

marmoteCore

Generated by Doxygen 1.8.13

Contents

1	Hierarchical Index	1
1.1	Class Hierarchy	1
2	Class Index	3
2.1	Class List	3
3	Class Documentation	5
3.1	bernoulliDistribution Class Reference	5
3.1.1	Detailed Description	7
3.1.2	Constructor & Destructor Documentation	7
3.1.2.1	bernoulliDistribution()	7
3.1.3	Member Function Documentation	7
3.1.3.1	cdf()	7
3.1.3.2	copy()	8
3.1.3.3	dLaplace()	8
3.1.3.4	getParameter()	9
3.1.3.5	hasMoment()	9
3.1.3.6	laplace()	10
3.1.3.7	mean()	11
3.1.3.8	moment()	11
3.1.3.9	proba()	12
3.1.3.10	rate()	12
3.1.3.11	rescale()	12
3.1.3.12	sample()	13

3.1.3.13	toString()	13
3.1.3.14	write()	13
3.2	binarySequence Class Reference	14
3.2.1	Detailed Description	15
3.2.2	Constructor & Destructor Documentation	15
3.2.2.1	binarySequence()	15
3.2.3	Member Function Documentation	15
3.2.3.1	decodeState()	16
3.2.3.2	firstState()	16
3.2.3.3	index()	16
3.2.3.4	isZero()	17
3.2.3.5	nextState()	17
3.2.3.6	printState()	17
3.3	binarySimplex Class Reference	18
3.3.1	Detailed Description	19
3.3.2	Constructor & Destructor Documentation	19
3.3.2.1	binarySimplex()	19
3.3.3	Member Function Documentation	20
3.3.3.1	decodeState()	20
3.3.3.2	firstState()	20
3.3.3.3	index()	20
3.3.3.4	isZero()	21
3.3.3.5	nextState()	21
3.3.3.6	printState()	22
3.4	diracDistribution Class Reference	22
3.4.1	Detailed Description	24
3.4.2	Constructor & Destructor Documentation	24
3.4.2.1	diracDistribution()	24
3.4.3	Member Function Documentation	24
3.4.3.1	cdf()	25

3.4.3.2	copy()	25
3.4.3.3	dLaplace()	26
3.4.3.4	getProba()	27
3.4.3.5	hasMoment()	27
3.4.3.6	iidSample()	28
3.4.3.7	laplace()	28
3.4.3.8	mean()	29
3.4.3.9	moment()	29
3.4.3.10	rate()	30
3.4.3.11	rescale()	30
3.4.3.12	sample()	31
3.4.3.13	toString()	31
3.4.3.14	value()	32
3.4.3.15	write()	32
3.5	discreteDistribution Class Reference	32
3.5.1	Detailed Description	35
3.5.2	Constructor & Destructor Documentation	35
3.5.2.1	discreteDistribution() [1/3]	35
3.5.2.2	discreteDistribution() [2/3]	35
3.5.2.3	discreteDistribution() [3/3]	36
3.5.2.4	~discreteDistribution()	36
3.5.3	Member Function Documentation	36
3.5.3.1	cdf()	37
3.5.3.2	copy()	37
3.5.3.3	distanceL1()	38
3.5.3.4	distanceL2()	38
3.5.3.5	distanceLinfinity()	39
3.5.3.6	dLaplace()	40
3.5.3.7	getProba()	41
3.5.3.8	getProbaByIndex()	41

3.5.3.9	<code>getValue()</code>	41
3.5.3.10	<code>hasMoment()</code>	42
3.5.3.11	<code>laplace()</code>	42
3.5.3.12	<code>mean()</code>	43
3.5.3.13	<code>moment()</code>	43
3.5.3.14	<code>nbVals()</code>	44
3.5.3.15	<code>probas()</code>	44
3.5.3.16	<code>rate()</code>	45
3.5.3.17	<code>rescale()</code>	45
3.5.3.18	<code>sample()</code>	46
3.5.3.19	<code>setProba()</code>	46
3.5.3.20	<code>toString()</code>	47
3.5.3.21	<code>values()</code>	47
3.5.3.22	<code>write()</code>	47
3.5.4	Member Data Documentation	48
3.5.4.1	<code>_nbVals</code>	48
3.5.4.2	<code>_probas</code>	48
3.5.4.3	<code>_values</code>	48
3.6	Distribution Class Reference	49
3.6.1	Detailed Description	50
3.6.2	Member Function Documentation	50
3.6.2.1	<code>ccdf()</code>	50
3.6.2.2	<code>cdf()</code>	51
3.6.2.3	<code>copy()</code>	51
3.6.2.4	<code>distanceL1()</code>	52
3.6.2.5	<code>dLaplace()</code>	52
3.6.2.6	<code>exponential()</code>	52
3.6.2.7	<code>hasMoment()</code>	53
3.6.2.8	<code>hasProperty()</code>	53
3.6.2.9	<code>iidSample()</code>	54

3.6.2.10	laplace()	54
3.6.2.11	mean()	54
3.6.2.12	moment()	55
3.6.2.13	name()	55
3.6.2.14	rate()	55
3.6.2.15	rescale()	56
3.6.2.16	sample()	56
3.6.2.17	toString()	56
3.6.2.18	u_0_1()	57
3.6.2.19	variance()	57
3.6.2.20	write()	57
3.6.3	Member Data Documentation	57
3.6.3.1	_mean	58
3.6.3.2	_name	58
3.6.3.3	VALUE_TOLERANCE	58
3.7	eventMixture Class Reference	58
3.7.1	Detailed Description	60
3.7.2	Constructor & Destructor Documentation	60
3.7.2.1	eventMixture() [1/2]	60
3.7.2.2	eventMixture() [2/2]	61
3.7.3	Member Function Documentation	61
3.7.3.1	copy()	61
3.7.3.2	embed()	61
3.7.3.3	evaluateMeasure() [1/2]	61
3.7.3.4	evaluateMeasure() [2/2]	62
3.7.3.5	evaluateValue()	62
3.7.3.6	evaluateValueState()	63
3.7.3.7	eventProba()	63
3.7.3.8	getCol()	63
3.7.3.9	getEntry()	64

3.7.3.10	getEntryByCol()	64
3.7.3.11	getNbElts()	65
3.7.3.12	getTransDistrib()	65
3.7.3.13	nbEvents()	66
3.7.3.14	rowSum()	66
3.7.3.15	setEntry()	66
3.7.3.16	uniformize()	67
3.7.3.17	write()	67
3.8	exponentialDistribution Class Reference	67
3.8.1	Detailed Description	69
3.8.2	Constructor & Destructor Documentation	69
3.8.2.1	exponentialDistribution()	69
3.8.3	Member Function Documentation	69
3.8.3.1	cdf()	69
3.8.3.2	copy()	70
3.8.3.3	dLaplace()	70
3.8.3.4	hasMoment()	71
3.8.3.5	laplace()	71
3.8.3.6	mean()	71
3.8.3.7	moment()	72
3.8.3.8	rate()	72
3.8.3.9	rescale()	72
3.8.3.10	sample()	73
3.8.3.11	toString()	73
3.8.3.12	write()	73
3.9	geometricDistribution Class Reference	74
3.9.1	Detailed Description	75
3.9.2	Constructor & Destructor Documentation	75
3.9.2.1	geometricDistribution()	76
3.9.3	Member Function Documentation	76

3.9.3.1	cdf()	76
3.9.3.2	copy()	77
3.9.3.3	dLaplace()	77
3.9.3.4	getProba()	78
3.9.3.5	getRatio()	79
3.9.3.6	hasMoment()	79
3.9.3.7	laplace()	80
3.9.3.8	mean()	80
3.9.3.9	moment()	81
3.9.3.10	p()	81
3.9.3.11	rate()	82
3.9.3.12	rescale()	82
3.9.3.13	sample()	82
3.9.3.14	toString()	83
3.9.3.15	write()	83
3.10	LAW_DESC Struct Reference	83
3.10.1	Detailed Description	84
3.10.2	Member Data Documentation	84
3.10.2.1	Law_Parameters	84
3.10.2.2	Name	84
3.10.2.3	Parameters	84
3.11	LAW_LIST Struct Reference	85
3.11.1	Detailed Description	85
3.11.2	Member Data Documentation	85
3.11.2.1	Next	85
3.11.2.2	Val	86
3.12	marmoteBox Class Reference	86
3.12.1	Detailed Description	87
3.12.2	Constructor & Destructor Documentation	87
3.12.2.1	marmoteBox() [1/2]	88

3.12.2.2	marmoteBox() [2/2]	88
3.12.3	Member Function Documentation	88
3.12.3.1	cardinalbyDim()	88
3.12.3.2	cardinalOutDim()	89
3.12.3.3	decodeState()	89
3.12.3.4	firstState()	90
3.12.3.5	firstStatebyDim()	90
3.12.3.6	firstStateOutDim()	90
3.12.3.7	index()	91
3.12.3.8	isZero()	91
3.12.3.9	nextState()	91
3.12.3.10	nextStatebyDim()	93
3.12.3.11	nextStateOutDim()	93
3.12.3.12	printStats()	94
3.13	marmoteInterval Class Reference	94
3.13.1	Detailed Description	95
3.13.2	Constructor & Destructor Documentation	96
3.13.2.1	marmoteInterval()	96
3.13.2.2	~marmoteInterval()	96
3.13.3	Member Function Documentation	96
3.13.3.1	decodeState()	96
3.13.3.2	firstState()	97
3.13.3.3	index()	97
3.13.3.4	isFinite()	97
3.13.3.5	isZero()	98
3.13.3.6	nextState()	98
3.13.3.7	printStats()	98
3.13.4	Member Data Documentation	99
3.13.4.1	_max	99
3.13.4.2	_min	99

3.14 marmoteSet Class Reference	99
3.14.1 Detailed Description	101
3.14.2 Member Enumeration Documentation	101
3.14.2.1 opType	101
3.14.3 Constructor & Destructor Documentation	102
3.14.3.1 marmoteSet()	102
3.14.3.2 ~marmoteSet()	102
3.14.4 Member Function Documentation	102
3.14.4.1 cardinal()	102
3.14.4.2 cardinalbyDim()	102
3.14.4.3 cardinalOutDim()	103
3.14.4.4 decodeState()	103
3.14.4.5 firstState()	104
3.14.4.6 firstStatebyDim()	104
3.14.4.7 firstStateOutDim()	104
3.14.4.8 index()	105
3.14.4.9 isFinite()	105
3.14.4.10 isProduct()	105
3.14.4.11 isSimple()	106
3.14.4.12 isUnion()	106
3.14.4.13 isZero()	106
3.14.4.14 nextState()	106
3.14.4.15 nextStatebyDim()	107
3.14.4.16 nextStateOutDim()	107
3.14.4.17 printState() [1/2]	108
3.14.4.18 printState() [2/2]	108
3.14.4.19 totNbDims()	108
3.14.5 Member Data Documentation	108
3.14.5.1 _cardinal	109
3.14.5.2 _dimension	109

3.14.5.3	_dimOffset	109
3.14.5.4	_idxOffset	109
3.14.5.5	_nbDimensions	109
3.14.5.6	_nbZones	109
3.14.5.7	_stateBuffer	109
3.14.5.8	_totNbDims	109
3.14.5.9	_type	110
3.14.5.10	_zeroState	110
3.14.5.11	_zone	110
3.15	multiDimHomTransition Class Reference	110
3.15.1	Detailed Description	112
3.15.2	Constructor & Destructor Documentation	112
3.15.2.1	multiDimHomTransition()	113
3.15.2.2	~multiDimHomTransition()	114
3.15.3	Member Function Documentation	114
3.15.3.1	copy()	114
3.15.3.2	dimSize()	114
3.15.3.3	embed()	115
3.15.3.4	evaluateMeasure() [1/2]	115
3.15.3.5	evaluateMeasure() [2/2]	115
3.15.3.6	evaluateValue()	116
3.15.3.7	getCol()	116
3.15.3.8	getEntry()	116
3.15.3.9	getEntryByCol()	117
3.15.3.10	getJumpDistribution()	117
3.15.3.11	getNbElts()	118
3.15.3.12	getTransDistrib()	118
3.15.3.13	p()	118
3.15.3.14	q()	119
3.15.3.15	rowSum()	119

3.15.3.16 setEntry()	119
3.15.3.17 uniformize()	120
3.15.3.18 write()	120
3.16 poissonDistribution Class Reference	121
3.16.1 Detailed Description	122
3.16.2 Constructor & Destructor Documentation	122
3.16.2.1 poissonDistribution()	123
3.16.2.2 ~poissonDistribution()	123
3.16.3 Member Function Documentation	123
3.16.3.1 ccdf()	123
3.16.3.2 cdf()	124
3.16.3.3 copy()	124
3.16.3.4 dLaplace()	125
3.16.3.5 getProba()	126
3.16.3.6 hasMoment()	126
3.16.3.7 iidSample()	127
3.16.3.8 lambda()	127
3.16.3.9 laplace()	127
3.16.3.10 mean()	128
3.16.3.11 moment()	128
3.16.3.12 rate()	129
3.16.3.13 rescale()	129
3.16.3.14 toString()	130
3.16.3.15 variance()	131
3.16.3.16 write()	131
3.17 SCC Struct Reference	131
3.18 sparseMatrix Class Reference	132
3.18.1 Constructor & Destructor Documentation	134
3.18.1.1 sparseMatrix() [1/2]	134
3.18.1.2 sparseMatrix() [2/2]	134

3.18.1.3	<code>~sparseMatrix()</code>	135
3.18.2	Member Function Documentation	135
3.18.2.1	<code>addToEntry()</code>	135
3.18.2.2	<code>copy()</code>	135
3.18.2.3	<code>diagnose()</code>	136
3.18.2.4	<code>embed()</code>	136
3.18.2.5	<code>evaluateMeasure()</code> [1/2]	137
3.18.2.6	<code>evaluateMeasure()</code> [2/2]	137
3.18.2.7	<code>evaluateValue()</code>	137
3.18.2.8	<code>evaluateValueState()</code>	138
3.18.2.9	<code>getCol()</code>	138
3.18.2.10	<code>getEntry()</code>	139
3.18.2.11	<code>getEntryByCol()</code>	139
3.18.2.12	<code>getNbElts()</code>	140
3.18.2.13	<code>getStronglyConnectedComponents()</code>	140
3.18.2.14	<code>getTransDistrib()</code>	140
3.18.2.15	<code>rowSum()</code>	141
3.18.2.16	<code>setEntry()</code>	141
3.18.2.17	<code>uniformize()</code>	142
3.18.2.18	<code>write()</code>	142
3.18.3	Friends And Related Function Documentation	143
3.18.3.1	<code>markovChain</code>	143
3.19	templateDistribution Class Reference	143
3.19.1	Detailed Description	144
3.19.2	Constructor & Destructor Documentation	144
3.19.2.1	<code>templateDistribution()</code>	145
3.19.3	Member Function Documentation	145
3.19.3.1	<code>ccdf()</code>	145
3.19.3.2	<code>cdf()</code>	145
3.19.3.3	<code>copy()</code>	146

3.19.3.4	dLaplace()	147
3.19.3.5	hasMoment()	147
3.19.3.6	iidSample()	148
3.19.3.7	laplace()	148
3.19.3.8	mean()	149
3.19.3.9	moment()	149
3.19.3.10	rate()	149
3.19.3.11	rescale()	150
3.19.3.12	sample()	151
3.19.3.13	toString()	151
3.19.3.14	variance()	152
3.19.3.15	write()	152
3.20	transitionStructure Class Reference	152
3.20.1	Detailed Description	155
3.20.2	Member Function Documentation	155
3.20.2.1	consolidate()	155
3.20.2.2	copy()	155
3.20.2.3	destSize()	156
3.20.2.4	embed()	156
3.20.2.5	evaluateMeasure() [1/2]	156
3.20.2.6	evaluateMeasure() [2/2]	156
3.20.2.7	evaluateValue()	157
3.20.2.8	getCol()	157
3.20.2.9	getEntry()	158
3.20.2.10	getEntryByCol()	158
3.20.2.11	getNbElts()	158
3.20.2.12	getTransDistrib()	159
3.20.2.13	origSize()	159
3.20.2.14	readEntry()	159
3.20.2.15	rowSum()	160

3.20.2.16	setEntry()	160
3.20.2.17	setType()	161
3.20.2.18	setUniformizationRate()	161
3.20.2.19	size()	161
3.20.2.20	toString()	161
3.20.2.21	type()	162
3.20.2.22	uniformizationRate()	162
3.20.2.23	uniformize()	162
3.20.2.24	write()	162
3.20.3	Member Data Documentation	163
3.20.3.1	_type	163
3.21	uniformDiscreteDistribution Class Reference	163
3.21.1	Detailed Description	165
3.21.2	Constructor & Destructor Documentation	165
3.21.2.1	uniformDiscreteDistribution()	165
3.21.3	Member Function Documentation	166
3.21.3.1	ccdf()	166
3.21.3.2	cdf()	166
3.21.3.3	copy()	167
3.21.3.4	dLaplace()	167
3.21.3.5	getProba()	167
3.21.3.6	hasMoment()	168
3.21.3.7	iidSample()	168
3.21.3.8	laplace()	169
3.21.3.9	mean()	169
3.21.3.10	moment()	169
3.21.3.11	rate()	170
3.21.3.12	rescale()	170
3.21.3.13	sample()	171
3.21.3.14	toString()	171

3.21.3.15 <code>valInf()</code>	171
3.21.3.16 <code>valSup()</code>	171
3.21.3.17 <code>variance()</code>	172
3.21.3.18 <code>write()</code>	172
3.22 uniformDistribution Class Reference	172
3.22.1 Detailed Description	174
3.22.2 Constructor & Destructor Documentation	174
3.22.2.1 <code>uniformDistribution()</code>	174
3.22.3 Member Function Documentation	175
3.22.3.1 <code>ccdf()</code>	175
3.22.3.2 <code>cdf()</code>	175
3.22.3.3 <code>copy()</code>	176
3.22.3.4 <code>dLaplace()</code>	176
3.22.3.5 <code>hasMoment()</code>	177
3.22.3.6 <code>iidSample()</code>	178
3.22.3.7 <code>laplace()</code>	178
3.22.3.8 <code>mean()</code>	179
3.22.3.9 <code>moment()</code>	179
3.22.3.10 <code>rate()</code>	180
3.22.3.11 <code>rescale()</code>	180
3.22.3.12 <code>sample()</code>	180
3.22.3.13 <code>toString()</code>	181
3.22.3.14 <code>valInf()</code>	181
3.22.3.15 <code>valSup()</code>	181
3.22.3.16 <code>variance()</code>	181
3.22.3.17 <code>write()</code>	181
Index	183

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Distribution	49
discreteDistribution	32
bernoulliDistribution	5
diracDistribution	22
uniformDiscreteDistribution	163
exponentialDistribution	67
geometricDistribution	74
poissonDistribution	121
templateDistribution	143
uniformDistribution	172
LAW_DESC	83
LAW_LIST	85
marmoteSet	99
binarySequence	14
binarySimplex	18
marmoteBox	86
marmoteInterval	94
SCC	131
transitionStructure	152
eventMixture	58
multiDimHomTransition	110
sparseMatrix	132

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

bernoulliDistribution	
The Bernoulli distribution with two values	5
binarySequence	
The class representing binary sequences. These sets are cartesian products of the basic set $\{0,1\}$	14
binarySimplex	
The class representing binary sequences with a specified number of "1"	18
diracDistribution	
The Dirac distribution concentrated at some point	22
discreteDistribution	
The general discrete distribution with finite support	32
Distribution	
A class for representing probability distributions	49
eventMixture	
A class representing probabilistic mixtures of "events", that is, elementary transition structures	58
exponentialDistribution	
The class representing the (negative) exponential distribution	67
geometricDistribution	
The geometric distribution with starting value 0. The parameter "p" is called "ratio". The Geometric distribution is discrete but does not inherit from discreteDistribution because its range is infinite	74
LAW_DESC	
Description of a distribution	83
LAW_LIST	
Chained list structure for lists of distributions	85
marmoteBox	
The class representing "rectangular" sets. Boxes are cartesian products of intervals	86
marmoteInterval	
The class describing a finite integer interval	94
marmoteSet	
The mother class representing abstract sets	99
multiDimHomTransition	
Class for multidimensional, homogeneous random walk transition structures. These are characterized by	110

poissonDistribution	
The Poisson distribution. The parameter is called "lambda". The Poisson distribution is discrete but does not inherit from discreteDistribution because its range is infinite	121
SCC	131
sparseMatrix	132
templateDistribution	
The general template distribution to be instantiated	143
transitionStructure	
Abstract class for transition structures. These are structures which describe transitions to one state to another one, to which is attached a numeric label. Typical instances should be one-step transition matrices of discrete-time Markov chains, and infinitesimal generators of continuous-time Markov chains. It is also possible that the origin state space and the destination state space are different	152
uniformDiscreteDistribution	
The uniform discrete distribution	163
uniformDistribution	
The continuous uniform distribution over some interval	172

Chapter 3

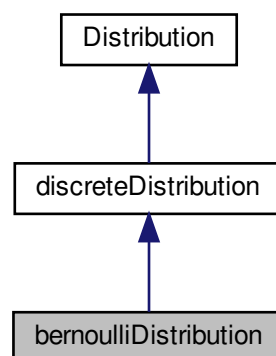
Class Documentation

3.1 bernoulliDistribution Class Reference

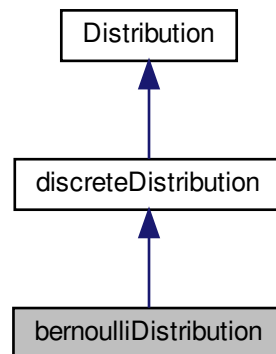
The Bernoulli distribution with two values.

```
#include <bernoulliDistribution.h>
```

Inheritance diagram for bernoulliDistribution:



Collaboration diagram for bernoulliDistribution:



Public Member Functions

- [bernoulliDistribution](#) (double)
Unique onstructor for a Bernoulli distribution from the probability that it is equal to 1.
- double [getParameter](#) ()
Accessor to the parameter of the distribution. Redundant with the standard accessor [proba\(\)](#) but more explicit.
- double [proba](#) ()
Accessor to the parameter of the distribution. Redundant with the accessor [getParameter\(\)](#) but conform to the coding standard.
- double [mean](#) ()
computing the mathematical expectation or mean
- double [rate](#) ()
computing the "rate", defined as the inverse of the mean
- double [moment](#) (int order)
Computing the moments of the distribution.
- double [laplace](#) (double s)
computing the Laplace transform of the distribution at real point
- double [dLaplace](#) (double s)
computing the derivative of the Laplace transform at real points
- double [cdf](#) (double x)
computing the cumulative distribution function at some real point x. This is the probability that the random variable is less or equal to x.
- bool [hasMoment](#) (int order)
test for the existence of moments of any order
- [bernoulliDistribution](#) * [rescale](#) (double factor)
Rescaling the distribution. Bernoulli distributions cannot be rescaled. Ac copy is returned and an error message is issued if the factor is not 1.0.
- [bernoulliDistribution](#) * [copy](#) ()
copying a distribution. Typically implemented as rescale(1.0).
- double [sample](#) ()
drawing a (pseudo)random value according to the distribution.

- `std::string toString ()`
an utility to convert the distribution into a string.
- `void write (FILE *out, int mode)`
an utility to write the distribution to some file, according to some format.

Additional Inherited Members

3.1.1 Detailed Description

The Bernoulli distribution with two values.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 bernoulliDistribution()

```
bernoulliDistribution::bernoulliDistribution (  
    double val )
```

Unique onstructor for a Bernoulli distribution from the probability that it is equal to 1.

Author

Alain Jean-Marie

Parameters

<i>val</i>	the value $P(X = 1)$
------------	----------------------

Returns

an object of type [bernoulliDistribution](#)

3.1.3 Member Function Documentation

3.1.3.1 cdf()

```
double bernoulliDistribution::cdf (  
    double x ) [virtual]
```

computing the cumulative distribution function at some real point x . This is the probability that the random variable is less or equal to x .

Computation of the cumulative density function at some real point x .

Parameters

x	the value at which to compute the cdf
-----	---------------------------------------

Returns

the value of the cdf

Author

Alain Jean-Marie

Parameters

x	value at which the CDF is computed
-----	------------------------------------

Returns

CDF(x)

Implements [Distribution](#).

3.1.3.2 copy()

```
bernoulliDistribution * bernoulliDistribution::copy ( ) [virtual]
```

copying a distribution. Typically implemented as `rescale(1.0)`.

See also

[rescale\(double\)](#)

Returns

a copy of the distribution

Implements [Distribution](#).

3.1.3.3 dLaplace()

```
double bernoulliDistribution::dLaplace (
    double s ) [virtual]
```

computing the derivative of the Laplace transform at real points

Computation of the derivative of the Laplace transform at some real point s .

Parameters

s	the value at which to compute
---	-------------------------------

Returns

the value of the derivative of the Laplace transform

Author

Alain Jean-Marie

Parameters

s	value at which the derivative is computed
---	-------------------------------------------

Returns

d LT(s)/ds

Implements [Distribution](#).

3.1.3.4 getParameter()

```
double bernoulliDistribution::getParameter ( ) [inline]
```

Accessor to the parameter of the distribution. Redundant with the standard accessor [proba\(\)](#) but more explicit.

Returns

the parameter

3.1.3.5 hasMoment()

```
bool bernoulliDistribution::hasMoment (
    int order ) [virtual]
```

test for the existence of moments of any order

Test of existence of a moment. These distributions always have one.

Parameters

order	the order of the moment to be tested
-------	--------------------------------------

Returns

true if the moment exists, false otherwise

Author

Alain Jean-Marie

Parameters

<i>order</i>	order of the moment
--------------	---------------------

Returns

true

Implements [Distribution](#).

3.1.3.6 laplace()

```
double bernoulliDistribution::laplace (  
    double s ) [virtual]
```

computing the Laplace transform of the distribution at real point

Computation of the Laplace transform at some real point s.

Parameters

<i>s</i>	the value at which to compute
----------	-------------------------------

Returns

the value of the Laplace transform

Author

Alain Jean-Marie

Parameters

<i>s</i>	value at which the derivative is computed
----------	-------------------------------------------

Returns

LT(s)

Implements [Distribution](#).

3.1.3.7 mean()

```
double bernoulliDistribution::mean ( ) [virtual]
```

computing the mathematical expectation or mean

Calculation of the mean. Returns the value since it is pre-computed.

Returns

the mean

Author

Alain Jean-Marie

Returns

the mathematical expectation of the distribution

Implements [Distribution](#).

3.1.3.8 moment()

```
double bernoulliDistribution::moment (
    int order ) [virtual]
```

Computing the moments of the distribution.

Calculation of the of order n.

Parameters

<i>order</i>	the order of the moment to be computed
--------------	----------------------------------------

Returns

the moment

Author

Alain Jean-Marie

Parameters

<i>order</i>	order of the moment
--------------	---------------------

Returns

the moment

Implements [Distribution](#).

3.1.3.9 proba()

```
double bernoulliDistribution::proba ( ) [inline]
```

Accessor to the parameter of the distribution. Redundant with the accessor [getParameter\(\)](#) but conform to the coding standard.

