericalSample

Value : a boolean which indicates if the type of data is conform to the type of the drawable (here a Staircase) : a NumericalSampl of dimension 2, giving the discontinuous points and their corresponding height

getPattern

Usage : *getPattern*()

Arguments : none

Value : a String, the pattern of the staircase.

All the methods *getColor*, *getLegendName*, *getLineStyle* and *getPattern* have their corresponding *setMethod*.

6.8 Pie

It inherits from the methods of the Drawable class.

Usage :

Pie(data)

Pie(data, labels, center, radius, palette)

Arguments :

data : a NumericalSample, of dimension 1, giving the percentiles of the pie

labels : a StringCollection, the names of each group. If not specified, by default equal to the description of the probabilistic input vector

center : a NumericalPoint, the center of the pie inside the bounding box. If not specified, by default equal to (0,0)

radius : a real positive value, the radius of the pie. If not specified, by default equal to 1

palette : a StringCollection, the names of the colors. If not specified, colors are successively taken from the list given below, in the same order

Some methods :

getCenter

Usage : *getCenter()*

Arguments : none

Value : a NumericalPoint, the center of the pie inside the bounding box

getData

Usage : *getData()*

Arguments : none

Value : a NumericalSample of dimension 1, giving the percentiles of the pie

getLabels

Usage : *getLabels()*

Arguments : none

Value : a StringCollection, the names of each group

getPalette

Usage : *getPalette()*

Arguments : none

Value : a StringCollection, the names of the colors used for the pie

getRadius

Usage : *getRadius()*

Arguments : none

Value : a real positive value, the radius of the pie

isConformData

Usage : *isConformData(data)*

Arguments : *data* : a NumericalSample

Value : a boolean which indicates if the type of data is conform to the type of the drawable (here a Pie) : a NumericalSample of dimension 1

getPattern

Usage : *getPattern()*

Arguments : none

Value :

All the methods *getColor*, *getLegendName*, *getLineStyle* and *getPattern* have their corresponding *setMethod*.

6.9 Show

The command *Show* enables to visualise a Graph within the TUI without creating the files .EPS, .PNG or .FIG.

Usage : *Show(graph)*

Arguments : *graph* : a Graph

Value : It shows the graph within the TUI without saving it in any file.

Details : For example, *Show(myDistribution.drawPDF())* where *myDistribution* is a Distribution.

7 Optimization

7.1 Minimization of the distance under an equality constraint

7.1.1 NearestPointAlgorithm

Usage :

NearestPointAlgorithm(levelFunction)

NearestPointAlgorithm(nearestPointAlgorithmImplementation)

Arguments :

levelFunction : a NumericalMathFunction, the constraint function of the constrained optimization problem.

nearestPointAlgorithmImplementation : a NearestPointAlgorithmImplementation, the implementation of the nearest point algorithm, which is *Cobylya* , *AbdoRackwitz* or *SQP*.

Details : a NearestPointAlgorithm, which resolves the optimization problem :

$$\min_{f(\underline{u})=0} \|\underline{U}\|^2$$

where f is the *levelFunction*.

in the first usage, the optimization algorithm is the Cobylya one with its default specific parameters.

in the second usage, the optimization algorithm is specified by the *nearestPointAlgorithmImplementation*.

Some methods :

getLevelFunction

Usage : *getLevelFunction()*

Arguments : none

Value : a NumericalMathFunction, the constraint function of the constrained optimization problem

getLevelValue

Usage : *getLevelFunction()*

Arguments : none

Value : a real value, the level value of the constraint function in the constrained optimization problem

getMaximumAbsoluteError

Usage : *getMaximumAbsoluteError()*

Arguments : none

Value : a positive real value, the maximum absolute error : maximum distance between two successive iterates

getMaximumConstraintError

Usage : *getMaximumConstraintError()*

Arguments : none

Value : a positive real value, the maximum absolute value of the constraint function minus the level value

getMaximumIterationsNumber

Usage : *getMaximumIterationsNumber()*

Arguments : none

Value : an integer, the maximum number of iterations of the algorithm

getMaximumRelativeError

Usage : *getMaximumRelativeError()*

Arguments : none

Value : a real value, the maximum relative distance between two successive iterates (with regards the second iterate)

getMaximumResidualError

Usage : *getMaximumResidualError()*

Arguments : none

Value : a real value, the maximum orthogonality error (lack of orthogonality between the vector Center - Iterate and the constraint surface)

getResult

Usage : *getResult()*

Arguments : none

Value : a NearestPointAlgorithmImplementationResult, the structure containing all the results of the constrained optimization problem

getStartingPoint

Usage : *getStartingPoint()*

Arguments : none

Value : a NumericalPoint, the starting point of the constrained optimization research

run

Usage : *run()*

Arguments : none

Value : none

Role : it creates a NearestPointAlgorithmImplementationResult, the optimization result which is accessible with the method *getResult()*.

Each *getMethod* is associated to a *setMethod*.

7.1.2 Cobyala

This class inherits from the NearestPointAlgorithmImplementation class.

Usage :

```
Cobyala()  
Cobyala(specificParameters, levelFunction)
```

Arguments :

specificParameters : a CobyalaSpecificParameters, the list of the parameters specific to the Cobyala algorithm

levelFunction : a NumericalMathFunction, the constraint function of the constrained optimization problem

Details : When no argument is specified, the parameters will have to be fulfilled after, for example, when used in a FORM algorithm (see the corresponding Use Case).

Some methods :

```
getSpecificParameters
```

Usage : *getSpecificParameters*()

Arguments : none

Value : a CobyalaSpecificParameters, the list of the parameters specific to the Cobyala algorithm

This *getMethod* is associated to a *setMethod*.

7.1.3 CobySpecificParameters

Usage :

CobySpecificParameters()

CobySpecificParameters(rhoBeg)

Arguments : *rhoBeg* : a real positive strictly value, a reasonable initial step to the variables.

Values :

in the first usage, the default values are considered for each parameter : *rhoBeg* = 0.1.

in the second usage, the parameter *rhoBeg* is specified.

Some methods :

getRhoBeg

Usage : *getRhoBeg()*

Arguments : none

Value : a real value >0 , a reasonable initial step to the variables

This *getMethod* is associated to a *setMethod*.

7.1.4 AbdoRackwitz

This class inherits from the `NearestPointAlgorithmImplementation` class.

Usage :

```
AbdoRackwitz()  
AbdoRackwitz(specificParameters, levelFunction)
```

Arguments :

specificParameters : a `AbdoRackwitzSpecificParameters`, the list of the parameters specific to the AbdoRackwitz algorithm
levelFunction : a `NumericalMathFunction`, the constraint function of the constrained optimization problem

Details : When no argument is specified, the parameters will have to be fulfilled after, for example, when used in a FORM algorithm (see the corresponding Use Case).

The AbdoRackwitz algorithm is a gradient-based constrained optimization method, thus the `NumericalMathFunction` has to provide its gradient:

- If the function is a meta-model generated by OpenTURNS, the analytical gradient is automatically provided.
- If it is an analytical function, OpenTURNS provides a gradient based on the centered finite difference method, that the user can change to better fit its will by another `GradientImplementation` (e.g. constructed by another finite difference method).
- If the function is loaded thanks to a wrapper and if the wrapper provide an implementation of the gradient, OpenTURNS uses it. If there is no gradient provided, OpenTURNS provides a gradient based on centered finite difference method, but with a parameterization different from the case of analytical functions (assuming a lower precision for the function evaluation than in the analytical case).

Be aware of the potential pitfalls associated with the use of finite differences and check the value of the finite difference epsilon for each dimension if AbdoRackwitz algorithm fails to converge.

Some methods :

```
getSpecificParameters
```

Usage : *getSpecificParameters*()

Arguments : none

Value : a `AbdoRackwitzSpecificParameters`, list of the parameters specific to the AbdoRackwitz algorithm

This *getMethod* is associated to a *setMethod*.

7.1.5 AbdoRackwitzSpecificParameters

Usage :

AbdoRackwitzSpecificParameters()

AbdoRackwitzSpecificParameters(Tau, Omega, Smooth)

Arguments :

Tau : a real positive value, the multiplicative decrease of the linear step. It must be < 1 .

Omega : a real strictly positive value, the Armijo factor. It must be < 1 and should be rather small.

Smooth : a real value > 1 , the increasing rate of the penalisation coefficient for the line search. It must be > 1 and should be rather near 1.

Values :

in the first usage, the default values are considered for each parameter : $Tau = 0.5$, $Omega = 10^{-4}$, $Smooth = 1.2$.

in the second usage, all the parameters are specified.

Some methods :

getOmega

Usage : *getOmega()*

Arguments : none

Value : a real value between 0 and 1, the Armijo factor

getSmooth

Usage : *getSmooth()*

Arguments : none

Value : a real value > 1 , the increasing rate of the penalisation coefficient for the line search

getTau

Usage : *getTau()*

Arguments : none

Value : a real value between 0 and 1, the multiplicative decrease of the linear step

This *getMethod* is associated to a *setMethod*.

7.1.6 SQP

This class inherits from the `NearestPointAlgorithmImplementation` class.

Usage : *SQP(specificParameters, levelFunction)*

Arguments :

specificParameters : a `SQPSpecificParameters`, list of the parameters specific to the SQP algorithm

levelFunction : a `NumericalMathFunction`, the constraint function of the constrained optimization problem

Some methods :

getSpecificParameters

Usage : *getSpecificParameters()*

Arguments : none

Value : a `SQPSpecificParameters`, list of the parameters specific to the SQP algorithm

This *getMethod* is associated to a *setMethod*.

7.1.7 SQPSpecificParameters

Usage :

SQPSpecificParameters()

SQPSpecificParameters(Omega, Smooth, Tau)

Arguments :

Omega : a real value, the Armijo factor, must be > 0 and < 1 but rather small.

Smooth : a real value, the increasing rate of the penalisation coefficient for the line search, must be > 1 but rather near 1.

Tau : a real value, the multiplicative decrease of the linear step, must be > 0 and < 1 .

Values :

in the first usage, the default values are considered for each parameter : $Omega = 10^{-4}$, $Smooth = 1.2$,
 $Tau = 0.5$.

in the second usage, all the parameters are specified.

Some methods :

getOmega

Usage : *getOmega()*

Arguments : none

Value : a real value between 0 and 1, Armijo factor

getSmooth

Usage : *getSmooth()*

Arguments : none

Value : a real value > 1 , the increasing rate of the penalisation coefficient for the line search

getTau

Usage : *getTau()*

Arguments : none

Value : a real value between 0 and 1, multiplicative decrease of the linear step

These *getMethod* are associated to *setMethod*.

7.1.8 NearestPointAlgorithmImplementationResult

Usage : structure created by the method `run()` of a `NearestPointAlgorithm` and obtained thanks to the method `getResult()`

Some methods :

getAbsoluteError

Usage : `getAbsoluteError()`

Arguments : none

Value : a `NumericalScalar`, the absolute error : the distance between the two last successive iterates when the algorithm stops

getConstraintError

Usage : `getConstraintError()`

Arguments : none

Value : a real value, the absolute value of the constraint function minus the level value at the last iterate point when the algorithm stops

getIterationsNumber

Usage : `getIterationsNumber()`

Arguments : none

Value : an integer, the number of performed iterations of the algorithm when the algorithm stops

getMinimizer

Usage : `getMinimizer()`

Arguments : none

Value : a `NumericalPoint`, the last iterate when the algorithm stops (the solution of the optimization problem)

getRelativeError

Usage : `getRelativeError()`

Arguments : none

Value : a real value, the relative distance between the two last successive iterates (with regards the last iterate)

getResidualError

Usage : `getResidualError()`

Arguments : none

Value : a real value, the orthogonality error of the solution point (lack of orthogonality between the vector Center - Last Iterate and the constraint surface)

7.2 Optimization of a function under an inequality constraint

7.2.1 BoundConstrainedAlgorithm

Usage :

*BoundConstrainedAlgorithm(objFct, boundCons, startingPt, ...
...BndConstAlgoImp.MINIMIZATION)*

*BoundConstrainedAlgorithm(objFct, boundCons, startingPt, ...
...BndConstAlgoImp.MAXIMIZATION)*

BoundConstrainedAlgorithm(BndConstAlgoImp)

Arguments :

objFct : a NumericalMathFunction, the function to minimize or maximize (objective function).

startingPt : a Interval, the interval $[a, b] \in \overline{\mathbb{R}}^n$.

startingPt : a NumericalPoint, the initial point of the optimization algorithm.

BndConstAlgoImp.MINIMIZATION : the code which specifies that it is a minimization optimization. It is possible to write *O* instead of it.

BndConstAlgoImp.MAXIMIZATION : the code which specifies that it is a maximization optimization. It is possible to write *1* instead of it.

boundConstrainAlgorithmImplementation : a BoundConstrainAlgorithmImplementation, the implementation of the bound constrain algorithm, which is *TNC*.

Value : a BoundConstrainedAlgorithm, which resolves the optimization problems :

$$\min_{a \leq u \leq b} f(u)$$

or

$$\max_{a \leq u \leq b} f(u)$$

where f is the objective function and $[a, b] \in \overline{\mathbb{R}}^n$.

In the first and second usages, the optimization algorithm is the TNC one with its default specific parameters.

In the third usage, the optimization problem is entirely specified by the implementation of the bound constrain algorithm.

Some methods :

getBoundConstraints

Usage : *getBoundConstraints()*

Arguments : none

Value : a Interval, the interval $[a, b] \in \overline{\mathbb{R}}^n$ where the optimization is performed.

getMaximumAbsoluteError

Usage : *getMaximumAbsoluteError()*

Arguments : none

Value : a positive real value, precision goal for the value of x in the stopping criterion (after applying x scaling factors). If negative 0.0, that parameter is set to $\sqrt{\text{machine} - \text{precision}}$.

getMaximumConstraintError

Usage : *getMaximumConstraintError()*

Arguments : none

Value : a positive real value, the precision goal for the value of the projected gradient in the stopping criterion (after applying x scaling factors) : if negative, that parameter is set to $1e-2 * \text{sqrt}(\text{accuracy})$. Setting it to 0.0 is not recommended.

getMaximumEvaluationsNumber

Usage : *getMaximumEvaluationsNumber()*

Arguments : none

Value : an integer, the maximum number of evaluations of the objective function.

getMaximumObjectiveError

Usage : *getMaximumObjectiveError()*

Arguments : none

Value : a positive real value, the precision goal for the value of the objective function in the stopping criterion. If negative, that parameter is set to accuracy.

getObjectiveFunction

Usage : *getObjectiveFunction()*

Arguments : none

Value : a NumericalMathFunction, the objective function.

getResult

Usage : *getResult()*

Arguments : none

Value : a BoundConstrainedAlgorithmImplementationResult, the structure containing all the results of the constrained optimization problem

getStartingPoint

Usage : *getStartingPoint()*

Arguments : none

Value : a NumericalPoint, the starting point of the constrained optimization research.

run

Usage : *run()*

Arguments : none

Value : none

Role : it creates a BoundConstrainedAlgorithmImplementationResult, the optimization result which is accessible with the method *getResult()*.

These *getMethod* are associated to *setMethod*.

7.2.2 TNC (Truncated Newton Constrained)

This class inherits from the `BoundConstrainedAlgorithmImplementation` class.

The TNC algorithm minimizes or maximizes an objective function f with variables x subjected to bounds $[a, b] \in \overline{\mathbb{R}}^n$, using gradient information.

Usage :

`TNC(specParam, objFct, boundCons, startPt, TNC.MINIMIZATION)`

`TNC(specParam, objFct, boundCons, startPt, TNC.MAXIMIZATION)`

Arguments :

specParam : a `TNCSpecificParameters`, the list of the parameters specific to the TNC algorithm

objFct : a `NumericalMathFunction`, the constraint function of the constrained problem

boundCons : a `Interval`, the interval $[a, b] \in \overline{\mathbb{R}}^n$ where the optimization is performed.

startingPoint : a `NumericalPoint`, the starting point of the constrained optimization research.

TNC.MINIMIZATION, TNC.MAXIMIZATION : the code which specifies whether it is a minimization or a maximization optimization. It is possible to write 0 instead of *TNC.MINIMIZATION* and 1 instead of *TNC.MAXIMIZATION*.

Some methods :

getSpecificParameters

Usage : `getSpecificParameters()`

Arguments : none

Value : a `TNCSpecificParameters`, the list of the parameters specific to the TNC algorithm

This *getMethod* is associated to a *setMethod*.

7.2.3 TNCSpecificParameters

The TNC algorithm resolves : $\min_{\underline{x} \in [a,b] \in \mathbb{R}^n} f(\underline{x})$ and proceeds as follows under the proper regularity of the objective function f :

$$\begin{cases} \underline{\nabla} f(\underline{x}) = \underline{0} \\ \underline{\nabla}_2 f(\underline{x}) \text{ is definite positive} \end{cases}$$

The Taylor development of second order of f around \underline{x}_k leads to the determination of the iterate \underline{x}_{k+1} such as :

$$\begin{cases} \underline{\Delta}_k & = \underline{x}_{k+1} - \underline{x}_k \\ \underline{\nabla}_2 f(\underline{x}_k) \underline{\Delta}_k & = -\underline{\nabla} f(\underline{x}_k) \end{cases} \quad (2)$$

The equation (2) is truncated : the iterative research of $\underline{\Delta}_k$ is stopped as soon as $\underline{\Delta}_k$ verifies :

$$\|\underline{\nabla}_2 f(\underline{x}_k) \underline{\Delta}_k + \underline{\nabla} f(\underline{x}_k)\| \leq \eta \|\underline{\nabla} f(\underline{x}_k)\|$$

At last, the iteration $k + 1$ is defined by :

$$\underline{x}_{k+1} = \underline{x}_k + \alpha_k \underline{\Delta}_k$$

where α_k is the parameter *stepmx*.

Usage :

```
TNCSpecificParameters()
TNCSpecificParameters(scale, offset, maxCGit, eta, stepmx, ...
...accuracy, fmin, rescale)
```

Arguments :

scale : a real value, the scaling factors to apply to each variable : if NULL, the factors are up-low for interval bounded variables and $1+|x|$ for the others.

offset : a real value, the constant to subtract to each variable. If NULL, the constant are (up+low)/2 for interval bounded variables and x for the others.

maxCGit : an integer, the max. number of hessian*vector evaluation per main iteration. If maxCGit == 0, the direction chosen is -gradient. If maxCGit < 0, maxCGit is set to $\max(1, \min(50, n/2))$.

eta : a positive real value, the severity of the line search. If < 0 or > 1, set to 0.25.

stepmx : a real value, the maximum step for the line search. may be increased during call. If too small, will be set to 10.0.

accuracy : a real value, the relative precision for finite difference calculations. If $\leq \text{machine-precision}$, set to $\sqrt{\text{machine-precision}}$.

fmin : a real value, the minimum function value estimate

rescale : a real value, the objective function scaling factor (in log10) used to trigger the objective function value rescaling : if 0, rescale at each iteration; if a big value, never rescale; if < 0, rescale is set to 1.3.

Value :

in the first usage, the default values are considered for each parameter : *MaxCGit* = 50, *Eta* = 0.25, *Stepmx* = 10.0, *Accuracy* = $1.0e - 4$, *Fmin* = 1.0, *Rescale* = 1.3.

in the second usage, all the parameters are specified.

Some methods :

getAccuracy

Usage : *getAccuracy()*

Arguments : none

Value : a real value, the *accuracy* parameter.

getEta

Usage : *getEta()*

Arguments : none

Value : a positive real value, the *eta* parameter.

getFmin

Usage : *getFmin()*

Arguments : none

Value : a real value, the *fmin* parameter.

getMaxCGit

Usage : *getMaxCGit()*

Arguments : none

Value : an integer, the *maxCGit* parameter.

getOffset

Usage : *getOffset()*

Arguments : none

Value : a real value, the *offset* parameter.

getRescale

Usage : *getRescale()*

Arguments : none

Value : a real value, the *rescale* parameter.

getScale

Usage : *getScale()*

Arguments : none

Value : a real value, the *scale* parameter.

getStepmx

Usage : *getStepmx()*

Arguments : none

Value : a real value, the *stepmx* parameter.

These *getMethod* are associated to *setMethod*.

7.2.4 BoundConstrainedAlgorithmImplementationResult

Usage : structure created by the method `run()` of a `BoundConstrainedAlgorithm` and obtained thanks to the method `getResult()`

Some methods :

getAbsoluteError

Usage : `getAbsoluteError()`

Arguments : none

Value : a positive real value, precision obtained for the value of `x` in the stopping criterion (after applying `x` scaling factors). Care : if the algorithm is TNC, there is no information on that parameter : the Maximum absolute error is returned.

getConstraintError

Usage : `getConstraintError()`

Arguments : none

Value : a positive real value, the precision obtained for the value of the projected gradient in the stopping criterion (after applying `x` scaling factors). Care : if the algorithm is TNC, there is no information on that parameter : the maximum constraint error is returned.

getEvaluationsNumber

Usage : `getEvaluationsNumber()`

Arguments : none

Value : an integer, the number of evaluations of the objective function.

getObjectiveError

Usage : `getObjectiveError()`

Arguments : none

Value : a positive real value, the precision obtained for the value of the objective function in the stopping criterion. Care : if the algorithm is TNC, there is no information on that parameter : the maximum objective error is returned.

getOptimalValue

Usage : `getOptimalValue()`

Arguments : none

Value : a real, the objective function value at the optimizer point.

getOptimizer

Usage : `getOptimizer()`

Arguments : none

Value : a `NumericalPoint`, solution of the constrained optimization problem.

8 Polynomials

8.1 UniVariatePolynomial

Usage : *UniVariatePolynomial(coefficients)*

Arguments : *coefficients* : a NumericalPoint, the list of the coefficients of each term x^k (no sparse representation)

Value : a UniVariatePolynomial

Some methods :

*

Usage :

*Pol1 * Pol2*

*lambda * Pol1*

Arguments :

(Pol1, Pol2) : two UniVariatePolynomial,

lambda : a NumericalScalar

Value :

usage 1 : a UniVariatePolynomial, the result of the multiplication *Pol1 * Pol2*

usage 2 : a UniVariatePolynomial, the result of the multiplication *lambda * Pol1*

+

Usage : *Pol1 + Pol2*

Arguments : *(Pol1, Pol2)* : two UniVariatePolynomial

Value : a UniVariatePolynomial, the result of the addition *Pol1 + Pol2*

–

Usage : *Pol1 – Pol2*

Arguments : *(Pol1, Pol2)* : two UniVariatePolynomial

Value : a UniVariatePolynomial, the result of the subtraction *Pol1 – Pol2*

derivate

Usage : *derivate()*

Arguments : none

Value : a UniVariatePolynomial, the derivated univariate polynomials

derivative

Usage : *derivative(point)*

Arguments : *point* : a NumericalScalar

Value : a NumericalScalar, the value of the derivated polynomials at point *point*

draw

Usage : *draw(min, max, pointNumber)*

Arguments :

min, max : a NumericalScalar

pointNumber : an integer, the number of points used for the graph

Value : a Graph, the polynomials curve on the range [*min, max*].

getCoefficients

Usage : *getCoefficients()*

Arguments : none

Value : a Coefficients, the coefficients of each x^k for $k \leq$ to the degree of the univariate polynomials (no sparse representation)

getDegree

Usage : *getDegree()*

Arguments : none

Value : an integer, the degree of the univariate polynomials

getRoots

Usage : *getRoots()*

Arguments : none

Value : a NumericalComplexCollection, the collection of complex roots of the univariate polynomials

incrementDegree

Usage : *incrementDegree(deg)*

Arguments : *deg* : an integer

Value : a UniVariatePolynomial obtained by multiplying the polynomial by x^{deg}

8.2 PolynomialCollection

Usage : *PolynomialCollection(size, univariatePol)*

Arguments :

size : an integer

univariatePol : a UniVariatePolynomial

Value : a PolynomialCollection, which contains *size* polynomials each equal to *univariatePol*

Some methods :

add

Usage : *add(univariatePol)*

Arguments : *univariatePol* : a UniVariatePolynomial,

Value : a PolynomialCollection which size has been increased of 1 and to which the polynomials *univariatePol* has been added

at

Usage : *at(i)*

Arguments : *i* : an integer

Value : a UniVariatePolynomial, the polynomials at position *i* in the collection

resize

Usage : *resize(newSize)*

Arguments : *i* : an integer

Value : a PolynomialCollection which size has been modified into *newSize* as follows : if *newSize* \leq *getSize()* then, the collection is truncated to the first *newSize* polynomials. Otherwise, the collection is increased until the size *newSize* : the added polynomials are the nul ones.

8.3 ProductPolynomialEvaluationImplementation

Usage : *ProductPolynomialEvaluationImplementation(polCollection)*

Arguments : *polCollection* : a PolynomialCollection, a collection of UniVariatePolynomial

Value : a ProductPolynomialEvaluationImplementation, the product of the polynomials of *polCollection*. The result polynomials is of input dimension n where n is the number of polynomials in *polCollection*.

Some methods :

Operator()

Usage : *Operator(point)*

Arguments : *point* : a NumericalPoint, which dimension is n where n is the number of polynomials in *polCollection*

Value : a NumericalPoint of dimension 1,

+

Usage : *Pol1 + Pol2*

Arguments : (*Pol1, Pol2*) : two UniVariatePolynomial

Value : a UniVariatePolynomial, the result of the addition *Pol1 + Pol2*

Details : The exact gradient and hessian evaluations have been implemented for the products of polynomials.

9 Response Surface : Parametric Approximation

9.1 Taylor approximation

9.1.1 LinearTaylor

Usage : *LinearTaylor*(*center*, *function*)

Arguments :

center : a NumericalPoint, the point where the Taylor expansion of the function *function* is performed

function : a NumericalMathFunction, the function to be approximated.

Value : a LinearTaylor

Some methods :

getInputFunction

Usage : *getInputFunction*

Arguments : none

Value : a NumericalMathFunction, the function *function*

getName

Usage : *getName*()

Arguments : none

Value : a string, the name of the LinearTaylor

getCenter

Usage : *getCenter*()

Arguments : none

Value : a NumericalPoint, around which the approximation has been made : *center*

run

Usage : *run*()

Arguments : none

Value : it performs the linear Taylor expansion around *center* (while this method has not been executed, only *getInputFunction*, *getName* and *setName* methods can be used)

getConstant

Usage : *getConstant*()

Arguments : none

Value : a NumericalPoint, the constant vector of the approximation, equal to *function*(*center*)

getLinear

Usage : *getLinear*()

Arguments : none

Value : a Matrix, the gradient of the function *function* at the point *center* (the transposition of the jacobian matrix)

getResponseSurface

Usage : *getResponseSurface*()

Arguments : none

Value : a NumericalMathFunction, an approximation of the function *function* by a linear Taylor expansion at the *center*

Links : [see docref_SurfRep_Taylor](#)

The methods *getInputFunction*, *getName*, *getCenter* have their associated *setMethod*.