Returns

the parameter

3.1.3.10 rate()

```
double bernoulliDistribution::rate ( ) [virtual]
```

computing the "rate", defined as the inverse of the mean

Calculation of the rate, which is the inverse of the mean.

Returns

the rate

Author

Alain Jean-Marie

Returns

the rate

Implements [Distribution](#).

3.1.3.11 rescale()

```
bernoulliDistribution * bernoulliDistribution::rescale (
    double factor ) [virtual]
```

Rescaling the distribution. Bernoulli distributions cannot be rescaled. A copy is returned and an error message is issued if the factor is not 1.0.

See also

[copy\(\)](#)

Parameters

<i>factor</i>	the real factor to be used
---------------	----------------------------

Returns

a new distribution object

Implements [Distribution](#).

3.1.3.12 sample()

```
double bernoulliDistribution::sample ( ) [virtual]
```

drawing a (pseudo)random value according to the distribution.

Returns

a sample

Implements [Distribution](#).

3.1.3.13 toString()

```
std::string bernoulliDistribution::toString ( ) [virtual]
```

an utility to convert the distribution into a string.

Printing a representation of the law into a string.

Returns

the string representing the distribution.

Author

Alain Jean-Marie

Returns

a string

Implements [Distribution](#).

3.1.3.14 write()

```
void bernoulliDistribution::write (
    FILE * out,
    int mode ) [virtual]
```

an utility to write the distribution to some file, according to some format.

Parameters

<i>out</i>	the file descriptor of the file
<i>mode</i>	a code for the format to be used

Implements [Distribution](#).

The documentation for this class was generated from the following files:

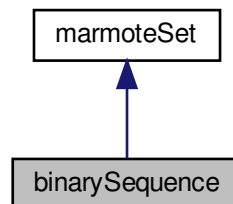
- Distribution/bernoulliDistribution.h
- Distribution/bernoulliDistribution.cpp

3.2 binarySequence Class Reference

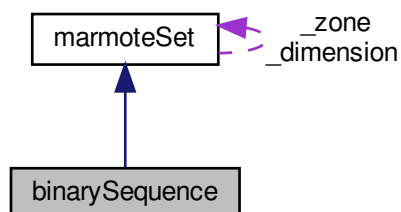
The class representing binary sequences. These sets are cartesian products of the basic set {0,1}.

```
#include <binarySequence.h>
```

Inheritance diagram for binarySequence:



Collaboration diagram for binarySequence:



Public Member Functions

- [binarySequence](#) (int n)
Constructor for a [binarySequence](#) from the parameters n.
- [~binarySequence](#) ()
Destructor.
- bool [isFinite](#) ()
Test whether the set is finite. Boxes are finite if and only if the size in each dimension is finite.
- bool [isZero](#) (int *buffer)
Tests that a state, given by its vector representation, is zero.
- void [firstState](#) (int *buffer)
Initializes some state buffer with the first state of the set.
- void [nextState](#) (int *buffer)
Procedure to compute the state following a given state in the set order. The new state is computed in place: the original state is erased. By convention, the state following the last state is the first state.
- void [decodeState](#) (int [index](#), int *buf)
utility to convert a state index into a state array
- int [index](#) (int *buf)
Utility to find the number of some state.
- void [printStats](#) (FILE *out, int *buffer)
Procedure for printing out a state.

Additional Inherited Members

3.2.1 Detailed Description

The class representing binary sequences. These sets are cartesian products of the basic set {0,1}.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 [binarySequence\(\)](#)

```
binarySequence::binarySequence (
    int n )
```

Constructor for a [binarySequence](#) from the parameters n.

Parameters

<i>n</i>	the number of elements
----------	------------------------

3.2.3 Member Function Documentation

3.2.3.1 decodeState()

```
void binarySequence::decodeState (
    int index,
    int * buf ) [virtual]
```

utility to convert a state index into a state array

Author

Alain Jean-Marie

Parameters

<i>index</i>	the state index
<i>buf</i>	the state buffer to be filled

Reimplemented from [marmoteSet](#).

3.2.3.2 firstState()

```
void binarySequence::firstState (
    int * buffer ) [virtual]
```

Initializes some state buffer with the first state of the set.

Parameters

<i>buffer</i>	the buffer to be set.
---------------	-----------------------

Reimplemented from [marmoteSet](#).

3.2.3.3 index()

```
int binarySequence::index (
    int * buf ) [virtual]
```

Utility to find the number of some state.

Parameters

<i>buf</i>	a state buffer
------------	----------------

Returns

the index of the state buffer

Reimplemented from [marmoteSet](#).

3.2.3.4 isZero()

```
bool binarySequence::isZero (
    int * buffer ) [virtual]
```

Tests that a state, given by its vector representation, is zero.

Parameters

<i>buffer</i>	the state to be tested
---------------	------------------------

Returns

true if the state is 0, false otherwise.

Reimplemented from [marmoteSet](#).

3.2.3.5 nextState()

```
void binarySequence::nextState (
    int * buffer ) [virtual]
```

Procedure to compute the state following a given state in the set order. The new state is computed in place: the original state is erased. By convention, the state following the last state is the first state.

Parameters

<i>buffer</i>	the reference state
---------------	---------------------

Reimplemented from [marmoteSet](#).

3.2.3.6 printState()

```
void binarySequence::printState (
    FILE * out,
    int * buffer ) [virtual]
```

Procedure for printing out a state.

Parameters

<i>out</i>	file descriptor of the stream to be used
<i>buffer</i>	the state buffer to be printed

Reimplemented from [marmoteSet](#).

The documentation for this class was generated from the following files:

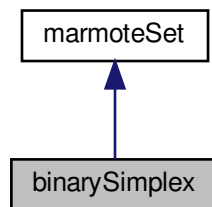
- Set/binarySequence.h
- Set/binarySequence.cpp

3.3 binarySimplex Class Reference

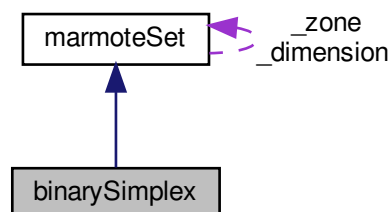
The class representing binary sequences with a specified number of "1".

```
#include <binarySimplex.h>
```

Inheritance diagram for binarySimplex:



Collaboration diagram for binarySimplex:



Public Member Functions

- [binarySimplex](#) (int n, int p)
Constructor for a [binarySimplex](#) from two parameters n and p.
- [~binarySimplex](#) ()
Destructor.
- bool [isFinite](#) ()
Test whether the set is finite. Boxes are finite if and only if the size in each dimension is finite.
- bool [isZero](#) (int *buffer)
Tests that a state, given by its vector representation, is zero.
- void [firstState](#) (int *buffer)
Initializes some state buffer with the first state of the set.
- void [nextState](#) (int *buffer)
Procedure to compute the state following a given state in the set order. The new state is computed in place: the original state is erased. By convention, the state following the last state is the first state.
- void [decodeState](#) (int [index](#), int *buf)
utility to convert a state index into a state array
- int [index](#) (int *buf)
Utility to find the number of some state.
- void [printState](#) (FILE *out, int *buffer)
Procedure for printing out a state.

Additional Inherited Members

3.3.1 Detailed Description

The class representing binary sequences with a specified number of "1".

3.3.2 Constructor & Destructor Documentation

3.3.2.1 [binarySimplex\(\)](#)

```
binarySimplex::binarySimplex (
    int n,
    int p )
```

Constructor for a [binarySimplex](#) from two parameters n and p.

Parameters

<i>n</i>	the number of positions nbPositions
<i>p</i>	the number of particles nbParticles

3.3.3 Member Function Documentation

3.3.3.1 decodeState()

```
void binarySimplex::decodeState (
    int index,
    int * buf ) [virtual]
```

utility to convert a state index into a state array

Author

Alain Jean-Marie

Parameters

<i>index</i>	the state index
<i>buf</i>	the state buffer to be filled

Reimplemented from [marmoteSet](#).

3.3.3.2 firstState()

```
void binarySimplex::firstState (
    int * buffer ) [virtual]
```

Initializes some state buffer with the first state of the set.

Parameters

<i>buffer</i>	the buffer to be set.
---------------	-----------------------

Reimplemented from [marmoteSet](#).

3.3.3.3 index()

```
int binarySimplex::index (
    int * buf ) [virtual]
```

Utility to find the number of some state.

Parameters

<i>buf</i>	a state buffer
------------	----------------

Returns

the index of the state buffer

Reimplemented from [marmoteSet](#).

3.3.3.4 isZero()

```
bool binarySimplex::isZero (
    int * buffer ) [virtual]
```

Tests that a state, given by its vector representation, is zero.

Parameters

<i>buffer</i>	the state to be tested
---------------	------------------------

Returns

true if the state is 0, false otherwise.

Reimplemented from [marmoteSet](#).

3.3.3.5 nextState()

```
void binarySimplex::nextState (
    int * buffer ) [virtual]
```

Procedure to compute the state following a given state in the set order. The new state is computed in place: the original state is erased. By convention, the state following the last state is the first state.

Parameters

<i>buffer</i>	the reference state
---------------	---------------------

Reimplemented from [marmoteSet](#).

3.3.3.6 printState()

```
void binarySimplex::printState (
    FILE * out,
    int * buffer ) [virtual]
```

Procedure for printing out a state.

Parameters

<i>out</i>	file descriptor of the stream to be used
<i>buffer</i>	the state buffer to be printed

Reimplemented from [marmoteSet](#).

The documentation for this class was generated from the following files:

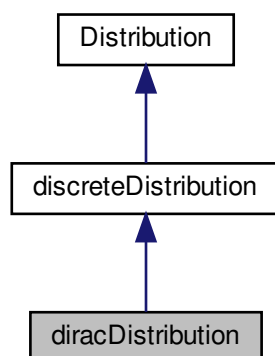
- Set/binarySimplex.h
- Set/binarySimplex.cpp

3.4 diracDistribution Class Reference

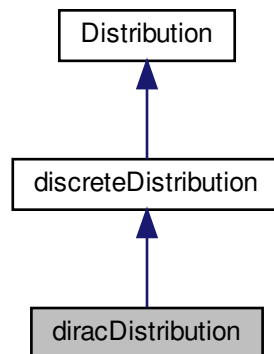
The Dirac distribution concentrated at some point.

```
#include <diracDistribution.h>
```

Inheritance diagram for diracDistribution:



Collaboration diagram for diracDistribution:



Public Member Functions

- `diracDistribution` (double val)
Unique constructor for the Dirac distribution from its value.
- double `value` (int)
Read accessor to the value of the Dirac distribution.
- double `getProba` (double value)
Calculation of the mean. Returns the value since it is pre-computed.
- double `mean` ()
computing the mathematical expectation or mean
- double `rate` ()
computing the "rate", defined as the inverse of the mean
- double `moment` (int order)
Computing the moments of the distribution.
- double `variance` ()
Variance of the Dirac distribution. Reimplemented to return always 0.
- double `laplace` (double s)
computing the Laplace transform of the distribution at real point
- double `dLaplace` (double s)
computing the derivative of the Laplace transform at real points
- double `cdf` (double x)
computing the cumulative distribution function at some real point x. This is the probability that the random variable is less or equal to x.
- bool `hasMoment` (int order)
test for the existence of moments of any order
- `diracDistribution` * `rescale` (double factor)
*rescaling a distribution by some real factor. Not all distributions allow this for any real factor. If the operation fails, or if the factor is 1.0, a copy of the distribution should be returned (**not** by using the `copy()` function).*
- `diracDistribution` * `copy` ()
copying a distribution. Typically implemented as `rescale(1.0)`.
- double `sample` ()

- *drawing a (pseudo)random value according to the distribution.*
- void `iidSample` (int n, double *s)
Sampling of i.i.d. values in a table. Reimplemented in order to avoid useless function calls.
- std::string `toString` ()
an utility to convert the distribution into a string.
- void `write` (FILE *out, int mode)
an utility to write the distribution to some file, according to some format.

Additional Inherited Members

3.4.1 Detailed Description

The Dirac distribution concentrated at some point.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 `diracDistribution()`

```
diracDistribution::diracDistribution (
    double val )
```

Unique constructor for the Dirac distribution from its value.

Parameters

<code>val</code>	the value at which the distribution is concentrated
------------------	-----------------------------------------------------

Constructor for a Dirac distribution. The mean is calculated at creation.

Author

Alain Jean-Marie

Parameters

<code>val</code>	the value of the deterministic law
------------------	------------------------------------

Returns

an object of type `diracDistribution`

3.4.3 Member Function Documentation

3.4.3.1 cdf()

```
double diracDistribution::cdf (
    double x ) [virtual]
```

computing the cumulative distribution function at some real point x. This is the probability that the random variable is less or equal to x.

Parameters

x	the value at which to compute the cdf
---	---------------------------------------

Returns

the value of the cdf

Computation of the cumulative density function at some real point x

Author

Alain Jean-Marie

Parameters

x	value at which the CDF is computed
---	------------------------------------

Returns

CDF(x)

Implements [Distribution](#).

3.4.3.2 copy()

```
diracDistribution * diracDistribution::copy ( ) [virtual]
```

copying a distribution. Typically implemented as `rescale(1.0)`.

See also

[rescale\(double\)](#)

Returns

a copy of the distribution

Implements [Distribution](#).

3.4.3.3 dLaplace()

```
double diracDistribution::dLaplace (  
    double s ) [virtual]
```

computing the derivative of the Laplace transform at real points

Parameters

s	the value at which to compute
---	-------------------------------

Returns

the value of the derivative of the Laplace transform

Computation of the derivative of the Laplace transform at some real point s

Author

Alain Jean-Marie

Parameters

s	value at which the derivative is computed
---	-------------------------------------------

Returns

d LT(s)/ds

Implements [Distribution](#).

3.4.3.4 getProba()

```
double diracDistribution::getProba (
    double value )
```

Calculation of the mean. Returns the value since it is pre-computed.

Author

Alain Jean-Marie

Returns

the mathematical expectation of the distribution

3.4.3.5 hasMoment()

```
bool diracDistribution::hasMoment (
    int order ) [virtual]
```

test for the existence of moments of any order

Parameters

<i>order</i>	the order of the moment to be tested
--------------	--------------------------------------

Returns

true if the moment exists, false otherwise

Test of existence of a moment. These distributions always have one.

Author

Alain Jean-Marie

Parameters

<i>order</i>	order of the moment
--------------	---------------------

Returns

true

Implements [Distribution](#).

3.4.3.6 iidSample()

```
void diracDistribution::iidSample (
    int n,
    double * s )
```

Sampling of i.i.d. values in a table. Reimplemented in order to avoid useless function calls.

Parameters

<i>n</i>	the number of values to sample
<i>s</i>	an array to be filled with the sample

3.4.3.7 laplace()

```
double diracDistribution::laplace (
    double s ) [virtual]
```

computing the Laplace transform of the distribution at real point

Parameters

s	the value at which to compute
---	-------------------------------

Returns

the value of the Laplace transform

Computation of the Laplace transform at some real point s

Author

Alain Jean-Marie

Parameters

s	value at which the derivative is computed
---	-------------------------------------------

Returns

LT(s)

Implements [Distribution](#).

3.4.3.8 mean()

```
double diracDistribution::mean ( ) [virtual]
```

computing the mathematical expectation or mean

Returns

the mean

Calculation of the mean. Returns the value since it is pre-computed

Author

Alain Jean-Marie

Returns

the mathematical expectation of the distribution

Implements [Distribution](#).

3.4.3.9 moment()

```
double diracDistribution::moment (
    int order ) [virtual]
```

Computing the moments of the distribution.

Parameters

<i>order</i>	the order of the moment to be computed
--------------	----------------------------------------

Returns

the moment

Implements [Distribution](#).

3.4.3.10 rate()

```
double diracDistribution::rate ( ) [virtual]
```

computing the "rate", defined as the inverse of the mean

Returns

the rate

Calculation of the rate, which is the inverse of the mean

Author

Alain Jean-Marie

Returns

the rate

Implements [Distribution](#).

3.4.3.11 rescale()

```
diracDistribution * diracDistribution::rescale (
    double factor ) [virtual]
```

rescaling a distribution by some real factor. Not all distributions allow this for any real factor. If the operation fails, or if the factor is 1.0, a copy of the distribution should be returned (**not** by using the [copy\(\)](#) function).

See also

[copy\(\)](#)

Parameters

<i>factor</i>	the real factor to be used
---------------	----------------------------

Returns

a new distribution object

Implements [Distribution](#).

3.4.3.12 sample()

```
double diracDistribution::sample ( ) [virtual]
```

drawing a (pseudo)random value according to the distribution.

Returns

a sample

Implements [Distribution](#).

3.4.3.13 toString()

```
std::string diracDistribution::toString ( ) [virtual]
```

an utility to convert the distribution into a string.

Returns

the string representing the distribution.

Printing a representation of the law into a string

Author

Alain Jean-Marie

Returns

a string

Implements [Distribution](#).

3.4.3.14 value()

```
double diracDistribution::value (
    int ) [inline]
```

Read accessor to the value of the Dirac distribution.

Returns

the value of the distribution

3.4.3.15 write()

```
void diracDistribution::write (
    FILE * out,
    int mode ) [virtual]
```

an utility to write the distribution to some file, according to some format.

Parameters

<i>out</i>	the file descriptor of the file
<i>mode</i>	a code for the format to be used

Implements [Distribution](#).

The documentation for this class was generated from the following files:

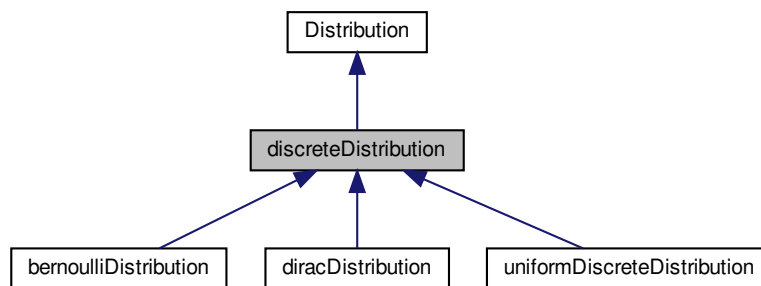
- Distribution/diracDistribution.h
- Distribution/diracDistribution.cpp

3.5 discreteDistribution Class Reference

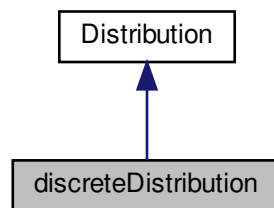
The general discrete distribution with finite support.

```
#include <discreteDistribution.h>
```

Inheritance diagram for discreteDistribution:



Collaboration diagram for discreteDistribution:



Public Member Functions

- [discreteDistribution](#) (int sz, double *vals, double *probas)
*Constructor for a general discrete distribution from arrays. The arrays are **copied**, not taken as pointer. The mean is calculated at creation.*
- [discreteDistribution](#) (int sz, char *name)
Constructor for a general discrete distribution from a file. The file is assumed to contain only the probas. The values are arbitrarily chosen between 0 and sz-1. The mean is calculated at creation.
- [~discreteDistribution](#) ()
Destructor for a general discrete distribution. The convention is that internal arrays for values and probas are freed at this moment. If they are useful for something else, they should be copied.
- double [getProbaByIndex](#) (int i)
Read accessor for the elements of the probas array. This is a pseudo-accessor since it performs additional checks.
- double [getProba](#) (double value)
Computes the probability of a particular value. The tolerance VALUE_TOLERANCE is applied to match values.
- double * [values](#) ()
Read accessor to the values (support set) of the distribution. DANGEROUS: the resulting table should be used uniquely as read-only.
- double * [probas](#) ()

Read accessor to the probabilities of the distribution. DANGEROUS: the resulting table should be used uniquely as read-only.

- double [getValue](#) (int i)

Read accessor for the values. This is a pseudo-accessor since it performs additional checks.

- int [nbVals](#) ()

Read accessor to the number of values in the distribution.

- bool [setProba](#) (int i, double v)

Write accessor for the probas. This is a pseudo-accessor since it performs additional checks.

- double [mean](#) ()

Calculation of the mean. Returns the value since it is pre-computed.

- double [rate](#) ()

Calculation of the rate, which is the inverse of the mean. If the mean is 0, the value INFINITE_RATE is returned.

- double [moment](#) (int order)

Computing the moments of the distribution.

- double [laplace](#) (double s)

computing the Laplace transform of the distribution at real point

- double [dLaplace](#) (double s)

computing the derivative of the Laplace transform at real points

- double [cdf](#) (double x)

Computation of the cumulative density function at some real point x.

- bool [hasMoment](#) (int order)

Test of existence of a moment. These distributions always have one.

- [discreteDistribution](#) * [rescale](#) (double factor)

rescaling a distribution by some real factor. Not all distributions allow this for any real factor. If the operation fails, or if the factor is 1.0, a copy of the distribution should be returned (**not** by using the [copy\(\)](#) function).

- [discreteDistribution](#) * [copy](#) ()

copying a distribution. Typically implemented as [rescale](#)(1.0).

- double [sample](#) ()

Samples a pseudo-random value of the distribution.

- double [distanceL2](#) ([discreteDistribution](#) *d)

Computes the L2 distance between the distribution and the one passed as parameter.

- double [distanceL1](#) ([discreteDistribution](#) *d)

Computes the L1 distance between the distribution and the one passed as parameter.

- double [distanceLinfinity](#) ([discreteDistribution](#) *d)

Computes the L-infinity (sup norm) distance between the distribution and the one passed as parameter.

- std::string [toString](#) ()

Printing a representation of the law into a string.

- void [write](#) (FILE *out, int mode)

Printing a representation of the law.

Protected Member Functions

- [discreteDistribution](#) (int sz)

Constructor for a general discrete distribution from its size. Arrays are created to this size but not initialized. DANGEROUS to use since methods cannot check whether these arrays are correct or not. Use reserved to sub-types such as [diracDistribution](#).

Protected Attributes

- int [_nbVals](#)
- double * [_values](#)
- double * [_probas](#)

Additional Inherited Members

3.5.1 Detailed Description

The general discrete distribution with finite support.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 discreteDistribution() [1/3]

```
discreteDistribution::discreteDistribution (
    int sz,
    double * vals,
    double * probas )
```

Constructor for a general discrete distribution from arrays. The arrays are ***copied***, not taken as pointer. The mean is calculated at creation.

Author

Alain Jean-Marie

Parameters

<i>sz</i>	number of values/probas
<i>vals</i>	array of values
<i>probas</i>	probability vector

Returns

an object of type [discreteDistribution](#)

3.5.2.2 discreteDistribution() [2/3]

```
discreteDistribution::discreteDistribution (
    int sz,
    char * name )
```

Constructor for a general discrete distribution from a file. The file is assumed to contain only the probas. The values are arbitrarily chosen between 0 and sz-1. The mean is calculated at creation.

Author

Alain Jean-Marie

Parameters

<i>sz</i>	number of values/probas
<i>name</i>	the name of the file

Returns

an object of type [discreteDistribution](#)

3.5.2.3 discreteDistribution() [3/3]

```
discreteDistribution::discreteDistribution (
    int sz ) [protected]
```

Constructor for a general discrete distribution from its size. Arrays are created to this size but not initialized. DANGEROUS to use since methods cannot check whether these arrays are correct or not. Use reserved to sub-types such as [diracDistribution](#).

Author

Alain Jean-Marie

Parameters

<i>sz</i>	number of values/probas
-----------	-------------------------

Returns

an object of type [discreteDistribution](#)

3.5.2.4 ~discreteDistribution()

```
discreteDistribution::~~discreteDistribution ( )
```

Destructor for a general discrete distribution. The convention is that internal arrays for values and probas are freed at this moment. If they are useful for something else, they should be copied.

Author

Alain Jean-Marie

3.5.3 Member Function Documentation

3.5.3.1 cdf()

```
double discreteDistribution::cdf (
    double x ) [virtual]
```

Computation of the cumulative density function at some real point x.

Parameters

x	
---	--

Returns

double

Author

Alain Jean-Marie

Parameters

x	value at which the CDF is computed
---	------------------------------------

Returns

CDF(x)

Implements [Distribution](#).

Reimplemented in [uniformDiscreteDistribution](#).

3.5.3.2 copy()

```
discreteDistribution * discreteDistribution::copy ( ) [virtual]
```

copying a distribution. Typically implemented as `rescale(1.0)`.

Copying the law.

See also

[rescale\(double\)](#)

Returns

a copy of the distribution

Author

Alain Jean-Marie

Returns

a copy of the law

Implements [Distribution](#).

Reimplemented in [uniformDiscreteDistribution](#).

3.5.3.3 distanceL1()

```
double discreteDistribution::distanceL1 (
    discreteDistribution * d )
```

Computes the L1 distance between the distribution and the one passed as parameter.

Computes the L1 distance between this distribution and another one. In case of incompatible or infinite sizes, a negative number is returned.

Parameters

<i>d</i>	
----------	--

Returns

double

Author

Alain Jean-Marie

Parameters

<i>d</i>	the second distribution
----------	-------------------------

Returns

the distance

3.5.3.4 distanceL2()

```
double discreteDistribution::distanceL2 (
    discreteDistribution * d )
```


Computes the L2 distance between the distribution and the one passed as parameter.

Computes the L2 distance between two distributions. In case of incompatible or infinite sizes, a negative number is returned.

Parameters

d	
-----	--

Returns

double

Author

Alain Jean-Marie

Parameters

d	the second distribution
-----	-------------------------

Returns

the distance

3.5.3.5 distanceLinfinity()

```
double discreteDistribution::distanceLinfinity (  
    discreteDistribution * d )
```

Computes the L-infinity (sup norm) distance between the distribution and the one passed as parameter.

Computes the L-infinity distance between two distributions. In case of incompatible or infinite sizes, a negative number is returned.

Parameters

d	
-----	--

Returns

double

Author

Alain Jean-Marie

Parameters

d	the second distribution
-----	-------------------------

Returns

the distance

3.5.3.6 dLaplace()

```
double discreteDistribution::dLaplace (  
    double  $s$  ) [virtual]
```

computing the derivative of the Laplace transform at real points

Computation of the derivative of the Laplace transform at some real point s .

Parameters

s	the value at which to compute
-----	-------------------------------

Returns

the value of the derivative of the Laplace transform

Author

Alain Jean-Marie

Parameters

s	value at which the derivative is computed
-----	-------------------------------------------

Returns

$d\text{ LT}(s)/ds$

Implements [Distribution](#).

Reimplemented in [uniformDiscreteDistribution](#).

3.5.3.7 getProba()

```
double discreteDistribution::getProba (
    double value )
```

Computes the probability of a particular value. The tolerance VALUE_TOLERANCE is applied to match values.

Author

Alain Jean-Marie

Parameters

<i>value</i>	the value at which the proba is computed
--------------	------------------------------------------

Returns

the probability of value v

3.5.3.8 getProbaByIndex()

```
double discreteDistribution::getProbaByIndex (
    int i )
```

Read accessor for the elements of the probas array. This is a pseudo-accessor since it performs additional checks.

Author

Alain Jean-Marie

Parameters

<i>i</i>	the index of the entry
----------	------------------------

Returns

the probability of entry i, or 0 if the index is out of range

3.5.3.9 getValue()

```
double discreteDistribution::getValue (
    int i )
```

Read accessor for the values. This is a pseudo-accessor since it performs additional checks.

Author

Alain Jean-Marie

Parameters

<i>i</i>	the index of the entry
----------	------------------------

Returns

the value of entry *i*, or 0 if the index is out of range

3.5.3.10 hasMoment()

```
bool discreteDistribution::hasMoment (
    int order ) [virtual]
```

Test of existence of a moment. These distributions always have one.

Author

Alain Jean-Marie

Parameters

<i>order</i>	order of the moment
--------------	---------------------

Returns

true

Implements [Distribution](#).

Reimplemented in [uniformDiscreteDistribution](#).

3.5.3.11 laplace()

```
double discreteDistribution::laplace (
    double s ) [virtual]
```

computing the Laplace transform of the distribution at real point

Computation of the Laplace transform at some real point *s*.

Parameters

<i>s</i>	the value at which to compute
----------	-------------------------------

Returns

the value of the Laplace transform

Author

Alain Jean-Marie

Parameters

s	value at which the derivative is computed
---	-------------------------------------------

Returns

LT(s)

Implements [Distribution](#).

Reimplemented in [uniformDiscreteDistribution](#).

3.5.3.12 mean()

```
double discreteDistribution::mean ( ) [inline], [virtual]
```

Calculation of the mean. Returns the value since it is pre-computed.

Author

Alain Jean-Marie

Returns

the mathematical expectation of the distribution

Implements [Distribution](#).

Reimplemented in [uniformDiscreteDistribution](#).

3.5.3.13 moment()

```
double discreteDistribution::moment (
    int order ) [virtual]
```

Computing the moments of the distribution.

Parameters

<i>order</i>	the order of the moment to be computed
--------------	----------------------------------------

Returns

the moment

Implements [Distribution](#).

Reimplemented in [uniformDiscreteDistribution](#).

3.5.3.14 nbVals()

```
int discreteDistribution::nbVals ( ) [inline]
```

Read accessor to the number of values in the distribution.

Returns

the number of values `_nbVals`

3.5.3.15 probas()

```
double* discreteDistribution::probas ( ) [inline]
```

Read accessor to the probabilities of the distribution. DANGEROUS: the resulting table should be used uniquely as read-only.

Author

Alain Jean-Marie

Returns

a pointer to the array of probabilities

3.5.3.16 rate()

```
double discreteDistribution::rate ( ) [virtual]
```

Calculation of the rate, which is the inverse of the mean. If the mean is 0, the value INFINITE_RATE is returned.

Author

Alain Jean-Marie

Returns

the rate

Implements [Distribution](#).

Reimplemented in [uniformDiscreteDistribution](#).

3.5.3.17 rescale()

```
discreteDistribution * discreteDistribution::rescale (
    double factor ) [virtual]
```

rescaling a distribution by some real factor. Not all distributions allow this for any real factor. If the operation fails, or if the factor is 1.0, a copy of the distribution should be returned (**not** by using the [copy\(\)](#) function).

Rescaling the law X by some real factor f.

See also

[copy\(\)](#)

Parameters

<i>factor</i>	the real factor to be used
---------------	----------------------------

Returns

a new distribution object

Author

Alain Jean-Marie

Parameters

<i>factor</i>	the factor f
---------------	--------------

Returns

the law $f \cdot X$

Implements [Distribution](#).

Reimplemented in [uniformDiscreteDistribution](#).

3.5.3.18 sample()

```
double discreteDistribution::sample ( ) [virtual]
```

Samples a pseudo-random value of the distribution.

Sampling from the law This is the straightforward, non optimized, linear-time algorithm.

Returns

double

Author

Alain Jean-Marie

Returns

a copy of the law

Implements [Distribution](#).

Reimplemented in [uniformDiscreteDistribution](#).

3.5.3.19 setProba()

```
bool discreteDistribution::setProba (
    int i,
    double v )
```

Write accessor for the probas. This is a pseudo-accessor since it performs additional checks.

Author

Alain Jean-Marie

Parameters

<i>i</i>	the index of the entry
<i>v</i>	the value to be given to the entry

Returns

true if ok, or false if the index is out of range

3.5.3.20 toString()

```
std::string discreteDistribution::toString ( ) [virtual]
```

Printing a representation of the law into a string.

Returns

std::string

Author

Alain Jean-Marie

Returns

a string

Implements [Distribution](#).

Reimplemented in [uniformDiscreteDistribution](#).

3.5.3.21 values()

```
double* discreteDistribution::values ( ) [inline]
```

Read accessor to the values (support set) of the distribution. DANGEROUS: the resulting table should be used uniquely as read-only.

Author

Alain Jean-Marie

Returns

a pointer to the array of values

3.5.3.22 write()

```
void discreteDistribution::write (
    FILE * out,
    int mode ) [virtual]
```

Printing a representation of the law.

Parameters

<i>out</i>	
<i>mode</i>	

Author

Alain Jean-Marie

Parameters

<i>out</i>	file descriptor of the output stream
<i>mode</i>	representation for the output

Implements [Distribution](#).

Reimplemented in [uniformDiscreteDistribution](#).

3.5.4 Member Data Documentation

3.5.4.1 `_nbVals`

```
int discreteDistribution::_nbVals [protected]
```

number of values in the distribution

3.5.4.2 `_probas`

```
double* discreteDistribution::_probas [protected]
```

table of probabilities

3.5.4.3 `_values`

```
double* discreteDistribution::_values [protected]
```

table of values

The documentation for this class was generated from the following files:

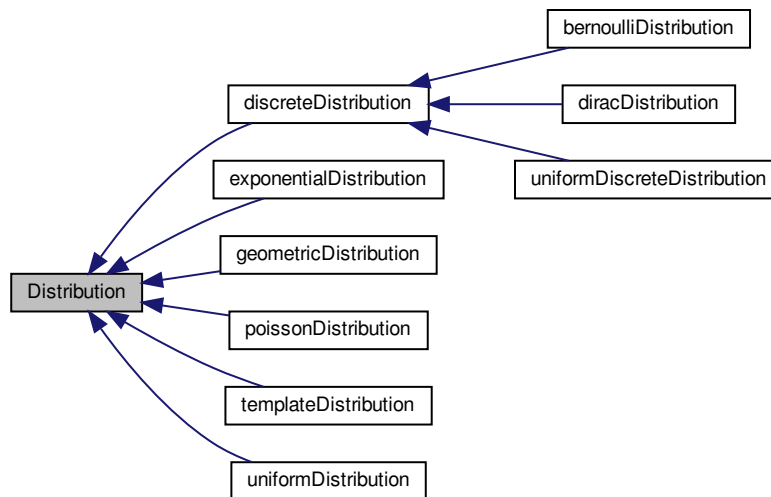
- `Distribution/discreteDistribution.h`
- `Distribution/discreteDistribution.cpp`

3.6 Distribution Class Reference

A class for representing probability distributions.

```
#include <Distribution.h>
```

Inheritance diagram for Distribution:



Public Member Functions

- virtual `~Distribution ()`
Standard destructor.
- `std::string name ()`
Read accessor to the type name of the distribution.
- virtual double `mean ()=0`
computing the mathematical expectation or mean
- virtual double `rate ()=0`
computing the "rate", defined as the inverse of the mean
- virtual double `moment (int order)=0`
Computing the moments of the distribution.
- double `variance ()`
Computing the variance of the random variable: the second moment minus the square of the first moment. Variance is the square of the coefficient of variation. The [Distribution](#) class offers a default implementation.
- virtual double `laplace (double s)=0`
computing the Laplace transform of the distribution at real point
- virtual double `dLaplace (double s)=0`
computing the derivative of the Laplace transform at real points
- virtual double `cdf (double x)=0`
computing the cumulative distribution function at some real point x. This is the probability that the random variable is less or equal to x.
- double `ccdf (double x)`

computing the complementary cumulative distributon function (or tail) at some real point x . This is the probability that the random variable is strictly larger than x . The [Distribution](#) class offers a default implementation.

- virtual bool [hasMoment](#) (int order)=0
test for the existence of moments of any order
- virtual [Distribution](#) * [rescale](#) (double factor)=0
rescaling a distribution by some real factor. Not all distributions allow this for any real factor. If the operation fails, or if the factor is 1.0, a copy of the distribution should be returned (**not** by using the [copy\(\)](#) function).
- virtual [Distribution](#) * [copy](#) ()=0
copying a distribution. Typically implemented as [rescale](#)(1.0).
- virtual double [sample](#) ()=0
drawing a (pseudo)random value according to the distribution.
- void [iidSample](#) (int n, double *s)
drawing an i.i.d. sample from the distribution. The result is returned in an array (that must have been already allocated) passed as a parameter. The [Distribution](#) class offers the default implementation with repeated call to [sample\(\)](#).
- virtual double [distanceL1](#) ([Distribution](#) *d)
Computing generally the L1 distance between distributions.
- virtual bool [hasProperty](#) (std::string pro)
Property test function. Current properties are:
- virtual std::string [toString](#) ()=0
an utility to convert the distribution into a string.
- virtual void [write](#) (FILE *out, int mode)=0
an utility to write the distribution to some file, according to some format.
- void [fprintf](#) ()
write on stdout with NORMAL_PRINT_MODE

Static Public Member Functions

- static double [u_0_1](#) (void)
- static double [exponential](#) (double [mean](#))

Protected Attributes

- std::string [_name](#)
- double [_mean](#)

Static Protected Attributes

- static const double [VALUE_TOLERANCE](#) = 1.0e-8

3.6.1 Detailed Description

A class for representing probability distributions.

3.6.2 Member Function Documentation

3.6.2.1 [ccdf\(\)](#)

```
double Distribution::ccdf (
    double x ) [inline]
```

computing the complementary cumulative distributon function (or tail) at some real point x . This is the probability that the random variable is strictly larger than x . The [Distribution](#) class offers a default implementation.

Parameters

<code>x</code>	the value at which to compute the ccdf
----------------	----------------------------------------

Returns

the value of the ccdf

3.6.2.2 cdf()

```
virtual double Distribution::cdf (
    double x ) [pure virtual]
```

computing the cumulative distribution function at some real point `x`. This is the probability that the random variable is less or equal to `x`.

Parameters

<code>x</code>	the value at which to compute the cdf
----------------	---------------------------------------

Returns

the value of the cdf

Implemented in [discreteDistribution](#), [poissonDistribution](#), [geometricDistribution](#), [uniformDiscreteDistribution](#), [diracDistribution](#), [bernoulliDistribution](#), [uniformDistribution](#), [templateDistribution](#), and [exponentialDistribution](#).

3.6.2.3 copy()

```
virtual Distribution\* Distribution::copy ( ) [pure virtual]
```

copying a distribution. Typically implemented as `rescale(1.0)`.

See also

[rescale\(double\)](#)

Returns

a copy of the distribution

Implemented in [discreteDistribution](#), [poissonDistribution](#), [geometricDistribution](#), [uniformDiscreteDistribution](#), [bernoulliDistribution](#), [uniformDistribution](#), [diracDistribution](#), [templateDistribution](#), and [exponentialDistribution](#).

3.6.2.4 distanceL1()

```
double Distribution::distanceL1 (
    Distribution * d ) [virtual]
```

Computing generally the L1 distance between distributions.

Author

Alain Jean-Marie

Parameters

<i>d</i>	the distribution with which the distance is computed
----------	------------------------------------------------------

Returns

the distance

3.6.2.5 dLaplace()

```
virtual double Distribution::dLaplace (
    double s ) [pure virtual]
```

computing the derivative of the Laplace transform at real points

Parameters

<i>s</i>	the value at which to compute
----------	-------------------------------

Returns

the value of the derivative of the Laplace transform

Implemented in [discreteDistribution](#), [poissonDistribution](#), [geometricDistribution](#), [uniformDiscreteDistribution](#), [diracDistribution](#), [bernoulliDistribution](#), [uniformDistribution](#), [templateDistribution](#), and [exponentialDistribution](#).

3.6.2.6 exponential()

```
double Distribution::exponential (
    double mean ) [static]
```

The generator for exponential distributions. Warning: the parameter is the mean.

Author

Alain Jean-Marie

Parameters

<i>mean</i>	the average (not the rate)
-------------	----------------------------

Returns

a sample of the Exponential distribution with the given mean

3.6.2.7 hasMoment()

```
virtual bool Distribution::hasMoment (
    int order ) [pure virtual]
```

test for the existence of moments of any order

Parameters

<i>order</i>	the order of the moment to be tested
--------------	--------------------------------------

Returns

true if the moment exists, false otherwise

Implemented in [discreteDistribution](#), [poissonDistribution](#), [geometricDistribution](#), [uniformDiscreteDistribution](#), [uniformDistribution](#), [diracDistribution](#), [bernoulliDistribution](#), [templateDistribution](#), and [exponentialDistribution](#).

3.6.2.8 hasProperty()

```
bool Distribution::hasProperty (
    std::string pro ) [virtual]
```

Property test function. Current properties are:

- [discreteDistribution](#): tests if the distribution is discrete and finite
- discrete: tests if the distribution is discrete
- continuous: tests if the distribution is continuous.

Author

Alain Jean-Marie

Parameters

<i>pro</i>	a string/key for the property to be tested
------------	--------------------------------------------

Returns

true if the distribution has the property, false otherwise

3.6.2.9 iidSample()

```
void Distribution::iidSample (
    int n,
    double * s )
```

drawing an i.i.d. sample from the distribution. The result is returned in an array (that must have been already allocated) passed as a parameter. The [Distribution](#) class offers the default implementation with repeated call to [sample\(\)](#).

Parameters

<i>n</i>	the number of values to sample
<i>s</i>	an array to be filled with the sample

3.6.2.10 laplace()

```
virtual double Distribution::laplace (
    double s ) [pure virtual]
```

computing the Laplace transform of the distribution at real point

Parameters

<i>s</i>	the value at which to compute
----------	-------------------------------

Returns

the value of the Laplace transform

Implemented in [discreteDistribution](#), [poissonDistribution](#), [geometricDistribution](#), [uniformDiscreteDistribution](#), [diracDistribution](#), [bernoulliDistribution](#), [uniformDistribution](#), [templateDistribution](#), and [exponentialDistribution](#).

3.6.2.11 mean()

```
virtual double Distribution::mean ( ) [pure virtual]
```

computing the mathematical expectation or mean

Returns

the mean

Implemented in [discreteDistribution](#), [poissonDistribution](#), [geometricDistribution](#), [uniformDiscreteDistribution](#), [bernoulliDistribution](#), [diracDistribution](#), [uniformDistribution](#), [exponentialDistribution](#), and [templateDistribution](#).

3.6.2.12 moment()

```
virtual double Distribution::moment (
    int order ) [pure virtual]
```

Computing the moments of the distribution.

Parameters

<i>order</i>	the order of the moment to be computed
--------------	----------------------------------------

Returns

the moment

Implemented in [discreteDistribution](#), [poissonDistribution](#), [geometricDistribution](#), [uniformDiscreteDistribution](#), [bernoulliDistribution](#), [diracDistribution](#), [uniformDistribution](#), [exponentialDistribution](#), and [templateDistribution](#).

3.6.2.13 name()

```
std::string Distribution::name ( ) [inline]
```

Read accessor to the type name of the distribution.

Returns

the name

3.6.2.14 rate()

```
virtual double Distribution::rate ( ) [pure virtual]
```

computing the "rate", defined as the inverse of the mean

Returns

the rate

Implemented in [discreteDistribution](#), [poissonDistribution](#), [geometricDistribution](#), [uniformDiscreteDistribution](#), [bernoulliDistribution](#), [diracDistribution](#), [uniformDistribution](#), [templateDistribution](#), and [exponentialDistribution](#).

3.6.2.15 rescale()

```
virtual Distribution\* Distribution::rescale (
    double factor ) [pure virtual]
```

rescaling a distribution by some real factor. Not all distributions allow this for any real factor. If the operation fails, or if the factor is 1.0, a copy of the distribution should be returned (**not** by using the [copy\(\)](#) function).

See also

[copy\(\)](#)

Parameters

<i>factor</i>	the real factor to be used
---------------	----------------------------

Returns

a new distribution object

Implemented in [discreteDistribution](#), [poissonDistribution](#), [geometricDistribution](#), [uniformDiscreteDistribution](#), [bernoulliDistribution](#), [uniformDistribution](#), [diracDistribution](#), [templateDistribution](#), and [exponentialDistribution](#).

3.6.2.16 sample()

```
virtual double Distribution::sample ( ) [pure virtual]
```

drawing a (pseudo)random value according to the distribution.

Returns

a sample

Implemented in [discreteDistribution](#), [poissonDistribution](#), [geometricDistribution](#), [uniformDiscreteDistribution](#), [bernoulliDistribution](#), [diracDistribution](#), [uniformDistribution](#), [templateDistribution](#), and [exponentialDistribution](#).

3.6.2.17 toString()

```
virtual std::string Distribution::toString ( ) [pure virtual]
```

an utility to convert the distribution into a string.

Returns

the string representing the distribution.

Implemented in [discreteDistribution](#), [poissonDistribution](#), [geometricDistribution](#), [uniformDiscreteDistribution](#), [diracDistribution](#), [uniformDistribution](#), [bernoulliDistribution](#), [templateDistribution](#), and [exponentialDistribution](#).

3.6.2.18 `u_0_1()`

```
double Distribution::u_0_1 (
    void ) [static]
```

The generator for the uniform distribution on $[0,1]$. Serves as the building block for most sampling algorithms.

Author

Alain Jean-Marie

Returns

a sample of the Uniform($[0,1]$) distribution

3.6.2.19 `variance()`

```
double Distribution::variance ( )
```

Computing the variance of the random variable: the second moment minus the square of the first moment. Variance is the square of the coefficient of variation. The [Distribution](#) class offers a default implementation.

Returns

the variance

3.6.2.20 `write()`

```
virtual void Distribution::write (
    FILE * out,
    int mode ) [pure virtual]
```

an utility to write the distribution to some file, according to some format.

Parameters

<i>out</i>	the file descriptor of the file
<i>mode</i>	a code for the format to be used

Implemented in [discreteDistribution](#), [poissonDistribution](#), [geometricDistribution](#), [uniformDiscreteDistribution](#), [diracDistribution](#), [uniformDistribution](#), [bernoulliDistribution](#), [exponentialDistribution](#), and [templateDistribution](#).

3.6.3 Member Data Documentation

3.6.3.1 `_mean`

```
double Distribution::_mean [protected]
```

the mathematical expectation of the distribution

3.6.3.2 `_name`

```
std::string Distribution::_name [protected]
```

a string uniquely representing the distribution

3.6.3.3 `VALUE_TOLERANCE`

```
const double Distribution::VALUE_TOLERANCE = 1.0e-8 [static], [protected]
```

tolerance accepted when looking for values

The documentation for this class was generated from the following files:

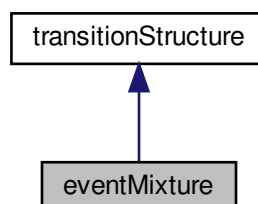
- Distribution/Distribution.h
- Distribution/Distribution.cpp

3.7 eventMixture Class Reference

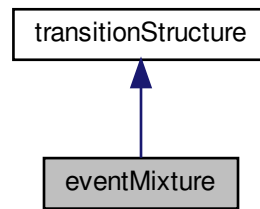
A class representing probabilistic mixtures of "events", that is, elementary transition structures.

```
#include <eventMixture.h>
```

Inheritance diagram for eventMixture:



Collaboration diagram for eventMixture:



Public Member Functions

- `eventMixture` (int `size`, int `nbEvents`, double `*probas`, std::string `*names`, int `**transitions`)
Constructor from arrays.
- `eventMixture` (sparseMatrix `*spMat`)
Contruction from a sparse matrix.
- `~eventMixture` ()
Destructor of the class.
- int `nbEvents` ()
Read accessor for the number of events.
- double `eventProba` (int `e`)
Read accessor for the probabilities associated to each event.
- bool `setEntry` (int `i`, int `j`, double `val`)
Method to set the value associated with some transition. Not applicable here.
- double `getEntry` (int `i`, int `j`)
Method to get the value associated with some transition. When state parameters are out of bounds, the returned value should be 0.
- int `getNbElts` (int `i`)
Method to get the number of non-zero entries in a transition from some state. This can be seen as the number of actually possible transitions from that state.
- int `getCol` (int `i`, int `k`)
*Method to get the **number** of the state corresponding to transition number `k` in the list of possible transitions from some state `i`.*
- double `getEntryByCol` (int `i`, int `k`)
*Method to get the **value** attached to transition number `k` in the list of possible transitions from some state `i`.*
- discreteDistribution `* getTransDistrib` (int `i`)
Method to get the transition from some state as a probability distribution.
- double `rowSum` (int `i`)
Sum of entries on some row `i`. Always 1.0 since this is a discrete-time transition structure.
- `eventMixture` `* copy` ()
Copying a transition structure.
- `eventMixture` `* uniformize` ()
Uniformizing a transition structure. Since the origin structure is already of discrete-time type, a copy is returned.
- `eventMixture` `* embed` ()
Embedding in a transition structure. Since the origin structure is already of discrete-time type, a copy is returned.
- void `evaluateMeasure` (double `*d`, double `*res`)

Computing the action of the transition structure on some measure, the measure being represented as a vector of real numbers. This corresponds to the multiplication vector/matrix, the row vector being interpreted as a signed measure.

- `discreteDistribution * evaluateMeasure (discreteDistribution *d)`

Computing the action of the transition structure on some probability distribution. This version with `discreteDistribution` objects uses `evaluateMeasure(double*,double*)`.

- `void evaluateValue (double *v, double *res)`

Evaluation of the expected value after a transition, for all states. Corresponds to matrix/vector multiplication. The result is stored in an array that must be already allocated.

- `double evaluateValueState (double *v, int stateIndex)`

Evaluation of the expected value after a transition from some state. Corresponds to matrix/vector multiplication for a specific state/line of the matrix.

- `void write (FILE *out, std::string format)`

Output method for the transition structure. Supported formats are: XBORNE, MARCA, Ers, Maple and PSI3 (experimental)

Additional Inherited Members

3.7.1 Detailed Description

A class representing probabilistic mixtures of "events", that is, elementary transition structures.

3.7.2 Constructor & Destructor Documentation

3.7.2.1 `eventMixture()` [1/2]

```
eventMixture::eventMixture (
    int size,
    int nbEvents,
    double * probas,
    std::string * names,
    int ** transitions )
```

Constructor from arrays.

Parameters

<i>size</i>	the number of states
<i>nbEvents</i>	the number of different events
<i>probas</i>	the array of probabilities. This array is copied
<i>names</i>	the names of events. This array is copied
<i>transitions</i>	the array of state-to-state mappings. This array is not copied .

3.7.2.2 eventMixture() [2/2]

```
eventMixture::eventMixture (
    sparseMatrix * spMat )
```

Contruction from a sparse matrix.

Parameters

<i>spMat</i>	the sparse matrix to convert into an event mixture
--------------	----------------------------------------------------

3.7.3 Member Function Documentation

3.7.3.1 copy()

```
eventMixture * eventMixture::copy ( ) [virtual]
```

Copying a transition structure.

Returns

[transitionStructure](#)

Implements [transitionStructure](#).

3.7.3.2 embed()

```
eventMixture * eventMixture::embed ( ) [virtual]
```

Embedding in a transition structure. Since the origin structure is already of discrete-time type, a copy is returned.

Returns

a discrete-time transition structure

Implements [transitionStructure](#).

3.7.3.3 evaluateMeasure() [1/2]

```
void eventMixture::evaluateMeasure (
    double * d,
    double * res ) [virtual]
```

Computing the action of the transition structure on some measure, the measure being represented as a vector of real numbers. This corresponds to the multiplication vector/matrix, the row vector being interpreted as a signed measure.

Parameters

<i>d</i>	the measure to evaluate
<i>res</i>	the resulting measure

Reimplemented from [transitionStructure](#).

3.7.3.4 `evaluateMeasure()` [2/2]

```
discreteDistribution * eventMixture::evaluateMeasure (
    discreteDistribution * d )
```

Computing the action of the transition structure on some probability distribution. This version with [discrete↔Distribution](#) objects uses [evaluateMeasure\(double*,double*\)](#).

Parameters

<i>d</i>	the distribution to evaluate
----------	------------------------------

Returns

the distribution resulting from the evaluation

3.7.3.5 `evaluateValue()`

```
void eventMixture::evaluateValue (
    double * v,
    double * res ) [virtual]
```

Evaluation of the expected value after a transition, for all states. Corresponds to matrix/vector multiplication. The result is stored in an array that must be already allocated.

Computing the action of the transition structure on some vector of values. This corresponds to the multiplication matrix/vector, the column vector being interpreted as a vector of values attached to the states.

Author

Alain Jean-Marie

Parameters

<i>v</i>	the value (column vector) to be multiplied
<i>res</i>	the resulting vector
<i>v</i>	the vector of values to evaluate
<i>res</i>	the resulting vector

Implements [transitionStructure](#).

3.7.3.6 evaluateValueState()

```
double eventMixture::evaluateValueState (
    double * v,
    int stateIndex )
```

Evaluation of the expected value after a transition from some state. Corresponds to matrix/vector multiplication for a specific state/line of the matrix.

Author

Alain Jean-Marie

Parameters

<i>v</i>	the value (column vector) to be multiplied
<i>stateIndex</i>	the index of the state for which to perform the multiplication

Returns

the value of the state

3.7.3.7 eventProba()

```
double eventMixture::eventProba (
    int e ) [inline]
```

Read accessor for the probabilities associated to each event.

Parameters

<i>e</i>	the event index
----------	-----------------

Returns

the probability that the event is e

3.7.3.8 getCol()

```
int eventMixture::getCol (
    int i,
    int k ) [virtual]
```

Method to get the **number** of the state corresponding to transition number k in the list of possible transitions from some state i .

Parameters

i	the origin state
k	the index of transition from state i

Returns

the destination state corresponding to the k -th possible transition from state i

Implements [transitionStructure](#).

3.7.3.9 getEntry()

```
double eventMixture::getEntry (
    int i,
    int j ) [virtual]
```

Method to get the value associated with some transition. When state parameters are out of bounds, the returned value should be 0.

Parameters

i	the origin state
j	the destination state

Returns

the value attached to the transition (i,j)

Implements [transitionStructure](#).

3.7.3.10 getEntryByCol()

```
double eventMixture::getEntryByCol (
    int i,
    int k ) [virtual]
```

Method to get the **value** attached to transition number k in the list of possible transitions from some state i .

Parameters

i	the origin state
k	the index of the transition from state i

Returns

the value attached to the k-th possible transition from state i

Implements [transitionStructure](#).

3.7.3.11 getNbElts()

```
int eventMixture::getNbElts (
    int i ) [virtual]
```

Method to get the number of non-zero entries in a transition from some state. This can be seen as the number of actually possible transitions from that state.

Parameters

<i>i</i>	the origin state
----------	------------------

Returns

int the number of possible transitions

Implements [transitionStructure](#).

3.7.3.12 getTransDistrib()

```
discreteDistribution * eventMixture::getTransDistrib (
    int i ) [virtual]
```

Method to get the transition from some state as a probability distribution.