9.1.2 QuadraticTaylor

Usage : *QuadraticTaylor(center, function)*

Arguments :

center : a NumericalPoint, the point where the quadratic Taylor expansion of the function *function* is performed

function : a NumericalMathFunction, the function to be approximated : the gradient and hessian of the NumericalMathFunction must be defined.

Value : a QuadraticTaylor

Some methods :

getInputFunction

Usage : *getInputFunction*

Arguments : none

Value : a NumericalMathFunction, the function *function*

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the QuadraticTaylor

getCenter

Usage : *getCenter()*

Arguments : none

Value : a NumericalPoint, around which the approximation has been made : *center*

run

Usage : *run()*

Arguments : none

Value : it performs the Quadratic Taylor expansion around *center* (while this method has not been executed, only *getInputFunction*, *getName* and *setName* methods can be used)

getConstant

Usage : *getConstant()*

Arguments : none

Value : a NumericalPoint, the constant vector of the approximation, equal to *function(center)*

getLinear

Usage : *getLinear()*

Arguments : none

Value : a Matrix, the gradient of the function *function* at the point *center* (the transposition of the jacobian matrix)

getQuadratic

Usage : *getQuadratic()*

Arguments : none

Value : a SymmetricTensor which contains the 0.5 * transposition of the hessian values of *function* at *center*

getResponseSurface

Usage : *getResponseSurface()*

Arguments : none

Value : a NumericalMathFunction, an approximation of the function *function* by a Quadratic Taylor expansion at *center*

Links : [see docref_SurfRep_Taylor](#)

The methods *getInputFunction*, *getName*, *getCenter* have their associated *setMethod*.

9.2 Least squares approximation

9.2.1 LinearLeastSquares

Usage :

LinearLeastSquares(dataIn, function)

LinearLeastSquares(dataIn, dataOut)

Arguments :

dataIn : a NumericalSample, the input variables

function : a NumericalMathFunction, the function to be approximated

dataOut : a NumericalSample, the output variables

Value : a LinearLeastSquares, the linear least squares approximation between :

the two samples *dataIn* and *dataOut* in the case of the second usage

the two samples *dataIn* and *function(dataIn)* in the case of the first usage

Some methods :

getInputFunction

Usage : *getInputFunction()*

Arguments : none

Value : a NumericalMathfunction the *function* parameter in the case of the first usage

getDataIn

Usage : *getDataIn()*

Arguments : none

Value : a NumericalSample, the *dataIn* parameter

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the LinearLeastSquares

run

Usage : *run()*

Arguments : none

Value : it performs the linear least squares approximation (while this method has not been executed, only *getInputfunctionion*, *getDataIn*, *getName* and *setName* methods can be used)

getDataOut

Usage : *getDataIn()*

Arguments : none

Value : a NumericalSample, it returns the output variable :
in the case of the first usage, it corresponds to the values of the function *function* at the input variables *dataIn* : *function(dataIn)*
in the case of the second usage, it corresponds to *dataOut*

getLinear

Usage : *getLinear()*

Arguments : none

Value : a Matrix, the gradient of the function *function* at the point *center* (the transposition of the jacobian matrix)

getResponseSurface

Usage : *getResponseSurface()*

Arguments : none

Value : a NumericalMathFunction, an approximation of the function *function* by Linear Least Squares

Links : see [docref_SurfRep_LeastSquare](#)

The methods *getInputFunction*, *getName*, *getDataIn* have their associated *setMethod*.

9.2.2 QuadraticLeastSquares

Usage : *QuadraticLeastSquares(dataIn, function)*

QuadraticLeastSquares(dataIn, dataOut)

Arguments :

dataIn : a NumericalSample, the input variables

function : a NumericalMathFunction, the function to be approximated

dataOut : a NumericalSample, the output variables

Value : a QuadraticLeastSquares, the quadratic least squares approximation between :

the two samples *dataIn* and *dataOut* in the case of the second usage

the two samples *dataIn* and *function(dataIn)* in the case of the first usage

Some methods :

getInputFunction

Usage : *getInputFunction*

Arguments : none

Value : a NumericalMathFunction, the function *function*

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the QuadraticLeastSquares

run

Usage : *run()*

Arguments : none

Value : it performs the quadratic least squares approximation (while this method has not been executed, only *getInputFunction*, *getName* and *setName* methods can be used)

getDataOut

Usage : *getDataIn()*

Arguments : none

Value : a NumericalSample, it returns the output variable :

in the case of the first usage, it corresponds to the values of the function *function* at the input variables *dataIn* : *function(dataIn)*

in the case of the second usage, it corresponds to *dataOut*

getConstant

Usage : *getConstant()*

Arguments : none

Value : a NumericalPoint, the constant vector of the approximation, equal to $function(center)$

getLinear

Usage : *getLinear()*

Arguments : none

Value : a Matrix, the linear matrix of the approximation

getQuadratic

Usage : *getQuadratic()*

Arguments : none

Value : a SymmetricTensor, the quadratic term of the approximation

getResponseSurface

Usage : *getResponseSurface()*

Arguments : none

Value : a NumericalMathFunction, an approximation of the function $function$ by Quadratic Least Squares

Links : [see docref_SurfRep_LeastSquare](#)

The methods *getInputFunction*, *getName*, *getDataIn* have their associated *setMethod*.

10 Response Surface : Functional Chaos Expansion

The polynomial chaos expansion enables to approximate the output random variable of interest :

$$\underline{Y} = g(\underline{X}) \in \mathbb{R}^p \quad (3)$$

where $g : \mathbb{R}^n \rightarrow \mathbb{R}^p$ is the model, \underline{X} is the input random vector of dimension n by the response surface :

$$\underline{\tilde{Y}} = \sum_{k \in K} \alpha_k \Psi_k \circ T(\underline{X}) \quad (4)$$

with :

$$\alpha_k \in \mathbb{R}^p \quad (5)$$

T is an isoprobabilistic transformation which maps the multivariate distribution of \underline{X} into the multivariate distribution,

$$\mu = \prod_{i=1}^n \mu_i \quad (6)$$

and $(\Psi_k)_{k \in \mathbb{N}}$ a multivariate polynomial basis of $\mathcal{L}_{\mu}^2(\mathbb{R}^n, \mathbb{R})$ which is orthonormal according to the distribution μ . K is a finite subset of \mathbb{N} .

The distribution μ is supposed to have an independent copula and Y be of finite second moment.

10.1 FunctionalChaosAlgorithm

Usage :

FunctionalChaosAlgorithm()

FunctionalChaosAlgorithm(model, distribution, adaptiveStrategy)

FunctionalChaosAlgorithm(model, distribution, adaptiveStrategy, projectionStrategy)

FunctionalChaosAlgorithm(inputSample, weights, outputSample, distribution, adaptiveStrategy, projectionStrategy)

Arguments :

model : a NumericalMathFunction, the physical model g defined in (3) that behaves as a NumericalMathFunction.

distribution : a Distribution, the joint probability density function of the physical input vector

inputSample, outputSample : NumericalSample, the input and output samples of a model evaluated apart.

weight : NumericalPoint, the weights of each point of the *inputSample*.

adaptiveStrategy : an AdaptiveStrategy, the strategy of selection of the basis (see AdaptiveStrategy above)

projectionStrategy : a ProjectionStrategy, that defines the strategy of projection of the empirical model results in the selected orthonormal basis (see ProjectionStrategy).

Value : a FunctionalChaosAlgorithm. In the second usage, the projection strategy considered is the Least-SquaresStrategy. In the last usage, the model is not required and replaced by some given input sample and output sample. The weights ω_i are determined such that $\sum_{i \in I} \omega_i \delta_{\underline{X}_i} \simeq p_{\underline{X}}$, where $p_{\underline{X}}$ is the distribution of the input random vector \underline{X} . When not specified, the *weight* are all equal to

$$\omega_i = \frac{1}{\text{card}I}$$

Some methods :*getMaximumResidual***Usage :** *getMaximumResidual()***Arguments :** none**Value :** a NumericalScalar, the residual value needed in the projection strategy. By default, its value is 0.*getProjectionStrategy***Usage :** *getProjectionStrategy()***Arguments :** none**Value :** a ProjectionStrategy, the projection strategy of the FunctionalChaosAlgorithm.*run***Usage :** *run()***Arguments :** none**Value :** execute the procedure of determination of coefficients using the projection strategy selected with respect to the AdaptiveStrategy selected. It provides the results as an object of type FunctionalChaosResult.*getResult***Usage :** *getResult()***Arguments :** none**Value :** a FunctionalChaosResult, which contains all results of the execution.

The method *getMaximumResidual* has its associated *setMaximumResidual*.

10.2 FunctionalChaosResult

Usage : structure created by the method *run()* of a FunctionalChaosAlgorithm and obtained thanks to the method *getResult()*.

Some methods :*getCoefficients***Usage :** *getCoefficients()***Arguments :** none**Value :** a NumericalSample, the collection of coefficients of the functional chaos $(\alpha_k)_{k \in K}$ *getComposedMetaModel***Usage :** *getComposedMetaModel()***Arguments :** none**Value :** a NumericalMathFunction, $\hat{h} = \sum_{k \in K} \alpha_k \Psi_k$.*getComposedModel*

Usage : *getComposedModel()*

Arguments : none

Value : a NumericalMathFunction, h which is the composition of the physical model g and the inverse iso-probabilistic transformation: $h : \underline{Z} \rightarrow \underline{Y} = g \circ T^{-1}(\underline{Z})$. We have $h = \sum_{k \in \mathbb{N}} \alpha_k \Psi_k$.

getDistribution

Usage : *getDistribution()*

Arguments : none

Value : a Distribution, the joint distribution of the physical input vector

getIndices

Usage : *getIndices()*

Arguments : none

Value : an Indices, the collection of integers that represent the indices of the final basis

getInverseTransformation

Usage : *getInverseTransformation()*

Arguments : none

Value : a NumericalMathFunction, the inverse iso-probabilistic transformation T^{-1} that transforms the data distributed according to the measure imposed by the selected orthonormal basis into the input data: $T^{-1} : \mathbb{R}^n \rightarrow \mathbb{R}^n$, $T^{-1}(\underline{Z}) = \underline{X}$

getMeasure

Usage : *getMeasure()*

Arguments : none

Value : a Distribution, the measure μ

getMetaModel

Usage : *getMetaModel()*

Arguments : none

Value : a NumericalMathFunction, $\hat{g} = \hat{h} \circ T$.

getModel

Usage : *getModel()*

Arguments : none

Value : a NumericalMathFunction, the physical model g defined in (3).

getReducedBasis

Usage : *getReducedBasis()*

Arguments : none

Value : a NumericalMathFunctionCollection, a collection of NumericalMathFunction that correspond to the basis of the functional chaos $(\Psi_k)_{k \in K}$

getRelativeErrors

Usage : *getResiduals()*

Arguments : none

Value : a NumericalPoint, the relative residual values parginal by marginal evaluated by the projection strategy.

getResiduals

Usage : *getResiduals()*

Arguments : none

Value : a NumericalPoint, the residual values parginal by marginal evaluated by the projection strategy.

getTransformation

Usage : *getTransformation()*

Arguments : none

Value : a NumericalMathFunction, the iso-probabilistic transformation T that transforms the input data into a data following the measure imposed by the selected orthonormal basis: $T : \mathbb{R}^n \rightarrow \mathbb{R}^n$,
 $T(\underline{X}) = \underline{Z}$

10.3 Construction of the multivariate orthogonal basis

10.3.1 OrthogonalBasis

Usage :

OrthogonalBasis()

OrthogonalBasis(orthogonalProductPolynomialFactory)

OrthogonalBasis(otherOrthogonalFunctionFactory)

Arguments :

orthogonalProductPolynomialFactory : an OrthogonalProductPolynomialFactory (see details below)

otherOrthogonalFunctionFactory : an OrthogonalFunctionFactory, it provides to the OrthogonalBasis the persistent types of the univariate orthogonal functions other than polynomials. For example, it can be a WaveletFunctionFactory that will be implemented with the response surface by wavelet expansion.

Value : an OrthogonalBasis, which is the interface class of the OrthogonalFunctionFactory implementation, which is an OrthogonalProductPolynomialFactory in the particular case of polynomial chaos expansion.

Some methods :

build

Usage : *build(index)*

Arguments : *index* : an integer that indicates the term of the basis which must be constructed.

In other words, *index* is used by a bijection from \mathbb{N} to \mathbb{N}^d (d is the dimension of the Basis); this bijection is detailed later in EnumerateFunction.

Value : a NumericalMathFunction, a term of the basis collection

getMeasure

Usage : *getMeasure()*

Arguments : none

Value : a Distribution, the joint measure that represents the product of the univariate distributions which are respectively orthogonal with respect to the selected univariate function families

10.3.2 OrthogonalProductPolynomialFactory

This class inherits from OrthogonalFunctionFactory

Usage :

OrthogonalProductPolynomialFactory()

OrthogonalProductPolynomialFactory(polColl)

OrthogonalProductPolynomialFactory(polColl, enumFct)

Arguments :

polColl : a PolynomialFamilyCollection, collection of orthogonal univariate polynomials factories. The collection must have the same dimension as the orthogonal basis.

enumFct : an EnumerateFunction, that associates to an integer its multi-index image in the \mathbb{N}^d dimension, which is the dimension of the basis. This multi-index represents the collection of degrees of the univariate polynomials

Value : an OrthogonalProductPolynomialFactory, represents a term of the basis that corresponds to the product of the univariate orthonormal polynomials. Let's note that the exact hessian and gradient have been implemented for the product of polynomials.

Some methods :

build

Usage : *build(index)*

Arguments : *index* : an integer that indicates the term of the basis which must be constructed. In other words, *index* is used by a bijection from \mathbb{N} to \mathbb{N}^{dim} (recalling that *dim* is the dimension of the *Basis*)

Value : a NumericalMathFunction, a term of the basis collection

getMeasure

Usage : *getMeasure()*

Arguments : none

Value : a Distribution, the joint measure that represents the product of the univariate distributions which are respectively orthogonal with respect to the selected univariate function families

Details : OrthogonalProductPolynomialFactory is a particular case of implementation of the OrthogonalFunctionFactory in the case of polynomial chaos expansion. It provides to the OrthogonalBasis the persistent types of the univariate orthogonal polynomials (e.g. Hermite, Legendre, Laguerre and Jacobi) needed to determine the distribution measure of projection of the input variable.

10.3.3 OrthogonalUniVariatePolynomialFamily

OrthogonalUniVariatePolynomialFamily is the interface of the OrthogonalUniVariatePolynomialFactory implementation.

Usage :

OrthogonalUniVariatePolynomialFamily()

OrthogonalUniVariatePolynomialFamily(orthogUniVarPolFactory)

Arguments :

orthogUniVarPolFactory : an OrthogonalUniVariatePolynomialFactory that builds particular univariate polynomial (e.g. Hermite, Legendre, Laguerre, ...).

Value : an OrthogonalUniVariatePolynomialFamily, represents the factory that allows the construction of any univariate orthonormal polynomial with any degree.

Some methods :

build

Usage : *build(n)*

Arguments : *n* : an integer, the degree of the orthogonal polynomial to be build. The polynomial is orthonormal with respect to its associated distribution.

Value : a UnivariatePolynomial of the same type as its factory

getMeasure

Usage : *getMeasure()*

Arguments : none

Value : a Distribution, the univariate measure with respect to which the polynomial basis is orthonormal : if the polynomials family is noted $(\Psi_k)_k$ then we have :

$$\langle \Psi_k, \Psi_l \rangle_{\mu} = \delta_{kl}. \quad (7)$$

buildCoefficients

Usage : *buildCoefficients(n)*

Arguments : *n* : an integer

Value : the sequence $(\alpha_i)_{0 \leq i \leq n}$ of the coefficients of the polynomial of degree *n* :

$$P(x) = \sum_{i=0}^n \alpha_i x^i \quad (8)$$

buildRecurrenceCoefficientsCollection

Usage : *buildRecurrenceCoefficientsCollection(n)*

Arguments : *n* : an integer

Value : a NumericalSample, which contains the sequence $(a_{0,i}, a_{1,i}, a_{2,i})_{1 \leq i \leq n}$ of the three-term recurrence relation that defines the polynomials :

$$P_{n+1}(x) = (a_{0,n}x + a_{1,n})P_n(x) + a_{2,n}P_{n-1}(x). \quad (9)$$

10.3.4 StandardDistributionPolynomialFactory

StandardDistributionPolynomialFactory inherits from OrthogonalUniVariatePolynomialFamily.

Usage :

```
StandardDistributionPolynomialFactory(distribution)  
StandardDistributionPolynomialFactory(myOrthoAlgo)
```

Arguments :

myOrthoAlgo : a OrthonormalizationAlgorithm,
distribution : a Distribution of dimension 1, defined by its standard parameters.

Value : a StandardDistributionPolynomialFactory that evaluates the coefficients of the three term linear recurrence (9) that defines the orthonormal polynomials associated to the distribution.

When the algorithm is not specified, the Gram Schmidt algorithm is used with the given distribution. This class detects if the distribution *distribution* directly given or defined in *myOrthoAlgo* is a measure for which the orthonormal polynomials have already been evaluated.

Some methods : The methods are described in the OrthogonalUniVariatePolynomialFamily class.

10.3.5 CharlierFactory

CharlierFactory inherits from OrthogonalUniVariatePolynomialFamily.

Usage : *CharlierFactory*(λ)

Arguments : λ : a real with $\lambda > 0$. By default, $\lambda = 1$.

Value : a CharlierFactory which builds the orthonormalized polynomials with respect to the discrete measure $\mu = \text{Poisson}(\lambda)$ as defined in (7).

Some methods :

getN

Usage : *getLambda*()

Arguments : none

Value : an real, the λ coefficient of the Charlier family.

getMeasure

Usage : *getMeasure*()

Arguments : none

Value : a Distribution, the univariate measure with respect to which the Charlier family is orthonormal. Here the *Poisson*(λ) distribution.

10.3.6 HermiteFactory

HermiteFactory inherits from OrthogonalUniVariatePolynomialFamily.

Usage : *HermiteFactory*()

Arguments : none

Value : an HermiteFactory which builds the orthonormalized polynomials with respect to the continuous measure *Normal*(0, 1) as defined in (7).

Some methods :

getMeasure

Usage : *getMeasure*()

Arguments : none

Value : a Distribution, the univariate measure with respect to which the Charlier family is orthonormal. Here the *Normal*(0, 1) distribution.

10.3.7 JacobiFactory

JacobiFactory inherits from OrthogonalUniVariatePolynomialFamily.

Usage : *JacobiFactory*(α, β)

Arguments : α, β : two NumericalScalars. Both which must be > -1 .

Value : an JacobiFactory which builds the orthonormalized polynomials with respect to the continuous measure *Beta*($\beta + 1, \alpha + \beta + 2, -1, 1$) as defined in (7).

Some methods :

getMeasure

Usage : *getMeasure*()

Arguments : none

Value : a Distribution, the univariate measure with respect to which the Charlier family is orthonormal. Here the *Beta*($\beta + 1, \alpha + \beta + 2, -1, 1$) distribution.

getAlpha

Usage : *getAlpha*()

Arguments : none

Value : a NumericalScalar, the α coefficient of the Jacobi family.

getBeta

Usage : *getBeta*()

Arguments : none

Value : a NumericalScalar, the β coefficient of the Jacobi family.

10.3.8 KrawtchoukFactory

KrawtchoukFactory inherits from OrthogonalUniVariatePolynomialFamily.

Usage : *KrawtchoukFactory*(N, p)

Arguments :

N : an integer, ≥ 1 . By default, $N = 1$.

p : a real, with $0 \leq p \leq 1$. By default, $p = \frac{1}{2}$.

Value : a KrawtchoukFactory which builds the orthonormalized polynomials with respect to the discrete measure $\mu = \text{Binomial}(N, p)$ as defined in (7).

Some methods :

getMeasure

Usage : *getMeasure*()

Arguments : none

Value : a Distribution, the univariate measure with respect to which the Charlier family is orthonormal. Here the *Binomial*(N, p) distribution.

getN

Usage : *getN*()

Arguments : none

Value : an integer, the N coefficient of the Krawtchouk family.

getP

Usage : *getP*()

Arguments : none

Value : a NumericalScalar, the p coefficient of the Krawtchouk family.

10.3.9 LaguerreFactory

LaguerreFactory inherits from OrthogonalUniVariatePolynomialFamily.

Usage : *LaguerreFactory*(k)

Arguments : k : a NumericalScalar which must be > -1 .

Value : an LaguerreFactory which builds the orthonormalized polynomials with respect to the continuous measure $\text{Gamma}(k + 1, 1, 0)$ as defined in (7)..

Some methods :

getMeasure

Usage : *getMeasure*()

Arguments : none

Value : a Distribution, the univariate measure with respect to which the Charlier family is orthonormal. Here the distribution $\text{Gamma}(k + 1, 1, 0)$.

getK

Usage : *getK*()

Arguments : none

Value : a NumericalScalar, the k coefficient of the Laguerre family.

10.3.10 LegendreFactory

LegendreFactory inherits from OrthogonalUniVariatePolynomialFamily.

Usage : *LegendreFactory*()

Arguments : none

Value : an LegendreFactory which builds the orthonormalized polynomials with respect to the continuous measure *Uniform*(-1, 1) distribution as defined in (7).

Some methods :

getMeasure

Usage : *getMeasure*()

Arguments : none

Value : a Distribution, the univariate measure with respect to which the Charlier family is orthonormal. Here the distribution *Uniform*(-1, 1).

10.3.11 MeixnerFactory

MeixnerFactory inherits from OrthogonalUniVariatePolynomialFamily.

Usage : *MeixnerFactory*(r, p)

Arguments :

r : a real value > 0 . By default, $r = 1$.

p : a real, with $0 \leq p \leq 1$. By default, $p = \frac{1}{2}$.

Value : an MeixnerFactory which builds the orthonormalized polynomials with respect to the continuous measure *NegativeBinomial*(1,0.5) distribution as defined in (7).

Some methods :

getR

Usage : *getR*()

Arguments : none

Value : a real value, the r coefficient of the Meixner family.

getP

Usage : *getP*()

Arguments : none

Value : a real value, the p coefficient of the Meixner family.

getMeasure

Usage : *getMeasure*()

Arguments : none

Value : a Distribution, the univariate measure with respect to which the Charlier family is orthonormal. Here the distribution *NegativeBinomial*(1,0.5).

10.3.12 OrthonormalizationAlgorithm

This class defines the algorithm used to build the orthonormal basis with respect to a specified distribution.

Usage :

OrthonormalizationAlgorithm(myOrthoAlgo)

OrthonormalizationAlgorithm(measure)

Arguments :

measure : a Distribution

myOrthoAlgo : a OrthonormalizationAlgorithmImplementation, which is the Chebychev or Gram Schmidt algorithm

value : a OrthonormalizationAlgorithm which enables to build the orthonormal polynomail basis with respect to a distribution.

in the first usage, the algorithm *myOrthoAlgo* is used (that contains the concerned distribution)

in the second usage, we use by default the Gram Schmidt algorithm. Only the distribution *measure* is specified.

10.3.13 ChebychevAlgorithm

This class inherits from OrthonormalizationAlgorithmImplementation.

Usage :

ChebychevAlgorithm(measure)

ChebychevAlgorithm(measure, referenceFamily)

Arguments :

measure : a Distribution

referenceFamily : a OrthogonalUniVariatePolynomialFamily. When not specified, the *referenceFamily* is the canonical one : $(1, X, X^2, \dots)$.

Value : a ChebychevAlgorithm, that builds the orthonormalized polynomials family with respect to the distribution *measure*, where the initial polynomials family is the one specified in *referenceFamily*. It implements the Chebychev algorithm.

Some methods :

getReferenceFamily

Usage : *getReferenceFamily()*

Arguments : *n* : an integer

Value : a OrthogonalUniVariatePolynomialFamily, the polynomials family from which the algorithm starts to build the polynomials family which is orthogonal to the distribution *measure*.

10.3.14 GramSchmidtAlgorithm

This class inherits from OrthonormalizationAlgorithmImplementation.

Usage :

GramSchmidtAlgorithm(measure)

GramSchmidtAlgorithm(measure, referenceFamily)

Arguments :

measure : a Distribution

referenceFamily : a OrthogonalUniVariatePolynomialFamily. When not specified, the *referenceFamily* is the canonical one : $(1, X, X^2, \dots)$.

Value : a GramSchmidtAlgorithm, that builds the orthonormalized polynomials family with respect to the distribuytion *measure*, where the initial polynomials family is the one specified in *referenceFamily*. It implements the Gram Schmidt algorithm.

Some methods :

getReferenceFamily

Usage : *getReferenceFamily()*

Arguments : *n* : an integer

Value : a OrthogonalUniVariatePolynomialFamily, the polynomials family from which the algorithm starts to build the polynomials family which is orthogonal to the distribution *measure*.

10.3.15 EnumerateFunction

Usage :

EnumerateFunction()
EnumerateFunction(dim)

Arguments :

dim : an integer, that represents the dimension of the EnumerateFunction. *dim* must be equal to the dimension of the OrthogonalBasis.

Value : an EnumerateFunction, which is a function that maps \mathbb{N} into \mathbb{N}^{dim} . In the default constructor *dim* = 0.

Some methods :

()

Usage : *inverse(multiIndex)*

Arguments : *multiIndex*: an Indices, which is a collection of integers

Value : an integer, represents the antecedent of the *multiIndex* in the EnumerateFunction.

getDimension

Usage : *getDimension()*

Arguments : none

Value : an integer, the dimension of the EnumerateFunction.

getStrateCardinal

Usage : *getStrateCardinal(strateIndex)*

Arguments : *strateIndex* : an integer, the index of the strate in the hierarchical basis. In the context of product of polynomial basis, this is the total polynomial degree.

Value : the number of members of the basis associated to the strate *strateIndex*. In the context of product of polynomial basis, this is the number of polynoms of the basis which total degree is *strateIndex*.

getStrateCumulatedCardinal

Usage : *getStrateCumulatedCardinal(strateIndex)*

Arguments : *strateIndex* : an integer, the index of the strate in the hierarchical basis. In the context of product of polynomial basis, this is the total polynomial degree.

Value : the number of members of the basis associated to the strates inferior or equal to *strateIndex*. In the context of product of polynomial basis, this is the number of polynoms of the basis which total degree is inferior or equal to *strateIndex*.

Details : *EnumerateFunction* represents a bijection from \mathbb{N} to \mathbb{N}^{dim} . This bijection is based on a particular procedure of enumerating the set of multi-indices. It begins from the multi-index $[0, 0, \dots, 0]$.

We associate a multi-index $[j_{p1}, j_{p2}, \dots, j_{pdim}]$ for every integer i_p in \mathbb{N} :

For more details, let us consider any $i_p, i_q \in \mathbb{N}$: if $|i_p - i_q| \leq 1$ then $|\sum_{k=1}^{dim} (j_{pk} - j_{qk})| \leq 1$. This proposition provides a necessary but insufficient condition for the construction of the bijection. Another

assumption was done indicating the way of iteration. Below an example showing this assumption.
Example: for $dim = 2$,

$$\begin{aligned} \mathit{phi}(0) &= [0 \ 0] \\ \mathit{phi}(1) &= [1 \ 0] \\ \mathit{phi}(2) &= [0 \ 1] \\ \mathit{phi}(3) &= [2 \ 0] \\ \mathit{phi}(4) &= [1 \ 1] \\ \mathit{phi}(5) &= [0 \ 2] \\ \mathit{phi}(6) &= [3 \ 0] \end{aligned}$$

For the functional expansion (respectively polynomial chaos expansion), the multi-index $\overline{i_p}$ represents the collection of degrees of the selected orthogonal functions (respectively orthogonal polynomials). In fact, after the selection of the type of orthogonal functions (respectively orthogonal polynomials) for the construction of the orthogonal basis, the *EnumerateFunction* characterizes the term of the basis by providing the degrees of the univariate functions (respectively univariate polynomials).

inverse

Usage : *inverse(multiIndex)*

Arguments : *multiIndex*: an Indices, which is a collection of integers

Value : an integer, represents the antecedent of the *multiIndex* in the *EnumerateFunction*

getDimension

Usage : *getDimension()*

Arguments : none

Value : an integer, the dimension of the *EnumerateFunction*

10.3.16 LinearEnumerateFunction

LinearEnumerateFunction inherits from EnumerateFunctionImplementation.

Usage :

LinearEnumerateFunction(dim)

Arguments :

dim : an integer, that represents the dimension of the EnumerateFunction. *dim* must be equal to the dimension of the OrthogonalBasis.

Value : a LinearEnumerateFunction

10.3.17 HyperbolicAnisotropicEnumerateFunction

HyperbolicAnisotropicEnumerateFunction inherits from EnumerateFunctionImplementation.

Usage :

HyperbolicAnisotropicEnumerateFunction(dim)
HyperbolicAnisotropicEnumerateFunction(dim, q)
HyperbolicAnisotropicEnumerateFunction(weight)
HyperbolicAnisotropicEnumerateFunction(weight, q)

Arguments :

dim : an UnsignedLong, that represents the dimension of the EnumerateFunction. *dim* must be equal to the dimension of the OrthogonalBasis.

q : a NumericalScalar, the q-quasi-norm parameter.

weight : a NumericalPoint, the weights of the indices in each dimension.

Value : an HyperbolicAnisotropicEnumerateFunction

10.4 Construction of the truncated multivariate orthogonal basis

getStrateCumulatedCardinal

Usage : *getStrateCumulatedCardinal(strateIndex)*

Arguments : *strateIndex* : an integer, the index of the strate in the hierarchical basis. In the context of product of polynomial basis, this is the total polynomial degree.

Value : the number of members of the basis associated to the strates inferior or equal to *strateIndex*. In the context of product of polynomial basis, this is the number of polynomials of the basis which total degree is inferior or equal to *strateIndex*.

Details : *EnumerateFunction* represents a bijection from \mathbb{N} to \mathbb{N}^{dim} . This bijection is based on a particular procedure of enumerating the set of multi-indices. It begins from the multi-index $[0, 0, \dots, 0]$.

We associate a multi-index $[j_{p1}, j_{p2}, \dots, j_{pdim}]$ for every integer i_p in \mathbb{N} :

For more details, let us consider any $i_p, i_q \in \mathbb{N}$: if $|i_p - i_q| \leq 1$ then $|\sum_{k=1}^{dim} (j_{pk} - j_{qk})| \leq 1$. This proposition provides a necessary but insufficient condition for the construction of the bijection. Another assumption was done indicating the way of iteration. Below an example showing this assumption.

Example: for $dim = 2$,

$$\begin{aligned} \mathit{phi}(0) &= [0 \ 0] \\ \mathit{phi}(1) &= [1 \ 0] \\ \mathit{phi}(2) &= [0 \ 1] \\ \mathit{phi}(3) &= [2 \ 0] \\ \mathit{phi}(4) &= [1 \ 1] \\ \mathit{phi}(5) &= [0 \ 2] \\ \mathit{phi}(6) &= [3 \ 0] \end{aligned}$$

For the functional expansion (respectively polynomial chaos expansion), the multi-index i_p represents the collection of degrees of the selected orthogonal functions (respectively orthogonal polynomials). In fact, after the selection of the type of orthogonal functions (respectively orthogonal polynomials) for the construction of the orthogonal basis, the *EnumerateFunction* characterizes the term of the basis by providing the degrees of the univariate functions (respectively univariate polynomials).

10.5 Construction of the truncated multivariate orthogonal basis

These strategies are conceived in such a way to be adapted for other orthogonal expansions (other than polynomial). They provide the strategies of selection of different terms of the basis in which the response surface by functional chaos is expressed. For the moment, their implementation are useful for the polynomial chaos expansion.

10.5.1 AdaptiveStrategy

Usage :

```
AdaptiveStrategy(orthogonalBasis, size)  
AdaptiveStrategy(adaptiveStrategyImplementation)
```

Arguments :

orthogonalBasis : an OrthogonalBasis

size : provides the number of terms of the basis. This first usage has the same implementation as the second (with a FixedStrategy). The difference is that the size of the basis is provided directly to the AdaptiveStrategy. (See below the FixedStrategy)

adaptiveStrategyImplementation : an adaptiveStrategyImplementation which is a FixedStrategy, a SequentialStrategy or a CleaningStrategy.

Value : an AdaptiveStrategy, such as :

in the first usage, the adaptive strategy is by default the FixedStrategy.

in the first usage, the adaptive strategy is specified.

Some methods :

```
getMaximumDimension
```

Usage : *getMaximumDimension()*

Arguments : none

Value : an integer which is the size of the truncated basis .

The *getMaximumDimension* has its associated *setMaximumDimension*.

10.5.2 FixedStrategy

FixedStrategy inherits from AdaptiveStrategyImplementation

Usage :

FixedStrategy(orthogonalBasis, size)

Arguments :

orthogonalBasis : an OrthogonalBasis

size : provides the number of terms of the basis.

Value : a FixedStrategy

Details : it is a fixed strategy in the sense that all terms of the basis are built once and for all : that means they can be determined before the projection in the basis. The basis is determined by iterating the *EnumerateFunction* a *size* - 1 times from the first term $i_1 = [0, \dots, 0]$. In this strategy, we can find the so-called *complete basis*, by setting $size = C_{dim+p}^{dim}$ with p the degree of the chaos expansion.

10.5.3 SequentialStrategy

SequentialStrategy inherits from AdaptiveStrategyImplementation

Usage :

SequentialStrategy(orthogonalBasis, maximumSize)

Arguments :

orthogonalBasis : an OrthogonalBasis

maximumSize : provides the maximum number of terms of the truncated basis.

Value : a SequentialStrategy.

Details : The SequentialStrategy generates, term by term, the basis using the Enumerate function. It begins from index = 0 and continues, till satisfying a convergence criterion (that is the residual of the least squares algorithm in the case of the LeastSquaresStrategy) or till reaching the maximal index of terms generation fixed by the user.

10.5.4 CleaningStrategy

CleaningStrategy inherits from AdaptiveStrategyImplementation

Usage :

```
CleaningStrategy(orthoBas, indexMaxOfBasis)
CleaningStrategy(orthoBas, indexMaxOfBasis, verbose)
CleaningStrategy(orthoBas, indexMaxOfBasis, ...
...maxSizeOfBasis, significanceFactor)
CleaningStrategy(orthoBas, indexMaxOfBasis, ...
...maxSizeOfBasis, significanceFactor, verbose)
```

Arguments :

orthoBas : an OrthogonalBasis

indexMaxOfBasis : an integer, the index maximum that can be used by the *EnumerateFunction* to determine the last term of the basis

maxSizeOfBasis : an integer, this parameter characterizes the *CleaningStrategy*. In fact, it represents the number of efficient coefficients of the basis. Its default value is equal to 20

significanceFactor : an integer, used as a threshold for selection the efficient coefficients of the basis. (the real threshold represents the multiplication of the significanceFactor with the maximum magnitude of the current coefficients determined). Its default value is equal to $1e^{-4}$

verbose : a boolean, needed when we are interested in the online monitoring of the current basis updates. (removed or added coefficients)

Value : a CleaningStrategy

Details : This strategy aims to filter the basis from insignificant terms. After choosing the maximum index and the size of the basis, the strategy calculates an initial basis of, for example, 20 terms, the insignificant coefficients are removed (w.r.t a significance factor) and another coefficients are calculated. This procedure continues till arriving to the *indexMaxOfBasis* of the basis or till satisfying the residual of the least squares algorithm.

10.6 Evaluation of the coefficients

These strategies are conceived in such a way to be adapted for other orthogonal expansions (other than polynomial). They provide different strategies of projection in the orthonormal basis. For the moment, their implementation are useful for the polynomial chaos expansion.

10.6.1 ProjectionStrategy

This class is not usable because it has sense only within the FunctionalChoasAlgorithm.

Usage :

ProjectionStrategy(*projectionStrategyImplementation*)

Arguments :

projectionStrategyImplementation : a ProjectionStrategyImplementation which is a LeastSquaresStrategy or a IntegrationStrategy (detailed further).

Value : a ProjectionStrategy, which is the interface of the ProjectionStrategyImplementation. It represents a generic class (virtual) for different strategies like: LeastSquaresStrategy, IntegrationStrategy.

10.6.2 LeastSquaresStrategy

This class inherits from ProjectionStrategyImplementation.

This class is not usable because it has sense only within the FunctionalChaosAlgorithm : the least squares strategy evaluates the coefficients $(\alpha_k)_{k \in K}$ of the polynomials decomposition as follows :

$$\underline{\alpha} = \underset{\alpha \in \mathbb{R}^K}{\operatorname{argmin}} E_{\mu} \left[\left(g \circ T^{-1}(\underline{Z}) - \sum_{k \in K} \alpha_k \Psi_k(\underline{Z}) \right)^2 \right] \quad (10)$$

where $\underline{Z} = T(\underline{X})$.

Then, the esperance E_{μ} is approximated by a relation of type :

$$E_{\mu} [f(\underline{Z})] \simeq \sum_{i \in I} \omega_i f(\Xi_i) \quad (11)$$

where f is a function $L_1(\mu)$ defined as :

$$f(\underline{Z}) = \left(g \circ T^{-1}(\underline{Z}) - \sum_{k \in K} \alpha_k \Psi_k(\underline{Z}) \right)^2 \quad (12)$$

In the approximation (14), the set I , the points $(\Xi_i)_{i \in I}$ and the weights $(\omega_i)_{i \in I}$ are evaluated from different methods implemented in OpenTURNS in the *WeightedExperiment*.

Usage :

```
LeastSquaresStrategy(weightedExperiment)
LeastSquaresStrategy(weightedExp, approxAlgoImpFact)
LeastSquaresStrategy(measure, approxAlgoImpFact)
LeastSquaresStrategy(measure, weightedExp, approxAlgoImpFact)
LeastSquaresStrategy(inputSample, weights, outputSample, approxAlgoImpFact)
```

Arguments :

weightedExp : a WeightedExperiment, the experimental design used for the transformed input data.
When not precised, Open TURNS uses a *MonteCarloExperiment*.

approxAlgoImpFact : an ApproximationAlgorithmImplementationFactory, the factory that builds the desired ApproximationAlgorithm. When not precised, Open TURNS uses the *PenalizedLeastSquaresAlgorithmFactory*.

measure : the Distribution μ with respect to which the basis is orthonormal. When not precised, Open TURNS uses the limit peasure defined within the WeightedExperiment.

inputSample, outputSample : two NumericalSample that describe the model.

weights : a NumericalPoint that are the weights associated to the input sample points such that the corresponding WeightedExperiment is a good approximation of μ .

Value : a LeastSquaresStrategy used to create a *FunctionalChaosAlgorithm*.

10.6.3 IntegrationStrategy

This class inherits from ProjectionStrategyImplementation.

This class is not usable because it has sense only within the FunctionalChaosAlgorithm : the integration strategy evaluates the coefficients $(\alpha_k)_{k \in K}$ of the polynomials decomposition as follows :

$$\underline{\alpha} = (E_\mu [g \circ T^{-1}(\underline{Z})\Psi_k(\underline{Z})])_k \quad (13)$$

where $\underline{Z} = T(\underline{X})$.

Then, the esperance E_μ is approximated by a relation of type :

$$E_\mu [f(\underline{Z})] \simeq \sum_{i \in I} \omega_i f(\xi_i) \quad (14)$$

where f is a function $L_1(\mu)$ defined as :

$$f(\underline{Z}) = g \circ T^{-1}(\underline{Z})\Psi_k(\underline{Z}) \quad (15)$$

In the approximation (14), the set I , the points $(\xi_i)_{i \in I}$ and the weights $(\omega_i)_{i \in I}$ are evaluated from different methods implemented in OpenTURNS in the *WeightedExperiment*.

Usage :

```
IntegrationStrategy(measure)
IntegrationStrategy(weightedExperiment)
IntegrationStrategy(measure, weightedExp)
IntegrationStrategy(inputSample, outputSample)
```

Arguments :

weightedExp : a WeightedExperiment, the experimental design used for the transformed input data.
When not precised, Open TURNS uses a *MonteCarloExperiment*.

approxAlgoImpFact : an ApproximationAlgorithmImplementationFactory, the factory that builds the desired ApproximationAlgorithm. When not precised, Open TURNS uses the *PenalizedLeastSquaresAlgorithmFactory*.

measure : the Distribution μ with respect to which the basis is orthonormal. When not precised, Open TURNS uses the limit peasure defined within the WeightedExperiment.

inputSample, outputSample : two NumericalSample that describe the model.

weights : a NumericalPoint that are the weights associated to the input sample points such that the corresponding WeightedExperiment is a good approximation of μ .

Value : a IntegrationStrategy used to create a *FunctionalChaosAlgorithm*.

10.6.4 ApproximationAlgorithmFactory

Usage :

ApproximationAlgorithmFactory(approxAlgoImpFact)

Arguments :

approxAlgoImpFact : an ApproximationAlgorithmImplementationFactory (detailed later).

Value : an ApproximationAlgorithmFactory, which is the interface of the ApproximationAlgorithmImplementationFactory. It represents a generic class (virtual) for different factories like: PenalizedLeastSquaresAlgorithmFactory, PenalizedLeastSquaresAlgorithmFactory ...

Some methods : This class is not usable because it has sense only within the FunctionalChaosAlgorithm.

10.6.5 PenalizedLeastSquaresAlgorithmFactory

PenalizedLeastSquaresAlgorithmFactory inherits from ApproximationAlgorithmImplementationFactory.

Usage :

PenalizedLeastSquaresAlgorithmFactory()

Value : a PenalizedLeastSquaresAlgorithmFactory, implementation of ApproximationAlgorithmFactory which builds an ApproximationAlgorithmImplementation.

Some methods : This class is not usable because it has sense only within the FunctionalChoasAlgorithm.

10.6.6 LeastSquaresMetaModelSelectionFactory

LeastSquaresMetaModelSelectionFactory inherits from ApproximationAlgorithmImplementationFactory.

Usage :

LeastSquaresMetaModelSelectionFactory()

LeastSquaresMetaModelSelectionFactory(basisSequenceFactory)

LeastSquaresMetaModelSelectionFactory(basisSequenceFactory, fittingAlgorithm)

Arguments :

basisSequenceFactory : a BasisSequenceFactory (detailed later).

fittingAlgorithm : a FittingAlgorithm (detailed later).

Value : a LeastSquaresMetaModelSelectionFactory, implementation of ApproximationAlgorithmImplementationFactory which builds an ApproximationAlgorithm.

Some methods : This class is not usable because it has sense only within the FunctionalChaosAlgorithm.

10.6.7 BasisSequenceFactory

Usage :

BasisSequenceFactory(basisSequenceFactoryImplementation)

Arguments :

basisSequenceFactoryImplementation : a BasisSequenceFactoryImplementation (detailed later).

Value : a BasisSequenceFactory, which is the interface of the BasisSequenceFactoryImplementation. It represents a generic class (virtual) for different strategies like: LAR.

Some methods : This class is not usable because it has sense only within the FunctionalChoasAlgorithm.

10.6.8 LAR

LAR inherits from BasisSequenceFactory.

Usage :

LAR()

Value : a LAR

Some methods : This class is not usable because it has sense only within the FunctionalChoasAlgorithm.

10.6.9 FittingAlgorithm

Usage :

FittingAlgorithm(fittingAlgorithmImplementation)

Arguments :

fittingAlgorithmImplementation : a FittingAlgorithmImplementation (detailed later).

Value : a FittingAlgorithm, which is the interface of the FittingAlgorithmImplementation. It represents a generic class (virtual) for different cross-validation algorithms like: CorrectedLeaveOneOut, KFold...

Some methods : This class is not usable because it has sense only within the FunctionalChaosAlgorithm.

10.6.10 CorrectedLeaveOneOut

CorrectedLeaveOneOut inherits from FittingAlgorithmImplementation.

Usage :

CorrectedLeaveOneOut()

Value : a CorrectedLeaveOneOut

Some methods : This class is not usable because it has sense only within BasisSequenceFactory.

10.6.11 KFold

KFold inherits from FittingAlgorithmImplementation.

Usage :

KFold()

KFold(k)

Arguments :

k : an integer, decides the number of folds in which the sample is splitted.

Value : a KFold

Some methods : This class is not usable because it has sense only within the FunctionalChaosAlgorithm.

10.7 FunctionalChaosRandomVector

This structure is created from a FunctionalChaosResult in order to evaluate the Sobol indices associated to the polynomial chaos decomposition of the model.

Usage : *FunctionalChaosRandomVector(FunctionalChaosResult)*

Arguments : *functionalChaosResult* : a FunctionalChaosResult resulting from a polynomial chaos decomposition of the model

Value : a FunctionalChaosRandomVector

Some methods :

getMean

Usage : *getMean()*

Arguments : none

Value : a NumericalPoint, the mean of the random vector defined as the result of the polynomial chaos approximation. The mean is the first coefficient (constant term) of the polynomial decomposition.

getCovariance

Usage : *getCovariance()*

Arguments : none

Value : a CovarianceMatrix, the covariance of the random vector defined as the result of the polynomial chaos approximation. The covariance is the sum of the square coefficients except the first one (constant term) minus the square of the constant term of the polynomial decomposition.

getFunctionalChaoResult

Usage : *getFunctionalChaoResult()*

Arguments : none

Value : the FunctionalChaosResult resulting from the polynomial chaos decomposition of the model

getSobolIndex

Usage :

getSobolIndex(variableIndices)

getSobolIndex(variableIndex)

Arguments :

variableIndices : a Indices, indicating the set of variables for which we want the associated Sobol indice

variableIndex : a UnsignedLong indicating the variable for which we want the associated Sobol indices

Value : a NumericalSaclar, the Sobol indice

getTotalSobolIndex

Usage :

getSobolTotalIndex(variableIndices)

getSobolTotalIndex(variableIndex)

Arguments :

variableIndices : a Indices, indicating the variables for which we want the associated Sobol indices

variableIndex : a UnsignedLong indicating the variable for which we want the associated Sobol indices

Value : a NumericalSaclar, the Total Sobol indice

11 Statistics on sample

11.1 Numerical Sample

11.1.1 NumericalComplexCollection

Usage :

NumericalComplexCollection(size)

NumericalComplexCollection(pythonList)

Arguments :

size : an integer, the size of the NumericalSample

pythonList : a list python of dimension 2. For example : [(1.0, 2.0), (3.0, 4.0)].

Value : a NumericalComplexCollection, containing *size* NumericalPoint, each one equal to :

in the first usage : *size* complex points (0, 0),

in the second usage, the list of the complex points mentioned in *pythonList*.

Some methods :

[]

Usage : *NumericalComplexCollection[i]*

Arguments : *i* : an integer, constraint : $0 \leq i \leq size - 1$

Value : the complex point of indice *i*.

11.1.2 NumericalSample

Usage :

NumericalSample(size, dim)
NumericalSample(size, numericalPoint)
NumericalSample(sequence)
NumericalSample(array)

Arguments :

size : an integer, the size of the NumericalSample
dim : an integer, the dimension each NumericalPoint of the NumericalSample
numericalPoint : a NumericalPoint
sequence : a tuple / list python of dimension 2. For example : [(1.0, 2.0), (3.0, 4.0)].
array: a Numpy array with dimension (ndim) 2

Value : a NumericalSample, containing *size* NumericalPoint, each one equal to :

$\underline{0} \in \mathbb{R}^{dim}$ in the first usage
numericalPoint in the second usage

Some methods :

[]

Usage : *NumericalSample[i]*

Arguments : *i* : an integer, constraint : $0 \leq i \leq size - 1$

Value : a NumericalPoint, the $(i + 1)$ - th NumericalPoint of the NumericalSample

[,]

Usage : *NumericalSample[i, j]*

Arguments : *i, j* : integers, constraint : $0 \leq i, j \leq size - 1$

Value : a real, the $(j + 1)$ - th component of the $(i + 1)$ - th NumericalPoint of the NumericalSample

add

Usage : *add(x)*

Arguments : *x* : a NumericalPoint

Value : a NumericalSample, of size $size + 1$ where the last NumericalPoint has been added, equal to *x*

computeCovariance

Usage : *computeCovariance()*

Arguments : none

Value : a CovarianceMatrix, the covariance matrix of the NumericalSample (a dim^2 matrix)

computeEmpiricalCDF

Usage :

computeEmpiricalCDF(x)

computeEmpiricalCDF(x, tail)

Arguments :

x : a NumericalPoint

tail : a Boolean, which indicates whether we compute the tail CDF (if *tail = True*) or the CDF at point *x* (if *tail = False*). The CDF at point \underline{x} is $\mathbb{P}(X_1 \leq x_1, \dots, X_n \leq x_n)$ and the tail CDF at point \underline{x} is $\mathbb{P}(X_1 > x_1, \dots, X_n > x_n)$. If *tail = True*, *computeEmpiricalCDF* evaluates the tail CDF. If not specified, *tail = False*.

Value : a numerical scalar, the Empirical Cumulative Distribution Function value of the NumericalSample at *x*

computeKendallTau

Usage : *computeKendallTau()*

Arguments : none

Value : a CorrelationMatrix, the Kendall rank correlation matrix of the NumericalSample

computeKurtosisPerComponent

Usage : *computeKurtosisPerComponent()*

Arguments : none

Value : a NumericalPoint, the value of the kurtosis of each component of the NumericalSample

computeMean

Usage : *computeMean()*

Arguments : none

Value : a NumericalPoint, the mean value vector of each component of the NumericalSample

computeMedianPerComponent

Usage : *computeMedianPerComponent()*

Arguments : none

Value : a NumericalPoint, the median value vector of each component of the NumericalSample

computePearsonCorrelation

Usage : *computePearsonCorrelation()*

Arguments : none

Value : a CorrelationMatrix, the Pearson correlation matrix of the NumericalSample (a dim^2 matrix)

*computeQuantile***Usage :** *computeQuantile(p)***Arguments :** *p*, a real value, constraint $0 \leq p \leq 1$, the value of a probability**Value :** a NumericalPoint, the empirical quantile value associated to probability *p*, determined from the empirical CDF of the NumericalSample as follows :

- $\forall q \in [\frac{1}{2n}, 1 - \frac{1}{2n}]$, then Open TURNS approximates the empirical cumulative density function by interpolating all the middles of the steps and then evaluates x_q from this continuous approximation.
- $\forall q \leq \frac{1}{2n}$, then Open TURNS returns $\min(X_i)$.
- $\forall q > 1 - \frac{1}{2n}$, then Open TURNS returns $\max(X_i)$.