Parameters

<i>i</i>	the origin state
----------	------------------

Returns

a discrete distribution object

Implements [transitionStructure](#).

3.7.3.13 nbEvents()

```
int eventMixture::nbEvents ( ) [inline]
```

Read accessor for the number of events.

Returns

the number of events

3.7.3.14 rowSum()

```
double eventMixture::rowSum (
    int i ) [virtual]
```

Sum of entries on some row i. Always 1.0 since this is a discrete-time transition structure.

Parameters

<i>i</i>	the row to be summed
----------	----------------------

Returns

1.0

Implements [transitionStructure](#).

3.7.3.15 setEntry()

```
bool eventMixture::setEntry (
    int i,
    int j,
    double val ) [virtual]
```

Method to set the value associated with some transition. Not applicable here.

Parameters

<i>i</i>	the origin state
<i>j</i>	the destination state
<i>val</i>	the value attached to the transition

Returns

false since it is not possible to set values in this structure

Implements [transitionStructure](#).

3.7.3.16 uniformize()

```
eventMixture * eventMixture::uniformize ( ) [virtual]
```

Uniformizing a transition structure. Since the origin structure is already of discrete-time type, a copy is returned.

Returns

a discrete-time transition structure

Implements [transitionStructure](#).

3.7.3.17 write()

```
void eventMixture::write (
    FILE * out,
    std::string format )
```

Output method for the transition structure. Supported formats are: XBORNE, MARCA, Ers, Maple and PSI3 (experimental)

Output method for the transition structure. Supported formats are: XBORNE, MARCA, Ers, Maple.

Parameters

<i>out</i>	the file descriptor to which the structure should be written.
<i>format</i>	the format/language to be used

The documentation for this class was generated from the following files:

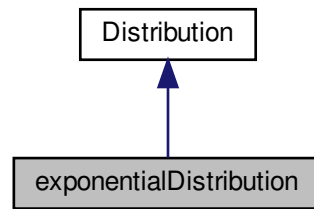
- transitionStructure/eventMixture.h
- transitionStructure/eventMixture.cpp

3.8 exponentialDistribution Class Reference

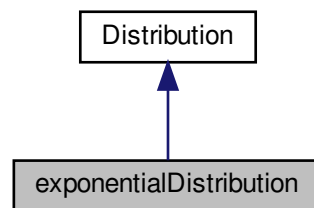
The class representing the (negative) exponential distribution.

```
#include <exponentialDistribution.h>
```

Inheritance diagram for exponentialDistribution:



Collaboration diagram for exponentialDistribution:



Public Member Functions

- `exponentialDistribution` (double val)
Unique constructor for the exponential distribution, from its average. The rate is computed at creation time.
- double `rate` ()
Read accessor for the rate.
- double `mean` ()
Calculation of the mean. Returns the value since it is pre-computed.
- double `moment` (int order)
Computing the moments of the distribution.
- double `laplace` (double s)
computing the Laplace transform of the distribution at real point
- double `dLaplace` (double s)
computing the derivative of the Laplace transform at real points
- double `cdf` (double x)
computing the cumulative distribution function at some real point x. This is the probability that the random variable is less or equal to x.
- bool `hasMoment` (int order)
Test of existence of a moment. These distributions always have one.
- `exponentialDistribution` * `rescale` (double factor)

*rescaling a distribution by some real factor. Not all distributions allow this for any real factor. If the operation fails, or if the factor is 1.0, a copy of the distribution should be returned (**not** by using the `copy()` function).*

- `exponentialDistribution * copy ()`
copying a distribution. Typically implemented as `rescale(1.0)`.
- `double sample ()`
Sampling from the law. Uses the mother class method.
- `std::string toString ()`
Printing a representation of the law into a string.
- `void write (FILE *out, int mode)`

Additional Inherited Members

3.8.1 Detailed Description

The class representing the (negative) exponential distribution.

3.8.2 Constructor & Destructor Documentation

3.8.2.1 exponentialDistribution()

```
exponentialDistribution::exponentialDistribution (
    double val )
```

Unique constructor for the exponential distribution, from its average. The rate is computed at creation time.

Author

Alain Jean-Marie

Parameters

<i>val</i>	the mean of the distribution
------------	------------------------------

3.8.3 Member Function Documentation

3.8.3.1 cdf()

```
double exponentialDistribution::cdf (
    double x ) [virtual]
```

computing the cumulative distribution function at some real point x. This is the probability that the random variable is less or equal to x.

Parameters

<code>x</code>	the value at which to compute the cdf
----------------	---------------------------------------

Returns

the value of the cdf

Implements [Distribution](#).

3.8.3.2 copy()

```
exponentialDistribution * exponentialDistribution::copy ( ) [virtual]
```

copying a distribution. Typically implemented as `rescale(1.0)`.

See also

[rescale\(double\)](#)

Returns

a copy of the distribution

Implements [Distribution](#).

3.8.3.3 dLaplace()

```
double exponentialDistribution::dLaplace (
    double s ) [virtual]
```

computing the derivative of the Laplace transform at real points

Parameters

<code>s</code>	the value at which to compute
----------------	-------------------------------

Returns

the value of the derivative of the Laplace transform

Implements [Distribution](#).

3.8.3.4 hasMoment()

```
bool exponentialDistribution::hasMoment (
    int order ) [virtual]
```

Test of existence of a moment. These distributions always have one.

Author

Alain Jean-Marie

Parameters

<i>order</i>	order of the moment
--------------	---------------------

Returns

true

Implements [Distribution](#).

3.8.3.5 laplace()

```
double exponentialDistribution::laplace (
    double s ) [virtual]
```

computing the Laplace transform of the distribution at real point

Parameters

<i>s</i>	the value at which to compute
----------	-------------------------------

Returns

the value of the Laplace transform

Implements [Distribution](#).

3.8.3.6 mean()

```
double exponentialDistribution::mean ( ) [inline], [virtual]
```

Calculation of the mean. Returns the value since it is pre-computed.

Author

Alain Jean-Marie

Returns

the mathematical expectation of the distribution

Implements [Distribution](#).

3.8.3.7 moment()

```
double exponentialDistribution::moment (
    int order ) [virtual]
```

Computing the moments of the distribution.

Parameters

<i>order</i>	the order of the moment to be computed
--------------	----------------------------------------

Returns

the moment

Implements [Distribution](#).

3.8.3.8 rate()

```
double exponentialDistribution::rate ( ) [inline], [virtual]
```

Read accessor for the rate.

Returns

the value of the rate

Implements [Distribution](#).

3.8.3.9 rescale()

```
exponentialDistribution * exponentialDistribution::rescale (
    double factor ) [virtual]
```

rescaling a distribution by some real factor. Not all distributions allow this for any real factor. If the operation fails, or if the factor is 1.0, a copy of the distribution should be returned (**not** by using the [copy\(\)](#) function).

See also

[copy\(\)](#)

Parameters

<i>factor</i>	the real factor to be used
---------------	----------------------------

Returns

a new distribution object

Implements [Distribution](#).

3.8.3.10 sample()

```
double exponentialDistribution::sample ( ) [virtual]
```

Sampling from the law. Uses the mother class method.

Returns

a sample from the exponential distribution

Implements [Distribution](#).

3.8.3.11 toString()

```
std::string exponentialDistribution::toString ( ) [virtual]
```

Printing a representation of the law into a string.

Author

Alain Jean-Marie

Returns

a string with a symbol for the law and its parameter

Implements [Distribution](#).

3.8.3.12 write()

```
void exponentialDistribution::write (
    FILE * out,
    int mode ) [virtual]
```

Printing a representation of the law

Author

Alain Jean-Marie

Parameters

<i>out</i>	the file descriptor of the output stream
<i>mode</i>	representation for the output

Implements [Distribution](#).

The documentation for this class was generated from the following files:

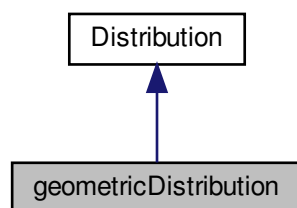
- Distribution/exponentialDistribution.h
- Distribution/exponentialDistribution.cpp

3.9 `geometricDistribution` Class Reference

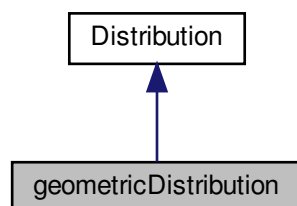
The geometric distribution with starting value 0. The parameter "p" is called "ratio". The Geometric distribution is discrete but does not inherit from [discreteDistribution](#) because its range is infinite.

```
#include <geometricDistribution.h>
```

Inheritance diagram for `geometricDistribution`:



Collaboration diagram for `geometricDistribution`:



Public Member Functions

- [geometricDistribution](#) (double p)
Unique constructor for the class, from its "ratio".
- double [getProba](#) (double k)
Function to obtain the probability of a specific value k.
- double [p](#) ()
Function to obtain the parameter (or ratio) of the distribution. Redundant with [p\(\)](#) but defined to be more explicit.
- double [getRatio](#) ()
Function to obtain the parameter (or ratio) of the distribution. Redundant with [getRatio\(\)](#) but defined according to the coding convention.
- double [mean](#) ()
Function to obtain the mean (expectation). Its value is $1/(1-p)$
- double [rate](#) ()
- double [moment](#) (int order)
Computing the moments of the distribution.
- double [laplace](#) (double s)
computing the Laplace transform of the distribution at real point
- double [dLaplace](#) (double s)
computing the derivative of the Laplace transform at real points
- double [cdf](#) (double x)
- bool [hasMoment](#) (int order)
Test of existence of a moment. These distributions always have one.
- [geometricDistribution](#) * [rescale](#) (double factor)
Rescaling the distribution. Geometric distributions cannot be rescaled. A copy is returned. A warning is issued if the factor is not 1.0.
- [geometricDistribution](#) * [copy](#) ()
copying a distribution. Typically implemented as [rescale\(1.0\)](#).
- double [sample](#) ()
Sampling from the distribution. The method uses the fact that the integer part of an exponential random variable is a geometric random variable.
- std::string [toString](#) ()
- void [write](#) (FILE *out, int mode)

Additional Inherited Members

3.9.1 Detailed Description

The geometric distribution with starting value 0. The parameter "p" is called "ratio". The Geometric distribution is discrete but does not inherit from [discreteDistribution](#) because its range is infinite.

Author

Alain Jean-Marie

3.9.2 Constructor & Destructor Documentation

3.9.2.1 `geometricDistribution()`

```
geometricDistribution::geometricDistribution (
    double p )
```

Unique constructor for the class, from its "ratio".

Author

Alain Jean-Marie

Parameters

p	the probability of being larger than 0
-----	----------------------------------------

Constructor for a geometric distribution The mean is calculated at creation. The value $p = 1$ is admitted, in which case the law is a Dirac at infinity and has no moments.

Author

Alain Jean-Marie

Parameters

p	the probability of being non 0
-----	--------------------------------

Returns

an object of type [geometricDistribution](#)

3.9.3 Member Function Documentation

3.9.3.1 `cdf()`

```
double geometricDistribution::cdf (
    double x ) [virtual]
```

Parameters

x	
-----	--

Returns

double

Computation of the cumulative density function at some real point x . Based on the formula $P(X \geq k) = p^k$ for integer k so that $P(X \leq x) = P(X \leq \text{floor}(x)) = 1 - P(X \geq \text{floor}(x) + 1) = 1 - p^{\text{floor}(x) + 1}$

Author

Alain Jean-Marie

Parameters

<i>x</i>	value at which the CDF is computed
----------	------------------------------------

Returns

CDF(*x*)

Implements [Distribution](#).

3.9.3.2 copy()

```
geometricDistribution * geometricDistribution::copy ( ) [virtual]
```

copying a distribution. Typically implemented as `rescale(1.0)`.

See also

[rescale\(double\)](#)

Returns

a copy of the distribution

Implements [Distribution](#).

3.9.3.3 dLaplace()

```
double geometricDistribution::dLaplace (
    double s ) [virtual]
```

computing the derivative of the Laplace transform at real points

Parameters

<i>s</i>	the value at which to compute
----------	-------------------------------

Returns

the value of the derivative of the Laplace transform

Computation of the derivative of the Laplace transform at some real point s

Author

Alain Jean-Marie

Parameters

s	value at which the derivative is computed
-----	-------------------------------------------

Returns

$d\text{LT}(s)/ds$

Implements [Distribution](#).

3.9.3.4 getProba()

```
double geometricDistribution::getProba (
    double k )
```

Function to obtain the probability of a specific value k .

Parameters

k	the value at which the probability should be computed
-----	-------------------------------------------------------

Returns

the probability that the random variable is k

Computation of the probability of some value

Author

Alain Jean-Marie

Parameters

k	value at which the probability is computed
-----	--------------------------------------------

Returns

$P(X = k)$

3.9.3.5 getRatio()

```
double geometricDistribution::getRatio ( ) [inline]
```

Function to obtain the parameter (or ratio) of the distribution. Redundant with [getRatio\(\)](#) but defined according to the coding convention.

Returns

double

3.9.3.6 hasMoment()

```
bool geometricDistribution::hasMoment (
    int order ) [virtual]
```

Test of existence of a moment. These distributions always have one.

Author

Alain Jean-Marie

Parameters

<i>order</i>	order of the moment
--------------	---------------------

Returns

true

Test of existence of a moment. These distributions always have one.

Author

Alain Jean-Marie

Parameters

<i>order</i>	order of the moment
--------------	---------------------

Returns

true iff $p < 1.0$

Implements [Distribution](#).

3.9.3.7 laplace()

```
double geometricDistribution::laplace (
    double s ) [virtual]
```

computing the Laplace transform of the distribution at real point

Parameters

s	the value at which to compute
---	-------------------------------

Returns

the value of the Laplace transform

Computation of the Laplace transform at some real point s

Author

Alain Jean-Marie

Parameters

s	value at which the derivative is computed
---	-------------------------------------------

Returns

LT(s)

Implements [Distribution](#).

3.9.3.8 mean()

```
double geometricDistribution::mean ( ) [virtual]
```

Function to obtain the mean (expectation). Its value is $1/(1-p)$

See also

`geometricDistribution::_p`

Returns

the mathematical expectation of the distribution

Calculation of the mean. Returns the value since it is pre-computed

Author

Alain Jean-Marie

Returns

the mathematical expectation of the distribution

Implements [Distribution](#).

3.9.3.9 moment()

```
double geometricDistribution::moment (
    int order ) [virtual]
```

Computing the moments of the distribution.

Parameters

<i>order</i>	the order of the moment to be computed
--------------	----------------------------------------

Returns

the moment

Calculation of the of order n

Author

Alain Jean-Marie

Parameters

<i>order</i>	order of the moment
--------------	---------------------

Returns

the moment

Implements [Distribution](#).

3.9.3.10 p()

```
double geometricDistribution::p ( ) [inline]
```

Function to obtain the parameter (or ratio) of the distribution. Redundant with [p\(\)](#) but defined to be more explicit.

Returns

the value of the ratio

3.9.3.11 rate()

```
double geometricDistribution::rate ( ) [virtual]
```

Returns

double

Calculation of the rate, which is the inverse of the mean

Author

Alain Jean-Marie

Returns

the rate

Implements [Distribution](#).

3.9.3.12 rescale()

```
geometricDistribution * geometricDistribution::rescale (
    double factor ) [virtual]
```

Rescaling the distribution. Geometric distributions cannot be rescaled. A copy is returned. A warning is issued if the factor is not 1.0.

Parameters

<i>factor</i>	the factor by which to rescale
---------------	--------------------------------

Returns

the rescaled distribution.

Implements [Distribution](#).

3.9.3.13 sample()

```
double geometricDistribution::sample ( ) [virtual]
```

Sampling from the distribution. The method uses the fact that the integer part of an exponential random variable is a geometric random variable.

Returns

a sample from the geometric distribution

Implements [Distribution](#).

3.9.3.14 toString()

```
std::string geometricDistribution::toString ( ) [virtual]
```

Returns

std::string

Implements [Distribution](#).

3.9.3.15 write()

```
void geometricDistribution::write (
    FILE * out,
    int mode ) [virtual]
```

Parameters

<i>out</i>	
<i>mode</i>	

Implements [Distribution](#).

The documentation for this class was generated from the following files:

- Distribution/geometricDistribution.h
- Distribution/geometricDistribution.cpp

3.10 LAW_DESC Struct Reference

Description of a distribution.

```
#include <law.h>
```

Collaboration diagram for LAW_DESC:



Public Attributes

- char [Name](#)
- double * [Parameters](#)
- struct [LAW_DESC](#) ** [Law_Parameters](#)

3.10.1 Detailed Description

Description of a distribution.

Author

Alain Jean-Maire

3.10.2 Member Data Documentation

3.10.2.1 Law_Parameters

```
struct LAW\_DESC** LAW_DESC::Law_Parameters
```

array of sub-distributions, e.g. for mixture or other compositions

3.10.2.2 Name

```
char LAW_DESC::Name
```

name-identifier of the distribution

3.10.2.3 Parameters

```
double* LAW_DESC::Parameters
```

array of standard parameters for the distribution

The documentation for this struct was generated from the following file:

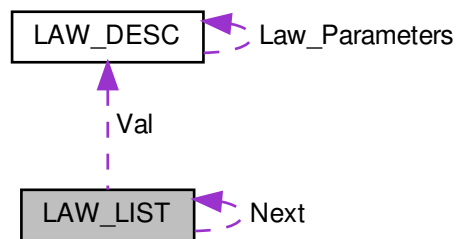
- [Distribution/law.h](#)

3.11 LAW_LIST Struct Reference

Chained list structure for lists of distributions.

```
#include <law.h>
```

Collaboration diagram for LAW_LIST:



Public Attributes

- [Law_Desc](#) Val
- struct [LAW_LIST](#) * [Next](#)

3.11.1 Detailed Description

Chained list structure for lists of distributions.

Author

Alain Jean-Marie

3.11.2 Member Data Documentation

3.11.2.1 Next

```
struct LAW\_LIST* LAW_LIST::Next
```

pointer to the next in the list

3.11.2.2 Val

`Law_Desc LAW_LIST:Val`

the distribution at the no

The documentation for this struct was generated from the following file:

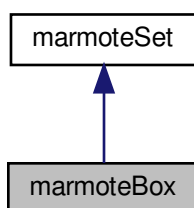
- Distribution/law.h

3.12 marmoteBox Class Reference

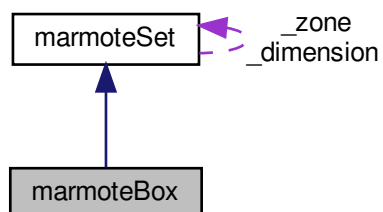
The class representing "rectangular" sets. Boxes are cartesian products of intervals.

```
#include <marmoteBox.h>
```

Inheritance diagram for marmoteBox:



Collaboration diagram for marmoteBox:



Public Member Functions

- [marmoteBox](#) (int nbDims, int *dimSize)
Constructor for a [marmoteBox](#) from the arrays of sizes in each dimension. By convention, the SW corner of the box is (0,...,0). All arrays passed as parameters are copied.
- [marmoteBox](#) (int nbDims, int *lower, int *upper)
Constructor for general boxes, from the arrays of lower/upper values. By convention, these lower/upper values belong to the box. All arrays passed as parameters are copied.
- [~marmoteBox](#) ()
Destructor.
- bool [isFinite](#) ()
Test whether the set is finite. Boxes are finite if and only if the size in each dimension is finite.
- bool [isZero](#) (int *buffer)
Tests that a state, given by its vector representation, is zero.
- void [firstState](#) (int *buffer)
Initializes some state buffer with the first state of the set.
- void [nextState](#) (int *buffer)
Procedure to compute the state following a given state in the set order. The new state is computed in place: the original state is erased. By convention, the state following the last state is the first state.
- void [decodeState](#) (int index, int *buf)
utility to convert a state index into a state array
- int [index](#) (int *buf)
Utility to find the number of some state.
- void [printStats](#) (FILE *out, int *buffer)
Procedure for printing out a state.
- int [nextStatebyDim](#) (int *buffer, int dim)
Procedure to compute the state following a given state following a dimension. The new state is computed in place: the original state is erased. By convention, the state following the last state is the first state.
- int [firstStatebyDim](#) (int *buffer, int dim)
Initializes some state buffer with the first state of the dimension the other parameters being unchanged.
- int [cardinalbyDim](#) (int dim)
procedure to know size of a dimension
- int [nextStateOutDim](#) (int *buffer, int dim)
Procedure to compute the state following a given state letting fix a dimension of the set. The new state is computed in place: the original state is erased. By convention, the state following the last state is the first state.
- int [firstStateOutDim](#) (int *buffer, int dim)
Procedure to initialize some state buffer without modifying the current dimension.
- int [cardinalOutDim](#) (int dim)
procedure to know the size of a set without considering one dimension

Additional Inherited Members

3.12.1 Detailed Description

The class representing "rectangular" sets. Boxes are cartesian products of intervals.

3.12.2 Constructor & Destructor Documentation

3.12.2.1 marmoteBox() [1/2]

```
marmoteBox::marmoteBox (
    int nbDims,
    int * dimSize )
```

Constructor for a [marmoteBox](#) from the arrays of sizes in each dimension. By convention, the SW corner of the box is (0,...,0). All arrays passed as parameters are copied.

Parameters

<i>nbDims</i>	the number of dimensions
<i>dimSize</i>	array of size in each dimension (possibly INFINITE_STATE_SPACE_SIZE)

3.12.2.2 marmoteBox() [2/2]

```
marmoteBox::marmoteBox (
    int nbDims,
    int * lower,
    int * upper )
```

Constructor for general boxes, from the arrays of lower/upper values. By convention, these lower/upper values belong to the box. All arrays passed as parameters are copied.

Parameters

<i>nbDims</i>	the number of dimensions
<i>lower</i>	array of lower values in each dimension
<i>upper</i>	array of upper values in each dimension (possibly INFINITE_STATE_SPACE_SIZE)

3.12.3 Member Function Documentation

3.12.3.1 cardinalbyDim()

```
int marmoteBox::cardinalbyDim (
    int dim ) [virtual]
```

procedure to know size of a dimension

Parameters

<i>dim</i>	the dimension
------------	---------------

Returns

the size

Reimplemented from [marmoteSet](#).

3.12.3.2 cardinalOutDim()

```
int marmoteBox::cardinalOutDim (
    int dim ) [virtual]
```

procedure to know the size of a set without considering one dimension

Parameters

<i>dim</i>	the dimension to not consider
------------	-------------------------------

Returns

the size

Reimplemented from [marmoteSet](#).

3.12.3.3 decodeState()

```
void marmoteBox::decodeState (
    int index,
    int * buf ) [virtual]
```

utility to convert a state index into a state array

Author

Alain Jean-Marie

Parameters

<i>index</i>	the state index
<i>buf</i>	the state buffer to be filled

Reimplemented from [marmoteSet](#).

3.12.3.4 firstState()

```
void marmoteBox::firstState (
    int * buffer ) [virtual]
```

Initializes some state buffer with the first state of the set.

Parameters

<i>buffer</i>	the buffer to be set.
---------------	-----------------------

Reimplemented from [marmoteSet](#).

3.12.3.5 firstStatebyDim()

```
int marmoteBox::firstStatebyDim (
    int * buffer,
    int dim ) [virtual]
```

Initializes some state buffer with the first state of the dimension the other parameters being unchanged.

Parameters

<i>buffer</i>	the buffer to be set.
<i>dim</i>	the dimension

Returns

the index of the state

Reimplemented from [marmoteSet](#).

3.12.3.6 firstStateOutDim()

```
int marmoteBox::firstStateOutDim (
    int * buffer,
    int dim ) [virtual]
```

Procedure to initialize some state buffer without modifying the current dimension.

Parameters

<i>buffer</i>	the buffer to be set.
<i>dim</i>	the dimension keeping unchnaged

Returns

the index of the state

Reimplemented from [marmoteSet](#).

3.12.3.7 index()

```
int marmoteBox::index (
    int * buf ) [virtual]
```

Utility to find the number of some state.

Parameters

<i>buf</i>	a state buffer
------------	----------------

Returns

the index of the state buffer

Reimplemented from [marmoteSet](#).

3.12.3.8 isZero()

```
bool marmoteBox::isZero (
    int * buffer ) [virtual]
```

Tests that a state, given by its vector representation, is zero.

Parameters

<i>buffer</i>	the state to be tested
---------------	------------------------

Returns

true if the state is 0, false otherwise.

Reimplemented from [marmoteSet](#).

3.12.3.9 nextState()

```
void marmoteBox::nextState (
    int * buffer ) [virtual]
```

Procedure to compute the state following a given state in the set order. The new state is computed in place: the original state is erased. By convention, the state following the last state is the first state.

Parameters

<i>buffer</i>	the reference state
---------------	---------------------

Reimplemented from [marmoteSet](#).

3.12.3.10 nextStatebyDim()

```
int marmoteBox::nextStatebyDim (  
    int * buffer,  
    int dim ) [virtual]
```

Procedure to compute the state following a given state following a dimension. The new state is computed in place: the original state is erased. By convention, the state following the last state is the first state.

Parameters

<i>buffer</i>	the reference state
<i>dim</i>	the dimension

Returns

the index of the state

Reimplemented from [marmoteSet](#).

3.12.3.11 nextStateOutDim()

```
int marmoteBox::nextStateOutDim (  
    int * buffer,  
    int dim ) [virtual]
```

Procedure to compute the state following a given state letting fix a dimension of the set. The new state is computed in place: the original state is erased. By convention, the state following the last state is the first state.

Parameters

<i>buffer</i>	the reference state
<i>dim</i>	the dimension

Returns

the index of the state

Reimplemented from [marmoteSet](#).

3.12.3.12 printState()

```
void marmoteBox::printState (
    FILE * out,
    int * buffer ) [virtual]
```

Procedure for printing out a state.

Parameters

<i>out</i>	file descriptor of the stream to be used
<i>buffer</i>	the state buffer to be printed

Reimplemented from [marmoteSet](#).

The documentation for this class was generated from the following files:

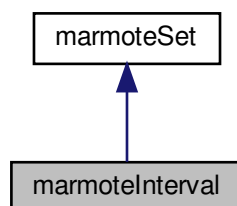
- Set/marmoteBox.h
- Set/marmoteBox.cpp

3.13 marmoteInterval Class Reference

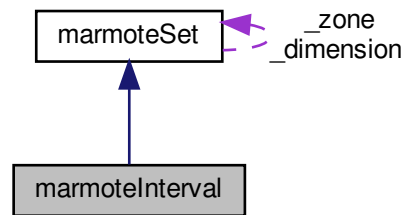
The class describing a **finite** integer interval.

```
#include <marmoteInterval.h>
```

Inheritance diagram for marmoteInterval:



Collaboration diagram for marmoteInterval:



Public Member Functions

- [marmoteInterval](#) (int min, int max)
- [~marmoteInterval](#) ()
- bool [isFinite](#) ()
Test if the set is finite. These sets always are.
- bool [isZero](#) (int *buffer)
Function that tests whether a state is the first state (state zero) or not.
- void [firstState](#) (int *buffer)
Initializes some state buffer with the first state of the set.
- void [nextState](#) (int *buffer)
Procedure to compute the state following a given state in the set order. The new state is computed in place: the original state is erased. By convention, the state following the last state is the first state.
- void [decodeState](#) (int index, int *buffer)
Procedure that converts an index into a state. The state is written in the buffer provided. The array must have been allocated before.
- int [index](#) (int *buffer)
Function that computes the number (index) of some state in the order of the set. The base class provides an implementation that uses solely [firstState\(\)](#), [nextState\(\)](#), and state comparison. For efficiency, it is advised to re-implement this method for derived classes.
- void [printStats](#) (FILE *out, int *buffer)
Procedure to print a state, given by its full description, to some file descriptor.
- void [enumerate](#) ()
Enumeration procedure.