*computeQuantilePerComponent***Usage :** *computeQuantilePerComponent(p)***Arguments :** *p*, a real value, constraint $0 \leq p \leq 1$, the value of a probability**Value :** a NumericalPoint, the empirical quantile value associated to probability *p* for each component, determined from the empirical CDF of each component of the NumericalSample*computeSkewnessPerComponent***Usage :** *computeSkewnessPerComponent()***Arguments :** none**Value :** a NumericalPoint, the skewness of each component of the NumericalSample*computeSpearmanCorrelation***Usage :** *computeSpearmanCorrelation()***Arguments :** none**Value :** a CorrelationMatrix, the Spearman correlation matrix of the NumericalSample (a dim^2 matrix)*computeStandardDeviation***Usage :** *computeStandardDeviation()***Arguments :** none**Value :** a SquareMatrix, the Cholesky factor \underline{L} of the covariance matrix $\underline{\Lambda}$: $\underline{L}\underline{L}^t = \underline{\Lambda}$, with \underline{L} triangular inferior*computeStandardDeviationPerComponent***Usage :** *computeStandardDeviationPerComponent()***Arguments :** none**Value :** a NumericalPoint, the standard Deviation value of each component of the NumericalSample*computeVariancePerComponent***Usage :** *computeVariancePerComponent()***Arguments :** none

Value : a NumericalPoint, the variance value of each component of the NumericalSample

getDimension

Usage : *getDimension()*

Arguments : none

Value : an integer, the dimension of each point which constitutes the NumericalSample (it returns *dim*)

getMin

Usage : *getMin()*

Arguments : none

Value : a NumericalPoint, each element of the NumericalPoint corresponds to the minimum of each component of the NumericalSample

getMax

Usage : *getMax()*

Arguments : none

Value : a NumericalPoint, each element of the NumericalPoint corresponds to the maximum of each component of the NumericalSample

getMarginal

Usage :

getMarginal(i)

getMarginal(indices)

Arguments :

i : a UnsignedLong, (integer)

indices : a Indices (collection of integers)

Value :

a NumericalSample : the NumericalSample of same size as the initial NumericalSample, of dimension 1, corresponding to the $i + 1$ coordinate of the NumericalPoints which constitute the initial NumericalSample

a NumericalSample : the NumericalSample of same size as the initial NumericalSample, of dimension *indices.getSize()*, corresponding to the associated coordinates of the NumericalPoints which constitute the initial NumericalSample. Care : indices start at 0.

getSize

Usage : *getSize()*

Arguments : none

Value : an integer, the size of the NumericalSample (which means the number of points which constitute the NumericalSample (it returns *size*))

rank

Usage : *rank()*

Arguments : none

Value : a NumericalSample, where each value has been replaced by the value of its rank (order set component by component)

scale

Usage : *scale(factor)*

Arguments : *factor* : a NumericalPoint

Value : a NumericalSample, where the components [i] of each NumericalPoint have been multiplied by the corresponding value *factor*[i]

sort

Usage : *sort(i)*

Arguments : *i* : UnsignedLong (an integer)

Value : a NumericalSample of same size as the initial NumericalSample, of dimension 1, constituted by the *i* + 1th component of the NumericalPoints that constitute the initial NumericalSample, all sorted in ascending order

sortAccordingAComponent

Usage : *sortAccordingAComponent(i)*

Arguments : *i* : UnsignedLong (an integer)

Value : a NumericalSample of same size and dimension as the initial NumericalSample, where the NumericalPoints have been reordered such that the (*i* + 1) component is sorted in ascending order

split

Usage : *split(i)*

Arguments : *i* : UnsignedLong (an integer)

Value : a NumericalSample. The initial NumericalSample has been modified and restricted to the *i* first NumericalPoints. The new NumericalSample contains the *size* - *i* last NumericalPoints of the initial NumericalPoints if size is the initial size of the sample. For example : initial= (1, 2, 3, 4) and newSplit = initial.split(1). Then initial=(1) and newSplit=(2,3,4).

str

Usage : *str()*

Arguments : none

Value : a string giving a brief description of the considered NumericalSample,

translate

Usage : *translate(translation)*

Arguments : *translation* : a NumericalPoint

Value : a NumericalSample, where the components [i] of each NumericalPoint have been added the corresponding value *translation*[i]

Details :

when two elements of the sample are equal, the rank of the first element appearing in the sample will be considered as the lower one (for computing Spearman correlation matrix)

11.2 Distribution factory

11.2.1 DistributionImplementationFactory

Usage : *DistributionImplementationFactory*()

Arguments : none

Value : a *DistributionImplementationFactory* is the implementation of the factory of one particular distribution.

Some methods :

build

Usage :

build(sample)

build(param)

build(sample, covarianceMat)

Arguments :

sample : a *NumericalSample*, of dimension $n \geq 1$

param : a *NumericalPointWithDescription*, the vector of parameters of the distribution.

covarianceMat : a *CovarianceMatrix*

Value : a *DistributionImplementation*, which is the implementation of the factory of one particular distribution. In the second usage, the covariance matrix *covarianceMat* is fulfilled with the covariance of the estimator of the parameter vector θ . The technique used is bootstrap. In case of asymptotical normal convergence of the estimator, it enables to build confidence intervals.

11.3 Correlation analysis

11.3.1 CorrelationAnalysis

Usage : *CorrelationAnalysis()*

Arguments : none

Some methods :

PCC

Usage : *PCC(sample1, sample2)*

Arguments :

sample1 : a NumericalSample, of dimension $n \geq 2$

sample2 : a NumericalSample, of dimension =1

Value : a NumericalPoint, the PCC (Partial Correlation Coefficient) coefficients evaluated between the *sample2* and each coordinate of *sample1*

PRCC

Usage : *PRCC(sample1, sample2)*

Arguments :

sample1 : a NumericalSample, of dimension $n \geq 2$

sample2 : a NumericalSample, of dimension =1

Value : a NumericalPoint, the PRCC (Partial Rank Correlation Coefficient) coefficients evaluated between the *sample2* and each coordinate of *sample1* (based on the rank values)

PearsonCorrelation

Usage : *PearsonCorrelation(sample1, sample2)*

Arguments :

sample1 : a NumericalSample, of dimension = 1

sample2 : a NumericalSample, of dimension = 1

Value : a real value, the Pearson Correlation coefficient evaluated between the *sample2* and *sample1*

SRC

Usage : *SRC(sample1, sample2)*

Arguments :

sample1 : a NumericalSample, of dimension $n \geq 1$

sample2 : a NumericalSample, of dimension = 1

Value : a NumericalPoint, the SRC (Standard Regression Coefficient) coefficients evaluated between the *sample2* and each coordinate of *sample1*

SRRC

Usage : *PRCC(sample1, sample2)*

Arguments :

sample1 : a NumericalSample, of dimension $n \geq 1$

sample2 : a NumericalSample, of dimension = 1

Value : a NumericalPoint, the SRRC (Standard Rank Regression Coefficient) coefficients evaluated between the *sample2* and each coordinate of *sample1* (based on the rank values)

SpearmanCorrelation

Usage : *SpearmanCorrelation(sample1, sample2)*

Arguments :

sample1 : a NumericalSample, of dimension = 1

sample2 : a NumericalSample, of dimension = 1

Value : a real value, the Spearman Correlation coefficient evaluated between the *sample2* and *sample1* (based on the rank values)

11.4 Sensitivity Analysis

11.4.1 SensitivityAnalysis

SensitivityAnalysis allows to compute the Sobol' sensitivity indices.

Usage :

SensitivityAnalysis(inputSample1, inputSample2, function)

Arguments :

inputSample1 : an input NumericalSample, which marginals are independently distributed

inputSample2 : an input NumericalSample of same dimension and size, independent from *inputSample1*

function : a NumericalMathFunction to be evaluated on the input samples

Details : Some indices are computed together when calling the corresponding accessors. The cost is $N(k+2)$ model evaluations for first and total order indices, or $N(2k+2)$ for first, second and total order indices. Be sure to retrieve higher order indices first to avoid extra model evaluations, i.e. call *getSecondOrderIndices* before *getFirstOrderIndices* if you need second order indices.

Some methods :

setBlockSize

Usage :

setBlockSize(k)

Arguments : *k* : an integer, the size of each block the sample is splitted into, this allows to save space while allowing multithreading, when available (wrapper function), set by default to 1

getFirstOrderIndices()

Usage :

getFirstOrderIndices()

getFirstOrderIndices(i)

Arguments : *i* : an integer, the index of the marginal of the function, equal to 0 by default

Value : a NumericalPoint containing first order Sobol' indices

getSecondOrderIndices()

Usage :

getSecondOrderIndices()

getSecondOrderIndices(i)

Arguments : *i* : an integer, the index of the marginal of the function, equal to 0 by default

Value : a SymmetricMatrix containing second order Sobol' indices

getTotalOrderIndices()

Usage :

getTotalOrderIndices()

getTotalOrderIndices(k)

Arguments : *i* : an integer, the index of the marginal of the function, equal to 0 by default

Value : a NumericalPoint containing total order Sobol' indices

11.5 Fitting test

11.5.1 TestResult

Usage : a TestResult is the result of a fitting test, of type NormalityTest or HypothesisTest.

Some methods :

getBinaryQualityMeasure

Usage : *getBinaryQualityMeasure()*

Arguments : none

Value : a boolean value, indicating the succes of the test : 1 for succes and 0 for failure

getPValue

Usage : *getPValue()*

Arguments : none

Value : a real positive value, <1 , the p-value of the test

getTestType

Usage : *getTestType()*

Arguments : none

Value : a string describing the type of the test

getThreshold

Usage : *getThreshold()*

Arguments : none

Value : a real positive value, < 1 , the p-value threshold

11.5.2 FittingTest

This class is used through its static methods in order to evaluate the adequation of samples to particular distributions.

Some methods :

BestModelBIC

Usage :

FittingTest.BestModelBIC(sample, factoryCollection)

FittingTest.BestModelBIC(sample, distributionCollection)

Arguments :

sample : a NumericalSample, the sample which will be tested

factoryCollection : a FactoryCollection, the collection of factories which are the structures which build the distribution from a sample

distributionCollection : a DistributionCollection, a collection of the distributions which will be tested through the BIC criteria

Value : a Distribution, the best one according to the BIC criteria

BestModelChiSquared

Usage :

FittingTest.BestModelChiSquared(sample, factoryCollection)

FittingTest.BestModelChiSquared(sample, distributionCollection)

Arguments :

sample : a NumericalSample, the sample which will be tested

factoryCollection : a FactoryCollection, a collection of the factories which are the structures which build distributions from a sample

distributionCollection : a DistributionCollection, the collection of the distributions which will be tested through the Chi Squared criteria

Value : a Distribution, the best one according to the ChiSquared criteria

BestModelKolmogorov

Usage :

FittingTest.BestModelKolmogorov(sample, factoryCollection)

FittingTest.BestModelKolmogorov(sample, distributionCollection)

Arguments :

sample : a NumericalSample, the sample which will be tested

factoryCollection : a FactoryCollection, a collection of the factories which are the structures which build distributions from a sample

distributionCollection : a DistributionCollection, the collection of the distributions which will be tested through the Kolmogorov criteria

Value : a Distribution, the best one according to the Kolmogorov criteria

BIC

Usage :

FittingTest.BIC(sample, factory)
FittingTest.BIC(sample, distribution)

Arguments :

sample : a NumericalSample, the sample which will be tested
factory : a Factory, the structure which builds the distribution from the sample which will be tested
distribution : a Distribution, which will be tested through the BIC criteria

Value : a real value, the BIC value of the distribution tested evaluated on the sample

*ChiSquared***Usage :**

FittingTest.ChiSquared(sample, factory, level)
FittingTest.ChiSquared(sample, distribution, level)

Arguments :

sample : a NumericalSample, the sample which will be tested
factory : a Factory, the structure which builds the distribution from the sample which will be tested
distribution : a Distribution, which will be tested through the ChiSquared criteria
level : a real value, constraint : $0 < level < 1$, such as $1 - level$ be the first type error of the fitting test (the probability you reject the distribution tested whereas you should not have). If not fulfilled, by default, $level = 0.95$.

Value : a TestResult, the structure which contains the result of the ChiSquared Test : the first usage tests a type of distribution, the second one tests a particular distribution

*Kolmogorov***Usage :**

FittingTest.Kolmogorov(sample, factory, level)
FittingTest.Kolmogorov(sample, distribution, level)

Arguments :

sample : a NumericalSample, the sample which will be tested
factory : a Factory, the structure which builds the distribution from the sample which will be tested
distribution : a Distribution, which will be tested through the Kolmogorov criteria
level : a real value, constraint : $0 < level < 1$, such as $1 - level$ be the first type error of the fitting test (the probability you reject the distribution tested whereas you should not have). If not fulfilled, by default, $level = 0.95$.

Value : a TestResult, the structure which contains the result of the Kolmogorov Test : the first usage tests a type of distribution, the second one tests a particular distribution

GetLastResult() This method gives access to the TestResult associated to the best suited distribution, among those tested. Thus, its method can be called after the methods *BestModelKolmogorov()* and *BestModelChiSquared()* have been applied. As the *BestModelBIC()* method is not based on a statistical test, it does not produce a TestResult object and the *GetLastResult()* can not be called.

11.5.3 VisualTest

This class is used through its static methods in order to graphically evaluate some hypothesis on samples : independence or monotonous relation.

Some methods :

DrawClouds

Usage :

VisualTest.DrawClouds(sample1, dist)
VisualTest.DrawClouds(sample1, sample2)

Arguments :

sample1 : a NumericalSample, drawn on the graph
sample2 : a NumericalSample, drawn on the graph
dist : a Distribution, the distribution which pdf is drawn

Value : a Graph, the structure which contains : one curve (pdf) and a cloud for the first usage or two clouds in the second usage

DrawCobWeb

Usage :

*VisualTest.DrawCobWeb(inputSample, outputSample, ...
 ...minValue, maxValue, color)*
*VisualTest.DrawCobWeb(inputSample, outputSample, ...
 ...minValue, maxValue, color, quantileScaleBoolean)*

Arguments :

inputSample : a NumericalSample of dimension n , the sample of the X data,
outputSample : a NumericalSample of dimension 1, the sample of the Y data,
minValue : a real valued, which must be in $[0, 1]$ if *quantileScaleBoolean* = *True* or not specified,
maxValue : a real valued, such that $maxValue \geq minValue$ and which must be in $[0, 1]$ if *quantileScaleBoolean* = *True* or not specified,
color : a String, the color of the specified curves
quantileScaleBoolean : a Boolean which indicates the scale of the *minValue* and *maxValue*.
 If *quantileScaleBoolean* = *True*, they are expressed in the rank based scale; otherwise, they are expressed in the Y -values scale. By default, *quantileScaleBoolean* = *True*.

Value : a Graph, the structure which contains the graph : one line for each point and the specific lines colored in *color*.

DrawEmpiricalCDF

Usage : *VisualTest.DrawEmpiricalCDF(sample, xMin, xMax)*

Arguments :

sample : a NumericalSample, drawn on the graph
xMin : a real value, the lower boundary of the graph
xMax : a real value, the upper boundary of the graph, must be $> xMin$

Value : a Graph, the structure which contains : one staircase curve which is the empirical cdf of the sample

DrawHenryLine

Usage : *VisualTest.DrawHenryLine(sample)*

Arguments : *sample* : a NumericalSample, drawn on the graph

Value : a Graph, the structure which contains the graph : one line (the first diagonal corresponding to the normal distribution (0,1)) and a cloud corresponding to the sample

DrawHistogram

Usage :

VisualTest.DrawHistogram(sample, barNumber)

VisualTest.DrawHistogram(sample)

Arguments :

sample : a NumericalSample which histogram is drawn

barNumber : an integer, the number of barplots used to draw the histogram. If not specified, it is automatically determined by Open TURNS according to the Silverman rule

Value : a Graph, the structure which contains the graph : one curve and a cloud for the first usage or 2 clouds for the second one

DrawKendallPlot

Usage :

VisualTest.DrawKendallPlot(data, copula)

VisualTest.DrawKendallPlot(sample1, sample2)

Arguments :

data : a bidimensional NumericalSample

copula : a bidimensional copula that we want to test visually as a possible dependence structure for the data.

sample1 : a bidimensional NumericalSample

sample2 : another bidimensional NumericalSample. We want to test visually if these two samples shares the same copula.

Value : a Graph, the structure which contains the graph : the Kendall curve that we compare to the main diagonal. The farther the curve is from the diagonal, the more different are the two dependence structures.

DrawLinearModelResidualTest

Usage : *VisualTest.DrawLinearModelResidualTest(sample1, sample2,linearModel)*

Arguments :

sample1 : a NumericalSample, X , of dimension 1

sample2 : a NumericalSample, Y , of dimension 1, which is scalar described by a linear model from $X : Y = aX + b$, a and b real values

linearModel : a LinearModel, the regression model

Value : a Graph, the structure which contains the cloud of the residual values, couples (residual i , residual $i+1$)

DrawLinearModelVisualTest

Usage : *VisualTest.DrawLinearModelVisualTest(sample1, sample2,linearModel)*

Arguments :

sample1 : a NumericalSample, X , of dimension 1

sample2 : a NumericalSample, Y , of dimension 1, which is described by a linear model from $X : Y = aX + b$, a and b real values

linearModel : a LinearModel, the regression model

Value : a Graph, the structure which contains the cloud of points (X_i, Y_i) and the linear model line $Y = aX + b$

DrawQQplot

Usage :

VisualTest.DrawQQplot(sample1, sample2)

VisualTest.DrawQQplot(sample1, sample2, pointNumber)

VisualTest.DrawQQplot(sample1, distribution)

VisualTest.DrawQQplot(sample1, distribution, pointNumber)

Arguments :

sample1 : a NumericalSample, used to build on the graph

sample2 : a NumericalSample, used to build the graph

distribution : a Distribution, the distribution which pdf is drawn

pointNumber : an integer, the number of points used to build the graph, equal to 20 by default

Value : a Graph, the structure which contains the corresponding empirical fractiles between the two samples in the two first usages, or between the sample and the distribution in the two last usages

11.5.4 NormalityTest

This class is used through its static methods in order to evaluate whether the sample follows a normal distribution. These two tests give more importance to extreme values.

Some methods :

AndersonDarlingNormal

Usage :

Normality.AndersonDarlingNormal(sample)

Normality.AndersonDarlingNormal(sample, level)

Arguments :

sample : a NumericalSample, of dimension 1 : the sample tested

level : a positive real value, the threshold p-value of the test (= 1- first type risk), must be < 1, equal to 0.95 by default

Value : a TestResult, the structure which contains the result of the test.

Details : the AndersonDarlingNormal Test is used to check whether the sample follows a normal distribution. This test gives more importance to extreme values

CramerVonMisesNormal

Usage :

Normality.CramerVonMisesNormal(sample)

Normality.CramerVonMisesNormal(sample, level)

Arguments :

sample : a NumericalSample, of dimension 1 : the sample tested

level : a positive real value, the threshold p-value of the test (= 1- first type risk), must be < 1, equal to 0.95 by default

Value : a TestResult, the structure which contains the result of the test.

Details : the CramerVonMisesNormal Test is used to check whether the sample follows a normal distribution. This test gives more importance to extreme values

11.5.5 HypothesisTest

This class is used through its static methods in order to evaluate some hypothesis on samples : independence or monotonous relation.

Some methods :

ChiSquared

Usage :

HypothesisTest.ChiSquared(firstSample, secondSample)

HypothesisTest.ChiSquared(firstSample, secondSample, level)

Arguments :

firstSample : a NumericalSample, of dimension 1 : the first sample tested

secondSample : a NumericalSample, of dimension 1 : the second sample tested

level : a positive real value, the threshold p-value of the test (= 1- first type risk), must be < 1, equal to 0.95 by default

Value : a TestResult, the structure which contains the result of the test

Details : the ChiSquared Test is used to check whether two discrete samples are independent

FullPearson

Usage :

HypothesisTest.FullPearson(firstSample, secondSample)

HypothesisTest.FullPearson(firstSample, secondSample, level)

Arguments :

firstSample : a NumericalSample, of dimension $n \geq 1$: the first sample tested

secondSample : a NumericalSample, of dimension 1 : the second sample tested

level : a positive real value, the threshold p-value of the test (= 1- first type risk), must be < 1, equal to 0.95 by default

Value : a TestResultCollection, the structure which contains the results of the successive tests.

Details : the FullPearson Test is the independence Pearson 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), FullPearson performs the Independence Pearson test simultaneously on firstSample[i] and secondSample. For all i, it is supposed that the couple (firstSample[i] and secondSample) is issued from a gaussian vector.

FullRegression

Usage :

HypothesisTest.FullRegression(firstSample, secondSample)

HypothesisTest.FullRegression(firstSample, secondSample, level)

Arguments :

firstSample : a NumericalSample, of dimension $n \geq 1$: the first sample tested

secondSample : a NumericalSample, of dimension 1 : the second sample tested

level : a positive real value, the threshold p-value of the test (= 1- first type risk), must be < 1, equal to 0.95 by default.

Value : a `TestResultCollection`, the structure which contains the results of the successive tests.

Details : the `FullRegression` Test is used to check the quality of the linear regression model between two samples : `firstSample` of dimension n and `secondSample` of dimension 1. If `firstSample[i]` is the numerical sample extracted from `firstSample` (i th coordinate of each point of the numerical sample), `FullRegression` performs the linear regression test simultaneously on all `firstSample[i]` and `secondSample`. The linear regression test tests if the linear regression model between two scalar numerical samples is significant. It is based on the deviation analysis of the regression. The Fisher distribution is used.

FullSpearman

Usage :

HypothesisTest.FullSpearman(firstSample, secondSample)
HypothesisTest.FullSpearman(firstSample, secondSample, level)

Arguments :

firstSample : a `NumericalSample`, of dimension $n \geq 1$: the first sample tested
secondSample : a `NumericalSample`, of dimension 1 : the second sample tested
level : a positive real value, the threshold p-value of the test (= 1- first type risk), must be < 1 , equal to 0.95 by default.

Value : a `TestResultCollection`, the structure which contains the results of the successive tests.

Details : the `FullSpearman` Test is used to check the hypothesis of monotonous relation between samples : `firstSample` of dimension n and `secondSample` of dimension 1. If `firstSample[i]` is the numerical sample extracted from `firstSample` (i th coordinate of each point of the numerical sample), `FullSpearman` performs the Independence Spearman test simultaneously on all `firstSample[i]` and `secondSample`.

PartialPearson

Usage :

HypothesisTest.PartialPearson(firstSample, secondSample, selection)
HypothesisTest.PartialPearson(firstSample, secondSample, selection, level)

Arguments :

firstSample : a `NumericalSample`, of dimension $n \geq 1$: the first sample tested
secondSample : a `NumericalSample`, of dimension 1 : the second sample tested
selection : a `Indices`, array of integers selecting the indices of the coordinates of the first sample which will successively be tested with the second sample through the Pearson Test
level : a positive real value, the threshold p-value of the test (= 1- first type risk), must be < 1 , equal to 0.95 by default.

Value : a `TestResultCollection`, the structure which contains the results of the successive tests.

Details : the `PartialPearson` Test is the independence Pearson test between 2 samples : `firstSample` of dimension n and `secondSample` of dimension 1. If `firstSample[i]` is the numerical sample extracted from `firstSample` (i th coordinate of each point of the numerical sample), `PartialPearson` performs the Independence Pearson test simultaneously on `firstSample[i]` and `secondSample`, for i in the selection. For all i , it is supposed that the couple (`firstSample[i]` and `secondSample`) is issued from a gaussian vector.

*PartialRegression***Usage :**

*HypothesisTest.PartialRegression(firstSample, secondSample, ...
...selection)*

*HypothesisTest.PartialRegression(firstSample, secondSample, ...
...selection, level)*

Arguments :

firstSample : a NumericalSample, of dimension $n \geq 1$: the first sample tested

secondSample : a NumericalSample, of dimension 1 : the second sample tested

selection : a Indices, array of integers selecting the indices of the coordinates of the first sample which will successively be tested with the second sample through the Regression Test

level : a positive real value, the threshold p-value of the test (= 1- first type risk), must be < 1, equal to 0.95 by default.

Value : a TestResult, the structure which contains the result of the test.

Details : the PartialRegression Test is used to check the quality of the linear regression model between two 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 linear regression test tests if the linear regression model between two scalar numerical samples is significant. It is based on the deviation analysis of the linear regression. The Fisher distribution is used.

*PartialSpearman***Usage :**

*HypothesisTest.PartialSpearman(firstSample, secondSample, ...
...selection)*

*HypothesisTest.PartialSpearman(firstSample, secondSample, ...
...selection, level)*

Arguments :

firstSample : a NumericalSample, of dimension $n \geq 1$: the first sample tested

secondSample : a NumericalSample, of dimension 1 : the second sample tested

selection : a Indices, array of integers selecting the indices of the coordinates of the first sample which will successively be tested with the second sample through the Spearman Test

level : a positive real value, the threshold p-value of the test (= 1- first type risk), must be < 1, equal to 0.95 by default.

Value : a TestResultCollection, the structure which contains the results of the successive tests.

Details : the PartialSpearman Test is used to check the hypothesis of monotonous relation between 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), PartialSpearman performs the Independence Spearman test simultaneously on firstSample[i] and secondSample, for i in the selection.

*Pearson***Usage :**

HypothesisTest.Pearson(firstSample, secondSample)

HypothesisTest.Pearson(firstSample, secondSample, level)

Arguments :

firstSample : a NumericalSample, of dimension 1 : the first sample tested

secondSample : a NumericalSample, of dimension 1 : the second sample tested

level : a positive real value, the threshold p-value of the test (= 1- first type risk), must be < 1, equal to 0.95 by default.

Value : a TestResult, the structure which contains the result of the test.

Details : the Test is used to check whether two samples which form a gaussian vector are independent (based on the evaluation of the linear correlation coefficient).

Smirnov

Usage :

HypothesisTest.Smirnov(firstSample, secondSample)

HypothesisTest.Smirnov(firstSample, secondSample, level)

Arguments :

firstSample : a NumericalSample, of dimension 1 : the first sample tested

secondSample : a NumericalSample, of dimension 1 : the second sample tested

level : a positive real value, the threshold p-value of the test (= 1- first type risk), must be < 1, equal to 0.95 by default.

Value : a TestResult, the structure which contains the result of the test.

Details : the Smirnov Test is used to check whether two continuous scalar samples (of sizes not necessarily equal) follow the same distribution.

Spearman

Usage :

HypothesisTest.Spearman(firstSample, secondSample)

HypothesisTest.Spearman(firstSample, secondSample, level)

Arguments :

firstSample : a NumericalSample, of dimension 1 : the first sample tested

secondSample : a NumericalSample, of dimension 1 : the second sample tested

level : a positive real value, the threshold p-value of the test (= 1- first type risk), must be < 1, equal to 0.95 by default.

Value : a TestResult, the structure which contains the result of the test.

Details : the Spearman Test is used to check whether two scalar samples have a monotonous relation.

11.5.6 LinearModelTest

This class is used through its static methods in order to evaluate the quality of the linear regression model between two samples (see 11.6). The linear regression model between the scalar variable Y and the n -dimensional one $\underline{X} = (X_i)_{i \leq n}$, as follows :

$$\tilde{Y} = a_0 + \sum_{i=1}^n a_i X_i + \varepsilon$$

where ε is the residual, supposed to follow the Normal(0.0, 1.0) distribution.

In the following, *firstSample* is a numerical sample from $\underline{X} = (X_i)_{i \leq n}$ and *secondSample* from Y .

Some methods :

LMAdjustedRSquared

Usage :

*LinearModelTest.LMAdjustedRSquared(firstSample, ...
...secondSample)*

*LinearModelTest.LMAdjustedRSquared(firstSample, ...
...secondSample, level)*

Arguments :

firstSample : a NumericalSample, of dimension n : the first sample tested

secondSample : a NumericalSample, of dimension 1 : the second sample tested

level : a positive real value, the threshold p-value of the test (= 1- first type risk), must be < 1, equal to 0.95 by default

Value : a TestResult, the structure which contains the result of the test

Details : The LMAdjustedRSquared test 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

Usage :

*LinearModelTest.LMAdjustedRSquared(firstSample, ...
...secondSample)*

*LinearModelTest.LMAdjustedRSquared(firstSample, ...
...secondSample, level)*

Arguments :

firstSample : a NumericalSample, of dimension n : the first sample tested

secondSample : a NumericalSample, of dimension 1 : the second sample tested

level : a positive real value, the threshold p-value of the test (= 1- first type risk), must be < 1, equal to 0.95 by default

Value : a TestResult, the structure which contains the result of the test

Details : The LMFisher Test tests the nullity of the regression linear model coefficients (Fisher distribution used)

LMRSquared

Usage :

LinearModelTest.LMAdjustedRSquared(firstSample, secondSample)
LinearModelTest.LMAdjustedRSquared(firstSample, ...
...secondSample, level)

Arguments :

firstSample : a NumericalSample, of dimension n : the first sample tested
secondSample : a NumericalSample, of dimension 1 : the second sample tested
level : a positive real value, the threshold p-value of the test (= 1- first type risk), must be < 1, equal to 0.95 by default

Value : a TestResult, the structure which contains the result of the test

Details : The LMRSquared test tests the quality of the linear regression model. It evaluates the indicator R^2 (regression variance analysis) and compares it to a level,

*LMResidualMean***Usage :**

LinearModelTest.LMAdjustedRSquared(firstSample, secondSample)
LinearModelTest.LMAdjustedRSquared(firstSample, secondSample, level)

Arguments :

firstSample : a NumericalSample, of dimension n : the first sample tested
secondSample : a NumericalSample, of dimension 1 : the second sample tested
level : a positive real value, the threshold p-value of the test (= 1- first type risk), must be < 1, equal to 0.95 by default

Value : a TestResult, the structure which contains the result of the test

Details : The 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).

11.6 Linear model

11.6.1 LinearModelFactory

This class is used in order to create a linear model from numerical samples. The linear regression model between the scalar variable Y and the n -dimensional one $\underline{X} = (X_i)_{i \leq n}$ writes as follows :

$$\tilde{Y} = a_0 + \sum_{i=1}^n a_i X_i + \varepsilon$$

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 accompanied by a confidence interval and a p-value (which tests if they are significantly different from 0.0).

In the following, *firstSample* is a numerical sample from $\underline{X} = (X_i)_{i \leq n}$ and *secondSample* from Y .

This class enables to test the quality of the model. It provides only numerical tests. If \underline{X} is scalar, a graphical validation test exists, that draws the residual couples $(\varepsilon_i, \varepsilon_{i+1})$, where the residual ε_i is evaluated on the samples $(Xsample, Ysample) : \varepsilon_i = Ysample_i - \tilde{Y}_i$ with $\tilde{Y}_i = a_0 + a_1 Xsample_i$. The Open TURNS method is *DrawLMResidualtest*, provided by the *VisualTest* class (see 11.5.3).

Usage : *LinearModelFactory()*

Arguments : none

Some methods :

build

Usage :

build(sampleX, sampleY)

build(sampleX, sampleY, level)

Arguments :

sampleX : a NumericalSample, of dimension $n \geq 1$

sampleY : a NumericalSample, of dimension 1

level : the level value of the confidence intervals of each coefficient of the linear model, equal to 0.95 by default

Value : a LinearModel, the linear model built from the samples $(sampleX, sampleY) : Y = a_0 + \sum_{i=1}^n a_i X_i + \varepsilon$, where ε is the random residual with zero mean.

11.6.2 LinearModel

Usage : A LinearModel is created through the method *build* of a LinearModelFactory.

Some methods :

getConfidenceIntervals

Usage : *getConfidenceIntervals()*

Arguments : none

Value : a ConfidenceIntervalPersistentCollection, the collection of the confidence intervals of the linear model coefficients, corresponding to the level precised when the LinearModel class has been created through the method *build*

getPValues

Usage : *getPValues()*

Arguments : none

Value : a NumericalScalarPersistentCollection, the collection of the p-values of the linear model coefficients

getPredict

Usage : *getPredict(sampleX)*

Arguments : *sampleX* : a NumericalSample, the sample we want to evaluate the response *Y* on

Value : a NumericalSample, of dimension 1, the response *Y* evaluated through the linear model on the sample *sampleX*

getRegression

Usage : *getRegression()*

Arguments : none

Value : a NumericalPoint, the coefficients of the linear model : (a_0, a_1, \dots, a_n)

getResidual

Usage : *getResidual(sampleX, sampleY)*

Arguments :

sampleX : a NumericalSample, the *sampleX* on which the linear model has been built

sampleY : a NumericalSample, the *sampleY* on which the linear model has been built

Value : a NumericalPoint, the residuals

12 Stochastic process

In this section a description of all general objects is given.

The notion of stochastic process induces many classes. This part is organized as following :

- we focus on the presentation of both the *Process*, *RegularGrid*, *TimeSeries* and *ProcessSample* class. These objects are used in the different following sections.
- We quickly present the *CompositeProcess* class,
- We present the *ARMA* class and its specific objects,
- The *NormalProcess* is presented both with its specific objects and its two variantes (*TemporalNormalProcess* and *SpectralNormalProcess*).
- The *WhiteNoise* is presented.

- Finally the *RandomWalk* is presented.

Be aware of the fact that for some uses in the TUI, it is necessary to explicitly cast a given process into the general *Process* class.

12.1 General common objects

12.1.1 Process

Usage :

Process(process)

Arguments :

process : a ProcessImplementation (which is a particular Process)

Value : a Process

This class enables to modelize a stochastic process.

Some methods :

getDescription

Usage : *getDescription()*

Arguments : none

Value : a string, the description of the process

getDimension

Usage : *getDimension()*

Arguments : none

Value : an integer, the dimension of the stochastic process

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the Process

getRealization

Usage : *getRealization()*

Arguments : none

Value : a TimeSeries, one realization of the considered random process

getMarginalProcess

Usage : *getMarginalProcess(k)*

Arguments : *k*, an integer

Value : a Process, the *k* – *th* marginal of the random process

getSample

Usage : *getSample(n)*

Arguments : *n*, an integer

Value : a ProcessSample , *n* realizations of the random process

getTimeGrid

Usage : *getTimeGrid()*

Arguments : none

Value : a RegularGrid, the timeGrid over which the process is observed

isComposite

Usage : *isComposite()*

Arguments : none

Value : a boolean. Tells if the process is composite (built upon a function and a process).

isStationary

Usage : *isStationary()*

Arguments : none

Value : a boolean, *true* if the process is stationary, *false* otherwise

isNormal

Usage : *isNormal()*

Arguments : none

Value : a boolean, *true* if the process is Normal, *false* otherwise

setDescription

Usage : *setDescription(description)*

Arguments : *description*, a Description

Value : none. Set a description to the process

setName

Usage : *setName(name)*

Arguments : *name* : a string

Value : the Process is named *name*

setTimeGrid

Usage : *setTimeGrid(timeGrid)*

Arguments : *timeGrid*, a RegularGrid

Value : Fix the time grid of observation of the process

12.1.2 RegularGrid

Usage :

RegularGrid()
RegularGrid(t_{Min} , Δt , N)

Arguments :

t_{Min} : a NumericalScalar, the initial time
 N : an integer, the number of time stamps
 Δt : a NumericalScalar, the step of the regular grid such as the k – th time stamp of the time grid is $t_k = t_{Min} + k * \Delta$

Value : RegularGrid

the RegularGrid is filled by fixing the step ΔT ; the final time corresponds to $t_{Min} + (N - 1) * \Delta$

Some methods :

getN

Usage : *getN()*
Arguments : none
Value : an integer : the number of time stamps

getStart

Usage : *getStart()*
Arguments : none
Value : a NumericalScalar : the initial time of the RegularGrid

getEnd

Usage : *getEnd()*
Arguments : none
Value : a NumericalScalar : the first time stamp out of the RegularGrid

getStep

Usage : *getStep()*
Arguments : none
Value : a NumericalScalar : the time step of the RegularGrid

getName

Usage : *getName()*
Arguments : none
Value : a string, the name of the RegularGrid

setName

Usage : *setName(name)*
Arguments : name : a string
Value : the time grid is named *name*

12.1.3 TimeSeries

Usage :

TimeSeries()

TimeSeries(timeGrid, sample)

Arguments :

timeGrid : a RegularGrid, the time values on which the realizations are observed

sample : a NumericalSample, the data values

Value : TimeSeries

the TimeSeries is filled with a time values issued by *timeGrid* and a values issued from *sample*

Some methods :

drawMarginal

Usage : *drawMarginal(i)*

Arguments : an integer

Value : a Graph of the *i* – *th* marginal as function of time

getDimension

Usage : *getDimension()*

Arguments : none

Value : an integer : the dimension of the TimeSeries

getTemporalMean

Usage : *getTemporalMean()*

Arguments : none

Value : a NumericalPoint : compute the temporal mean which corresponds to the mean of its values

getTimeGrid

Usage : *getTimeGrid()*

Arguments : none

Value : a RegularGrid : the time stamps of observation

getValueAtIndex

Usage : *getValueAtIndex(k)*

Arguments : an integer

Value : a NumericalPoint : the *k* – *th* value of the TimeSeries

getValueAtNearestTime

Usage : *getValueAtNearestTime(t)*

Arguments : a NumericalScalar

Value : a NumericalPoint : the value of the TimeSeries observed at time t

getNumericalSample

Usage : *getNumericalSample()*

Arguments : none

Value : a NumericalSample : the data of the TimeSeries

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the TimeSeries

setValueAtIndex

Usage : *setValueAtIndex(i, np)*

Arguments : an integer, a NumericalPoint

Value : None ; the value np is set at the index i

setValueAtNearestTime

Usage : *setValueAtNearestTime(t, np)*

Arguments : a NumericalScalar, a NumericalPoint

Value : None ; the value np is set at the index which corresponds to the nearest time t

setName

Usage : *setName(name)*

Arguments : name : a string

Value : the TimeSeries is named $name$

12.1.4 ProcessSample

Usage :

ProcessSample()
ProcessSample(timeGrid, size, dimension)
ProcessSample(size, timeSeries)
ProcessSample(collection)

Arguments :

timeGrid : a RegularGrid
size : an integer, size of the ProcessSample
dimension : an integer, dimension of the ProcessSample
collection : a TimeSeriesCollection, collection of TimeSeries with the same timeGrid

Value : ProcessSample

while using the second parameters set, the ProcessSample has timeGrid as grid of time stamps, its size and dimension are fixed (sample is filled with values are zero)

while using the third parameters set, the ProcessSample is filled with *size* copies of the TimeSeries *timeSeries*

while using the fourth parameters set, the ProcessSample is filled with the collection of TimeSeries (with common RegularGrid)

Some methods :

drawMarginal

Usage : *drawMarginal(i)*

Arguments : an integer

Value : a Graph of the *i* – *th* marginals as function of time

getDimension

Usage : *getDimension()*

Arguments : none

Value : an integer : the dimension of the ProcessSample

getSize

Usage : *getSize()*

Arguments : none

Value : an integer : the size of the ProcessSample

getTimeGrid

Usage : *getTimeGrid()*

Arguments : none

Value : a RegularGrid : the time stamps of observation of the ProcessSample

add

Usage : *add(timeSeries)*

Arguments : *timeSeries*, a TimeSeries

Value : Extend the ProcessSample by adding *timeSeries* to the collection (the TimeSeries should have the same time grid as the ProcessSample). The new size of the sample is updated

computeMean

Usage : *computeMean()*

Arguments : none

Value : a TimeSeries : the mean of the ProcessSample. Its *k* – *th* time stamp is the mean value of the *k* – *th* time stamp of all its TimeSeries

computeQuantilePerComponent

Usage : *computeQuantilePerComponent(prob)*

Arguments : *prob*: a real value in (0, 1).

Value : a TimeSeries with size and dimension parameters the same as the ProcessSample's ones. The values at a given time index are component-wise estimates of the quantiles of level *prob* of the underlying process at this time value, using the empirical quantile.

computeTemporalMean

Usage : *computeTemporalMean()*

Arguments : none

Value : a NumericalSample with size and dimension parameters the same as the ProcessSample's ones. The *k* – *th* value corresponds to the temporal mean of the *k* – *th* TimeSeries of the ProcessSample

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the ProcessSample

setName

Usage : *setName(name)*

Arguments : *name* : a string

Value : the ProcessSample is named *name*

12.1.5 TimeSeriesCollection

Usage :

TimeSeriesCollection()

TimeSeriesCollection(size)

TimeSeriesCollection(size, ts)

Arguments :

size : an integer, the size of the collection (number of elements that it contains)

ts : a TimeSeries

Value : a TimeSeriesCollection, collection of TimeSeries

while using the second parameters set, the TimeSeriesCollection is filled with *size* copy of the TimeSeries *ts*

Some methods :

add

Usage : *add(ts)*

Arguments : TimeSeries

Value : Add *ts* to the collection of TimeSeries

clear

Usage : *clear()*

Arguments : None

Value : Remove all elements of the collection. The new size is 0

getSize

Usage : *getSize()*

Arguments : none

Value : an integer : the size of the collection

resize

Usage : *resize(n)*

Arguments : an integer

Value : Fix the new size of the collection

12.2 Temporal information

12.2.1 CovarianceModel

This class is the interface of *CovarianceModelImplementation*.

Usage : *CovarianceModel(myCovarianceModelImplementation)*

Arguments : *myCovarianceModelImplementation* : the implementation of a covariance model. For example, a stationary covariance model, as the Exponential one.

Value : a CovarianceModel

Some methods :

getDimension

Usage : *getDimension()*

Arguments : none

Value : an integer, the dimension of the model d

isStationary

Usage : *isStationary()*

Arguments : none

Value : a boolean, *true* if the model is stationary

computeCovariance

Usage :

computeCovariance(t, s)

computeCovariance(τ)

Arguments : t, s : NumericalScalar

τ : NumericalScalar

]

Value : a CovarianceMatrix $\in \mathcal{M}_{d \times d}(\mathbb{R})$ that evaluates the covariance model between time stamps t and s :

$$\underline{C}(s, t) = \mathbb{E} [(\underline{X}_s - \underline{m}(s))(\underline{X}_t - \underline{m}(t))^t]$$

where \underline{m} is the mean function defined for all $t > 0$ by $\underline{m}(t) = E[\underline{X}_t]$.

In the second usage, the covariance model must be stationary.

discretizeCovariance

Usage : *discretizeCovariance(tg)*

Arguments : tg : a RegularGrid, the time grid (t_0, \dots, t_{n-1}) of size n

Value : a CovarianceMatrix $\in \mathcal{M}_{nd \times nd}(\mathbb{R})$ (if the process is of dimension d) makes a discretization of the model on the time grid tg and returns the covariance matrix :

$$\underline{\underline{C}}_{1,\dots,k} = \begin{pmatrix} \underline{\underline{C}}(t_0, t_0) & \underline{\underline{C}}(t_0, t_1) & \dots & \underline{\underline{C}}(t_0, t_{n-1}) \\ \dots & \underline{\underline{C}}(t_1, t_1) & \dots & \underline{\underline{C}}(t_1, t_{n-1}) \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \underline{\underline{C}}(t_{n-1}, t_{n-1}) \end{pmatrix} \quad (16)$$

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the CovarianceModel

setName

Usage : *setName(name)*

Arguments : name : a string

Value : the CovarianceModel, named *name*

12.2.2 StationaryCovarianceModel

This class inherits from CovarianceModel. This class is not used alone.

Usage : *StationaryCovarianceModel()*

Arguments : none.

Value : a StationaryCovarianceModel. For example, the Exponential one.

description : The covariance matrix $\underline{\underline{C}}(s, t)$ is stationnary when it only depends on $t - s$:

$$\forall (s, t, h) > 0, \quad \underline{\underline{C}}(s, s + h) = \underline{\underline{C}}(t, t + h) \quad (17)$$

12.2.3 ExponentialModel

This class inherits from *StationaryCovarianceModel* class. The Exponential model defines the covariance function $\underline{\underline{C}}^{stat}(\tau)$ such that :

$$\forall \tau \in \mathbb{R}, \quad \underline{\underline{C}}^{stat}(\tau) = \sqrt{\underline{\underline{\Delta}}(\tau)} \underline{\underline{R}} \sqrt{\underline{\underline{\Delta}}(\tau)} \quad (18)$$

where

$$\sqrt{\underline{\underline{\Delta}}(\tau)} = \begin{pmatrix} a_1 e^{-\lambda_1 |\tau|/2} & & 0 \\ & \ddots & \\ 0 & & a_d e^{-\lambda_d |\tau|/2} \end{pmatrix} \quad (19)$$

and $\underline{\underline{R}}$ a *spatial* correlation matrix such that :

$$\forall t \in \mathbb{R}, \quad \underline{\underline{Cor}}(\underline{\underline{X}}(t)) = \underline{\underline{R}} \quad (20)$$

where $\lambda_i > 0$ and $a_i > 0$ for all i .

It is possible to define the exponential model from the spatial covariance function $\underline{\underline{C}}^s(\tau)$ defined in

$$\forall t \in \mathbb{R}, \quad \underline{\underline{\mathbb{E}}} [\underline{\underline{X}}(t) \underline{\underline{X}}^t(t)] = \underline{\underline{C}}^s \quad (21)$$

since we have the relation :

$$\underline{\underline{C}}^s = \underline{\underline{A}} \underline{\underline{R}} \underline{\underline{A}} \quad (22)$$

where

$$\underline{\underline{A}} = \begin{pmatrix} a_1 & & 0 \\ & \ddots & \\ 0 & & a_d \end{pmatrix} \quad (23)$$

The spectral model associated is the Cauchy model (12.3.2).

Usage :

```
ExponentialModel()
ExponentialModel(amplitude, scale)
ExponentialModel(amplitude, scale, spatialCorrelation)
ExponentialModel(amplitude, scale, spatialCovariance)
```

Arguments :

amplitude : a NumericalPoint of dimension d , the amplitude \underline{a} of the model,
scale : a NumericalPoint of size d , the scale $\underline{\lambda}$ of the model,
spatialCorrelation : a CorrelationMatrix of dimension $d \times d$, the correlation matrix $\underline{\underline{R}}$,
spatialCovariance : a CovarianceMatrix of dimension $d \times d$, the correlation matrix $\underline{\underline{C}}^s$.

Value : an ExponentialModel

in the first usage, we fix dimension to 1, scale, amplitude and spatialCorrelation to 1.0

in the second usage, we fix the scale \underline{a} and the amplitude $\underline{\lambda}$. By default, the spatial correlation matrix \underline{R} is $Id(d)$. The dimension d is deduced.

in the third usage, we fix the scale \underline{a} , the amplitude $\underline{\lambda}$ and the spatial correlation matrix \underline{R} . The dimension d is deduced.

in the last usage, we fix the scale \underline{a} , the amplitude $\underline{\lambda}$ and the spatial covariance matrix \underline{C}^s . The dimension d is deduced.

Some methods :

getAmplitude

Usage : *getAmplitude()*

Arguments : none

Value : a NumericalPoint, the amplitude \underline{a} of the Exponential model.

getScale

Usage : *getScale()*

Arguments : none

Value : a NumericalPoint, the scale $\underline{\lambda}$ of the Exponential model.

getSpatialCorrelation

Usage : *getSpatialCorrelation()*

Arguments : none

Value : a CorrelationeMatrix, the spatial correlation matrix \underline{R} which gives the correlation between the i-th and j-th components at each instant t .

12.3 Spectral information

We recall that If $\underline{X}(\omega, t)$ is a stationary process, we define the *bilateral spectral density function* $\underline{S}(f) \in \mathcal{M}(\mathbb{R}^d \times \mathbb{R}^n)$ as the Fourier transform of the covariance function \underline{C}^{stat} :

$$\forall f \in \mathbb{R}, \underline{S}(f) = \int_{\mathbb{R}} \exp\{-2i\pi f\tau\} \underline{C}^{stat}(\tau) d\tau \quad (24)$$

and the *unilateral spectral density function* $\underline{G}(f)$ defined by :

$$\forall f \geq 0, \underline{G}(f) = 2\underline{S}(f) \quad (25)$$

Inversely, the covariance function \underline{C}^{stat} may be evaluated from the spectral density function $\underline{S}(f)$ if \underline{S} is $L^1(\mathcal{M}_{dd}(\mathbb{C}))$ as follows :

$$\underline{C}^{stat}(\tau) = \int_{\mathbb{R}} \exp\{2i\pi f\tau\} \underline{S}(f) df \quad (26)$$

12.3.1 SpectralModel

This class is the interface of SpectralModelImplementation.

Usage : *SpectralModel(mySpectralModelImplementation)*

Arguments : *mySpectralModelImplementation* : the implementation of a spectral model. For example, the Cauchy model.

Value : a SpectralModel

Some methods :

getDimension

Usage : *getDimension()*

Arguments : none

Value : an integer, the dimension of the model d

computeSpectralDensity

Usage : *computeSpectralDensity(f)*

Arguments : f , a NumericalScalar

Value : an HermitianMatrix of size d , computes the unilateral spectral function $\underline{G}(f)$ at the frequency f ($f \geq 0$)

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the SpectralModel

setName

Usage : *setName(name)*

Arguments : name : a string

Value : the SpectralModel, named *name*

12.3.2 CauchyModel

This class inherits from SpectralModel class. It implements the spectral model associated to the Exponential covariance model detailed in (12.2.3). The spectral density function is defined by :

$$S_{ij}(f) = \frac{4R_{ij}a_i a_j (\lambda_i + \lambda_j)}{(\lambda_i + \lambda_j)^2 + (4\pi f)^2} \quad (27)$$

where \underline{R} , \underline{a} and $\underline{\lambda}$ are the parameters of the Exponential covariance model. The relation (??) can be explicitated with the spatial covariance function $\underline{C}^{stat}(\tau)$ defined in (22).

Usage :

```
CauchyModel()
CauchyModel(amplitude, scale)
CauchyModel(amplitude, scale, spatialCorrelation)
CauchyModel(amplitude, scale, spatialCovariance)
```

Arguments :

amplitude : a NumericalPoint of size d , the amplitude of the model in each dimension,
scale : a NumericalPoint of size d , the scale of the model in each dimension,
spatialCorrelation : a CorrelationMatrix of size $d \times d$, the correlation between i -th and j -th dimension,
spatialCovariance : a CovarianceMatrix of dimension $d \times d$, the correlation matrix \underline{C}^s .

Value : an CauchyModel

in the first usage, we fix dimension to 1, scale, amplitude and spatialCorrelation to 1.0
in the second usage, we fix the dimension d , scale and amplitude values. Correlation used here is IdentityMatrix(d)
in the third usage, we fix the dimension, scale, amplitude and correlation.
in the last usage, we fix the scale \underline{a} , the amplitude $\underline{\lambda}$ and the spatial covariance matrix \underline{C}^s . The dimension d is deduced.

Some methods :

getAmplitude

Usage : *getAmplitude()*

Arguments : none

Value : a NumericalPoint, the amplitude of the Cauchy model in each dimension

getScale

Usage : *getScale()*

Arguments : none

Value : a NumericalPoint, the scale of the model in each dimension

getSpatialCorrelation

Usage : *getSpatialCorrelation()*

Arguments : none

Value : a CorrelationeMatrix, the correlation between the i-th and j-th dimension

12.3.3 UserDefinedSpectralModel

This class inherits from `SpectralModel`.

Usage :

UserDefinedSpectralModel()

UserDefinedSpectralModel(*frequency*, *densityCollectionFunction*)

Arguments :

frequency : a `RegularGrid` which containing the frequency values on which the model has been built;

densityCollectionFunction : a collection of `HermitianMatrix` of size *d*, the density functions of the `SpectralModel`

Value : a `UserDefinedSpectralModel`

in the second usage, we fix the spectral density function over the frequency grid *frequency*

Description : The class enables user to quickly implement a `SpectralModel` from retrn of experience in their area by fixing a regular frequency grid and matrices collection.

This class is also used for the estimation of a `SpectralModel`

12.3.4 SpectralModelFactory

Usage :

SpectralModelFactory()

Value : a `SpectralModelFactory`

The default implementation refers to the `WelchFactory` class

Some methods :

getFFTAlgorithm

Usage : *getFFTAlgorithm*()

Arguments : none

Value : a FFT, the used FFT algorithm for the Fourier transform

setFFTAlgorithm

Usage : *setFFTAlgorithm*(*fft*)

Arguments : an FFT

Value : set *fft* as the algorithm for the Fourier transform

build

Usage : *build(sample)*

Arguments : a ProcessSample

Value : a SpectralModel. Build an estimation of a SpectralModel on the sample.

build

Usage : *build(timeSerie)*

Arguments : a TimeSeries

Value : a SpectralModel. Build an estimation of a SpectralModel on the timeSerie.

12.3.5 WelchFactory

This class inherits from *SpectralModelFactory*. It implements the *Welch* method.

Usage :

WelchFactory()

WelchFactory(*window*, *bloc*, *overlap*)

Value : a WelchFactory

With the first usage, the Hanning is fixed as the window, the number of blocs is 1 and overlap = 0

In the second usage, we fix the filtering window, the number of blocs and the overlap parameter

Some methods :

getFilteringWindows

Usage : *getFilteringWindows*()

Arguments : none

Value : a FilteringWindows. The used window algorithm

setFilteringWindows

Usage : *setFilteringWindows*(*window*)

Arguments : *window*, a FilteringWindows

Value : Set *window* as the FilteringWindows

getBloc

Usage : *getBloc*()

Arguments : none

Value : an integer. The number of blocs of the Welch method

setBloc

Usage : *setBloc*(*bloc*)

Arguments : *bloc*, an integer

Value : Set the number of blocs

getOverlap

Usage : *getOverlap*()

Arguments : none

Value : an integer. The size of overlap within the Welch method

setOverlap

Usage : *setOverlap*(*overlap*)

Arguments : *overlap*, an integer

Value : Set the size of the overlap

12.3.6 FilteringWindows

Usage :

FilteringWindows()

Value : a FilteringWindows

The FilteringWindows is built over the interval $[0, 1]$. This class is the interface of *FilteringWindowsImplement*

Some methods :

() operand

Usage : (α)

Arguments : α , a NumericalScalar

Value : a NumericalScalar, the evaluation of the filtering windows on α

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the filtering windows

setName

Usage : *setName(name)*

Arguments : name, a string

Value : the filtering windows is named *name*

12.3.7 Hanning

The Hanning filtering windows inherits from the FilteringWindows class

Usage :

Hanning()

Value : a Hanning

Description :

The Hanning windows is implemented using the formula $w(t) = \sqrt{\frac{8}{3}} * \sin(\pi t)^2$ for $t \in [0, 1]$, 0 otherwise.