Protected Attributes

- int [_min](#)
- int [_max](#)

Additional Inherited Members

3.13.1 Detailed Description

The class describing a **finite** integer interval.

3.13.2 Constructor & Destructor Documentation

3.13.2.1 marmoteInterval()

```
marmoteInterval::marmoteInterval (
    int min,
    int max )
```

Constructor for an interval. By convention, if $\text{max} < \text{min}$, then the interval is empty. Otherwise, both *min* and *max* are inside the interval.

Author

Alain Jean-Marie

Parameters

<i>min</i>	the low end of the interval
<i>max</i>	the high end of the interval

3.13.2.2 ~marmoteInterval()

```
marmoteInterval::~~marmoteInterval ( )
```

Standard destructor for an interval.

Author

Alain Jean-Marie

3.13.3 Member Function Documentation

3.13.3.1 decodeState()

```
void marmoteInterval::decodeState (
    int index,
    int * buffer ) [virtual]
```

Procedure that converts an index into a state. The state is written in the buffer provided. The array must have been allocated before.

Parameters

<i>index</i>	index of the state to be decoded
<i>buffer</i>	state buffer containing the resulting state

Reimplemented from [marmoteSet](#).

3.13.3.2 firstState()

```
void marmoteInterval::firstState (
    int * buffer ) [virtual]
```

Initializes some state buffer with the first state of the set.

Parameters

<i>buffer</i>	the buffer to be set.
---------------	-----------------------

Reimplemented from [marmoteSet](#).

3.13.3.3 index()

```
int marmoteInterval::index (
    int * buffer ) [virtual]
```

Function that computes the number (index) of some state in the order of the set. The base class provides an implementation that uses solely [firstState\(\)](#), [nextState\(\)](#), and state comparison. For efficiency, it is advised to re-implement this method for derived classes.

Parameters

<i>buffer</i>	state buffer containing the state
---------------	-----------------------------------

Returns

the index of the state

Reimplemented from [marmoteSet](#).

3.13.3.4 isFinite()

```
bool marmoteInterval::isFinite ( ) [inline], [virtual]
```

Test if the set is finite. These sets always are.

Returns

true

Reimplemented from [marmoteSet](#).

3.13.3.5 isZero()

```
bool marmoteInterval::isZero (
    int * buffer ) [virtual]
```

Function that tests whether a state is the first state (state zero) or not.

Parameters

<i>buffer</i>	the state to be tested
---------------	------------------------

Returns

true if the state is zero, false otherwise.

Reimplemented from [marmoteSet](#).

3.13.3.6 nextState()

```
void marmoteInterval::nextState (
    int * buffer ) [virtual]
```

Procedure to compute the state following a given state in the set order. The new state is computed in place: the original state is erased. By convention, the state following the last state is the first state.

Parameters

<i>buffer</i>	the reference state
---------------	---------------------

Reimplemented from [marmoteSet](#).

3.13.3.7 printState()

```
void marmoteInterval::printState (
    FILE * out,
    int * buffer ) [virtual]
```

Procedure to print a state, given by its full description, to some file descriptor.

Parameters

<i>out</i>	the file descriptor to be used
<i>buffer</i>	the state to be printed

Reimplemented from [marmoteSet](#).

3.13.4 Member Data Documentation

3.13.4.1 `_max`

```
int marmoteInterval::_max [protected]
```

the higher end of the interval

3.13.4.2 `_min`

```
int marmoteInterval::_min [protected]
```

the lower end of the interval

The documentation for this class was generated from the following files:

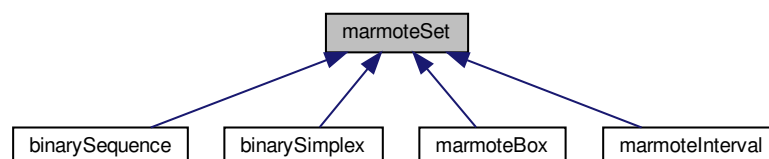
- Set/marmoteInterval.h
- Set/marmoteInterval.cpp

3.14 marmoteSet Class Reference

The mother class representing abstract sets.

```
#include <marmoteSet.h>
```

Inheritance diagram for marmoteSet:



Collaboration diagram for marmoteSet:



Public Types

- enum `opType` { **UNION**, **PRODUCT**, **SIMPLE** }
The different composition types for sets.

Public Member Functions

- `marmoteSet (marmoteSet **list, int nb, opType t)`
Constructor for composite sets, from smaller sets.
- virtual `~marmoteSet ()`
- virtual long int `cardinal ()`
Cardinal of the set.
- virtual bool `isFinite ()`
Test if the set is finite.
- bool `isSimple ()`
Test if the set is a simple, elementary set.
- bool `isUnion ()`
Test if the set is a composite, product set.
- bool `isProduct ()`
Test if the set is a composite, product set.
- int `totNbDims ()`
Read accessor to the total number of dimensions.
- virtual void `enumerate ()`
Enumerates the set: lists all elements of the set. PROBLEM: enumerate where? in a file? to be specified more precisely.
- const void `test_index_decode ()`
Tests the `index()` and `decodeState()` methods by enumerating all states. Results written to standard output.
- virtual void `firstState (int *buffer)`
Procedure that initializes the state buffer to the first state in the set.
- virtual void `nextState (int *buffer)`
Procedure to compute the state following a given state in the set order. The new state is computed in place: the original state is erased. By convention, the state following the last state is the first state.
- virtual void `decodeState (int index, int *buffer)`
Procedure that converts an index into a state. The state is written in the buffer provided. The array must have been allocated before.
- virtual int `index (int *buffer)`
Function that computes the number (index) of some state in the order of the set. The base class provides an implementation that uses solely `firstState()`, `nextState()`, and state comparison. For efficiency, it is advised to re-implement this method for derived classes.

- virtual bool `isZero` (int *buffer)
Function that tests whether a state is the first state (state zero) or not.
- virtual void `printState` (FILE *out, int *buffer)
Procedure to print a state, given by its full description, to some file descriptor.
- virtual void `printState` (FILE *out, int index)
Procedure to print a state, given by its index, to some file descriptor.
- virtual int `nextStatebyDim` (int *buffer, int dim)
Procedure to compute the state following a given state following a dimension of the set. The new state is computed in place: the original state is erased. By convention, the state following the last state is the first state.
- virtual int `firstStatebyDim` (int *buffer, int dim)
Initializes some state buffer with the first state of the dimension.
- virtual int `cardinalbyDim` (int dim)
procedure to know the size of a dimension
- virtual int `nextStateOutDim` (int *buffer, int dim)
Procedure to compute the state following a given state letting fix a dimension of the set. The new state is computed in place: the original state is erased. By convention, the state following the last state is the first state.
- virtual int `firstStateOutDim` (int *buffer, int dim)
Procedure to initialize some state buffer without modifying the current dimension.
- virtual int `cardinalOutDim` (int dim)
procedure to know the size of a set without considering one dimension

Protected Attributes

- `opType _type`
- `int _nbDimensions`
- `int _nbZones`
- `long int _cardinal`
- `marmoteSet ** _zone`
- `marmoteSet ** _dimension`
- `int * _stateBuffer`
- `int * _dimOffset`
- `int * _idxOffset`
- `int _totNbDims`
- `int * _zeroState`

3.14.1 Detailed Description

The mother class representing abstract sets.

3.14.2 Member Enumeration Documentation

3.14.2.1 opType

```
enum marmoteSet::opType
```

The different composition types for sets.

- SIMPLE: the "atomic" typedef
- PRODUCT: a cartesian product of sets
- UNION: a disjoint union of subsets

3.14.3 Constructor & Destructor Documentation

3.14.3.1 marmoteSet()

```
marmoteSet::marmoteSet (
    marmoteSet ** list,
    int nb,
    opType t )
```

Constructor for composite sets, from smaller sets.

Parameters

<i>list</i>	the list of elements in the composite
<i>nb</i>	the number of elements in the composite
<i>t</i>	the type of composite: UNION or PRODUCT

3.14.3.2 ~marmoteSet()

```
marmoteSet::~~marmoteSet ( ) [virtual]
```

Destructor for general sets. Essentially for composite structures.

3.14.4 Member Function Documentation

3.14.4.1 cardinal()

```
long int marmoteSet::cardinal ( ) [virtual]
```

Cardinal of the set.

Returns

an integer, the cardinal of the set, or INFINITE_STATE_SPACE_SIZE

3.14.4.2 cardinalbyDim()

```
int marmoteSet::cardinalbyDim (
    int dim ) [virtual]
```

procedure to know the size of a dimension

Parameters

<i>dim</i>	the dimension
------------	---------------

Returns

the size

Reimplemented in [marmoteBox](#).

3.14.4.3 cardinalOutDim()

```
int marmoteSet::cardinalOutDim (
    int dim ) [virtual]
```

procedure to know the size of a set without considering one dimension

Parameters

<i>dim</i>	the dimension to not consider
------------	-------------------------------

Returns

the size

Reimplemented in [marmoteBox](#).

3.14.4.4 decodeState()

```
void marmoteSet::decodeState (
    int index,
    int * buffer ) [virtual]
```

Procedure that converts an index into a state. The state is written in the buffer provided. The array must have been allocated before.

Parameters

<i>index</i>	index of the state to be decoded
<i>buffer</i>	state buffer containing the resulting state

Reimplemented in [marmoteBox](#), [marmoteInterval](#), [binarySimplex](#), and [binarySequence](#).

3.14.4.5 firstState()

```
void marmoteSet::firstState (
    int * buffer ) [virtual]
```

Procedure that initializes the state buffer to the first state in the set.

Parameters

<i>buffer</i>	the buffer to be initialized. Must have been allocated before.
---------------	----------------------------------------------------------------

Reimplemented in [marmoteInterval](#), [marmoteBox](#), [binarySimplex](#), and [binarySequence](#).

3.14.4.6 firstStatebyDim()

```
int marmoteSet::firstStatebyDim (
    int * buffer,
    int dim ) [virtual]
```

Initializes some state buffer with the first state of the dimension.

Parameters

<i>buffer</i>	the buffer to be set.
<i>dim</i>	the dimension

Returns

the index of the state

Reimplemented in [marmoteBox](#).

3.14.4.7 firstStateOutDim()

```
int marmoteSet::firstStateOutDim (
    int * buffer,
    int dim ) [virtual]
```

Procedure to initialize some state buffer without modifying the current dimension.

Parameters

<i>buffer</i>	the buffer to be set.
<i>dim</i>	the dimension keeping unchnaged

Returns

the index of the state

Reimplemented in [marmoteBox](#).

3.14.4.8 index()

```
int marmoteSet::index (
    int * buffer ) [virtual]
```

Function that computes the number (index) of some state in the order of the set. The base class provides an implementation that uses solely [firstState\(\)](#), [nextState\(\)](#), and state comparison. For efficiency, it is advised to re-implement this method for derived classes.

Parameters

<i>buffer</i>	state buffer containing the state
---------------	-----------------------------------

Returns

the index of the state

Reimplemented in [marmoteBox](#), [marmoteInterval](#), [binarySimplex](#), and [binarySequence](#).

3.14.4.9 isFinite()

```
bool marmoteSet::isFinite ( ) [virtual]
```

Test if the set is finite.

Returns

true if the set is finite, false if not.

Reimplemented in [marmoteInterval](#), [marmoteBox](#), [binarySimplex](#), and [binarySequence](#).

3.14.4.10 isProduct()

```
bool marmoteSet::isProduct ( ) [inline]
```

Test if the set is a composite, product set.

Returns

true if the set is a product, false otherwise.

3.14.4.11 isSimple()

```
bool marmoteSet::isSimple ( ) [inline]
```

Test if the set is a simple, elementary set.

Returns

true if the set is elementary, false otherwise.

3.14.4.12 isUnion()

```
bool marmoteSet::isUnion ( ) [inline]
```

Test if the set is a composite, product set.

Returns

true if the set is a product, false otherwise.

3.14.4.13 isZero()

```
bool marmoteSet::isZero (
    int * buffer ) [virtual]
```

Function that tests whether a state is the first state (state zero) or not.

Parameters

<i>buffer</i>	the state to be tested
---------------	------------------------

Returns

true if the state is zero, false otherwise.

Reimplemented in [marmoteInterval](#), [marmoteBox](#), [binarySimplex](#), and [binarySequence](#).

3.14.4.14 nextState()

```
void marmoteSet::nextState (
    int * buffer ) [virtual]
```

Procedure to compute the state following a given state in the set order. The new state is computed in place: the original state is erased. By convention, the state following the last state is the first state.

Parameters

<i>buffer</i>	the reference state
---------------	---------------------

Reimplemented in [marmoteInterval](#), [marmoteBox](#), [binarySimplex](#), and [binarySequence](#).

3.14.4.15 nextStatebyDim()

```
int marmoteSet::nextStatebyDim (  
    int * buffer,  
    int dim ) [virtual]
```

Procedure to compute the state following a given state following a dimension of the set. The new state is computed in place: the original state is erased. By convention, the state following the last state is the first state.

Parameters

<i>buffer</i>	the reference state
<i>dim</i>	the dimension

Returns

the index of the state

Reimplemented in [marmoteBox](#).

3.14.4.16 nextStateOutDim()

```
int marmoteSet::nextStateOutDim (  
    int * buffer,  
    int dim ) [virtual]
```

Procedure to compute the state following a given state letting fix a dimension of the set. The new state is computed in place: the original state is erased. By convention, the state following the last state is the first state.

Parameters

<i>buffer</i>	the reference state
<i>dim</i>	the dimension

Returns

the index of the state

Reimplemented in [marmoteBox](#).

3.14.4.17 printState() [1/2]

```
void marmoteSet::printState (
    FILE * out,
    int * buffer ) [virtual]
```

Procedure to print a state, given by its full description, to some file descriptor.

Parameters

<i>out</i>	the file descriptor to be used
<i>buffer</i>	the state to be printed

Reimplemented in [marmoteBox](#), [marmoteInterval](#), [binarySimplex](#), and [binarySequence](#).

3.14.4.18 printState() [2/2]

```
void marmoteSet::printState (
    FILE * out,
    int index ) [virtual]
```

Procedure to print a state, given by its index, to some file descriptor.

Parameters

<i>out</i>	the file descriptor to be used
<i>index</i>	index of the state to be printed

3.14.4.19 totNbDims()

```
int marmoteSet::totNbDims ( ) [inline]
```

Read accessor to the total number of dimensions.

Returns

The total number of dimensions of a vector representation.

3.14.5 Member Data Documentation

3.14.5.1 `_cardinal`

```
long int marmoteSet::_cardinal [protected]
```

cardinal of the set

3.14.5.2 `_dimension`

```
marmoteSet** marmoteSet::_dimension [protected]
```

array of dimensions, in case of a product

3.14.5.3 `_dimOffset`

```
int* marmoteSet::_dimOffset [protected]
```

table used for products to keep track of nested dimensions

3.14.5.4 `_idxOffset`

```
int* marmoteSet::_idxOffset [protected]
```

table used for products to compute index offsets in dimensions

3.14.5.5 `_nbDimensions`

```
int marmoteSet::_nbDimensions [protected]
```

number of dimensions if this is a product

3.14.5.6 `_nbZones`

```
int marmoteSet::_nbZones [protected]
```

number of subsets if this is a union

3.14.5.7 `_stateBuffer`

```
int* marmoteSet::_stateBuffer [protected]
```

a buffer to store the representation of a state as a vector of integers

3.14.5.8 `_totNbDims`

```
int marmoteSet::_totNbDims [protected]
```

total number of dimensions of the vector representation

3.14.5.9 `_type`

```
opType marmoteSet::_type [protected]
```

type of the composition

3.14.5.10 `_zeroState`

```
int* marmoteSet::_zeroState [protected]
```

the representation of the initial state (static)

3.14.5.11 `_zone`

```
marmoteSet** marmoteSet::_zone [protected]
```

array of zones, in case of a union

The documentation for this class was generated from the following files:

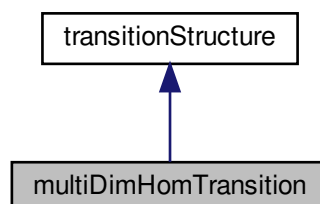
- Set/marmoteSet.h
- Set/marmoteSet.cpp

3.15 multiDimHomTransition Class Reference

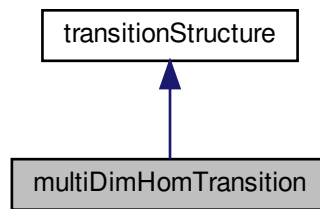
Class for multidimensional, homogeneous random walk transition structures. These are characterized by.

```
#include <multiDimHomTransition.h>
```

Inheritance diagram for multiDimHomTransition:



Collaboration diagram for multiDimHomTransition:



Public Member Functions

- `multiDimHomTransition` (int nbDims, int *dimSize, double *p, double *q)
General constructor of the class.
- `~multiDimHomTransition` ()
Destructor of the class.
- int dimSize (int d)
Read accessor for the size of the state space in each dimension.
- double p (int d)
Read accessor for the jump probability to the right in each dimension.
- double q (int d)
Read accessor for the jump probability to the right in each dimension.
- bool setEntry (int i, int j, double val)
Method to set the value associated with some transition. Not applicable here.
- double getEntry (int i, int j)
Method to get the value associated with some transition.
- int getNbElts (int i)
Method to get the number of non-zero entries in a transition from some state. This can be seen as the number of actually possible transitions from that state.
- int getCol (int i, int k)
*Method to get the **number** of the state corresponding to transition number k in the list of possible transitions from some state i.*
- double getEntryByCol (int i, int k)
*Method to get the **value** attached to transition number k in the list of possible transitions from some state i.*
- `discreteDistribution * getTransDistrib` (int i)
Method to get the transition from some state as a probability distribution.
- double rowSum (int i)
Sum of entries on some row i. Always 1.0 since this is a discrete-time transition structure.
- `multiDimHomTransition * copy` ()
Copying a transition structure.
- `multiDimHomTransition * uniformize` ()
Uniformizing a transition structure. Since the origin structure is already of discrete-time type, a copy is returned.
- `multiDimHomTransition * embed` ()
Embedding in the transition structure. Since the origin structure is already of discrete-time type, a copy is returned.
- void evaluateMeasure (double *d, double *res)

Computing the action of the transition structure on some measure, the measure being represented as a vector of real numbers. This corresponds to the multiplication vector/matrix, the row vector being interpreted as a signed measure. The result is placed in an array that must have been previously allocated.

- void [evaluateValue](#) (double *v, double *res)

Computing the action of the transition structure on some vector of values. This corresponds to the multiplication matrix/vector, the column vector being interpreted as a vector of values attached to the states.

- void [write](#) (FILE *out, string format)

Output method for the transition structure. Supported formats are: XBORNE, MARCA, Ers, Maple.

- [discreteDistribution](#) * [evaluateMeasure](#) ([discreteDistribution](#) *d)

Computing the action of the transition structure on some probability distribution. This version with [discreteDistribution](#) objects uses [evaluateMeasure\(double,double*\)](#).*

- [discreteDistribution](#) * [getJumpDistribution](#) ()

*Getting a discrete distribution representing the generic jumps. The distribution has $2*nbDims + 1$ values: 0, +/- 1, ..., +/- nDims. The coding is:*

Protected Attributes

- int [_nbDims](#)
the number of dimensions
- int * [_dimSize](#)
size of the state space in each dimension
- double * [_p](#)
probas to jump to the right in each dimension
- double * [_q](#)
probas to jump to the left in each dimension

Additional Inherited Members

3.15.1 Detailed Description

Class for multidimensional, homogeneous random walk transition structures. These are characterized by.

- their dimension d
- d sizes for each dimension (possibly infinite)
- d transitions probabilities to the right in each dimension, p_i ,
- d transition probabilities to the left in each dimension, q_i . The probability to stay in the current location is $1 - \sum_i (p_i + q_i)$. The boundaries are absorbing: when a transition goes out of bounds, it is assumed to stay on the boundary.

Author

Alain Jean-Marie

3.15.2 Constructor & Destructor Documentation

3.15.2.1 multiDimHomTransition()

```
multiDimHomTransition::multiDimHomTransition (
    int nbDims,
    int * dimSize,
    double * p,
    double * q )
```

General constructor of the class.

Parameters

<i>nbDims</i>	the number of dimensions
<i>dimSize</i>	the array of sizes
<i>p</i>	the array of jump probabilities to the right
<i>q</i>	the array of jump probabilities to the left

3.15.2.2 `~multiDimHomTransition()`

```
multiDimHomTransition::~~multiDimHomTransition ( )
```

Destructor of the class.

Destructor.

3.15.3 Member Function Documentation**3.15.3.1 `copy()`**

```
multiDimHomTransition * multiDimHomTransition::copy ( ) [virtual]
```

Copying a transition structure.

Returns

[transitionStructure](#)

Implements [transitionStructure](#).

3.15.3.2 `dimSize()`

```
int multiDimHomTransition::dimSize (
    int d ) [inline]
```

Read accessor for the size of the state space in each dimension.

Parameters

<i>d</i>	the dimension
----------	---------------

Returns

the size of the state space in dimension d

3.15.3.3 embed()

```
multiDimHomTransition * multiDimHomTransition::embed ( ) [virtual]
```

Embedding in the transition structure. Since the origin structure is already of discrete-time type, a copy is returned.

Returns

a discrete-time transition structure

Implements [transitionStructure](#).

3.15.3.4 evaluateMeasure() [1/2]

```
void multiDimHomTransition::evaluateMeasure (
    double * pi,
    double * res ) [virtual]
```

Computing the action of the transition structure on some measure, the measure being represented as a vector of real numbers. This corresponds to the multiplication vector/matrix, the row vector being interpreted as a signed measure. The result is placed in an array that must have been previously allocated.

Parameters

<i>pi</i>	the measure to evaluate
<i>res</i>	the resulting measure

Reimplemented from [transitionStructure](#).

3.15.3.5 evaluateMeasure() [2/2]

```
discreteDistribution * multiDimHomTransition::evaluateMeasure (
    discreteDistribution * d )
```

Computing the action of the transition structure on some probability distribution. This version with [discrete↔Distribution](#) objects uses [evaluateMeasure\(double*,double*\)](#).

Parameters

<i>d</i>	the distribution to evaluate
----------	------------------------------

Returns

the distribution resulting from the evaluation

3.15.3.6 evaluateValue()

```
void multiDimHomTransition::evaluateValue (
    double * v,
    double * res ) [virtual]
```

Computing the action of the transition structure on some vector of values. This corresponds to the multiplication matrix/vector, the column vector being interpreted as a vector of values attached to the states.

Parameters

<i>v</i>	the vector of values to evaluate
<i>res</i>	the resulting vector

Implements [transitionStructure](#).

3.15.3.7 getCol()

```
int multiDimHomTransition::getCol (
    int i,
    int k ) [virtual]
```

Method to get the **number** of the state corresponding to transition number k in the list of possible transitions from some state i.

Parameters

<i>i</i>	the origin state
<i>k</i>	the index of transition from state i

Returns

the destination state corresponding to the k-th possible transition from state i

Implements [transitionStructure](#).

3.15.3.8 getEntry()

```
double multiDimHomTransition::getEntry (
    int i,
    int j ) [virtual]
```

Method to get the value associated with some transition.

Parameters

<i>i</i>	the origin state
<i>j</i>	the destination state

Returns

the value attached to the transition (i,j)

Implements [transitionStructure](#).

3.15.3.9 getEntryByCol()

```
double multiDimHomTransition::getEntryByCol (
    int i,
    int k ) [virtual]
```

Method to get the **value** attached to transition number k in the list of possible transitions from some state i.

Parameters

<i>i</i>	the origin state
<i>k</i>	the index of the transition from state i

Returns

the value attached to the k-th possible transition from state i

Implements [transitionStructure](#).

3.15.3.10 getJumpDistribution()

```
discreteDistribution * multiDimHomTransition::getJumpDistribution ( )
```

Getting a discrete distribution representing the generic jumps. The distribution has $2 \cdot \text{nbDims} + 1$ values: 0, +/- 1, ..., +/- nDims. The coding is:

- 0 codes the self jump
- +d codes a jump upwards in dimension (d-1).
- -d codes a jump downwards in dimension (d-1).

Returns

a discrete distribution object

3.15.3.11 getNbElts()

```
int multiDimHomTransition::getNbElts (
    int i ) [virtual]
```

Method to get the number of non-zero entries in a transition from some state. This can be seen as the number of actually possible transitions from that state.

Parameters

<i>i</i>	the origin state
----------	------------------

Returns

int the number of possible transitions

Implements [transitionStructure](#).

3.15.3.12 getTransDistrib()

```
discreteDistribution * multiDimHomTransition::getTransDistrib (
    int i ) [virtual]
```

Method to get the transition from some state as a probability distribution.

Parameters

<i>i</i>	the origin state
----------	------------------

Returns

a discrete distribution object

Implements [transitionStructure](#).

3.15.3.13 p()

```
double multiDimHomTransition::p (
    int d ) [inline]
```

Read accessor for the jump probability to the right in each dimension.

Parameters

<i>d</i>	the dimension
----------	---------------

Returns

the jump probability

3.15.3.14 q()

```
double multiDimHomTransition::q (
    int d ) [inline]
```

Read accessor for the jump probability to the right in each dimension.

Parameters

<i>d</i>	the dimension
----------	---------------

Returns

the jump probability

3.15.3.15 rowSum()

```
double multiDimHomTransition::rowSum (
    int i ) [virtual]
```

Sum of entries on some row i. Always 1.0 since this is a discrete-time transition structure.

Parameters

<i>i</i>	the row to be summed
----------	----------------------

Returns

1.0

Implements [transitionStructure](#).

3.15.3.16 setEntry()

```
bool multiDimHomTransition::setEntry (
    int i,
    int j,
    double val ) [virtual]
```

Method to set the value associated with some transition. Not applicable here.

Parameters

<i>i</i>	the origin state
<i>j</i>	the destination state
<i>val</i>	the value attached to the transition

Returns

true if the operation was successful, false otherwise (out of bounds; wrong numeric value)

Implements [transitionStructure](#).

3.15.3.17 uniformize()

```
multiDimHomTransition * multiDimHomTransition::uniformize ( ) [virtual]
```

Uniformizing a transition structure. Since the origin structure is already of discrete-time type, a copy is returned.

Returns

a discrete-time transition structure

Implements [transitionStructure](#).

3.15.3.18 write()

```
void multiDimHomTransition::write (
    FILE * out,
    string format )
```

Output method for the transition structure. Supported formats are: XBORNE, MARCA, Ers, Maple.