12.3.8 Hamming

The Hamming filtering windows inherits from the FilteringWindows class

Usage :

Hamming()

Value : a Hamming

Description :

The Hamming windows is implemented using the formula :

$$w(t) = C * (0.54 - 0.46 * \cos(2.0\pi t)) \text{ with } C = \frac{1.0}{0.54^2 + 0.5 * 0.46^2}.$$

The formula is available for $t \in [0, 1]$. Otherwise, the filtering windows is zero.

The normalization C is done such as $\int_0^1 w(t)^2 dt = 1$

12.4 Link Temporal - Spectral information

12.4.1 SecondOrderModel

The class is the interface of SecondOrderModelImplementation.

This aims at regrouping a *StationaryCovarianceModel* and a *SpectralModel* that are in correspondence.

The issued SecondOrderModel both has spectral and covariance functions.

Usage :

SecondOrderModel()

SecondOrderModel(*stationaryCovarianceModel*, *spectralModel*)

Arguments :

Value : a SecondOrderModel

Some methods :

The methods of the SecondOrderModel are both those implemented for StationaryCovarianceModel and SpectralModel.

getDimension

Usage : *getDimension*()

Arguments : none

Value : an integer, the dimension of the model d

computeCovariance

Usage : *computeCovariance*(τ)

Arguments : t : NumericalScalar

Value : a CovarianceMatrix of size $d \times d$, computes the covariance function for different τ

discretizeCovariance

Usage : *discretizeCovariance*(tg)

Arguments : tg : a RegularGrid

Value : a CovarianceMatrix (of size $d \times n$), with n the size of the time grid, which is the discretization of the covariance function on the time grid tg .

computeSpectralDensity

Usage : *computeSpectralDensity*(f)

Arguments : f , a NumericalScalar

Value : an HermitianMatrix of size $d \times d$, computes the unilateral spectral density model \underline{G} at the frequency f ($f \geq 0$)

getSpectralModel

Usage : *getSpectralModel()*

Arguments : none

Value : a SpectralModel, the spectral model of the SecondOrderModel

setSpectralModel

Usage : *setSpectralModel(spectralModel)*

Arguments : a SpectralModel

Value : set *spectralModel* as the spectral model of the SecondOrderModel

getStationaryCovarianceModel

Usage : *getStationaryCovarianceModel()*

Arguments : none

Value : a StationaryCovarianceModel, the covariance function of the SecondOrderModel

setStationaryCovarianceModel

Usage : *setStationaryCovarianceModel(covarianceModel)*

Arguments : a StationaryCovarianceModel

Value : sets *covarianceModel* as the covariance function of the SecondOrderModel

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the SecondOrderModel

setName

Usage : *setName(name)*

Arguments : name : a string

Value : the SecondOrderModel, named *name*

12.4.2 ExponentialCauchy

This class inherits from SecondOrderModel class.

It aims at regrouping the Exponential Model of the covariance function defined in (12.2.3) and the Cauchy Model for the spectral density function defined in (12.3.2).

Usage :

ExponentialCauchy()

ExponentialCauchy(amplitude, scale)

ExponentialCauchy(amplitude, scale, spatialCorrelation)

ExponentialCauchy(amplitude, scale, spatialCovariance)

Arguments :

amplitude : a NumericalPoint of size d , the amplitude of the model in each dimension.

scale : a NumericalPoint of size d , the scale of the model in each dimension.

spatialCorrelation : a CorrelationMatrix of size $d \times d$, the correlation between the i -th and j -th dimension.

spatialCovariance : a CovarianceMatrix of dimension $d \times d$, the correlation matrix $\underline{\underline{C}}^s$.

Value : an ExponentialCauchy

in the first usage, we fix dimension to 1, scale amplitude and spatialCorrelation to 1.0

in the second usage, we fix the dimension, scale and amplitude values. spatialCorrelation here is IdentityMatrix(d)

in the third usage, we fix the dimension, scale, amplitude and correlation.

in the last usage, we fix the scale \underline{a} , the amplitude $\underline{\lambda}$ and the spatial covariance matrix $\underline{\underline{C}}^s$. The dimension d is deduced.

Some methods :

getAmplitude

Usage : *getAmplitude()*

Arguments : none

Value : a NumericalPoint, the amplitude of the Exponential model in each dimension

getScale

Usage : *getScale()*

Arguments : none

Value : a NumericalPoint, the scale of the Exponential model in each dimension

getSpatialCorrelation

Usage : *getSpatialCorrelation()*

Arguments : none

Value : a CorrelationMatrix, the correlation between the i-th and j-th dimension

Remark :

The class regroups the *ExponentialModel* and *CauchyModel* of same parameters.

The density function is, from a mathematical point of view, the Fourier transform of the CovarianceModel.

12.5 Normal process

The class *NormalProcess* inherits from *Process*

12.5.1 NormalProcess

Usage : *NormalProcess(timeGrid, model)*

Arguments :

timeGrid : a RegularGrid, the time grid over which the process is implemented

modes : a SecondOrderModel

Value : a NormalProcess process, parametered by the given time grid and the given second order model (which contains the covariance information and the spectral one).

Nota : This class enables to group the models which are based on temporal and spectral domains.

Some methods :

getSecondOrderModel

Usage : *getSecondOrderModel()*

Arguments : none

Value : a SecondOrderModel, if the process has been created from a second order used by the NormalProcess.

12.5.2 SpectralNormalProcess

Usage :

```

SpectralNormalProcess()
SpectralNormalProcess(model, timeGrid)
SpectralNormalProcess(model, maximalFrequency, N)
SpectralNormalProcess(spectralModel, timeGrid)
SpectralNormalProcess(spectralModel, maximalFrequency, N)

```

Arguments :

spectralModel : a SpectralModel
model : a SecondOrderModel
timeGrid : a RegularGrid; the time grid over which the process is implemented
maximalFrequency : a NumericalScalar; the maximal frequency
N : an integer; the size of discretization of the frequency domain

Value : a SpectralNormalProcess process

in the second usage, we fix the time grid and the second order model (spectral density model) which implements the process; Frequency values are induced by the time values.

in the third usage, conversely to the previous usage, the process is fixed in the frequency domain. *maximalFrequency* value and *N* induce the time grid.

Be aware that the maximal frequency used in the computation is not *maximalFrequency* but $\frac{\text{maximalFrequency} * (N - 1)}{N}$.

in the fourth usage (respectively the fifth usage), the spectral model is given instead of a complete second order model and the other arguments are the same as the second (respectively the third) usage.

Comments :

The SpectralNormalProcess enables to model the normal processes in the spectral domain. This class inherits from the NormalProcess class.

The first call of *getRealization* might be time consuming because it computes *N* hermitian matrices of size $d \times d$, where *d* is the dimension of the spectral model. These matrices are factorized and stored in order to be used for each call of the *getRealization* method.

Some methods :

getFrequentiaGrid

Usage : *getFrequentiaGrid*()

Arguments : none

Value : a *RegularGrid*, the frequencies used in the computation.

12.5.3 TemporalNormalProcess

Usage :

TemporalNormalProcess()
TemporalNormalProcess(model, timeGrid)
TemporalNormalProcess(covarianceModel, timeGrid)

Arguments :

covarianceModel : a StationaryCovarianceModel
model : a SecondOrderModel
timeGrid : a *RegularGrid*; the time grid over which the process is implemented

Value : a TemporalNormalProcess process

in the second usage, we fix the time grid and the model of second order (the associated temporal covariance model and spectral one)

in the third usage, we only give the temporal covariance model.

Comments : This class inherits from the NormalProcess class. The initialization stores the second model in order to reuse it for the realizations.

When calling the getRealization method, the first call might be time consuming because it calls the discretize method of the covariance model in order to compute the covariance model on the time grid and get its Cholesky factor. This is done once only.

12.6 ARMA

We suppose that the stochastic process $(X_t)_t$ follows the linear recurrence :

$$\underline{X}_t + \underline{A}_1 \underline{X}_{t-1} + \dots + \underline{A}_p \underline{X}_{t-p} = \underline{\varepsilon}_t + \underline{B}_1 \underline{\varepsilon}_{t-1} + \dots + \underline{B}_q \underline{\varepsilon}_{t-q} \quad (28)$$

that writes in dimension 1 :

$$X_t + a_1 X_{t-1} + \dots + a_p X_{t-p} = \varepsilon_t + b_1 \varepsilon_{t-1} + \dots + b_q \varepsilon_{t-q} \quad (29)$$

where $(a_i, b_i) \in \mathbb{R}$.

This class inherits from *Process*

12.6.1 ARMA

Usage :

```
ARMA()
ARMA(ARCoefficients, MACoefficients, whiteNoise)
ARMA(ARCoefficients, MACoefficients, whiteNoise, state)
```

Arguments :

ARCoefficients : an ARMACoefficients , the coefficients of the AR part of the recurrence : (a_1, \dots, a_p) of (29) and $(\underline{A}_1, \dots, \underline{A}_p)$ of (28).

MACoefficients : an ARMACoefficients , the coefficients of the MA part of the recurrence : (b_1, \dots, b_q) of (29) and $(\underline{B}_1, \dots, \underline{B}_q)$ of (28).

whiteNoise : a WhiteNoise, the white noise used for the random noise ε .

state : an ARMAState , the state of the ARMA process ie the last values pf the process.

Value : an ARMA process

in the first usage, an *ARMA(0,0)* is built with default options : the time grid is $\{0,1\}$, and the ε distribution is $\mathcal{N}(0,1)$.

in the second usage, we fix the coefficients of the linear recurrence (and the dimension of the process is deduced) and the distribution of the random noise ε .

in the third usage, we also fix the initial state which is the last p values of the process and the last q values of the noise.

Some methods :

getARCoefficients

Usage : *getARCoefficients()*

Arguments : none

Value : an ARMACoefficients, the coefficients of the linear recurrence.

*getMACoefficients***Usage :** *getMACoefficients()***Arguments :** none**Value :** an ARMACoefficients, the coefficients of the linear recurrence.*getFuture***Usage :***getFuture(N_{it})**getFuture(N_{it}, N_{real})***Arguments :***N_{it}* : an integer, the number of future instants where the process is extended*N_{real}* : an integer, the number of futures that are evaluated**Value :**in the first usage, a TimeSeries, which is one possible realization of the future using the current state of the process over the *N_{it}* next instants.in the second usage, a ProcessSample, which contains *N_{real}* possible realizations of the future of the process on the *N_{it}* next instants. Note that the time grid of each possible future begins at the last instant of the time grid associated to the time series which is extended.*getState***Usage :** *getState()***Arguments :** none**Value :** an ARMAState, the state of the ARMA which is the last *p* values of the process and the last *q* values of the random noise ε .*getWhiteNoise***Usage :** *getWhiteNoise()***Arguments :** none**Value :** a WhiteNoise, the white noise ε of (28).*setWhiteNoise***Usage :** *setWhiteNoise(whiteNoise)***Arguments :** *whiteNoise*, a WhiteNoise**Value :** Fix the white noise ε of (28).*computeNThermalization***Usage :** *computeNThermalization(ε)***Arguments :** ε , a positive real**Value :** an integer, the number of iterations of the ARMA process before being stationary and independent of its current state evaluated with the precision ε :

$$N_{ther} > E\left[\frac{\ln \varepsilon}{\ln \max_{i,j} |r_{ij}|}\right] \quad (30)$$

where $E[]$ is the integer part and the r_i are the roots of the polynomials (given here ion dimension 1) :

$$\Phi(\underline{r}) = \underline{r}^p + \sum_{i=1}^p a_i \underline{r}^{p-i} \quad (31)$$

getNThermalization

Usage : *getNThermalization()*

Arguments : none

Value : an integer, the number of iterations that the ARMA process uses before being independent from its actual state evaluated with the default precision $\varepsilon = 2^{-53} \equiv 10^{-16}$.

setNThermalization

Usage : *setNThermalization(n)*

Arguments : n , an integer

Value : Set the number of iterations of initialization for the ARMA process

12.6.2 ARMACoefficients

Usage :

ARMACoefficients(size, dimension)

ARMACoefficients(point)

ARMACoefficients(collection)

Arguments :

size : a integer, the number of elements of the ARMACoefficients class

dimension : an integer, the dimension of each coefficient

point : a NumericalPoint, the coefficients of the recurrence (in dimension 1)

collection : a SquareMatrixCollection which contains SquareMatrix of same dimension

Value : an ARMACoefficients

in the first usage, we fix the dimension and the number of elements.

in the second usage, we fix the dimension to 1 and we get the coefficients of the recurrence

in the third usage, we get the coefficients of the recurrence as matrix (and so the dimension)

Some methods :*getSize*

Usage : *getSize()*

Arguments : none

Value : an integer, the size of the ARMACoefficients (number of coefficients)

getDimension

Usage : *getDimension()*

Arguments : none

Value : an integer, the dimension each coefficients

add

Usage : *add(coeffcient)*

Arguments : *coeffcient* : a SquareMatrix

Value : an ARMACoefficients of size *size* + 1

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the ARMACoefficients

setName

Usage : *setName(name)*

Arguments : name : a string

Value : the ARMACoefficients is named *name*

12.6.3 ARMAState

Usage :

ARMAState(values, noise)

Arguments :

values : a NumericalSample, last observations of an ARMA process

noise : a NumericalSample, last observations of a random noise

Value : an ARMAState

Some methods :

getSize

Usage : *getSize()*

Arguments : none

Value : an integer, the size of the ARMACoefficients (number of coefficients)

getDimension

Usage : *getDimension()*

Arguments : none

Value : an integer, the dimension of the state (and thus of the process)

getX

Usage : *getX()*

Arguments : none

Value : a NumericalSample (of size p), the last p values of the ARMA

getEpsilon

Usage : *getEpsilon()*

Arguments : none

Value : a NumericalSample (of size q), the last q noise values of the ARMA

setX

Usage : *setX(myLastProcessValues)*

Arguments : *myLastValues*, a NumericalSample of size p

Value : None. We fix the last p values of the process

setEpsilon

Usage : *setEpsilon(myLastNoiseValues)*

Arguments : *myLastNoiseValues*, a NumericalSample of size q

Value : None. We fix the last q values of the random noise

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the ARMAState

setName

Usage : *setName(name)*

Arguments : name : a string

Value : the ARMAState is named *name*

12.6.4 BoxCoxFactory

Usage :

```
BoxCoxFactory()
BoxCoxFactory(alpha)
```

Arguments :

alpha : a NumericalScalar. The value to be fixed by user to ensure positive character of all values

Value : a BoxCoxFactory

in the first usage, the used value is 0

in the second usage, the user fix the "translate" value

Some methods :

build

Usage : *build*(*inTS*)

Arguments : *inTS* : a TimeSeries , the time series from which the Box Cox transformation parameter is estimated

Value : a BoxCoxTransform which enables to transform the time series \underline{Y}_t into $h_\lambda(\underline{Y}_t)$ such that $\text{Var}[h_\lambda(\underline{Y}_t)]$ is constant with respect to the time. In the scalar case, we note h_λ is parametered by a scalar λ :

$$h_\lambda(y) = \begin{cases} \frac{y^\lambda - 1}{\lambda} & \lambda \neq 0 \\ \log(y) & \lambda = 0 \end{cases} \quad (32)$$

For time series of dimension $p > 1$, we operate component by component as in (33).

getName

Usage : *getName*()

Arguments : none

Value : a string, the name of the BoxCoxFactory

setName

Usage : *setName*(*name*)

Arguments : *name* : a string

Value : the BoxCoxFactory is named *name*

12.6.5 BoxCoxTransform

This class inherits from *SpatialFunction*. The Box Cox transformation is defined as following. In the scalar case, we note h_λ , parametered by a scalar λ :

$$h_\lambda(y) = \begin{cases} \frac{y^\lambda - 1}{\lambda} & \lambda \neq 0 \\ \log(y) & \lambda = 0 \end{cases} \quad (33)$$

For time series of dimension $p > 1$, we operate component by component as in (33), which defines the parameter $\underline{\lambda}$.

Usage :

```
BoxCoxTransform()
BoxCoxTransform(lambdaPoint)
BoxCoxTransform(lambdaCollection)
BoxCoxTransform(lambdaScalar)
```

Arguments : λ : the parameter of the Box Cox transformation h_λ . This last one might be :

```
lambdaPoint : a NumericalPoint.
lambdaCollection : a NumericalScalarCollection.
lambdaCollection : a NumericalScalar.
```

Value : a BoxCoxTransform that contains the Box Cox transformation and its inverse.

in the second and third usages, we fix the dimension and the value of $\underline{\lambda}$.
in the fourth usage, the dimension is one.

Some methods :

```
()
```

Usage : (y)

Arguments : y : a TimeSeries, the time series $(\underline{y}_{t_1}, \dots, \underline{y}_{t_N})$

Value : a TimeSeries, the time series $(\underline{z}_{t_1}, \dots, \underline{z}_{t_N})$ with stabilized variance, such that for each instant t_i we apply the function :

$$\begin{aligned} \mathbb{R}^{p+1} &\mapsto \mathbb{R}^p \\ (t_i, \underline{y}_{t_i}) &\mapsto h_\lambda(\underline{y}_{t_i}) \end{aligned} \quad (34)$$

or

$$\begin{aligned} \mathbb{R}^{p+1} &\mapsto \mathbb{R}^p \\ (t_i, \underline{y}_{t_i}) &\mapsto h_\lambda(\underline{y}_{t_i} + \underline{\alpha}) \end{aligned} \quad (35)$$

```
getInverse
```

Usage : *getInverse()*

Arguments : none

Value : an InverseBoxCoxTransform, the inverse Box Cox transformation that enables to refind the initial time series $(\underline{y}_{t_1}, \dots, \underline{y}_{t_N})$ from the time series $(z_{t_1}, \dots, z_{t_N})$ such that :

$$\begin{aligned} \mathbb{R}^{p+1} &\mapsto \mathbb{R}^p \\ (t_i, z_{t_i}) &\mapsto h_{\underline{\lambda}}^{-1}(z_{t_i}) \end{aligned} \quad (36)$$

or

$$\begin{aligned} \mathbb{R}^{p+1} &\mapsto \mathbb{R}^p \\ (t_i, z_{t_i}) &\mapsto h_{\underline{\lambda}}^{-1}(z_{t_i}) - \underline{\alpha} \end{aligned} \quad (37)$$

getLambda

Usage : *getLambda()*

Arguments : none

Value : a NumericalPoint. The parameter $\underline{\lambda}$ of $h_{\underline{\lambda}}$

getInputDimension

Usage : *getInputDimension()*

Arguments : none

Value : an integer, the input dimension e.g the dimension that should have the input *TimeSeries*

getOutputDimension

Usage : *getOutputDimension()*

Arguments : none

Value : an integer, the output dimension of the transform function. Its equal here to the input dimension.

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the BoxCoxTransform

setName

Usage : *setName(name)*

Arguments : name : a string

Value : the BoxCoxTransform is named *name*

12.6.6 InverseBoxCoxTransform

This class inherits from *SpatialFunction*.

The inverse of the Box Cox transformation is defined as following.

In the scalar case, h_λ^{-1} is parametered by a scalar λ :

$$h_\lambda^{-1}(y) = \begin{cases} (\lambda y + 1)^{\frac{1}{\lambda}} & \lambda \neq 0 \\ \exp(y) & \lambda = 0 \end{cases} \quad (38)$$

For time series of dimension $p > 1$, we operate component by component as in (38), which defines the parameter $\underline{\lambda}$.

Usage : Generally, it is obtained as the result of a *BoxCoxTransform.getInverse()*. But it is possible to directly define the inverse transformation.

InverseBoxCoxTransform()

InverseBoxCoxTransform(lambdaPoint)

InverseBoxCoxTransform(lambdaCollection)

InverseBoxCoxTransform(lambdaScalar)

Arguments :

lambdaPoint : a NumericalPoint.

lambdaCollection : a NumericalScalarCollection.

lambdaCollection : a NumericalScalar.

Value : an InverseBoxCoxTransform, the inverse Box Cox transformation.

in the second and third usages, we fix the dimension and the value of $\underline{\lambda}$.

in the fourth usage, the dimension is one.

Some methods :

()

Usage : (*Z*)

Arguments : a *Z* : *TimeSeries*, the time series $(z_{t_1}, \dots, z_{t_N})$.

Value : a *TimeSeries*, the time series $(\underline{y}_{t_1}, \dots, \underline{y}_{t_N})$ such that for each instant t_i we apply the function

$$\begin{aligned} &: \\ &\mathbb{R}^{p+1} \quad \mapsto \quad \mathbb{R}^p \\ &(t_i, z_{t_i}) \quad \mapsto \quad \underline{y}_{t_i} = h_{\underline{\lambda}}^{-1}(z_{t_i}) \end{aligned} \quad (39)$$

getLambda

Usage : *getLambda()*

Arguments : none

Value : a NumericalPoint. The lambda values of the transform

getInputDimension

Usage : *getInputDimension()*

Arguments : none

Value : an integer, the input dimension e.g the dimension p of the associated times series.

getOutputDimension

Usage : *getOutputDimension()*

Arguments : none

Value : an integer, the output dimension of the transform function. Its equal here to the input dimension p of the associated times series.

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the InverseBoxCoxTransform

setName

Usage : *setName(name)*

Arguments : name : a string

Value : the InverseBoxCoxTransform is named *name*

12.6.7 TrendFactory

Usage :

TrendFactory(basisSequence, fittingAlgorithm)

Arguments :

myBasisSequenceFactory : a BasisSequenceFactory such as *LAR*

myFittingAlgorithm : a FittingAlgorithm such as *KFold* or *CorrectedLeaveOneOut*

Value : a TrendFactory

Some methods :*build*

Usage : *build(inTS, basis)*

Arguments :

inTS: a TimeSeries, the time series on which the trend is built

inTS : a NumericalMathFunctionCollection, on which the trend is decompos

Value : a TrendTransform which contains the best dual combination of *basis* in order to model the time series *inTS*, using the regression strategy *myBasisSequenceFactory* and the fitting algorithm *myFittingAlgorithm*.

getBasisSequenceFactory

Usage : *getBasisSequenceFactory()*

Arguments : none

Value : a *BasisSequenceFactory*, the one used the TrendFactory.

getFittingAlgorithm

Usage : *getFittingAlgorithm()*

Arguments : none

Value : a FittingAlgorithm, the algorithm used by the TrendFactory.

setBasisSequenceFactory

Usage : *setBasisSequenceFactory(basis)*

Arguments : *basis* : a BasisSequenceFactory

Value : None. We fix the basis sequence factory.

setFittingAlgorithm

Usage : *setFittingAlgorithm(fittingAlgorithm)*

Arguments : fittingAlgorithm, a FittingAlgorithm that estimates the empirical error on each sub-basis

Value : None. We fix the fitting algorithm.

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the TrendFactory.

setName

Usage : *setName(name)*

Arguments : *name* : a string

Value : None. The TrendFactory is named *name*.

12.6.8 TrendTransform

This class inherits from *TemporalFunction*. Let $(\underline{Y}_t)_t$ be a time series of dimension p such that :

$$\underline{Y}_t = \underline{f}(t) + \underline{X}_t \quad (40)$$

where $f : \mathbb{R} \rightarrow \mathbb{R}^p$ is the trend function and \underline{X}_t is stationary .

Usage : *TrendTransform(f)*. Generally, it is obtained as the result of a *TrendFactory.build(...)*. But it is possible to directly fix the evaluation function to be used.

Arguments :

f : a NumericalMathFunction

Value : a TrendTransform, the trend function.

Some methods :

()

Usage : (X)

Arguments : X : TimeSeries, the time series $(\underline{x}_{t_1}, \dots, \underline{x}_{t_N})$

Value : a TimeSeries, the time series $(\underline{y}_{t_1}, \dots, \underline{y}_{t_N})$ such that :

$$\begin{aligned} \mathbb{R}^{p+1} &\mapsto \mathbb{R}^p \\ (t_i, \underline{x}_{t_i}) &\mapsto \underline{y}_{t_i} = \underline{x}_{t_i} + \underline{f}(t_i) \end{aligned} \quad (41)$$

getFunction

Usage : *getFunction()*

Arguments : none

Value : a NumericalMathFunction, the function that evaluates the trend :

$$\begin{aligned} \mathbb{R} &\mapsto \mathbb{R}^p \\ t &\mapsto \underline{f}(t) = \sum_{j=1}^k \alpha_j \underline{f}_j(t) \end{aligned} \quad (42)$$

getInputDimension

Usage : *getInputDimension()*

Arguments : none

Value : an integer, the input dimension of the trend function. As we consider a trend (time dependency), the input dimension of the function should be one.

getOutputDimension

Usage : *getOutputDimension()*

Arguments : none

Value : an integer, the output dimension p of the trend function.

getInverse

Usage : *getInverse()*

Arguments : none.

Value : an InverseTrendTransform which contains the inverse trend function that can be used on a time series $(\underline{y}_{t_1}, \dots, \underline{y}_{t_N})$ thanks to the operand $()$, in order to create the time series $(\underline{x}_{t_1}, \dots, \underline{x}_{t_N})$ such that :

$$\begin{aligned} \mathbb{R}^{p+1} &\mapsto \mathbb{R}^p \\ (t, \underline{y}_{t_i}) &\mapsto \underline{x}_{t_i} = \underline{y}_{t_i} - \underline{f}(t_i) \end{aligned} \quad (43)$$

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the TrendTransform.

setName

Usage : *setName(name)*

Arguments : name : a string

Value : the TrendTransform is named *name*.

12.6.9 InverseTrendTransform

This class inherits from *TemporalFunction*.

Usage : Generally, it is obtained as the result of a *TrendTransform.getInverse()*. But it is possible to directly fix the evaluation function $f : \text{InverseTrendTransform}(f)$

Arguments :

f : a NumericalMathFunction

Value : an InverseTrendTransform, the inverse trend function.

Some methods :

()

Usage : (X)

Arguments : X : TimeSeries, the time series $(\underline{x}_{t_1}, \dots, \underline{x}_{t_N})$.

Value : a TimeSeries, the time series $(\underline{y}_{t_1}, \dots, \underline{y}_{t_N})$ such that for each instant t_i we apply the function

$$\begin{aligned} \mathbb{R}^{p+1} &\mapsto \mathbb{R}^p \\ (t_i, \underline{x}_{t_i}) &\mapsto \underline{y}_{t_i} = \underline{x}_{t_i} - \underline{f}(t_i) \end{aligned} \quad (44)$$

getInputDimension

Usage : *getInputDimension()*

Arguments : none

Value : an integer, the input dimension of the trend function (here 1 as it is a time dependency).

getOutputDimension

Usage : *getOutputDimension()*

Arguments : none.