Parameters

<i>out</i>	the file descriptor to which the structure should be written.
<i>format</i>	the format/language to be used

The documentation for this class was generated from the following files:

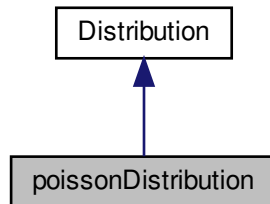
- transitionStructure/multiDimHomTransition.h
- transitionStructure/multiDimHomTransition.cpp

3.16 poissonDistribution Class Reference

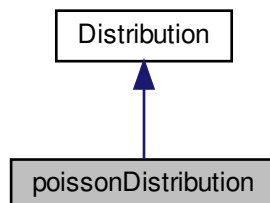
The Poisson distribution. The parameter is called "lambda". The Poisson distribution is discrete but does not inherit from [discreteDistribution](#) because its range is infinite.

```
#include <poissonDistribution.h>
```

Inheritance diagram for poissonDistribution:



Collaboration diagram for poissonDistribution:



Public Member Functions

- [poissonDistribution](#) (double [lambda](#))
Constructor for a Poisson distribution from its "lambda" parameter. The mean is calculated at creation.
- [~poissonDistribution](#) ()
Destructor for a Poisson distribution.
- double [getProba](#) (double k)
Function to obtain the probability of a specific value k.
- double [lambda](#) ()
Function to obtain the parameter (or ratio) of the distribution. Redundant with p() but defined to be more explicit.
- double [mean](#) ()
computing the mathematical expectation or mean
- double [rate](#) ()

- computing the "rate", defined as the inverse of the mean*

 - double `moment` (int order)

Computing the moments of the distribution.
- double `variance` ()

Computing the variance of the random variable: the second moment minus the square of the first moment. Variance is the square of the coefficient of variation. The `Distribution` class offers a default implementation.
- double `laplace` (double s)

computing the Laplace transform of the distribution at real point
- double `dLaplace` (double s)

computing the derivative of the Laplace transform at real points
- double `cdf` (double x)

computing the cumulative distribution function at some real point x. This is the probability that the random variable is less or equal to x.
- double `ccdf` (double x)

computing the complementary cumulative distributon function (or tail) at some real point x. This is the probability that the random variable is strictly larger than x. The `Distribution` class offers a default implementation.
- bool `hasMoment` (int order)

test for the existence of moments of any order
- `poissonDistribution` * `rescale` (double factor)

*rescaling a distribution by some real factor. Not all distributions allow this for any real factor. If the operation fails, or if the factor is 1.0, a copy of the distribution should be returned (**not** by using the `copy()` function).*
- `poissonDistribution` * `copy` ()

copying a distribution. Typically implemented as `rescale(1.0)`.
- double `sample` ()

Sample from the Poisson distribution. Uses the "R" package.
- void `iidSample` (int n, double *s)

drawing an i.i.d. sample from the distribution. The result is returned in an array (that must have been already allocated) passed as a parameter. The `Distribution` class offers the default implementation with repeated call to `sample()`.
- std::string `toString` ()

an utility to convert the distribution into a string.
- void `write` (FILE *out, int mode)

an utility to write the distribution to some file, according to some format.

Additional Inherited Members

3.16.1 Detailed Description

The Poisson distribution. The parameter is called "lambda". The Poisson distribution is discrete but does not inherit from `discreteDistribution` because its range is infinite.

Author

Alain Jean-Marie

3.16.2 Constructor & Destructor Documentation

3.16.2.1 poissonDistribution()

```
poissonDistribution::poissonDistribution (
    double lambda )
```

Constructor for a Poisson distribution from its "lambda" parameter. The mean is calculated at creation.

Author

Alain Jean-Marie

Parameters

<i>lambda</i>	a non-negative real number
---------------	----------------------------

Returns

an object of type [poissonDistribution](#)

3.16.2.2 ~poissonDistribution()

```
poissonDistribution::~~poissonDistribution ( )
```

Destructor for a Poisson distribution.

Author

Alain Jean-Marie

3.16.3 Member Function Documentation

3.16.3.1 ccdf()

```
double poissonDistribution::ccdf (
    double x )
```

computing the complementary cumulative distributon function (or tail) at some real point x. This is the probability that the random variable is strictly larger than x. The [Distribution](#) class offers a default implementation.

Parameters

<i>x</i>	the value at which to compute the ccdf
----------	----------------------------------------

Returns

the value of the ccdf

3.16.3.2 cdf()

```
double poissonDistribution::cdf (  
    double x ) [virtual]
```

computing the cumulative distribution function at some real point x. This is the probability that the random variable is less or equal to x.

Parameters

x	the value at which to compute the cdf
---	---------------------------------------

Returns

the value of the cdf

Computation of the cumulative density function at some real point x

Author

Alain Jean-Marie

Parameters

x	value at which the CDF is computed
---	------------------------------------

Returns

CDF(x)

Implements [Distribution](#).

3.16.3.3 copy()

```
poissonDistribution * poissonDistribution::copy ( ) [virtual]
```

copying a distribution. Typically implemented as rescale(1.0).

See also

[rescale\(double\)](#)

Returns

a copy of the distribution

Copying the law

Author

Alain Jean-Marie

Returns

a copy of the law

Implements [Distribution](#).

3.16.3.4 dLaplace()

```
double poissonDistribution::dLaplace (  
    double s ) [virtual]
```

computing the derivative of the Laplace transform at real points

Parameters

s	the value at which to compute
---	-------------------------------

Returns

the value of the derivative of the Laplace transform

Computation of the derivative of the Laplace transform at some real point s

Author

Alain Jean-Marie

Parameters

s	value at which the derivative is computed
---	-------------------------------------------

Returns

d LT(s)/ds

Implements [Distribution](#).

3.16.3.5 getProba()

```
double poissonDistribution::getProba (
    double k )
```

Function to obtain the probability of a specific value k.

Parameters

<i>k</i>	the value at which the probability should be computed
----------	-------------------------------------------------------

Returns

the probability that the random variable is k

Computation of the probability of some value

Author

Alain Jean-Marie

Parameters

<i>k</i>	value at which the probability is computed
----------	--------------------------------------------

Returns

$P(X = k)$

3.16.3.6 hasMoment()

```
bool poissonDistribution::hasMoment (
    int order ) [virtual]
```

test for the existence of moments of any order

Parameters

<i>order</i>	the order of the moment to be tested
--------------	--------------------------------------

Returns

true if the moment exists, false otherwise

Implements [Distribution](#).

3.16.3.7 iidSample()

```
void poissonDistribution::iidSample (
    int n,
    double * s )
```

drawing an i.i.d. sample from the distribution. The result is returned in an array (that must have been already allocated) passed as a parameter. The [Distribution](#) class offers the default implementation with repeated call to [sample\(\)](#).

Parameters

<i>n</i>	the number of values to sample
<i>s</i>	an array to be filled with the sample

3.16.3.8 lambda()

```
double poissonDistribution::lambda ( ) [inline]
```

Function to obtain the parameter (or ratio) of the distribution. Redundant with [p\(\)](#) but defined to be more explicit.

Returns

the value of the ratio

3.16.3.9 laplace()

```
double poissonDistribution::laplace (
    double s ) [virtual]
```

computing the Laplace transform of the distribution at real point

Parameters

<i>s</i>	the value at which to compute
----------	-------------------------------

Returns

the value of the Laplace transform

Computation of the Laplace transform at some real point *s*

Author

Alain Jean-Marie

Parameters

<i>s</i>	value at which the derivative is computed
----------	-------------------------------------------

Returns

LT(s)

Implements [Distribution](#).

3.16.3.10 mean()

```
double poissonDistribution::mean ( ) [virtual]
```

computing the mathematical expectation or mean

Returns

the mean

Calculation of the mean. Returns the value since it is pre-computed

Author

Alain Jean-Marie

Returns

the mathematical expectation of the distribution

Implements [Distribution](#).

3.16.3.11 moment()

```
double poissonDistribution::moment (
    int order ) [virtual]
```

Computing the moments of the distribution.

Parameters

<i>order</i>	the order of the moment to be computed
--------------	----------------------------------------

Returns

the moment

Implements [Distribution](#).

3.16.3.12 rate()

```
double poissonDistribution::rate ( ) [virtual]
```

computing the "rate", defined as the inverse of the mean

Returns

the rate

Calculation of the rate, which is the inverse of the mean

Author

Alain Jean-Marie

Returns

the rate

Implements [Distribution](#).

3.16.3.13 rescale()

```
poissonDistribution * poissonDistribution::rescale (
    double factor ) [virtual]
```

rescaling a distribution by some real factor. Not all distributions allow this for any real factor. If the operation fails, or if the factor is 1.0, a copy of the distribution should be returned (**not** by using the [copy\(\)](#) function).

See also

[copy\(\)](#)

Parameters

<i>factor</i>	the real factor to be used
---------------	----------------------------

Returns

a new distribution object

Rescaling the law X by some real factor f

Author

Alain Jean-Marie

Parameters

<i>factor</i>	the factor f
---------------	----------------

Returns

the law $f \cdot X$

Implements [Distribution](#).

3.16.3.14 toString()

```
std::string poissonDistribution::toString ( ) [virtual]
```

an utility to convert the distribution into a string.

Returns

the string representing the distribution.

Printing a representation of the law into a string

Author

Alain Jean-Marie

Returns

a string

Implements [Distribution](#).

3.16.3.15 variance()

```
double poissonDistribution::variance ( )
```

Computing the variance of the random variable: the second moment minus the square of the first moment. Variance is the square of the coefficient of variation. The [Distribution](#) class offers a default implementation.

Returns

the variance

3.16.3.16 write()

```
void poissonDistribution::write (
    FILE * out,
    int mode ) [virtual]
```

an utility to write the distribution to some file, according to some format.

Parameters

<i>out</i>	the file descriptor of the file
<i>mode</i>	a code for the format to be used

Printing a representation of the law

Author

Alain Jean-Marie

Parameters

<i>out</i>	the output stream
<i>mode</i>	representation for the output

Implements [Distribution](#).

The documentation for this class was generated from the following files:

- Distribution/poissonDistribution.h
- Distribution/poissonDistribution.cpp

3.17 SCC Struct Reference

Public Member Functions

- bool **operator**< (const [SCC](#) &a)

Public Attributes

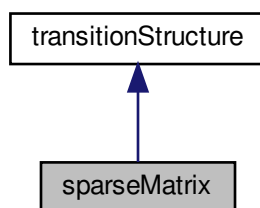
- int **id**
- int **period**
- std::set< int > **states**

The documentation for this struct was generated from the following file:

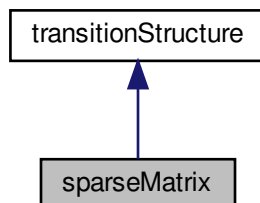
- transitionStructure/sparseMatrix.h

3.18 sparseMatrix Class Reference

Inheritance diagram for sparseMatrix:



Collaboration diagram for sparseMatrix:



Public Member Functions

- [sparseMatrix](#) (int size)
Standard constructor for sparse matrices. The internal structures are initialized. The result is the null matrix.
- [sparseMatrix](#) (int rowSize, int colSize)
Constructor for non-square sparse matrices. The internal structures are initialized. The result is the null matrix.
- [~sparseMatrix](#) ()
Destructor of the class.
- bool [setEntry](#) (int row, int col, double val)
Inserting an element in the matrix. IMPORTANT NOTE: there is no check for the existence of the column This is because in the context of Markov Reward Processes, there may be several transitions with different reward. It is simpler to handle this by allowing replicated entries in the neighborhood lists. The multiplication algorithm handles this in a natural way. BUT the [getEntry\(int,int\)](#) method must handle this situation carefully.
- double [getEntry](#) (int, int)
Retrieving an entry from the matrix. Takes into account the fact that columns may be replicated: the different values are then added.
- int [getNbElts](#) (int row)
Retrieving a the number of elements on some row.
- int [getCol](#) (int row, int numCol)
Retrieving an low-level entry from the matrix.
- double [getEntryByCol](#) (int row, int numCol)
Retrieving an low-level entry from the matrix.
- [discreteDistribution](#) * [getTransDistrib](#) (int row)
Retrieving the transitions from some state as a discrete distribution.
- double [rowSum](#) (int row)
Summing a row.
- void [evaluateMeasure](#) (double *m, double *res)
Sparse vector/matrix multiplication. The result is stored in an array that must be already allocated. Anciently the "multVM" method.
- [discreteDistribution](#) * [evaluateMeasure](#) ([discreteDistribution](#) *d)
- void [evaluateValue](#) (double *v, double *res)
Sparse matrix/vector multiplication. The result is stored in an array that must be already allocated. Anciently the "multMV" method.
- double [evaluateValueState](#) (double *v, int stateIndex)
Sparse matrix/vector multiplication, for a specific state/line of the matrix.
- [sparseMatrix](#) * [copy](#) ()
Sparse matrix copy. This is a straightforward copy. No optimization is performed, viz replicated columns.
- [sparseMatrix](#) * [uniformize](#) ()
Sparse matrix uniformization. Returns a sparse matrix and sets the uniformization factor. If the type is already discrete, returns a copy.
- [sparseMatrix](#) * [embed](#) ()
Sparse matrix embedding. Returns a sparse matrix. If the type is already discrete, returns a copy.
- void [diagnose](#) (FILE *out)
Performs various diagnostics on the transition structure (only written for [sparseMatrix](#) objects at the moment)
- void [write](#) (FILE *out, std::string format)
Writing a sparse matrix to a file. Several formats could be chosen. Supported formats are: "Ers", "Full", "Matrix↔Market-sparse", "MatrixMarket-full", "Maple", "MARCA", "R", "scilab", "XBORNE".
- bool [addToEntry](#) (int row, int col, double val)
Adding a value to an element to the matrix.
- void [normalize](#) ()
Method to normalize transition probabilities/rates across the matrix. The normalized form has strictly increasing column indices for each row.
- std::pair< std::vector< [SCC](#) > *, [sparseMatrix](#) * > [getStronglyConnectedComponents](#) (double ignore=0.0)
Method for computing strongly connected components of the Markov chain which utilize Divide-and-Conquer algorithm.

Friends

- class [markovChain](#)

Additional Inherited Members

3.18.1 Constructor & Destructor Documentation

3.18.1.1 `sparseMatrix()` [1/2]

```
sparseMatrix::sparseMatrix (  
    int size )
```

Standard constructor for sparse matrices. The internal structures are initialized. The result is the null matrix.

Parameters

<i>size</i>	the number of states/rows and columns in the structure
-------------	--------------------------------------------------------

Returns

a sparse matrix object

Standard constructor for square sparse matrices. The internal structures are initialized. Type is set to UNKNOWN. The result is the null matrix.

3.18.1.2 `sparseMatrix()` [2/2]

```
sparseMatrix::sparseMatrix (  
    int rowSize,  
    int colSize )
```

Constructor for non-square sparse matrices. The internal structures are initialized. The result is the null matrix.

Parameters

<i>rowSize</i>	the number of origin states/rows in the structure
<i>colSize</i>	the number of destination states/columns in the structure

Returns

a sparse matrix object

3.18.1.3 ~sparseMatrix()

```
sparseMatrix::~sparseMatrix ( )
```

Destructor of the class.

Standard destructor.

Author

Alain Jean-Marie

3.18.2 Member Function Documentation

3.18.2.1 addToEntry()

```
bool sparseMatrix::addToEntry (
    int row,
    int col,
    double val )
```

Adding a value to an element to the matrix.

Author

Alain Jean-Marie

Parameters

<i>row</i>	number of the row
<i>col</i>	number of the column
<i>val</i>	value to be added

Returns

true if successful,

See also

[setEntry\(\)](#)

3.18.2.2 copy()

```
sparseMatrix * sparseMatrix::copy ( ) [virtual]
```

Sparse matrix copy. This is a straightforward copy. No optimization is performed, viz replicated columns.

Author

Alain Jean-Marie

Returns

a copy of the generator

Implements [transitionStructure](#).

3.18.2.3 diagnose()

```
void sparseMatrix::diagnose (
    FILE * out )
```

Performs various diagnostics on the transition structure (only written for [sparseMatrix](#) objects at the moment)

Author

Alain Jean-Marie

Parameters

<i>out</i>	the file descriptor to which the diagnostic is written
------------	--------------------------------------------------------

3.18.2.4 embed()

```
sparseMatrix * sparseMatrix::embed ( ) [virtual]
```

Sparse matrix embedding. Returns a sparse matrix. If the type is already discrete, returns a copy.

Author

Alain Jean-Marie

Returns

the embedded version of the generator

Implements [transitionStructure](#).

3.18.2.5 evaluateMeasure() [1/2]

```
void sparseMatrix::evaluateMeasure (
    double * m,
    double * res ) [virtual]
```

Sparse vector/matrix multiplication. The result is stored in an array that must be already allocated. Anciently the "multVM" method.

Author

Alain Jean-Marie

Parameters

<i>m</i>	the measure (row vector) to be multiplied
<i>res</i>	the resulting vector

Reimplemented from [transitionStructure](#).

3.18.2.6 evaluateMeasure() [2/2]

```
discreteDistribution * sparseMatrix::evaluateMeasure (
    discreteDistribution * d )
```

Version of the vector/matrix multiplication that acts on [discreteDistribution](#) objects.

Author

Alain Jean-Marie

Parameters

<i>d</i>	the distribution to be multiplied
----------	-----------------------------------

3.18.2.7 evaluateValue()

```
void sparseMatrix::evaluateValue (
    double * v,
    double * res ) [virtual]
```

Sparse matrix/vector multiplication. The result is stored in an array that must be already allocated. Anciently the "multMV" method.

Author

Alain Jean-Marie

Parameters

<i>v</i>	the value (column vector) to be multiplied
<i>res</i>	the resulting vector

Implements [transitionStructure](#).

3.18.2.8 evaluateValueState()

```
double sparseMatrix::evaluateValueState (
    double * v,
    int stateIndex )
```

Sparse matrix/vector multiplication, for a specific state/line of the matrix.

Author

Emmanuel Hyon and Abood Mourad

Parameters

<i>v</i>	the value (column vector) to be multiplied
<i>stateIndex</i>	the index of the state for which to perform the multiplication

Returns

the value of the state

3.18.2.9 getCol()

```
int sparseMatrix::getCol (
    int row,
    int numCol ) [virtual]
```

Retrieving an low-level entry from the matrix.

Author

Alain Jean-Marie

Parameters

<i>row</i>	the number of the row
<i>numCol</i>	the number of the entry in the sparse structure

Implements [transitionStructure](#).

3.18.2.10 getEntry()

```
double sparseMatrix::getEntry (
    int row,
    int col ) [virtual]
```

Retrieving an entry from the matrix. Takes into account the fact that columns may be replicated: the different values are then added.

See also

[setEntry\(\)](#)

Author

Alain Jean-Marie

Parameters

<i>row</i>	
<i>col</i>	

Implements [transitionStructure](#).

3.18.2.11 getEntryByCol()

```
double sparseMatrix::getEntryByCol (
    int row,
    int numCol ) [virtual]
```

Retrieving an low-level entry from the matrix.

Author

Alain Jean-Marie

Parameters

<i>row</i>	the number of the row
<i>numCol</i>	the number of the entry in the sparse structure

Implements [transitionStructure](#).

3.18.2.12 getNbElts()

```
int sparseMatrix::getNbElts (
    int row ) [virtual]
```

Retrieving a the number of elements on some row.

Author

Alain Jean-Marie

Parameters

<i>row</i>	the number of the row
------------	-----------------------

Returns

the number of columns

Implements [transitionStructure](#).

3.18.2.13 getStronglyConnectedComponents()

```
std::pair<std::vector<SCC> *, sparseMatrix *> sparseMatrix::getStronglyConnectedComponents (
    double ignore = 0.0 )
```

Method for computing strongly connected components of the Markov chain which utilize Divide-and-Conquer algorithm.

Author

Hlib Mykhailenko

3.18.2.14 getTransDistrib()

```
discreteDistribution * sparseMatrix::getTransDistrib (
    int row ) [virtual]
```

Retrieving the transitions from some state as a discrete distribution.

Author

Alain Jean-Marie

Parameters

<i>row</i>	the number of the row
------------	-----------------------

Returns

the distribution: next states with the corresponding probas

Implements [transitionStructure](#).

3.18.2.15 rowSum()

```
double sparseMatrix::rowSum (  
    int row ) [virtual]
```

Summing a row.

Author

Alain Jean-Marie

Parameters

<i>row</i>	the number of the row
------------	-----------------------

Returns

the sum of the entries of the row (including diagonal)

Implements [transitionStructure](#).

3.18.2.16 setEntry()

```
bool sparseMatrix::setEntry (  
    int row,  
    int col,  
    double val ) [virtual]
```

Inserting an element in the matrix. IMPORTANT NOTE: there is no check for the existence of the column This is because in the context of Markov Reward Processes, there may be several transitions with different reward. It is simpler to handle this by allowing replicated entries in the neighborhood lists. The multiplication algorithm handles this in a natural way. BUT the [getEntry\(int,int\)](#) method must handle this situation carefully.

Author

Alain Jean-Marie

Parameters

<i>row</i>	the row number
<i>col</i>	the column number
<i>val</i>	the value to be inserted

Returns

true if successful, that is, if ranges are OK and memory allocation is OK.

Implements [transitionStructure](#).

3.18.2.17 uniformize()

```
sparseMatrix * sparseMatrix::uniformize ( ) [virtual]
```

Sparse matrix uniformization. Returns a sparse matrix and sets the uniformization factor. If the type is already discrete, returns a copy.

Author

Alain Jean-Marie

Returns

the uniformized version of the generator

Implements [transitionStructure](#).

3.18.2.18 write()

```
void sparseMatrix::write (
    FILE * out,
    std::string format )
```

Writing a sparse matrix to a file. Several formats could be chosen. Supported formats are: "Ers", "Full", "Matrix↔Market-sparse", "MatrixMarket-full", "Maple", "MARCA", "R", "scilab", "XBORNE".

Author

Alain Jean-Marie

Parameters

<i>out</i>	the file descriptor to which the matrix is written
<i>format</i>	the name of the format/language to use

3.18.3 Friends And Related Function Documentation

3.18.3.1 markovChain

```
friend class markovChain [friend]
```

markovChain class is a friend class because we want markovChain to be the main class for user.

The documentation for this class was generated from the following files:

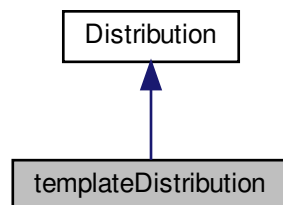
- transitionStructure/sparseMatrix.h
- transitionStructure/sparseMatrix.cpp

3.19 templateDistribution Class Reference

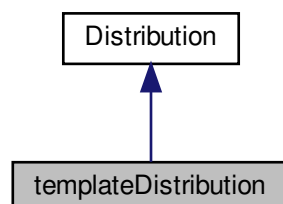
The general template distribution to be instantiated.

```
#include <templateDistribution.h>
```

Inheritance diagram for templateDistribution:



Collaboration diagram for templateDistribution:



Public Member Functions

- [templateDistribution](#) (int x, double *y, double *z)
Constructor for some distribution of some type from things and other. The mean is calculated at creation (or not).
- double [mean](#) ()
computing the mathematical expectation or mean
- double [rate](#) ()
computing the "rate", defined as the inverse of the mean
- double [moment](#) (int order)
Computing the moments of the distribution.
- double [variance](#) ()
Computing the variance of the random variable: the second moment minus the square of the first moment. Variance is the square of the coefficient of variation. The [Distribution](#) class offers a default implementation.
- double [laplace](#) (double s)
computing the Laplace transform of the distribution at real point
- double [dLaplace](#) (double s)
computing the derivative of the Laplace transform at real points
- double [cdf](#) (double x)
computing the cumulative distribution function at some real point x. This is the probability that the random variable is less or equal to x.
- double [ccdf](#) (double x)
computing the complementary cumulative distribution function (or tail) at some real point x. This is the probability that the random variable is strictly larger than x. The [Distribution](#) class offers a default implementation.
- bool [hasMoment](#) (int order)
test for the existence of moments of any order
- [templateDistribution](#) * [rescale](#) (double factor)
*rescaling a distribution by some real factor. Not all distributions allow this for any real factor. If the operation fails, or if the factor is 1.0, a copy of the distribution should be returned (**not** by using the [copy\(\)](#) function).*
- [templateDistribution](#) * [copy](#) ()
copying a distribution. Typically implemented as [rescale](#)(1.0).
- double [sample](#) ()
drawing a (pseudo)random value according to the distribution.
- void [iidSample](#) (int n, double *s)
drawing an i.i.d. sample from the distribution. The result is returned in an array (that must have been already allocated) passed as a parameter. The [Distribution](#) class offers the default implementation with repeated call to [sample\(\)](#).
- std::string [toString](#) ()
an utility to convert the distribution into a string.
- void [write](#) (FILE *out, int mode)
an utility to write the distribution to some file, according to some format.

Additional Inherited Members

3.19.1 Detailed Description

The general template distribution to be instantiated.

3.19.2 Constructor & Destructor Documentation

3.19.2.1 templateDistribution()

```
templateDistribution::templateDistribution (
    int x,
    double * y,
    double * z )
```

Constructor for some distribution of some type from things and other. The mean is calculated at creation (or not).

Author

Alain Jean-Marie

Parameters

<i>x</i>	un truc
<i>y</i>	un machin
<i>z</i>	une chose

Returns

an object of type [templateDistribution](#)

3.19.3 Member Function Documentation

3.19.3.1 ccdf()

```
double templateDistribution::ccdf (
    double x )
```

computing the complementary cumulative distributon function (or tail) at some real point x. This is the probability that the random variable is strictly larger than x. The [Distribution](#) class offers a default implementation.

Parameters

<i>x</i>	the value at which to compute the ccdf
----------	----------------------------------------

Returns

the value of the ccdf

3.19.3.2 cdf()

```
double templateDistribution::cdf (
    double x ) [virtual]
```

computing the cumulative distribution function at some real point x . This is the probability that the random variable is less or equal to x .

Parameters

x	the value at which to compute the cdf
-----	---------------------------------------

Returns

the value of the cdf

Computation of the cumulative density function at some real point x

Author

Alain Jean-Marie

Parameters

x	value at which the CDF is computed
-----	------------------------------------

Returns

CDF(x)

Implements [Distribution](#).

3.19.3.3 copy()

```
templateDistribution * templateDistribution::copy ( ) [virtual]
```

copying a distribution. Typically implemented as `rescale(1.0)`.

See also

[rescale\(double\)](#)

Returns

a copy of the distribution

Copying the law

Author

Alain Jean-Marie

Returns

a copy of the law

Implements [Distribution](#).

3.19.3.4 dLaplace()

```
double templateDistribution::dLaplace (
    double s ) [virtual]
```

computing the derivative of the Laplace transform at real points

Parameters

<i>s</i>	the value at which to compute
----------	-------------------------------

Returns

the value of the derivative of the Laplace transform

Computation of the derivative of the Laplace transform at some real point *s*

Author

Alain Jean-Marie

Parameters

<i>s</i>	value at which the derivative is computed
----------	-------------------------------------------

Returns

$d\text{LT}(s)/ds$

Implements [Distribution](#).