Value : an integer, the output dimension p of the transform function.

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the InverseTrendTransform.

setName

Usage : *setName(name)*

Arguments : *name* : a string

Value : the InverseTrendTransform is named *name*.

12.7 ARMAFactory

The class enables to estimate the coefficients of an ARMA process using a realization or a sample of realizations

12.7.1 ARMAFactory

This class is the interface of *ARMAFactoryImplementation*.

Usage :

ARMAFactory(myARMAFactoryImplementation)

Arguments : *myARMAFactoryImplementation* : the implementation of an ARMA factory. For example, a Whittle factory.

Value : a ARMAFactory

Some methods :

getName

Usage : *getName()*

Arguments : none

Value : a string, the name of the ARMAFactory

setName

Usage : *setName(name)*

Arguments : name : a string

Value : the ARMAFactory, named *name*

12.7.2 WhittleFactoryState

This class inherits from *PersistenObject*.

Usage :

WhittleFactoryState()

WhittleFactoryState(p, theta, sigma2, criteria, timeGrid)

Arguments :

p : integer. The order of the AR part of the estimate.

theta : NumericalPoint. The coefficients of the scalar ARMA process, with the *p* coefficients of the AR part first, followed by the *q* coefficients of the MA part.

sigma2 : real. Estimate of the white noise variance.

criteria : NumericalPoint. The information criteria associated with the estimate.

timeGrid : TimeGrid. The time grid over which the estimated ARMA process is defined. m estimate

Value : a WhittleFactoryState

in the second usage, the object stores the minimal sufficient information on a particular estimation step of the WhittleFactory class, from which all the relevant other information can be recovered.

Some methods :

getARCoefficients

Usage : *getARCoefficients()*

Arguments : None

Value : ARMACoefficient. the AR coefficients of the estimate.

getARMA

Usage : *getARMA()*

Arguments : None

Value : ARMA. The ARMA process associated with the estimated parameters.

getMACoefficients

Usage : *getMACoefficients()*

Arguments : None

Value : ARMACoefficient. the MA coefficients of the estimate.

getInformationCriteria

Usage : *getInformationCriteria()*

Arguments : None

Value : NumericalPoint. The information criteria associated with the estimate. There are 3 information criterion that are computed: the corrected AIC, the AIC and the BIC, stored in this order.

getP

Usage : *getP()*

Arguments : None

Value : integer. The AR order of the estimate.

getQ

Usage : *getQ()*

Arguments : None

Value : integer. The MA order of the estimate.

getSigma2

Usage : *getSigma2()*

Arguments : None

Value : real. The estimate of the white noise variance.

getTheta

Usage : *getTheta()*

Arguments : None

Value : NumericalPoint. The ARMA coefficients estimate as manipulated during the optimization step.

getTimeGrid

Usage : *getTimeGrid()*

Arguments : None

Value : TimeGrid. The time grid over which the ARMA process is defined, as given by the data.

getWhiteNoise

Usage : *getWhiteNoise()*

Arguments : None

Value : WhiteNoise. The scalar white noise process associated with the ARMA process. It is build using a Normal distribution with zero mean and with the estimated variance.

12.7.3 WhittleFactory

This class inherits from *ARMAFactoryImplementation*.

Usage :

WhittleFactory()

WhittleFactory(p, q, invertible)

WhittleFactory(pIndices, qIndices, invertible)

Arguments :

p : integer. The order of the AR part to be tested.

q : integer. The order of the MA part to be tested.

invertible : boolean. Flag to restrict the estimation to invertible ARMA processes.

pIndices : Indices. The possible orders of the AR part to be tested.

qIndices : Indices. The possible orders of the MA part to be tested. m estimate

Value : a WhittleFactory

in the first usage, the Whittle factory is such that a normal white noise will be identified.

in the second usage, we fix the order of the ARMA (AR size and MA size) process that will be identified, with a control on its invertibility,

in the third usage, we fix a range of orders for the AR and the MA parts. All the possible combinations will be tested.

Some methods :

build

Usage : *build(series)*

build(series, criteria)

build(sample)

build(sample, criteria)

Arguments : *series* : a TimeSeries

criteria : a NumericalPoint

sample : a ProcessSample

Value : an ARMA process, the model that matches the best the given time series or process sample. The comparison is made using the spectral density built using the given data and the theoretical spectral density of the ARMA process.

If given, the *criteria* vector is filled by three scalar values: the corrected AIC criterion, the AIC criterion and the BIC criterion. The best ARMA process is selected according to the corrected AIC criterion.

getSpectralModelFactory

Usage : *getSpectralModelFactory()*

Arguments : None.

Value : a SpectralModelFactory object. Returns the spectral factory used to estimate the spectral density based on the data during the estimation step.

setSpectralModelFactory

Usage : *setSpectralModelFactory(spectralFactory)*

Arguments : a SpectralModelFactory.

Value : None. Set the spectral factory used to estimate the spectral density based on the data during the estimation step.

disableHistory

Usage : *disableHistory()*

Arguments : None.

Value : None. Deactivate the history mechanism, it means the trace of all the tested models and their associated information criteria.

enableHistory

Usage : *enableHistory()*

Arguments : None.

Value : None. Activate the history mechanism, it means the trace of all the tested models and their associated information criteria.

isHistoryEnabled

Usage : *isHistoryEnabled()*

Arguments : None.

Value : a logical value. True if the history mechanism is activated. It is activated by default.

resetHistory

Usage : *resetHistory()*

Arguments : None.

Value : None. Clear the history of the factory.

getHistory

Usage : *getHistory()*

Arguments : None.

Value : a collection of WhittleFactoryState objects. Returns the collection of all the states that have been built during the estimation phase.

setVerbose

Usage : *setVerbose(verbose)*

Arguments : a logical value.

Value : None. Activate the verbose mode of the factory during both the exploration of the possible models and the optimization steps.

getVerbose

Usage : *getVerbose()*

Arguments : None.

Value : a logical value telling if the verbose mode has been activated.

setStartingPoints

Usage : *setStartingPoints(points)*

Arguments : a collection of NumericalPoints.

Value : None. Set the starting points in the (p, q) parameters space during the optimization step, for each pair of orders that will be tested.

getStartingPoints

Usage : *getStartingPoints()*

Arguments : None.

Value : a collection of NumericalPoints. Returns the starting points in the (p, q) parameters space during the optimization step, for each pair of orders that will be tested.

12.8 RandomWalk

Usage :

RandomWalk(origin, distribution)

RandomWalk(origin, distribution, timeGrid)

Arguments :

origin : a NumericalPoint, the starting point of the random walk.

distribution : a Distribution, the distribution used for the steps of the random walk.

timeGrid : a RegularGrid, the time grid over which the realizations are observed

Value : a RandomWalk process

in the first usage, we fix the the origin and distribution of the process. They must have a common dimension, which is also the dimension of the process. The time grid is by default reduced to one time stamp $t = 0$.

in the second usage, we fix also the time grid of the observations.

Some methods :

getOrigin

Usage : *getOrigine()*

Arguments : none

Value : a NumericalPoint, the the starting point of the random walk.

setOrigin

Usage : *setOrigin(origin)*

Arguments : *origin*, a NumericalPoint

Value : none. Fix the starting point of the random walk.

getDistribution

Usage : *getDistribution()*

Arguments : none

Value : a Distribution, the distribution used to generate the steps of the random walk.

setDistribution

Usage : *setDistribution(distribution)*

Arguments : *distribution*, a Distribution

Value : none. Fix the distribution for the steps of the random walk.

12.9 WhiteNoise

Usage :

WhiteNoise(distribution)

WhiteNoise(distribution,timeGrid)

Arguments :

distribution : a Distribution, the distribution used for the realization of the white noise. Care! A white noise has a zero mean and finite standard deviation distribution!

timeGrid : a RegularGrid, the time grid over which the realizations are observed

Value : a WhiteNoise process

in the first usage, we fix the distribution and so the dimension of the process. The time grid is by default reduced to one time stamp $t = 0$.

in the second usage, we fix also the time grid of the observations.

Some methods :

getDistribution

Usage : *getDistribution()*

Arguments : none

Value : a Distribution, the distribution used to generate random noise at time t

setDistribution

Usage : *setDistribution(distribution)*

Arguments : *distribution*, a Distribution

Value : none. Fix the distribution for the realization of noise at time t

12.10 CompositeProcess

This class inherits from *Process*.

The objective is to build a stochastic process upon a function and a process such as *ARMA* or *NormalProcess* for example.

12.10.1 CompositeProcess

Usage :

CompositeProcess()

CompositeProcess(*function*, *AntecedentProcess*)

Arguments :

function : a *DynamicalFunction*, the function of composition

AntecedentProcess : a *Process*. The input process considered.

Value : a *CompositeProcess* process

in the second usage, we fix the process all elements of composition $\underline{Y} = \text{function}(\underline{X})$ with \underline{X} a process.

Some methods :

getFunction

Usage : *getFunction*()

Arguments : none

Value : a *DynamicalFunction*, the function used for the evaluation.

getAntecedent

Usage : *getAntecedent*()

Arguments : none

Value : a *Process*, the proces \underline{X} such as the current proces is obtained by $\underline{Y} = f(\underline{X})$.

13 Threshold probability : Reliability algorithms

13.1 Reliability Algorithms

13.1.1 Analytical

Usage : *Analytical(nearestPointAlgorithm, event, physicalStartingPoint)*

Arguments :

nearestPointAlgorithm : a NearestPointAlgorithm, the optimization algorithm which will be used to research the design point

event : a Event, the event we want to evaluate the probability

physicalStartingPoint : a NumericalPoint, the starting point of the optimization research, declared in the physical space

Some methods :

getAnalyticalResult

Usage : *getAnalyticalResult()*

Arguments : none

Value : a AlgorithmAnalyticalResult, the result structure which contains results

getEvent

Usage : *getEvent()*

Arguments : none

Value : a Event, the event we want to evaluate the probability

getNearestPointAlgorithm

Usage : *getNearestPointAlgorithm()*

Arguments : none

Value : a NearestPointAlgorithm, the optimization algorithm which will be used to research the design point

getPhysicalStartingPoint

Usage : *getPhysicalStartingPoint()*

Arguments : none

Value : a NumericalPoint, the starting point of the optimization research, declared in the physical space

run

Usage : *run()*

Arguments : none

Value : it performs the research design point and creates a AnalyticalResult, the structure result which is accessible with the method *getAnalyticalResult()*.

The methods *getEvent*, *getNearestPointAlgorithm* and *getPhysicalStartingPoint* are associated to a *setMethod*.

Derivative Classes : FORM ans SORM

13.1.2 AnalyticalResult

Usage : structure created by the method `run()` of a `Analytical` and obtained thanks to the method `getAnalyticalResult`

Some methods :

drawHasoferReliabilityIndexSensitivity

Usage : `drawHasoferReliabilityIndexSensitivity()`

Arguments : none

Value : a `GraphCollection` (the collection of two barplots) drawing the sensitivity of the Hasofer Reliability Index to the parameters of the marginals of the probabilistic input vector (first graph) and to the parameters of the dependence structure of the probabilistic input vector (second graph).

drawImportanceFactors

Usage :

`drawImportanceFactors()`

`drawImportanceFactors(True)`

Arguments : `True` : a Boolean. When 'True', the importance factors are evaluated as the square of the co-factors of the design point in the \mathbf{U} -space (see (45)) :

$$\alpha_i^2 = \frac{(u_i^*)^2}{\beta_{HL}^2} \quad (45)$$

When not specified, the importance factors are evaluated as the square of the co-factors of the design point in the \mathbf{Y} -space (see (46)) :

$$\alpha_i^2 = \frac{(y_i^*)^2}{\|\underline{y}^*\|^2} \quad (46)$$

where

$$\mathbf{Y}^* = \begin{pmatrix} E^{-1} \circ F_1(X_1^*) \\ E^{-1} \circ F_2(X_2^*) \\ \vdots \\ E^{-1} \circ F_n(X_n^*) \end{pmatrix}. \quad (47)$$

whith \underline{X}^* is the design point in the physical space and E the univariate standard CDF of the elliptical space.

In the case where the input distribution of \underline{X} has an elliptical copula C_E , then E has the same type as C_E .

In the case where the input distribution of \underline{X} has a copula C which is not elliptical, then $E = \Phi$ where Φ is the CDF of the standard normal.

Value : a `Graph`, the pie of the importance factors of the probabilistic variables

getHasoferReliabilityIndex

Usage : `getHasoferReliabilityIndex()`

Arguments : none

Value : a real positive value, the Hasofer Reliability Index

getHasoferReliabilityIndexSensitivity

Usage : *getHasoferReliabilityIndexSensitivity()*

Arguments : none

Value : a Sensitivity, the sensitivities of the Hasofer Reliability Index to the parameters of the probabilistic input vector (marginals and dependence structure)

getImportanceFactors

Usage :

getImportanceFactors()

getImportanceFactors(True)

Arguments : *True* : a Boolean. When 'True', the importance factors are evaluated as the square of the co-factors of the design point in the \mathbf{U} -space (see (45)).

When not specified, the importance factors are evaluated as the square of the co-factors of the design point in the \mathbf{Y} -space (see (46)).

Value : a NumericalPoint, the importance factors of probabilistic variables

getIsStandardPointOriginInFailureSpace

Usage : *getIsStandardPointOriginInFailureSpace()*

Arguments : none

Value : a boolean which indicates whether the origin of the standard space is in the failure space

getLimitStateVariable

Usage : *getLimitStateVariable()*

Arguments : none

Value : a Event, the event we evaluated the probability

getMeanPointInStandardEventDomain

Usage : *getMeanPointInStandardEventDomain()*

Arguments : none

Value : a NumericalPoint, the mean point of the standard space distribution restricted to the event domain : $\frac{1}{E_1(-\beta)} \int_{\beta}^{\infty} u_1 p_1(u_1) du_1$ where E_1 is the spheric univariate distribution of the standard space and β the reliability index.

getPhysicalSpaceDesignPoint

Usage : *getPhysicalSpaceDesignPoint()*

Arguments : none

Value : a NumericalPoint, the starting point of the optimization research, declared in the physical space

getStandardSpaceDesignPoint

Usage : *getStandardSpaceDesignPoint()*

Arguments : none

Value : a NumericalPoint, the starting point of the optimization research, declared in the standard space

Derivative Classes : FORMResult and SORMResult

13.1.3 Event

Usage :

Event(antecedent, comparisonOperator, threshold)
Event(antecedent, comparisonOperator, threshold, name)
Event(antecedent, domain)
Event(antecedent, domain, name)

Arguments :

antecedent : a RandomVector, of dimension 1 : the output variable of interest
comparisonOperator : a ComparisonOperator, the comparison operator which is equal to *Less*, *Greater*, *LessOrEqual* or *GreaterOrEqual*
threshold : a real value, the threshold we want to compare to *antecedent*
domain : a domain, the domain failure
name : a string, the name of the event

Some methods :

getDimension

Usage : *getDimension()*

Arguments : none

Value : an integer, the dimension of the probabilistic input vector

getNumericalSample

Usage : *getNumericalSample(size)*

Arguments : *size* : an integer, the size of the numerical sample generated

Value : a NumericalSample filled with boolean values (1 for the realization of the event and 0 else)
 : *size* realizations of the event (considered as a Bernoulli variable)

getOperator

Usage : *getOperator()*

Arguments : none

Value : a ComparisonOperator, the comparison operator of the event

getRealization

Usage : *getRealization()*

Arguments : none

Value : a NumericalPoint of dimension 1, filled with a boolean value (1 for the realization of the event and 0 else) : one realization of the event (considered as a Bernoulli variable)

getThreshold

Usage : *getThreshold()*

Arguments : none

Value : a real value, the threshold of the event

Derivative Class : StandardEvent

13.1.4 StandardEvent

This class inherits from Event.

Usage :

StandardEvent(antecedent, comparisonOperator, threshold)
StandardEvent(antecedent, comparisonOperator, threshold, name)
StandardEvent(event)

Arguments :

antecedent : a RandomVector, of dimension 1 : the output variable of interest
comparisonOperator : a ComparisonOperator, the comparison operator which is equal to *Less*, *Greater*, *LessOrEqual* or *GreaterOrEqual*
threshold : a real value, the threshold we want to compare to *antecedent*
name : a string, the name of the event
event : a Event, the physical event associated to the standard event

13.1.5 FORM

This class inherits from Analytical.

Usage : *FORM(nearestPointAlgorithm, event, physicalStartingPoint)*

Arguments :

nearestPointAlgorithm : a NearestPointAlgorithm, the optimization algorithm which will be used to research the design point

event : a Event, the event in the physical space we want to evaluate the probability

physicalStartingPoint : a NumericalPoint, the starting point of the optimization research, declared in the physical space

Some methods :

getResult

Usage : *getResult()*

Arguments : none

Value : a FORMResult, structure containing all the results of the FORM analysis

run

Usage : *run()*

Arguments : none

Value : it creates a FORMResult, the optimization result which is accessible with the method *getResult()*.

13.1.6 FORMResult

This class inherits from AnalyticalResult.

Usage : structure created by the method `run()` of a FORM and obtained thanks to the method `getResult()`

Some methods :

drawEventProbabilitySensitivity

Usage : *drawEventProbabilitySensitivity()*

Arguments : none

Value : a GraphCollection (the collection of two barplots) drawing the sensitivities of the FORM Probability with regards to the parameters of the probabilistic input vector (first graph) and to parameters of the dependence structure of the probabilistic input vector (second graph)

getEventProbability

Usage : *getEventProbability()*

Arguments : none

Value : a positive real value, the FORM probability of the event

getEventProbabilitySensitivity

Usage : *getEventProbabilitySensitivity()*

Arguments : none

Value : a Sensitivity, the sensitivities of the FORM Probability with regards to the parameters of the probabilistic input vector and to parameters of the dependence structure of the probabilistic input vector

getGeneralisedReliabilityIndex

Usage : *getGeneralisedReliabilityIndexHohenBichler()*

Arguments : none

Value : a real value, the generalised reliability index evaluated from the FORM Probability. The generalised reliability index from the FORM probability is equal to \pm the Hasofer reliability index according to the fact the standard space center fulfills the event or not

13.1.7 SORM

This class inherits from Analytical.

Usage : *SORM(nearestPointAlgorithm, event, physicalStartingPoint)*

Arguments :

nearestPointAlgorithm : a NearestPointAlgorithm, the optimization algorithm which will be used to research the design point

event : a Event, the event in the physical space we want to evaluate the probability

physicalStartingPoint : a NumericalPoint, the starting point of the optimization research, declared in the physical space

Some methods :

getResult

Usage : *getResult()*

Arguments : none

Value : a SORMResult, structure containing all the results of the SORM analysis

run

Usage : *run()*

Arguments : none

Value : it creates a SORMResult, the optimization result which is accessible with the method *getResult()*.

13.1.8 SORMResult

This class inherits from AnalyticalResult.

Usage : structure created by the method `run()` of a SORM and obtained thanks to the method `getResult()`

Some methods :

getEventProbabilityBreitung

Usage : `getEventProbabilityBreitung()`

Arguments : none

Value : a positive real value, the SORM Probability according to the Breitung approximation

getEventProbabilityHohenBichler

Usage : `getEventProbabilityHohenBichler()`

Arguments : none

Value : a positive real value, the SORM Probability according to the Hohen Bichler approximation

getEventProbabilityTvedt

Usage : `getEventProbabilityTvedt()`

Arguments : none

Value : a positive real value, the SORM Probability according to the Tvedt approximation

getGeneralisedReliabilityIndexBreitung

Usage : `getGeneralisedReliabilityIndexBreitung()`

Arguments : none

Value : a real value, the generalised reliability index evaluated from the Breitung SORM Probability (positive or negative according to the fact the standard space center fulfills the event or not)

getGeneralisedReliabilityIndexHohenBichler

Usage : `getGeneralisedReliabilityIndexHohenBichler()`

Arguments : none

Value : a real value, the generalised reliability index evaluated from the Hohen Bichler SORM Probability (positive or negative according to the fact the standard space center fulfills the event or not)

getGeneralisedReliabilityIndexTvedt

Usage : `getGeneralisedReliabilityIndexTvedt()`

Arguments : none

Value : a real value, the generalised reliability index evaluated from the Tvedt SORM Probability (positive or negative according to the fact the standard space center fulfills the event or not)

13.2 The Strong Maximum Test

13.2.1 StrongMaximumTest

Usage :

*StrongMaximumTest(event, standardSpaceDesignPoint, importanceLevel, ...
accuracyLevel, confidenceLevel)*

*StrongMaximumTest(event, standardSpaceDesignPoint, importanceLevel, ...
accuracyLevel, pointNumber)*

Arguments :

event : a StandardEvent,

standardSpaceDesignPoint : a NumericalPoint,

importanceLevel : a real value $\in]0, 1]$, the importance level ε .

accuracyLevel : a positive real value, the accuracy Level τ . It is recommended to take $\tau \leq 4$.

confidenceLevel : a positive real value, the confidenceLevel $(1 - q)$, must be < 1 .

pointNumber : the number of points used to perform the Strong Maximum Test on which the limit state function is evaluated

Some methods :

getAccuracyLevel

Usage : *getAccuracyLevel()*

Arguments : none

Value : a positive real value, the accuracy Level τ

getConfidenceLevel

Usage : *getConfidenceLevel()*

Arguments : none

Value : a positive real value, the confidenceLevel $(1 - q)$

getEvent

Usage : *getEvent()*

Arguments : none

Value : a StandardEvent, the event in the standard space on which is based the Strong Maximum Test

getFarDesignPointVerifyingEventPoints

Usage : *getFarDesignPointVerifyingEventPoints()*

Arguments : none

Value : a NumericalSample, the list of points of the discretized sphere which are out of the vicinity of the standard design point and which verify the event

getFarDesignPointVerifyingEventValues

Usage : *getFarDesignPointVerifyingEventValues()*

Arguments : none

Value : a NumericalSample, the list of the values of the limit state function on the points of the discretized sphere which are out of the vicinity of the standard design point and which verify the event

getFarDesignPointViolatingEventPoints

Usage : *getFarDesignPointViolatingEventPoints()*

Arguments : none

Value : a NumericalSample, the list of points of the discretized sphere which are out of the vicinity of the standard design point and which don't verify the event

getFarDesignPointViolatingEventValues

Usage : *getFarDesignPointViolatingEventValues()*

Arguments : none

Value : a NumericalSample, the list of the values of the limit state function on the points of the discretized sphere which are out of the vicinity of the standard design point and which don't verify the event

getImportanceLevel

Usage : *getImportanceLevel()*

Arguments : none

Value : a positive real value, the importance level ε

getNearDesignPointVerifyingEventPoints

Usage : *getNearDesignPointVerifyingEventPoints()*

Arguments : none

Value : a NumericalSample, the list of points of the discretized sphere which are inside the vicinity of the standard design point and which verify the event

getNearDesignPointVerifyingEventValues

Usage : *getNearDesignPointVerifyingEventValues()*

Arguments : none

Value : a NumericalSample, the list of the values of the limit state function on the points of the discretized sphere which are inside the vicinity of the standard design point and which verify the event

getNearDesignPointViolatingEventPoints

Usage : *getNearDesignPointViolatingEventPoints()*

Arguments : none

Value : a NumericalSample, the list of points of the discretized sphere which are out of the vicinity of the standard design point and which don't verify the event

getNearDesignPointViolatingEventValues

Usage : *getNearDesignPointViolatingEventValues()*

Arguments : none

Value :

a NumericalSample, the list of the values of the limit state function on the points of the discretized sphere which are inside the vicinity of the standard design point and which don't verify the event

getPointNumber

Usage : *getPointNumber()*

Arguments : none

Value : an integer, the number of points used to perform the Strong Maximum Test, evaluated by the limit state function

getStandardSpaceDesignPoint

Usage : *getStandardSpaceDesignPoint()*

Arguments : none

Value : a NumericalPoint, the standard space design point

run

Usage : *run()*

Arguments : none

Value : it performs the Strong Maximum Test.

14 Threshold probability : Simulation algorithms

14.1 RandomGenerator

Usage : *RandomGenerator()*

Arguments : none

Some methods :

Generate

Usage :

Generate()

Generate(size)

Arguments : *size* : an integer, the number of realizations required. When not given, by default taken equal to 1

Value : a NumericalPoint, the list of the required realizations of a uniform distribution on $[0, 1]$

GetState

Usage : *GetState()*

Arguments : : none

Value : a RandomGeneratorState, the state of the random generator

SetSeed

Usage : *SetSeed(n)*

Arguments : *n* : an integer which enables an easy initialisation of the random generator

Value : none. It initializes the state of the random generator

SetState

Usage : *SetState(state)*

Arguments : *state* : a RandomGeneratorState, the state of the random generator

Value : none. It initializes the state of the random generator

The *GetState* method is associated to a *SetState* one.

14.2 Wilks

This class is a static class which enables the evaluation of the Wilks number : the minimal sample size $N_{\alpha,\beta,i}$ to perform in order to guarantee that the empirical quantile α , noted $\tilde{q}_\alpha(N_{\alpha,\beta,i})$ evaluated with the $(n - i)$ th maximum of the sample, noted X_{N-i} be greater than the theoretical quantile q_α with a probability at least β :

$$\mathbb{P}(\tilde{q}_\alpha(N_{\alpha,\beta,i}) > q_\alpha) > \beta$$

where $\tilde{q}_\alpha(N_{\alpha,\beta,i}) = X_{n-i}$

This class proposes the methods :

ComputeSampleSize

Usage : *ComputeSampleSize(alpha, beta, i)*

Arguments :

alpha : a real value, order of the quantile we want the evaluate, must be in (0, 1).

beta : confidence on the evaluation of the empirical quantile, must be in (0, 1).

i : rank of the maximum which will evaluate the empirical quantile, by default $i = 0$ (maximum of the sample)

Value : an integer, the Wilks number.

computeQuantileBound

Usage : *computeQuantileBound(alpha, beta, i)*

Arguments :

alpha : a real value, order of the quantile we want the evaluate, must be in (0, 1).

beta : confidence on the evaluation of the empirical quantile, must be in (0, 1).

i : rank of the maximum which will evaluate the empirical quantile, by default $i = 0$ (maximum of the sample)

Value : a real value, the estimate of the quantile upper bound for the given quantile level, at the given confidence level and using the given upper statistics.

14.3 Simulation

Usage : A *Simulation* object can be created only through its derived classes: *DirectionalSampling*, *ImportanceSampling*, *LHS*, *MonteCarlo*, *PostAnalyticalControlledImportanceSampling*, *PostAnalyticalImportanceSampling*, *QuasiMonteCarlo*, *RandomizedLHS*, *RandomizedQuasiMonteCarlo*.