3.19.3.5 hasMoment()

```
bool templateDistribution::hasMoment (
    int order ) [virtual]
```

test for the existence of moments of any order

Parameters

<i>order</i>	the order of the moment to be tested
--------------	--------------------------------------

Returns

true if the moment exists, false otherwise

Implements [Distribution](#).

3.19.3.6 iidSample()

```
void templateDistribution::iidSample (
    int n,
    double * s )
```

drawing an i.i.d. sample from the distribution. The result is returned in an array (that must have been already allocated) passed as a parameter. The [Distribution](#) class offers the default implementation with repeated call to [sample\(\)](#).

Parameters

<i>n</i>	the number of values to sample
<i>s</i>	an array to be filled with the sample

3.19.3.7 laplace()

```
double templateDistribution::laplace (
    double s ) [virtual]
```

computing the Laplace transform of the distribution at real point

Parameters

<i>s</i>	the value at which to compute
----------	-------------------------------

Returns

the value of the Laplace transform

Computation of the Laplace transform at some real point *s*

Author

Alain Jean-Marie

Parameters

<i>s</i>	value at which the derivative is computed
----------	-------------------------------------------

Returns

LT(*s*)

Implements [Distribution](#).

3.19.3.8 mean()

```
double templateDistribution::mean ( ) [virtual]
```

computing the mathematical expectation or mean

Returns

the mean

Calculation of the mean. Returns the value since it is pre-computed

Author

Alain Jean-Marie

Returns

the mathematical expectation of the distribution

Implements [Distribution](#).

3.19.3.9 moment()

```
double templateDistribution::moment (
    int order ) [virtual]
```

Computing the moments of the distribution.

Parameters

<i>order</i>	the order of the moment to be computed
--------------	----------------------------------------

Returns

the moment

Implements [Distribution](#).

3.19.3.10 rate()

```
double templateDistribution::rate ( ) [virtual]
```

computing the "rate", defined as the inverse of the mean

Returns

the rate

Calculation of the rate, which is the inverse of the mean

Author

Alain Jean-Marie

Returns

the rate

Implements [Distribution](#).

3.19.3.11 rescale()

```
templateDistribution * templateDistribution::rescale (  
    double factor ) [virtual]
```

rescaling a distribution by some real factor. Not all distributions allow this for any real factor. If the operation fails, or if the factor is 1.0, a copy of the distribution should be returned (**not** by using the [copy\(\)](#) function).

See also

[copy\(\)](#)

Parameters

<i>factor</i>	the real factor to be used
---------------	----------------------------

Returns

a new distribution object

Rescaling the law X by some real factor f

Author

Alain Jean-Marie

Parameters

<i>factor</i>	the factor f
---------------	--------------

Returns

the law $f \cdot X$

Implements [Distribution](#).

3.19.3.12 sample()

```
double templateDistribution::sample ( ) [virtual]
```

drawing a (pseudo)random value according to the distribution.

Returns

a sample

Sampling from the law This is the straightforward, non optimized, linear-time algorithm

Author

Alain Jean-Marie

Returns

a copy of the law

Implements [Distribution](#).

3.19.3.13 toString()

```
std::string templateDistribution::toString ( ) [virtual]
```

an utility to convert the distribution into a string.

Returns

the string representing the distribution.

Printing a representation of the law into a string

Author

Alain Jean-Marie

Returns

a string

Implements [Distribution](#).

3.19.3.14 variance()

```
double templateDistribution::variance ( )
```

Computing the variance of the random variable: the second moment minus the square of the first moment. Variance is the square of the coefficient of variation. The [Distribution](#) class offers a default implementation.

Returns

the variance

3.19.3.15 write()

```
void templateDistribution::write (
    FILE * out,
    int mode ) [virtual]
```

an utility to write the distribution to some file, according to some format.

Parameters

<i>out</i>	the file descriptor of the file
<i>mode</i>	a code for the format to be used

Printing a representation of the law

Author

Alain Jean-Marie

Parameters

<i>out</i>	the output stream
<i>mode</i>	representation for the output

Implements [Distribution](#).

The documentation for this class was generated from the following files:

- Distribution/templateDistribution.h
- Distribution/templateDistribution.cpp

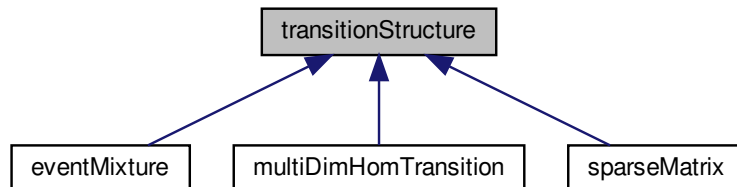
3.20 transitionStructure Class Reference

Abstract class for transition structures. These are structures which describe transitions to one state to another one, to which is attached a numeric label. Typical instances should be one-step transition matrices of discrete-time

Markov chains, and infinitesimal generators of continuous-time Markov chains. It is also possible that the origin state space and the destination state space are different.

```
#include <transitionStructure.h>
```

Inheritance diagram for transitionStructure:



Public Member Functions

- virtual `~transitionStructure ()`
Destructor of the class.
- int `size ()`
Read accessor for the size of the state space. This is the origin state space by default.
- int `origSize ()`
Read accessor for the size of the origin state space.
- int `destSize ()`
Read accessor for the size of the destination state space.
- timeType `type ()`
Read accessor for the time type.
- double `uniformizationRate ()`
Read accessor for the uniformization rate. Relevant mostly for discrete-time structures created by uniformization of a continuous-time one.
- void `setType (timeType t)`
Write accessor for the time type.
- void `setUniformizationRate (double rate)`
Write accessor for the uniformization rate.
- virtual bool `setEntry (int i, int j, double val)=0`
Method to set the value associated with some transition.
- virtual double `getEntry (int i, int j)=0`
Method to get the value associated with some transition. When state parameters are out of bounds, the returned value should be 0.
- virtual int `getNbElts (int i)=0`
Method to get the number of non-zero entries in a transition from some state. This can be seen as the number of actually possible transitions from that state.
- virtual int `getCol (int i, int k)=0`
*Method to get the **number** of the state corresponding to transition number k in the list of possible transitions from some state i.*
- virtual double `getEntryByCol (int i, int k)=0`
*Method to get the **value** attached to transition number k in the list of possible transitions from some state i.*

- virtual [discreteDistribution](#) * [getTransDistrib](#) (int i)=0
Method to get the transition from some state as a probability distribution.
- bool [readEntry](#) (FILE *input)
Reading from a file, and adding an element to the matrix. The field must be in the form "row column value" with blank spaces as separators.
- virtual double [rowSum](#) (int i)=0
Calculating the sum of values corresponding to transitions from some state.
- virtual [transitionStructure](#) * [copy](#) ()=0
Copying a transition structure.
- virtual [transitionStructure](#) * [uniformize](#) ()=0
Uniformizing a transition structure. The structure should be of continuous time type. The resulting one will be of discrete time type. If uniformization fails, a NULL pointer should be returned. If the origin structure is already of discrete-time type, a copy should be returned.
- virtual [transitionStructure](#) * [embed](#) ()=0
Embedding a discrete-time transition structure at jump times of a transition structure. The structure should be of continuous time type. The resulting one will be of discrete time type. If embedding fails, a NULL pointer should be returned. If the origin structure is already of discrete-time type, a copy should be returned.
- virtual void [evaluateMeasure](#) (double *d, double *res)
Computing the action of the transition structure on some measure, the measure being represented as a vector of real numbers. This corresponds to the multiplication vector/matrix, the row vector being interpreted as a signed measure.
- [discreteDistribution](#) * [evaluateMeasure](#) ([discreteDistribution](#) *d)
Computing the action of the transition structure on some probability distribution. This version with [discreteDistribution](#) objects uses [evaluateMeasure\(double,double*\)](#).*
- virtual void [evaluateValue](#) (double *v, double *res)=0
Computing the action of the transition structure on some vector of values. This corresponds to the multiplication matrix/vector, the column vector being interpreted as a vector of values attached to the states.
- void [write](#) (FILE *out, std::string format)
File output method for a transition structure.
- std::string [toString](#) (std::string format)
String serialization method for a transition structure. Supported formats are XBORNE (Rii variant: by increasing row and increasing columns), MARCA, Matrix-Market sparse and full, Ers, Maple, R, SCILAB, Full, and Matlab.

Protected Member Functions

- int [consolidate](#) (int i, int *destinations, double *values)
Method to consolidate (aggregate) transition probabilities from a given state. The method returns the number of different destination states, and modifies the arrays which will contain the lists of destinations and the corresponding probabilities. These arrays must have been allocated beforehand, with a size large enough to handle safely all possibilities.

Protected Attributes

- timeType [_type](#)
the time type of the structure: discrete or continuous.
- long int [_origSize](#)
size of the origin state space.
- long int [_destSize](#)
size of the destination state space.
- double [_uniformizationRate](#)
value related to the uniformization procedure.

3.20.1 Detailed Description

Abstract class for transition structures. These are structures which describe transitions to one state to another one, to which is attached a numeric label. Typical instances should be one-step transition matrices of discrete-time Markov chains, and infinitesimal generators of continuous-time Markov chains. It is also possible that the origin state space and the destination state space are different.

Author

Alain Jean-Marie and Issam Rabhi

3.20.2 Member Function Documentation

3.20.2.1 consolidate()

```
int transitionStructure::consolidate (
    int i,
    int * destinations,
    double * values ) [protected]
```

Method to consolidate (aggregate) transition probabilities from a given state. The method returns the number of different destination states, and modifies the arrays which will contain the lists of destinations and the corresponding probabilities. These arrays must have been allocated beforehand, with a size large enough to handle safely all possibilities.

Parameters

<i>i</i>	the origin state
<i>destinations</i>	the list of destinations
<i>values</i>	the list of probabilities

Returns

the number of different destinations

3.20.2.2 copy()

```
virtual transitionStructure\* transitionStructure::copy ( ) [pure virtual]
```

Copying a transition structure.

Returns

[transitionStructure](#)

Implemented in [sparseMatrix](#), [multiDimHomTransition](#), and [eventMixture](#).

3.20.2.3 destSize()

```
int transitionStructure::destSize ( ) [inline]
```

Read accessor for the size of the destination state space.

Returns

the size of the destination state space

3.20.2.4 embed()

```
virtual transitionStructure* transitionStructure::embed ( ) [pure virtual]
```

Embedding a discrete-time transition structure at jump times of a transition structure. The structure should be of continuous time type. The resulting one will be of discrete time type. If embedding fails, a NULL pointer should be returned. If the origin structure is already of discrete-time type, a copy should be returned.

Returns

a discrete-time transition structure

Implemented in [sparseMatrix](#), [multiDimHomTransition](#), and [eventMixture](#).

3.20.2.5 evaluateMeasure() [1/2]

```
virtual void transitionStructure::evaluateMeasure (
    double * d,
    double * res ) [inline], [virtual]
```

Computing the action of the transition structure on some measure, the measure being represented as a vector of real numbers. This corresponds to the multiplication vector/matrix, the row vector being interpreted as a signed measure.

Parameters

<i>d</i>	the measure to evaluate
<i>res</i>	the resulting measure

Reimplemented in [sparseMatrix](#), [multiDimHomTransition](#), and [eventMixture](#).

3.20.2.6 evaluateMeasure() [2/2]

```
discreteDistribution * transitionStructure::evaluateMeasure (
    discreteDistribution * d )
```

Computing the action of the transition structure on some probability distribution. This version with [discrete↔Distribution](#) objects uses [evaluateMeasure\(double*,double*\)](#).

Parameters

<i>d</i>	the distribution to evaluate
----------	------------------------------

Returns

the distribution resulting from the evaluation

3.20.2.7 evaluateValue()

```
virtual void transitionStructure::evaluateValue (
    double * v,
    double * res ) [pure virtual]
```

Computing the action of the transition structure on some vector of values. This corresponds to the multiplication matrix/vector, the column vector being interpreted as a vector of values attached to the states.

Parameters

<i>v</i>	the vector of values to evaluate
<i>res</i>	the resulting vector

Implemented in [sparseMatrix](#), [multiDimHomTransition](#), and [eventMixture](#).

3.20.2.8 getCol()

```
virtual int transitionStructure::getCol (
    int i,
    int k ) [pure virtual]
```

Method to get the **number** of the state corresponding to transition number k in the list of possible transitions from some state i.

Parameters

<i>i</i>	the origin state
<i>k</i>	the index of transition from state i

Returns

the destination state corresponding to the k-th possible transition from state i

Implemented in [sparseMatrix](#), [multiDimHomTransition](#), and [eventMixture](#).

3.20.2.9 getEntry()

```
virtual double transitionStructure::getEntry (
    int i,
    int j ) [pure virtual]
```

Method to get the value associated with some transition. When state parameters are out of bounds, the returned value should be 0.

Parameters

<i>i</i>	the origin state
<i>j</i>	the destination state

Returns

the value attached to the transition (i,j)

Implemented in [sparseMatrix](#), [multiDimHomTransition](#), and [eventMixture](#).

3.20.2.10 getEntryByCol()

```
virtual double transitionStructure::getEntryByCol (
    int i,
    int k ) [pure virtual]
```

Method to get the **value** attached to transition number k in the list of possible transitions from some state i.

Parameters

<i>i</i>	the origin state
<i>k</i>	the index of the transition from state i

Returns

the value attached to the k-th possible transition from state i

Implemented in [sparseMatrix](#), [multiDimHomTransition](#), and [eventMixture](#).

3.20.2.11 getNbElts()

```
virtual int transitionStructure::getNbElts (
    int i ) [pure virtual]
```

Method to get the number of non-zero entries in a transition from some state. This can be seen as the number of actually possible transitions from that state.

Parameters

<i>i</i>	the origin state
----------	------------------

Returns

int the number of possible transitions

Implemented in [sparseMatrix](#), [multiDimHomTransition](#), and [eventMixture](#).

3.20.2.12 getTransDistrib()

```
virtual discreteDistribution* transitionStructure::getTransDistrib (
    int i ) [pure virtual]
```

Method to get the transition from some state as a probability distribution.

Parameters

<i>i</i>	the origin state
----------	------------------

Returns

a discrete distribution object

Implemented in [sparseMatrix](#), [multiDimHomTransition](#), and [eventMixture](#).

3.20.2.13 origSize()

```
int transitionStructure::origSize ( ) [inline]
```

Read accessor for the size of the origin state space.

Returns

the size of the origin state space

3.20.2.14 readEntry()

```
bool transitionStructure::readEntry (
    FILE * input )
```

Reading from a file, and adding an element to the matrix. The field must be in the form "row column value" with blank spaces as separators.

Author

Alain Jean-Marie

Parameters

<i>input</i>	is the input flow from which the data is read
--------------	-----------------------------------------------

Returns

true if successful, false otherwise

3.20.2.15 rowSum()

```
virtual double transitionStructure::rowSum (
    int i ) [pure virtual]
```

Calculating the sum of values corresponding to transitions from some state.

Parameters

<i>i</i>	the state
----------	-----------

Returns

the sum

Implemented in [sparseMatrix](#), [multiDimHomTransition](#), and [eventMixture](#).

3.20.2.16 setEntry()

```
virtual bool transitionStructure::setEntry (
    int i,
    int j,
    double val ) [pure virtual]
```

Method to set the value associated with some transition.

Parameters

<i>i</i>	the origin state
<i>j</i>	the destination state
<i>val</i>	the value attached to the transition

Returns

true if the operation was successful, false otherwise (out of bounds; wrong numeric value)

Implemented in [sparseMatrix](#), [multiDimHomTransition](#), and [eventMixture](#).

3.20.2.17 setType()

```
void transitionStructure::setType (
    timeType t ) [inline]
```

Write accessor for the time type.

Parameters

<i>t</i>	the time type to be set
----------	-------------------------

3.20.2.18 setUniformizationRate()

```
void transitionStructure::setUniformizationRate (
    double rate ) [inline]
```

Write accessor for the uniformization rate.

Parameters

<i>rate</i>	the rate to be set
-------------	--------------------

3.20.2.19 size()

```
int transitionStructure::size ( ) [inline]
```

Read accessor for the size of the state space. This is the origin state space by default.

Returns

the size of the origin state space

3.20.2.20 toString()

```
std::string transitionStructure::toString (
    std::string format )
```

String serialization method for a transition structure. Supported formats are XBORNE (Rii variant: by increasing row and increasing columns), MARCA, Matrix-Market sparse and full, Ers, Maple, R, SCILAB, Full, and Matlab.

Author

Issam Rabhi (R language), Alain Jean-Marie (XBORNE language)

Parameters

<i>format</i>	the format/language to be used.
---------------	---------------------------------

3.20.2.21 `type()`

```
timeType transitionStructure::type ( ) [inline]
```

Read accessor for the time type.

Returns

the time type

3.20.2.22 `uniformizationRate()`

```
double transitionStructure::uniformizationRate ( ) [inline]
```

Read accessor for the uniformization rate. Relevant mostly for discrete-time structures created by uniformization of a continuous-time one.

Returns

the uniformization rate

3.20.2.23 `uniformize()`

```
virtual transitionStructure\* transitionStructure::uniformize ( ) [pure virtual]
```

Uniformizing a transition structure. The structure should be of continuous time type. The resulting one will be of discrete time type. If uniformization fails, a NULL pointer should be returned. If the origin structure is already of discrete-time type, a copy should be returned.

Returns

a discrete-time transition structure

Implemented in [sparseMatrix](#), [multiDimHomTransition](#), and [eventMixture](#).

3.20.2.24 `write()`

```
void transitionStructure::write (
    FILE * out,
    std::string format )
```

File output method for a transition structure.

Parameters

<i>out</i>	the file descriptor to which the structure should be written.
<i>format</i>	the format/language to be used.

3.20.3 Member Data Documentation

3.20.3.1 `_type`

```
timeType transitionStructure::_type [protected]
```

the time type of the structure: discrete or continuous.

This determines the nature of the numeric label: probabilities or rates.

The documentation for this class was generated from the following files:

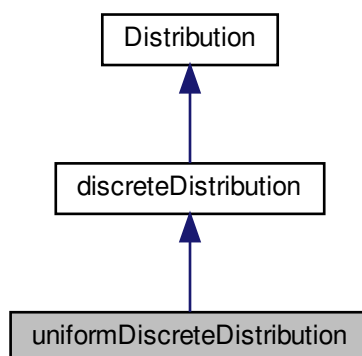
- transitionStructure/transitionStructure.h
- transitionStructure/transitionStructure.cpp

3.21 uniformDiscreteDistribution Class Reference

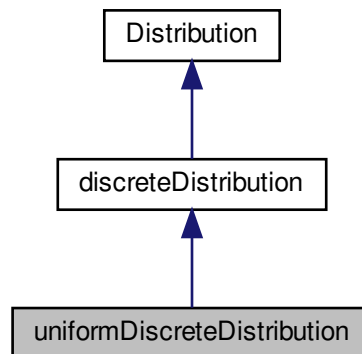
The uniform discrete distribution.

```
#include <uniformDiscreteDistribution.h>
```

Inheritance diagram for uniformDiscreteDistribution:



Collaboration diagram for uniformDiscreteDistribution:



Public Member Functions

- `uniformDiscreteDistribution` (int `valInf`, int `valSup`)
The unique constructor for the uniform discrete distribution on some interval $[a..b]$.
- int `valInf` ()
Read accessor to the lower end of the interval.
- int `valSup` ()
Read accessor to the upper end of the interval.
- double `getProba` (double value)
Computes the probability of a particular value. The tolerance `VALUE_TOLERANCE` is applied to match values.
- double `mean` ()
computing the mathematical expectation or mean
- double `rate` ()
computing the "rate", defined as the inverse of the mean
- double `moment` (int order)
Computing the moments of the distribution.
- double `variance` ()
Computing the variance of the random variable: the second moment minus the square of the first moment. Variance is the square of the coefficient of variation. The `Distribution` class offers a default implementation.
- double `laplace` (double s)
computing the Laplace transform of the distribution at real point
- double `dLaplace` (double s)
computing the derivative of the Laplace transform at real points
- double `cdf` (double x)
computing the cumulative distribution function at some real point x. This is the probability that the random variable is less or equal to x.
- double `ccdf` (double x)
computing the complementary cumulative distributon function (or tail) at some real point x. This is the probability that the random variable is strictly larger than x. The `Distribution` class offers a default implementation.
- bool `hasMoment` (int order)
test for the existence of moments of any order

- [uniformDiscreteDistribution](#) * [rescale](#) (double factor)
*rescaling a distribution by some real factor. Not all distributions allow this for any real factor. If the operation fails, or if the factor is 1.0, a copy of the distribution should be returned (**not** by using the [copy\(\)](#) function).*
- [uniformDiscreteDistribution](#) * [copy](#) ()
copying a distribution. Typically implemented as [rescale\(1.0\)](#).
- double [sample](#) ()
drawing a (pseudo)random value according to the distribution.
- void [iidSample](#) (int n, double *s)
drawing an i.i.d. sample from the distribution. The result is returned in an array (that must have been already allocated) passed as a parameter. The [Distribution](#) class offers the default implementation with repeated call to [sample\(\)](#).
- std::string [toString](#) ()
an utility to convert the distribution into a string.
- void [write](#) (FILE *out, int mode)
an utility to write the distribution to some file, according to some format.

Additional Inherited Members

3.21.1 Detailed Description

The uniform discrete distribution.

3.21.2 Constructor & Destructor Documentation

3.21.2.1 uniformDiscreteDistribution()

```
uniformDiscreteDistribution::uniformDiscreteDistribution (
    int valInf,
    int valSup )
```

The unique constructor for the uniform discrete distribution on some interval [a..b].

Parameters

<i>valInf</i>	the lower bound (a)
<i>valSup</i>	the upper bound (b)

Constructor for a discrete uniform distribution on [a..b] The mean is calculated at creation.

Author

Alain Jean-Marie

Parameters

<i>valInf</i>	the value of a
<i>valSup</i>	the value of b

Returns

an object of type [uniformDiscreteDistribution](#)

3.21.3 Member Function Documentation

3.21.3.1 ccdf()

```
double uniformDiscreteDistribution::ccdf (  
    double x )
```

computing the complementary cumulative distributon function (or tail) at some real point x. This is the probability that the random variable is strictly larger than x. The [Distribution](#) class offers a default implementation.

Parameters

x	the value at which to compute the ccdf
---	----------------------------------------

Returns

the value of the ccdf

3.21.3.2 cdf()

```
double uniformDiscreteDistribution::cdf (  
    double x ) [virtual]
```

computing the cumulative distribution function at some real point x. This is the probability that the random variable is less or equal to x.

Parameters

x	the value at which to compute the cdf
---	---------------------------------------

Returns

the value of the cdf

Computation of the cumulative density function at some real point x

Reimplemented from [discreteDistribution](#).

3.21.3.3 copy()

```
uniformDiscreteDistribution * uniformDiscreteDistribution::copy ( ) [virtual]
```

copying a distribution. Typically implemented as `rescale(1.0)`.

See also

[rescale\(double\)](#)

Returns

a copy of the distribution

Copying the law

Reimplemented from [discreteDistribution](#).

3.21.3.4 dLaplace()

```
double uniformDiscreteDistribution::dLaplace (
    double s ) [virtual]
```

computing the derivative of the Laplace transform at real points

Parameters

<code>s</code>	the value at which to compute
----------------	-------------------------------

Returns

the value of the derivative of the Laplace transform

Computation of the derivative of the Laplace transform at some real point `s`

Reimplemented from [discreteDistribution](#).

3.21.3.5 getProba()

```
double uniformDiscreteDistribution::getProba (
    double value )
```

Computes the probability of a particular value. The tolerance `VALUE_TOLERANCE` is applied to match values.

Author

Alain Jean-Marie

Parameters

<i>value</i>	the value at which the proba is computed
--------------	------------------------------------------

Returns

the probability of value *v*

3.21.3.6 hasMoment()

```
bool uniformDiscreteDistribution::hasMoment (
    int order ) [virtual]
```

test for the existence of moments of any order

Parameters

<i>order</i>	the order of the moment to be tested
--------------	--------------------------------------

Returns

true if the moment exists, false otherwise

These distributions have moments of all orders.

Returns

true

Reimplemented from [discreteDistribution](#).

3.21.3.7 iidSample()

```
void uniformDiscreteDistribution::iidSample (
    int n,
    double * s )
```

drawing an i.i.d. sample from the distribution. The result is returned in an array (that must have been already allocated) passed as a parameter. The [Distribution](#) class offers the default implementation with repeated call to [sample\(\)](#).

Parameters

<i>n</i>	the number of values to sample
<i>s</i>	an array to be filled with the sample

lid samples from the law in a table

3.21.3.8 laplace()

```
double uniformDiscreteDistribution::laplace (
    double s ) [virtual]
```

computing the Laplace transform of the distribution at real point

Parameters

<i>s</i>	the value at which to compute
----------	-------------------------------

Returns

the value of the Laplace transform

Computation of the Laplace transform at some real point *s*

Reimplemented from [discreteDistribution](#).

3.21.3.9 mean()

```
double uniformDiscreteDistribution::mean ( ) [virtual]
```

computing the mathematical expectation or mean

Returns

the mean

Returns the value since the mean is pre-computed at creation time.

Reimplemented from [discreteDistribution](#).

3.21.3.10 moment()

```
double uniformDiscreteDistribution::moment (
    int order ) [virtual]
```

Computing the moments of the distribution.

Parameters

<i>order</i>	the order of the moment to be computed
--------------	----------------------------------------

Returns

the moment

Reimplemented from [discreteDistribution](#).

3.21.3.11 rate()

```
double uniformDiscreteDistribution::rate ( ) [virtual]
```

computing the "rate", defined as the inverse of the mean

Returns

the rate

Calculation of the rate, which is the inverse of the mean

Reimplemented from [discreteDistribution](#).

3.21.3.12 rescale()

```
uniformDiscreteDistribution * uniformDiscreteDistribution::rescale (
    double factor ) [virtual]
```

rescaling a distribution by some real factor. Not all distributions allow this for any real factor. If the operation fails, or if the factor is 1.0, a copy of the distribution should be returned (**not** by using the [copy\(\)](#) function).

See also

[copy\(\)](#)

Parameters

<i>factor</i>	the real factor to be used
---------------	----------------------------

Returns

a new distribution object

Rescaling the law X by some real factor f

Reimplemented from [discreteDistribution](#).

3.21.3.13 sample()

```
double uniformDiscreteDistribution::sample ( ) [virtual]
```

drawing a (pseudo)random value according to the distribution.

Returns

a sample

Sampling from the law

Reimplemented from [discreteDistribution](#).

3.21.3.14 toString()

```
std::string uniformDiscreteDistribution::toString ( ) [virtual]
```

an utility to convert the distribution into a string.

Returns

the string representing the distribution.

Printing a representation of the law into a string

Reimplemented from [discreteDistribution](#).

3.21.3.15 valInf()

```
int uniformDiscreteDistribution::valInf ( )
```

Read accessor to the lower end of the interval.

Returns

the lower end of the interval

3.21.3.16 valSup()

```
int uniformDiscreteDistribution::valSup ( )
```

Read accessor to the upper end of the interval.