Some methods :

getBlockSize

Usage : *getBlockSize()*

Arguments : none

Value : an integer, the number of terms in the probability simulation estimator grouped together

Details : for Monte Carlo, LHS and Importance Sampling methods, we recommend to use *BlockSize* = number of available CPUs; for the Directional Sampling, we recommend to use *BlockSize* = 1

getConvergenceStrategy

Usage : *getConvergenceStrategy()*

Arguments : none

Value : a *HistoryStrategy*, the storage strategy used to store the values of the probability estimator and its variance during the simulation algorithm

getEvent

Usage : *getEvent()*

Arguments : none

Value : an *Event*, which we want to evaluate the probability

getInputStrategy (removed starting from the 0.16 release, see *NumericalMathFunction.getInputHistory()*)

Usage : *getInputStrategy()*

Arguments : none

Value : a *HistoryStrategy*, the storage strategy used to store the input random vector sample used to evaluate the probability estimator of the event probability

getMaximumCoefficientOfVariation

Usage : *getMaximumCoefficientOfVariation()*

Arguments : none

Value : a real value, the maximum coefficient of variation of the simulated sample

getMaximumOuterSampling

Usage : *getMaximumOuterSampling()*

Arguments : none

Value : an integer, the maximum number of groups of terms in the probability simulation estimator

Details : for all the methods excepted the *DirectionalSampling* one, the maximum number of evaluations of the limit state function defining the event is : $MaximumOuterSampling \times BlockSize$

getMaximumStandardDeviation

Usage : *getMaximumStandardDeviation()*

Arguments : none

Value : a positive real value , the standard deviation maximum of the estimator

getOutputStrategy (removed starting from the 0.16 release, see `NumericalMathFunction.getOutputHistory()`)

Usage : *getOutputStrategy()*

Arguments : none

Value : a `HistoryStrategy`, the storage strategy used to store the output random vector sample used to evaluate the probability estimator of the event probability

getResult

Usage : *getResult()*

Arguments : none

Value : a `SimulationResult`, the structure containing all the results obtained after simulation and created by the method `run()`

run

Usage : *run()*

Arguments : none

Value : it launches the simulation and creates a `SimulationResult`, structure containing all the results obtained after simulation

setConvergenceStrategy

Usage : *setConvergenceStrategy(myHistoryStrategy)*

Arguments : *myHistoryStrategy* : a `HistoryStrategy`, the storage strategy used to store the values of the probability estimator and its variance during the simulation algorithm.

Value : none

setInputOutputStrategy (removed starting from the 0.16 release, see `NumericalMathFunction.enableHistory()`)

Usage : *setInputOutputStrategy(myInputOutputStrategy)*

Arguments : *myInputOutputStrategy* : a `HistoryStrategy`, the storage strategy used to store the input and output random vector samples used during the simulation. The two strategies (the one for the input vectors and the one for the output vectors) have to be of the same kind with the same parameters, which is enforced by this method.

Value : none

Only the *getBlockSize*, *getMaximumCoefficientOfVariation*, *getMaximumOuterSampling* methods have an associated *setMethod*.

Derivative Classes :

DirectionalSampling

ImportanceSampling

LHS

MonteCarlo
PostAnalyticalControlledImportanceSampling
PostAnalyticalImportanceSampling
QuasiMonteCarlo
RandomizedLHS
RandomizedQuasiMonteCarlo

14.4 MonteCarlo

This class inherits from Simulation.

Usage : *MonteCarlo(event)*

Arguments : *event* : an Event, the event we want to evaluate the probability

14.5 LHS

This class inherits from Simulation.

Usage : $LHS(event)$

Arguments : $event$: an Event, the event we want to evaluate the probability

Details : Be carefull, to be valid, the LHS sampling method requires that the multi-variate distribution have an independent copula. All events can be reduced to this case using the Roseblatt transformation.

14.6 RandomizedLHS

This class inherits from Simulation.

Usage : *RandomizedLHS(event)*

Arguments : *event* : an Event, the event we want to evaluate the probability

Details : Be carefull, to be valid, the RandomizedLHS sampling method requires that the multi-variate distribution have an independent copula. All events can be reduced to this case using the Rosemblatt transformation.

14.7 DirectionalSampling

14.7.1 DirectionalSampling

This class inherits from Simulation.

The Directional Sampling simulation operates in the standard space.

Usage :

DirectionalSampling(event)

DirectionalSampling(event, rootStrategy, samplingStrategy)

Arguments :

event : an Event, the event we want to evaluate the probability

rootStrategy : a RootStrategy, the strategy adopted to evaluate the intersections of each direction with the limit state function and take into account the contribution of the direction to the event probability. By default, *rootStrategy* = *RootStrategy(SafeAndSlow)*

samplingStrategy : a SamplingStrategy, the strategy adopted to sample directions. By default,

samplingStrategy = *SamplingStrategy(RandomDirection)*

Some methods :

getRootStrategy

Usage : *getRootStrategy()*

Arguments : none

Value : a RootStrategy, the root strategy adopted

getSamplingStrategy

Usage : *getSamplingStrategy()*

Arguments : none

Value : a SamplingStrategy, the direction sampling strategy adopted

Each *getMethod* is associated to a *setMethod*.

14.7.2 RootStrategy

Usage :

```
RootStrategy()  
RootStrategy(rootStrategyImplementation)
```

Arguments :

rootStrategyImplementation : a RootStrategyImplementation, the implementation of the root strategy adopted, which is *RiskyAndFast*, *MediumSafe* or *SafeAndSlow*

When not given, by default, *rootStrategyImplementation* = *SafeAndSlow*.

Some methods :

getMaximumDistance

Usage : *getMaximumDistance()*

Arguments : none

Value : a positive real value, the distance from the center of the standard space until which we research an intersection with the limit state function along each direction. By default, the maximum distance is equal to 8

getOriginValue

Usage : *getOriginValue()*

Arguments : none

Value : a real value, the value of the limit state function at the center of the standard space

getStepSize

Usage : *getStepSize()*

Arguments : none

Value : a real value, the length of each segment inside which the root research is performed.

Each *getMethod* is associated to a *setMethod*.

14.7.3 RiskyAndFast

The RiskyAndFast strategy is the following : for each direction, we check whether there is a sign changement 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 changement, we search one root in the segment [origin, maximum distant point] with the selected non linear solver.

As soon as founded, the segment [root, infinity point] is considered within the failure space.

It inherits from the methods of the RootStrategy class.

Usage :

RiskyAndFast()

RiskyAndFast(solver)

RiskyAndFast(solver, maximumDistance, stepSize)

Arguments :

solver : a Solver, the non linear solver used to research the intersection of the limit state function with the direction, on each segment of length *stepSize*, between the center of the space and *maximumDistance* (root research)

maximumDistance : a real strictly positive value, the maximum distance within which the root research is performed along each direction

stepSize : a real value, the length of each segment along a direction inside which the root research is performed

By default, *solver* = *Brent*, *maximumDistance* = 8, *stepSize* = 1

Some methods :

getSolver

Usage : *getSolver()*

Arguments : none

Value : a Solver, the non linear solver which will research the root in a segment

solve

Usage : *solve(function, value)*

Arguments :

function : a NumericalMathFunction, from \mathbb{R} into \mathbb{R}

value : a real value

Value : a ScalarCollection of dimension 1, the real value x such as $function(x) = value$ researched within [*origin*, *maximumDistance*]

Each *getMethod* is associated to a *setMethod*.

14.7.4 MediumSafe

The MediumSafe strategy is the following : 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 changement on each segment so formed.

At the first sign changement, we research one root in the concerned segment with the selected 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 origin.

It inherits from the methods of the RootStrategy class.

Usage :

MediumSafe()

MediumSafe(solver)

MediumSafe(solver, maximumDistance, stepSize)

Arguments :

solver : a Solver, the non linear solver used to research the intersection of the limit state function with the direction, on each segment of length *stepSize*, between the center of the space and *maximumDistance* (root research),

maximumDistance : a real strictly positive value, the maximum distance within which the root research is performed along each direction

stepSize : a real value. CARE : this value is not taken into account in the root research : *stepSize* = *maximumDistance* automatically on the algorithm according to this root strategy

By default, *solver* = *Brent*, *maximumDistance* = 8

Some methods :

getSolver

Usage : *getSolver()*

Arguments : none

Value : a Solver, the non linear solver which will research the root in a segment

solve

Usage : *solve(function, value)*

Arguments :

function : a NumericalMathFunction, from \mathbb{R} into \mathbb{R}

value : a real value

Value : a ScalarCollection of dimension 1 (one root) : the real value x such as $function(x) = value$ researched the first segment of length *stepsize*, within [*origin*, *maximumDistance*] where a sign changement of *function* has been detected

Each *getMethod* is associated to a *setMethod*.

14.7.5 SafeAndSlow

The SafeAndSlow strategy is the following : 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 chagement on each segment so formed.

We go until the maximum distant point. Then, for all the segments where we detected a the presence of a root, we research the root with the selected 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.

It inherits from the methods of the RootStrategy class.

Usage :

SafeAndSlow()

SafeAndSlow(*solver*)

SafeAndSlow(*solver*, *maximumDistance*, *stepSize*)

Arguments :

solver : a Solver, the non linear solver used to research the intersection of the limit state function with the direction, on each segment of length *stepSize*, between the center of the space and *maximumDistance* (root research),

maximumDistance : a real strictly positive value, the maximum distance within which the root research is performed along each direction

stepSize : a real value, the length of each segment along a direction inside which the root research is performed.

By default, *solver* = *Brent*, *maximumDistance* = 8, *stepSize* = 1

Some methods :

getSolver

Usage : *getSolver*()

Arguments : none

Value : a Solver, the non linear solver which will research the root in a segment

solve

Usage : *solve*(*function*, *value*)

Arguments :

function : a NumericalMathFunction, from \mathbb{R} into \mathbb{R}

value : a real value

Value : a ScalarCollection, all the real values x such as $function(x) = value$ researched in each segment of length *stepsize*, within [*origin*, *maximumDistance*]

Each *getMethod* is associated to a *setMethod*.

14.7.6 SamplingStrategy

Usage :

SamplingStrategy()
SamplingStrategy(samplingStrategyImplementation)
SamplingStrategy(dimension)

Arguments :

samplingStrategyImplementation : a *SamplingStrategyImplementation*, the implementation of the sampling strategy adopted, which is *RandomDirection*, or *OrthogonalDirection*

dimension : an integer, the dimension of the standard space

By default, *samplingStrategyImplementation* = *RandomDirection* and *dimension* = 0 but the dimension automatically updated by the calling class

Some methods :

getDimension

Usage : *getDimension()*

Arguments : none

Value : an integer, the dimension of the standard space

Each *getMethod* is associated to a *setMethod*.

14.7.7 RandomDirection

The RandomDirection strategy is the following : we generate some points on the sphere unity in the standard space according to the uniform distribution and we consider both opposite directions so built.

It inherits from the methods of the SamplingStrategy class.

Usage :

RandomDirection()
RandomDirection(dimension)

Arguments :

dimension : an integer, the dimension of the standard space
By default, *dimension* = 0 but automatically updated by the calling class

Some methods :

generate

Usage : *generate()*

Arguments : none

Value : a NumericalSample of size 2, two opposite random directions generated

getUniformUnitVectorRealization

Usage : *getUniformUnitVectorRealization(dimension)*

Arguments : *dimension* : an integer, the dimension of the sphere unity (which is the dimension of the standard space)

Value : a NumericalPoint, a realization of a vector on the sphere unity, according to the uniform distribution

Each *getMethod* is associated to a *setMethod*.

14.7.8 OrthogonalDirection

The OrthogonalDirection strategy is the following : this strategy is parameterized by $k \in \mathbb{N}$. We generate one direct orthonormalized base (e_1, \dots, e_n) within the set of orthonormalized bases. We consider all the renormalized 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 .

If $k = 1$, we consider all the axes of the standard space.

It inherits from the methods of the SamplingStrategy class.

Usage :

```
OrthogonalDirection()
OrthogonalDirection(dimension, size)
```

Arguments :

dimension : an integer, dimension of the standard space

size : an integer, the number of elements in the linear combinations described here above

By default, *size* = 1 and *dimension* = 0 but automatically updated by the calling class.

Some methods :

generate

Usage : *generate*()

Arguments : none

Value : a NumericalSample, a realization of a random direction according to the algorithm described here above.

getUniformUnitVectorRealization

Usage : *getUniformUnitVectorRealization*(*dimension*)

Arguments : *dimension* : an integer, the dimension of the sphere unity (dimension of the standard space)

Value : a NumericalPoint, a realization of a vector on the sphere unity, according to the uniform distribution

Each *getMethod* is associated to a *setMethod*.

14.7.9 Solver

This class enables to solve 1D non linear equations :

$$f(x) = \text{value}, \text{ for } x \in (\text{infPoint}, \text{supPoint})$$

if f is a continuous function from \mathbb{R} in \mathbb{R} , $\text{infPoint}, \text{supPoint} \in \mathbb{R}$ and if f is such that $f(\text{infPoint})f(\text{supPoint}) < 0$, then f has at least a zero in the interval $(\text{infPoint}, \text{supPoint})$.. In particular, it is used in the root research of a directional sampling simulation.

Usage :

Solver(solverImplementation)

Solver(absoluteError, relativeError, maximumFunctionEvaluation)

Arguments :

solverImplementation : a SolverImplementation, the implementation of particular solver which is *Bisection*, *Brent* or *Secant*,

absoluteError : a real positive value, absolute error : distance between two successive iterates at the end point

relativeError : a real positive value, relative distance between the two last successive iterates (with regards the last iterate)

maximumFunctionEvaluation : an integer, the maximum number of evaluations of the function

Some methods :

getAbsoluteError

Usage : *getAbsoluteError()*

Arguments : none

Value : a real positive value, the absolute error : distance between two successive iterates at the end point

getMaximumFunctionEvaluation

Usage : *getMaximumFunctionEvaluation()*

Arguments : none

Value : an integer, the maximum number of evaluations of the function

getRelativeError

Usage : *getRelativeError()*

Arguments : none

Value : a real positive value, the relative distance between the two last successive iterates (with regards the last iterate)

14.7.10 Bisection

The Bisection solver is a bisection algorithm.

Usage :

Bisection()

Bisection(absoluteError, relativeError, maximumFunctionEvaluation)

Arguments :

absoluteError : a real positive value, absolute error : distance between two successive iterates at the end point

relativeError : a real positive value, relative distance between the two last successive iterates (with regards the last iterate)

maximumFunctionEvaluation : an integer, the maximum number of evaluations of the function

By default, *absoluteError* = 10^{-5} , *relativeError* = 10^{-5} , *maximumFunctionEvaluation* = 100

Some methods :

solve

Usage :

solve(function, value, infPoint, supPoint)

solve(function, value, infPoint, supPoint, infValue, supValue)

Arguments :

function : a NumericalMathFunction, the function of the equation $function(x) = value$ we want to solve on $(infPoint, supPoint)$

value : a real value, the value of the equation $function(x) = value$ we want to solve on $(infPoint, supPoint)$

infPoint : a real value, the lower bound of the interval where we want to solve the equation

supPoint : a real value, the upper bound of the interval where we want to solve the equation

infValue: a real value, the value of *function* on the point *infPoint* : $function(infPoint)$, must be of opposite sign of *supValue*

supValue: a real value, a real value, the value of *function* on the point *supPoint* : $function(supPoint)$ must be of opposite sign of *infValue*

Value : a real value, the result of the root research, in $(infPoint, supPoint)$

Details : If the function f is continuous, the Bisection solver will converge towards a root of the equation $function(x) = value$ on $(infPoint, supPoint)$. If not, it will converge towards either a root or a discontinuity point of f on $(infPoint, supPoint)$. Bisection guarantees a convergence.

Bisection may fail.

14.7.11 Brent

The Brent solver is a mix of Bisection, Secant and inverse quadratic interpolation.

Usage :

Brent()

Brent(absoluteError, relativeError, maximumFunctionEvaluation)

Arguments :

absoluteError : a real positive value, the absolute error : distance between two successive iterates at the end point

relativeError : a real positive value, the relative distance between the two last successive iterates (with regards the last iterate)

maximumFunctionEvaluation : an integer, the maximum number of evaluations of the function

By default, *absoluteError* = 10^{-5} , *relativeError* = 10^{-5} , *maximumFunctionEvaluation* = 100

Some methods :

solve

Usage :

solve(function, value, infPoint, supPoint)

solve(function, value, infPoint, supPoint, infValue, supValue)

Arguments :

function : a NumericalMathFunction, the function of the equation $function(x) = value$ we want to solve on $(infPoint, supPoint)$

value : a real value, the value of the equation $f(x) = value$ we want to solve on $(infPoint, supPoint)$

infPoint : a real value, the lower bound of the interval where we want to solve the equation

supPoint : a real value, the upper bound of the interval where we want to solve the equation

infValue: a real value, the value of *function* on the point *infPoint* : $function(infPoint)$, must be of opposite sign of *supValue*

supValue: a real value, a real value, the value of *function* on the point *supPoint* : $function(supPoint)$ must be of opposite sign of *infValue*

Value : the result of the root research, in $(infPoint, supPoint)$

Details : If the function *f* is continuous, the Brent solver will converge towards a root of the equation $function(x) = value$ on $(infPoint, supPoint)$. If not, it will converge towards either a root or a discontinuity point of *f* on $(infPoint, supPoint)$. Brent guarantees a convergence.

14.7.12 Secant

The Secant solver is based on the evaluation of a segment between the two last iterated points.

Usage :

Secant()
Secant(absoluteError, relativeError, maximumFunctionEvaluation)

Arguments :

absoluteError : a real positive value, absolute error : distance between two successive iterates at the end point

relativeError : a real positive value, relative distance between the two last successive iterates (with regards the last iterate)

maximumFunctionEvaluation : an integer, the maximum number of evaluations of the function
 By default, *absoluteError* = 10^{-5} , *relativeError* = 10^{-5} , *maximumFunctionEvaluation* = 100

Some methods :

solve

Usage :

solve(function, value, infPoint, supPoint)
solve(function, value, infPoint, supPoint, infValue, supValue)

Arguments :

function : a NumericalMathFunction, the function of the equation $function(x) = value$ we want to solve on $(infPoint, supPoint)$

value : a real value, the value of the equation $function(x) = value$ we want to solve on $(infPoint, supPoint)$

infPoint : a real value, the lower bound of the interval where we want to solve the equation

supPoint : a real value, the upper bound of the interval where we want to solve the equation

infValue: a real value, the value of *function* on the point *infPoint* : $function(infPoint)$, must be of opposite sign of *supValue*

supValue: a real value, a real value, the value of *function* on the point *supPoint* : $function(supPoint)$, must be of opposite sign of *infValue*

Value : the result of the root research, in $(infPoint, supPoint)$

Details : Secant might fail and not converge.

14.8 ImportanceSampling

This class inherits from Simulation.

Usage : *ImportanceSampling(event, importanceDistribution)*

Arguments :

event : a Event, the event we want to evaluate the probability

importanceDistribution : a Distribution, the importance distribution of the Importance Sampling simulation method.

Some methods :

getImportanceDistribution

Usage : *getImportanceDistribution()*

Arguments : none

Value : a Distribution, the importance distribution of the Importance Sampling simulation method

14.9 PostAnalyticalSimulation

This class inherits from Simulation.

The principle is to perform a simulation study to evaluate the threshold exceedance probability according to an importance density centered around the design point, in the standard space. The importance distribution is the standard distribution of the standard space

Usage : A *PostAnalyticalSimulation* object can be created only through its derivative classes : *PostAnalyticalImportanceSampling* or *PostAnalyticalControlledImportanceSampling*.

Some methods :

getAnalyticalResult() :

Usage : *getAnalyticalResult()*

Arguments : none

Value : a *AnalyticalResult*, which contains the results of the analytical study which has been performed just before the simulation study centered around the importance factor.

14.10 PostAnalyticalImportanceSampling

This class inherits from PostAnalyticalSimulation.

Usage : *PostAnalyticalImportanceSampling(analyticalResult)*

Arguments : *analyticalResult* : an AnalyticalResult which contains the whole information on the analytical study performed before the simulation study : in particular, the standard distribution of the standard space and the standard space design point.

Some methods :

run :

Usage : *run()*

Arguments : none

Value : it launches the simulation and creates a SimulationResult, structure containing all the results obtained after simulation

14.11 PostAnalyticalControlledImportanceSampling

This class inherits from PostAnalyticalSimulation.

Usage : *PostAnalyticalControlledImportanceSampling(analyticalResult)*

Arguments : *analyticalResult* : an AnalyticalResult which contains the whole information on the analytical study performed before the simulation study : in particular, the standard distribution of the standard space and the standard space design point.

Some methods :

run :

Usage : *run()*

Arguments : none

Value : it launches the controlled simulation and creates a SimulationResult, structure containing all the results obtained after simulation

14.12 QuasiMonteCarlo

This class inherits from Simulation.

Usage :

QuasiMonteCarlo(event)

QuasiMonteCarlo(event, sequence)

Arguments :

event : an Event, the event we want to evaluate the probability

sequence : a LowDiscrepancySequence, which is the low-discrepancy sequence used to generate the samples (by default the Sobol sequence is used (SobolSequence))

Details : Be carefull, to be valid, the QuasiMonteCarlo sampling method requires that the multi-variate distribution have an independent copula. All events can be reduced to this case using the Roseblatt transformation.

14.13 RandomizedQuasiMonteCarlo

This class inherits from Simulation.

Usage :

RandomizedQuasiMonteCarlo(event)
RandomizedQuasiMonteCarlo(event, sequence)

Arguments :

event : an Event, the event we want to evaluate the probability
sequence : a LowDiscrepancySequence, which is the low-discrepancy sequence used to generate the samples (by default the Sobol sequence is used (SobolSequence))

Details : Be carefull, to be valid, the RandomizedQuasiMonteCarlo sampling method requires that the multivariate distribution have an independent copula. All events can be reduced to this case using the Roseblatt transformation.

14.14 SimulationResult

Usage : structure created by the method `run()` of a `Simulation`, and obtained thanks to the method `getResult()`

Some methods :

computeImportanceFactors

Usage : `computeImportanceFactors`

Arguments : none

Value : a `NumericalPointWithDescription`, the importance factors (49) evaluated from the coordinates of the mean point (48) of event domain, mapped into the standard space as follows :

$$\underline{X}_{event}^* = \frac{1}{n} \sum_{i=1}^n \underline{X}_i 1_{event}(\underline{X}_i) \quad (48)$$

$$\alpha_i = \frac{(U_i^*)^2}{\|\underline{U}^*\|^2} \quad (49)$$

where

$$\underline{U}^* = T(\underline{X}_{event}^*) \quad (50)$$

Be carefull : this notion is only valuable for Monte Carlo or LHS sampling as the mean is evaluated from the (48) relation (only uniform weights over the realizations \underline{X}_i).

computeMeanPointInEventDomain

Usage : `computeMeanPointInEventDomain()`

Arguments : none

Value : a `NumericalPoint` which is the mean point in the physical space of all the simulations generated by the `Simulation` algorithm that failed into the event domain. Be carefull : this notion is only valuable for Monte Carlo or LHS sampling as the mean is evaluated from the (48) relation (only uniform weights over the realizations \underline{X}_i).

getBlockSize

Usage : `getBlockSize()`

Arguments : none

Value : an integer, the number of terms in the probability simulation estimator grouped together

getCoefficientOfVariation

Usage : `getCoefficientOfVariation()`

Arguments : none

Value : a real value, the coefficient of variation of the simulated sample

getConfidenceLength

Usage : `getConfidenceLength()`

Arguments : none

Value : a positive real value, the length of any confidence interval equal to the double of the variance of the Monte Carlo estimator

getOuterSampling

Usage : *getOuterSampling()*

Arguments : none

Value : an integer, the number of groups of terms in the probability simulation estimator

Details : for Monte Carlo, LHS and Importance Sampling methods, the number of evaluations of the limit state function defining the event is : $\text{OuterSampling} * \text{BlockSize}$

getProbabilityEstimate

Usage : *getProbabilityEstimate()*

Arguments : none

Value : a positive real value, the Monte Carlo estimate of the event probability

getStandardDeviation

Usage : *getStandardDeviation()*

Arguments : none

Value : a positive real value , the standard deviation of the estimator at the end of the simulation

getVarianceEstimate

Usage : *getVarianceEstimate()*

Arguments : none

Value : a positive real value, the variance of the Monte Carlo estimator, equal to the $M_n(1 - M_n)/n$ if M_n is the Monte Carlo probability estimator and n the size of the simulated sample

15 Taylor decomposition of the limit state function

15.1 QuadraticCumul

Usage : *QuadraticCumul(randVect)*

Arguments : *randVect* : a RandomVector, constraint : this RandomVector must be of type Composite, which means it must have been defined with the second usage of declaration of a RandomVector (from a NumericalMathFunction and an antecedent Distribution)

Value : a QuadraticCumul

Some methods :

drawImportanceFactors

Usage : *drawImportanceFactors()*

Arguments : none

Value : a Graph, the structure containing the pie corresponding to the importance factors of the probabilistic variables

getCovariance

Usage : *getCovariance()*

Arguments : none

Value : a CovarianceMatrix, approximation of first order of the covariance matrix of the random vector

getImportanceFactors

Usage : *getImportanceFactors()*

Arguments : none

Value : a NumericalPoint, the importance factors of the inputs : only when *randVect* is of dimension 1

getMeanFirstOrder

Usage : *getMeanFirstOrder()*

Arguments : none

Value : a NumericalPoint, approximation at the first order of the mean of the random vector

getMeanSecondOrder

Usage : *getMeanSecondOrder()*

Arguments : none

Value : a NumericalPoint, approximation at the second order of the mean of the random vector (it requires that the hessian of the NumericalMathFunction has been defined)

getValueAtMean

Usage : *getValueAtMean()*

Arguments : none

Value : a NumericalPoint, the value of the NumericalMathFunction which defines the random vector at the mean point of the input random vector

getGradientAtMean

Usage : *getGradientAtMean()*

Arguments : none

Value : a Matrix, the gradient of the NumericalMathFunction which defines the random vector at the mean point of the input random vector

getHessianAtMean

Usage : *getHessianAtMean()*

Arguments : none

Value : a SymmetricTensor, the hessian of the NumericalMathFunction which defines the random vector at the mean point of the input random vector