Returns

the upper end of the interval

3.21.3.17 variance()

```
double uniformDiscreteDistribution::variance ( )
```

Computing the variance of the random variable: the second moment minus the square of the first moment. Variance is the square of the coefficient of variation. The [Distribution](#) class offers a default implementation.

Returns

the variance

3.21.3.18 write()

```
void uniformDiscreteDistribution::write (
    FILE * out,
    int mode ) [virtual]
```

an utility to write the distribution to some file, according to some format.

Parameters

<i>out</i>	the file descriptor of the file
<i>mode</i>	a code for the format to be used

Printing a representation of the law

Reimplemented from [discreteDistribution](#).

The documentation for this class was generated from the following files:

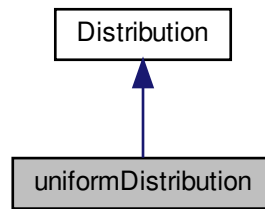
- Distribution/uniformDiscreteDistribution.h
- Distribution/uniformDiscreteDistribution.cpp

3.22 uniformDistribution Class Reference

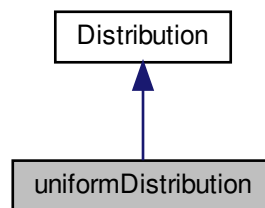
The continuous uniform distribution over some interval.

```
#include <uniformDistribution.h>
```

Inheritance diagram for uniformDistribution:



Collaboration diagram for uniformDistribution:



Public Member Functions

- `uniformDistribution` (double, double)
standard constructor from extremities of the interval
- double `valInf` ()
Read accessor to the lower end of the interval.
- double `valSup` ()
Read accessor to the upper end of the interval.
- double `mean` ()
computing the mathematical expectation or mean
- double `rate` ()
computing the "rate", defined as the inverse of the mean
- double `moment` (int order)
Computing the moments of the distribution.
- double `variance` ()
Computing the variance of the random variable: the second moment minus the square of the first moment. Variance is the square of the coefficient of variation. The `Distribution` class offers a default implementation.
- double `laplace` (double s)
computing the Laplace transform of the distribution at real point
- double `dLaplace` (double s)

- computing the derivative of the Laplace transform at real points*
- double `cdf` (double x)

computing the cumulative distribution function at some real point x. This is the probability that the random variable is less or equal to x.
- double `ccdf` (double x)

computing the complementary cumulative distributon function (or tail) at some real point x. This is the probability that the random variable is strictly larger than x. The [Distribution](#) class offers a default implementation.
- bool `hasMoment` (int order)

test for the existence of moments of any order
- `uniformDistribution * rescale` (double factor)

*rescaling a distribution by some real factor. Not all distributions allow this for any real factor. If the operation fails, or if the factor is 1.0, a copy of the distribution should be returned (**not** by using the `copy()` function).*
- `uniformDistribution * copy` ()

copying a distribution. Typically implemented as `rescale(1.0)`.
- double `sample` ()

drawing a (pseudo)random value according to the distribution.
- void `iidSample` (int n, double *s)

drawing an i.i.d. sample from the distribution. The result is returned in an array (that must have been already allocated) passed as a parameter. The [Distribution](#) class offers the default implementation with repeated call to `sample()`.
- std::string `toString` ()

an utility to convert the distribution into a string.
- void `write` (FILE *out, int mode)

an utility to write the distribution to some file, according to some format.

Additional Inherited Members

3.22.1 Detailed Description

The continuous uniform distribution over some interval.

3.22.2 Constructor & Destructor Documentation

3.22.2.1 uniformDistribution()

```
uniformDistribution::uniformDistribution (
    double inf,
    double sup )
```

standard constructor from extremities of the interval

Constructor for a continuous uniform distribution on [a,b] The mean is calculated at creation.

Author

Alain Jean-Marie

Parameters

<i>inf</i>	the value of a
<i>sup</i>	the value of b

Returns

an object of type [uniformDistribution](#)

3.22.3 Member Function Documentation**3.22.3.1 ccdf()**

```
double uniformDistribution::ccdf (  
    double x )
```

computing the complementary cumulative distributon function (or tail) at some real point x. This is the probability that the random variable is strictly larger than x. The [Distribution](#) class offers a default implementation.

Parameters

<i>x</i>	the value at which to compute the ccdf
----------	----------------------------------------

Returns

the value of the ccdf

3.22.3.2 cdf()

```
double uniformDistribution::cdf (  
    double x ) [virtual]
```

computing the cumulative distribution function at some real point x. This is the probability that the random variable is less or equal to x.

Parameters

<i>x</i>	the value at which to compute the cdf
----------	---------------------------------------

Returns

the value of the cdf

Computation of the cumulative density function at some real point x

Implements [Distribution](#).

3.22.3.3 `copy()`

```
uniformDistribution * uniformDistribution::copy ( ) [virtual]
```

copying a distribution. Typically implemented as `rescale(1.0)`.

See also

[rescale\(double\)](#)

Returns

a copy of the distribution

Copying the law

Implements [Distribution](#).

3.22.3.4 `dLaplace()`

```
double uniformDistribution::dLaplace (
    double s ) [virtual]
```

computing the derivative of the Laplace transform at real points

Parameters

s	the value at which to compute
-----	-------------------------------

Returns

the value of the derivative of the Laplace transform

Computation of the derivative of the Laplace transform at some real point s

Implements [Distribution](#).

3.22.3.5 hasMoment()

```
bool uniformDistribution::hasMoment (
    int order ) [virtual]
```

test for the existence of moments of any order

Parameters

<i>order</i>	the order of the moment to be tested
--------------	--------------------------------------

Returns

true if the moment exists, false otherwise

These distributions have moments of all orders.

Returns

true

Implements [Distribution](#).

3.22.3.6 iidSample()

```
void uniformDistribution::iidSample (
    int n,
    double * s )
```

drawing an i.i.d. sample from the distribution. The result is returned in an array (that must have been already allocated) passed as a parameter. The [Distribution](#) class offers the default implementation with repeated call to [sample\(\)](#).

Parameters

<i>n</i>	the number of values to sample
<i>s</i>	an array to be filled with the sample

lid samples from the law in a table

3.22.3.7 laplace()

```
double uniformDistribution::laplace (
    double s ) [virtual]
```

computing the Laplace transform of the distribution at real point

Parameters

<i>s</i>	the value at which to compute
----------	-------------------------------

Returns

the value of the Laplace transform

Computation of the Laplace transform at some real point s

Implements [Distribution](#).

3.22.3.8 mean()

```
double uniformDistribution::mean ( ) [virtual]
```

computing the mathematical expectation or mean

Returns

the mean

Returns the value since the mean is pre-computed at creation time.

Implements [Distribution](#).

3.22.3.9 moment()

```
double uniformDistribution::moment (
    int order ) [virtual]
```

Computing the moments of the distribution.

Parameters

<i>order</i>	the order of the moment to be computed
--------------	----------------------------------------

Returns

the moment

Calculation of the of order n

Implements [Distribution](#).

3.22.3.10 rate()

```
double uniformDistribution::rate ( ) [virtual]
```

computing the "rate", defined as the inverse of the mean

Returns

the rate

Calculation of the rate, which is the inverse of the mean

Implements [Distribution](#).

3.22.3.11 rescale()

```
uniformDistribution * uniformDistribution::rescale (
    double factor ) [virtual]
```

rescaling a distribution by some real factor. Not all distributions allow this for any real factor. If the operation fails, or if the factor is 1.0, a copy of the distribution should be returned (**not** by using the [copy\(\)](#) function).

See also

[copy\(\)](#)

Parameters

<i>factor</i>	the real factor to be used
---------------	----------------------------

Returns

a new distribution object

Rescaling the law X by some real factor f

Implements [Distribution](#).

3.22.3.12 sample()

```
double uniformDistribution::sample ( ) [virtual]
```

drawing a (pseudo)random value according to the distribution.

Returns

a sample

Sampling from the law

Implements [Distribution](#).

3.22.3.13 toString()

```
std::string uniformDistribution::toString ( ) [virtual]
```

an utility to convert the distribution into a string.

Returns

the string representing the distribution.

Printing a representation of the law into a string

Implements [Distribution](#).

3.22.3.14 valInf()

```
double uniformDistribution::valInf ( )
```

Read accessor to the lower end of the interval.

Returns

the lower end of the interval

3.22.3.15 valSup()

```
double uniformDistribution::valSup ( )
```

Read accessor to the upper end of the interval.

Returns

the upper end of the interval

3.22.3.16 variance()

```
double uniformDistribution::variance ( )
```

Computing the variance of the random variable: the second moment minus the square of the first moment. Variance is the square of the coefficient of variation. The [Distribution](#) class offers a default implementation.

Returns

the variance

3.22.3.17 write()

```
void uniformDistribution::write (
    FILE * out,
    int mode ) [virtual]
```

an utility to write the distribution to some file, according to some format.

Parameters

<i>out</i>	the file descriptor of the file
<i>mode</i>	a code for the format to be used

Printing a representation of the law

Implements [Distribution](#).

The documentation for this class was generated from the following files:

- Distribution/uniformDistribution.h
- Distribution/uniformDistribution.cpp

Index

- `_cardinal`
 - `marmoteSet`, 108
 - `_dimOffset`
 - `marmoteSet`, 109
 - `_dimension`
 - `marmoteSet`, 109
 - `_idxOffset`
 - `marmoteSet`, 109
 - `_max`
 - `marmoteInterval`, 99
 - `_mean`
 - `Distribution`, 57
 - `_min`
 - `marmoteInterval`, 99
 - `_name`
 - `Distribution`, 58
 - `_nbDimensions`
 - `marmoteSet`, 109
 - `_nbVals`
 - `discreteDistribution`, 48
 - `_nbZones`
 - `marmoteSet`, 109
 - `_probas`
 - `discreteDistribution`, 48
 - `_stateBuffer`
 - `marmoteSet`, 109
 - `_totNbDims`
 - `marmoteSet`, 109
 - `_type`
 - `marmoteSet`, 109
 - `transitionStructure`, 163
 - `_values`
 - `discreteDistribution`, 48
 - `_zeroState`
 - `marmoteSet`, 110
 - `_zone`
 - `marmoteSet`, 110
- `~discreteDistribution`
 - `discreteDistribution`, 36
- `~marmoteInterval`
 - `marmoteInterval`, 96
- `~marmoteSet`
 - `marmoteSet`, 102
- `~multiDimHomTransition`
 - `multiDimHomTransition`, 114
- `~poissonDistribution`
 - `poissonDistribution`, 123
- `~sparseMatrix`
 - `sparseMatrix`, 134

- `addToEntry`
 - `sparseMatrix`, 135
- `bernoulliDistribution`, 5
 - `bernoulliDistribution`, 7
 - `cdf`, 7
 - `copy`, 8
 - `dLaplace`, 8
 - `getParameter`, 9
 - `hasMoment`, 9
 - `laplace`, 10
 - `mean`, 11
 - `moment`, 11
 - `proba`, 12
 - `rate`, 12
 - `rescale`, 12
 - `sample`, 13
 - `toString`, 13
 - `write`, 13
- `binarySequence`, 14
 - `binarySequence`, 15
 - `decodeState`, 15
 - `firstState`, 16
 - `index`, 16
 - `isZero`, 17
 - `nextState`, 17
 - `printState`, 17
- `binarySimplex`, 18
 - `binarySimplex`, 19
 - `decodeState`, 20
 - `firstState`, 20
 - `index`, 20
 - `isZero`, 21
 - `nextState`, 21
 - `printState`, 21
- `cardinal`
 - `marmoteSet`, 102
- `cardinalOutDim`
 - `marmoteBox`, 89
 - `marmoteSet`, 103
- `cardinalbyDim`
 - `marmoteBox`, 88
 - `marmoteSet`, 102
- `ccdf`
 - `Distribution`, 50
 - `poissonDistribution`, 123
 - `templateDistribution`, 145
 - `uniformDiscreteDistribution`, 166
 - `uniformDistribution`, 175

- cdf
 - bernoulliDistribution, 7
 - diracDistribution, 24
 - discreteDistribution, 36
 - Distribution, 51
 - exponentialDistribution, 69
 - geometricDistribution, 76
 - poissonDistribution, 124
 - templateDistribution, 145
 - uniformDiscreteDistribution, 166
 - uniformDistribution, 175
- consolidate
 - transitionStructure, 155
- copy
 - bernoulliDistribution, 8
 - diracDistribution, 25
 - discreteDistribution, 37
 - Distribution, 51
 - eventMixture, 61
 - exponentialDistribution, 70
 - geometricDistribution, 77
 - multiDimHomTransition, 114
 - poissonDistribution, 124
 - sparseMatrix, 135
 - templateDistribution, 146
 - transitionStructure, 155
 - uniformDiscreteDistribution, 166
 - uniformDistribution, 176
- dLaplace
 - bernoulliDistribution, 8
 - diracDistribution, 25
 - discreteDistribution, 40
 - Distribution, 52
 - exponentialDistribution, 70
 - geometricDistribution, 77
 - poissonDistribution, 125
 - templateDistribution, 146
 - uniformDiscreteDistribution, 167
 - uniformDistribution, 176
- decodeState
 - binarySequence, 15
 - binarySimplex, 20
 - marmoteBox, 89
 - marmoteInterval, 96
 - marmoteSet, 103
- destSize
 - transitionStructure, 155
- diagnose
 - sparseMatrix, 136
- dimSize
 - multiDimHomTransition, 114
- diracDistribution, 22
 - cdf, 24
 - copy, 25
 - dLaplace, 25
 - diracDistribution, 24
 - getProba, 27
 - hasMoment, 27
 - iidSample, 28
 - laplace, 28
 - mean, 29
 - moment, 29
 - rate, 30
 - rescale, 30
 - sample, 31
 - toString, 31
 - value, 31
 - write, 32
- discreteDistribution, 32
 - _nbVals, 48
 - _probas, 48
 - _values, 48
 - ~discreteDistribution, 36
 - cdf, 36
 - copy, 37
 - dLaplace, 40
 - discreteDistribution, 35, 36
 - distanceL1, 38
 - distanceL2, 38
 - distanceLinfinity, 39
 - getProba, 40
 - getProbaByIndex, 41
 - getValue, 41
 - hasMoment, 42
 - laplace, 42
 - mean, 43
 - moment, 43
 - nbVals, 44
 - probas, 44
 - rate, 44
 - rescale, 45
 - sample, 46
 - setProba, 46
 - toString, 47
 - values, 47
 - write, 47
- distanceL1
 - discreteDistribution, 38
 - Distribution, 51
- distanceL2
 - discreteDistribution, 38
- distanceLinfinity
 - discreteDistribution, 39
- Distribution, 49
 - _mean, 57
 - _name, 58
 - ccdf, 50
 - cdf, 51
 - copy, 51
 - dLaplace, 52
 - distanceL1, 51
 - exponential, 52
 - hasMoment, 53
 - hasProperty, 53
 - iidSample, 54
 - laplace, 54

- mean, 54
 - moment, 55
 - name, 55
 - rate, 55
 - rescale, 55
 - sample, 56
 - toString, 56
 - u_0_1, 56
 - VALUE_TOLERANCE, 58
 - variance, 57
 - write, 57
- embed
- eventMixture, 61
 - multiDimHomTransition, 115
 - sparseMatrix, 136
 - transitionStructure, 156
- evaluateMeasure
- eventMixture, 61, 62
 - multiDimHomTransition, 115
 - sparseMatrix, 136, 137
 - transitionStructure, 156
- evaluateValue
- eventMixture, 62
 - multiDimHomTransition, 116
 - sparseMatrix, 137
 - transitionStructure, 157
- evaluateValueState
- eventMixture, 63
 - sparseMatrix, 138
- eventMixture, 58
- copy, 61
 - embed, 61
 - evaluateMeasure, 61, 62
 - evaluateValue, 62
 - evaluateValueState, 63
 - eventMixture, 60
 - eventProba, 63
 - getCol, 63
 - getEntry, 64
 - getEntryByCol, 64
 - getNbElts, 65
 - getTransDistrib, 65
 - nbEvents, 65
 - rowSum, 66
 - setEntry, 66
 - uniformize, 67
 - write, 67
- eventProba
- eventMixture, 63
- exponential
- Distribution, 52
- exponentialDistribution, 67
- cdf, 69
 - copy, 70
 - dLaplace, 70
 - exponentialDistribution, 69
 - hasMoment, 70
 - laplace, 71
 - mean, 71
 - moment, 72
 - rate, 72
 - rescale, 72
 - sample, 73
 - toString, 73
 - write, 73
- firstState
- binarySequence, 16
 - binarySimplex, 20
 - marmoteBox, 89
 - marmoteInterval, 97
 - marmoteSet, 103
- firstStateOutDim
- marmoteBox, 90
 - marmoteSet, 104
- firstStatebyDim
- marmoteBox, 90
 - marmoteSet, 104
- geometricDistribution, 74
- cdf, 76
 - copy, 77
 - dLaplace, 77
 - geometricDistribution, 75
 - getProba, 78
 - getRatio, 78
 - hasMoment, 79
 - laplace, 79
 - mean, 80
 - moment, 80
 - p, 81
 - rate, 81
 - rescale, 82
 - sample, 82
 - toString, 83
 - write, 83
- getCol
- eventMixture, 63
 - multiDimHomTransition, 116
 - sparseMatrix, 138
 - transitionStructure, 157
- getEntry
- eventMixture, 64
 - multiDimHomTransition, 116
 - sparseMatrix, 139
 - transitionStructure, 158
- getEntryByCol
- eventMixture, 64
 - multiDimHomTransition, 117
 - sparseMatrix, 139
 - transitionStructure, 158
- getJumpDistribution
- multiDimHomTransition, 117
- getNbElts
- eventMixture, 65
 - multiDimHomTransition, 117
 - sparseMatrix, 139

- transitionStructure, 158
- getParameter
 - bernoulliDistribution, 9
- getProba
 - diracDistribution, 27
 - discreteDistribution, 40
 - geometricDistribution, 78
 - poissonDistribution, 125
 - uniformDiscreteDistribution, 167
- getProbaByIndex
 - discreteDistribution, 41
- getRatio
 - geometricDistribution, 78
- getStronglyConnectedComponents
 - sparseMatrix, 140
- getTransDistrib
 - eventMixture, 65
 - multiDimHomTransition, 118
 - sparseMatrix, 140
 - transitionStructure, 159
- getValue
 - discreteDistribution, 41
- hasMoment
 - bernoulliDistribution, 9
 - diracDistribution, 27
 - discreteDistribution, 42
 - Distribution, 53
 - exponentialDistribution, 70
 - geometricDistribution, 79
 - poissonDistribution, 126
 - templateDistribution, 147
 - uniformDiscreteDistribution, 168
 - uniformDistribution, 176
- hasProperty
 - Distribution, 53
- iidSample
 - diracDistribution, 28
 - Distribution, 54
 - poissonDistribution, 126
 - templateDistribution, 147
 - uniformDiscreteDistribution, 168
 - uniformDistribution, 178
- index
 - binarySequence, 16
 - binarySimplex, 20
 - marmoteBox, 91
 - marmoteInterval, 97
 - marmoteSet, 105
- isFinite
 - marmoteInterval, 97
 - marmoteSet, 105
- isProduct
 - marmoteSet, 105
- isSimple
 - marmoteSet, 105
- isUnion
 - marmoteSet, 106
- isZero
 - binarySequence, 17
 - binarySimplex, 21
 - marmoteBox, 91
 - marmoteInterval, 98
 - marmoteSet, 106
- LAW_DESC, 83
 - Law_Parameters, 84
 - Name, 84
 - Parameters, 84
- LAW_LIST, 85
 - Next, 85
 - Val, 85
- lambda
 - poissonDistribution, 127
- laplace
 - bernoulliDistribution, 10
 - diracDistribution, 28
 - discreteDistribution, 42
 - Distribution, 54
 - exponentialDistribution, 71
 - geometricDistribution, 79
 - poissonDistribution, 127
 - templateDistribution, 148
 - uniformDiscreteDistribution, 169
 - uniformDistribution, 178
- Law_Parameters
 - LAW_DESC, 84
- markovChain
 - sparseMatrix, 143
- marmoteBox, 86
 - cardinalOutDim, 89
 - cardinalbyDim, 88
 - decodeState, 89
 - firstState, 89
 - firstStateOutDim, 90
 - firstStatebyDim, 90
 - index, 91
 - isZero, 91
 - marmoteBox, 87, 88
 - nextState, 91
 - nextStateOutDim, 93
 - nextStatebyDim, 93
 - printState, 93
- marmoteInterval, 94
 - _max, 99
 - _min, 99
 - ~marmoteInterval, 96
 - decodeState, 96
 - firstState, 97
 - index, 97
 - isFinite, 97
 - isZero, 98
 - marmoteInterval, 96
 - nextState, 98
 - printState, 98
- marmoteSet, 99

- [_cardinal](#), 108
- [_dimOffset](#), 109
- [_dimension](#), 109
- [_idxOffset](#), 109
- [_nbDimensions](#), 109
- [_nbZones](#), 109
- [_stateBuffer](#), 109
- [_totNbDims](#), 109
- [_type](#), 109
- [_zeroState](#), 110
- [_zone](#), 110
- [~marmoteSet](#), 102
- [cardinal](#), 102
- [cardinalOutDim](#), 103
- [cardinalbyDim](#), 102
- [decodeState](#), 103
- [firstState](#), 103
- [firstStateOutDim](#), 104
- [firstStatebyDim](#), 104
- [index](#), 105
- [isFinite](#), 105
- [isProduct](#), 105
- [isSimple](#), 105
- [isUnion](#), 106
- [isZero](#), 106
- [marmoteSet](#), 102
- [nextState](#), 106
- [nextStateOutDim](#), 107
- [nextStatebyDim](#), 107
- [opType](#), 101
- [printState](#), 107, 108
- [totNbDims](#), 108
- [mean](#)
 - [bernoulliDistribution](#), 11
 - [diracDistribution](#), 29
 - [discreteDistribution](#), 43
 - [Distribution](#), 54
 - [exponentialDistribution](#), 71
 - [geometricDistribution](#), 80
 - [poissonDistribution](#), 128
 - [templateDistribution](#), 148
 - [uniformDiscreteDistribution](#), 169
 - [uniformDistribution](#), 179
- [moment](#)
 - [bernoulliDistribution](#), 11
 - [diracDistribution](#), 29
 - [discreteDistribution](#), 43
 - [Distribution](#), 55
 - [exponentialDistribution](#), 72
 - [geometricDistribution](#), 80
 - [poissonDistribution](#), 128
 - [templateDistribution](#), 149
 - [uniformDiscreteDistribution](#), 169
 - [uniformDistribution](#), 179
- [multiDimHomTransition](#), 110
 - [~multiDimHomTransition](#), 114
 - [copy](#), 114
 - [dimSize](#), 114
 - [embed](#), 115
 - [evaluateMeasure](#), 115
 - [evaluateValue](#), 116
 - [getCol](#), 116
 - [getEntry](#), 116
 - [getEntryByCol](#), 117
 - [getJumpDistribution](#), 117
 - [getNbEls](#), 117
 - [getTransDistrib](#), 118
 - [multiDimHomTransition](#), 112
 - [p](#), 118
 - [q](#), 119
 - [rowSum](#), 119
 - [setEntry](#), 119
 - [uniformize](#), 120
 - [write](#), 120
- [Name](#)
 - [LAW_DESC](#), 84
- [name](#)
 - [Distribution](#), 55
- [nbEvents](#)
 - [eventMixture](#), 65
- [nbVals](#)
 - [discreteDistribution](#), 44
- [Next](#)
 - [LAW_LIST](#), 85
- [nextState](#)
 - [binarySequence](#), 17
 - [binarySimplex](#), 21
 - [marmoteBox](#), 91
 - [marmoteInterval](#), 98
 - [marmoteSet](#), 106
- [nextStateOutDim](#)
 - [marmoteBox](#), 93
 - [marmoteSet](#), 107
- [nextStatebyDim](#)
 - [marmoteBox](#), 93
 - [marmoteSet](#), 107
- [opType](#)
 - [marmoteSet](#), 101
- [origSize](#)
 - [transitionStructure](#), 159
- [p](#)
 - [geometricDistribution](#), 81
 - [multiDimHomTransition](#), 118
- [Parameters](#)
 - [LAW_DESC](#), 84
- [poissonDistribution](#), 121
 - [~poissonDistribution](#), 123
 - [ccdf](#), 123
 - [cdf](#), 124
 - [copy](#), 124
 - [dLaplace](#), 125
 - [getProba](#), 125
 - [hasMoment](#), 126
 - [iidSample](#), 126

- lambda, [127](#)
- laplace, [127](#)
- mean, [128](#)
- moment, [128](#)
- poissonDistribution, [122](#)
- rate, [129](#)
- rescale, [129](#)
- toString, [130](#)
- variance, [130](#)
- write, [131](#)
- printStats
 - binarySequence, [17](#)
 - binarySimplex, [21](#)
 - marmoteBox, [93](#)
 - marmoteInterval, [98](#)
 - marmoteSet, [107](#), [108](#)
- proba
 - bernoulliDistribution, [12](#)
- probas
 - discreteDistribution, [44](#)
- q
 - multiDimHomTransition, [119](#)
- rate
 - bernoulliDistribution, [12](#)
 - diracDistribution, [30](#)
 - discreteDistribution, [44](#)
 - Distribution, [55](#)
 - exponentialDistribution, [72](#)
 - geometricDistribution, [81](#)
 - poissonDistribution, [129](#)
 - templateDistribution, [149](#)
 - uniformDiscreteDistribution, [170](#)
 - uniformDistribution, [179](#)
- readEntry
 - transitionStructure, [159](#)
- rescale
 - bernoulliDistribution, [12](#)
 - diracDistribution, [30](#)
 - discreteDistribution, [45](#)
 - Distribution, [55](#)
 - exponentialDistribution, [72](#)
 - geometricDistribution, [82](#)
 - poissonDistribution, [129](#)
 - templateDistribution, [150](#)
 - uniformDiscreteDistribution, [170](#)
 - uniformDistribution, [180](#)
- rowSum
 - eventMixture, [66](#)
 - multiDimHomTransition, [119](#)
 - sparseMatrix, [141](#)
 - transitionStructure, [160](#)
- SCC, [131](#)
- sample
 - bernoulliDistribution, [13](#)
 - diracDistribution, [31](#)
 - discreteDistribution, [46](#)
 - Distribution, [56](#)
 - exponentialDistribution, [73](#)
 - geometricDistribution, [82](#)
 - templateDistribution, [151](#)
 - uniformDiscreteDistribution, [170](#)
 - uniformDistribution, [180](#)
- setEntry
 - eventMixture, [66](#)
 - multiDimHomTransition, [119](#)
 - sparseMatrix, [141](#)
 - transitionStructure, [160](#)
- setProba
 - discreteDistribution, [46](#)
- setType
 - transitionStructure, [161](#)
- setUniformizationRate
 - transitionStructure, [161](#)
- size
 - transitionStructure, [161](#)
- sparseMatrix, [132](#)
 - ~sparseMatrix, [134](#)
 - addToEntry, [135](#)
 - copy, [135](#)
 - diagnose, [136](#)
 - embed, [136](#)
 - evaluateMeasure, [136](#), [137](#)
 - evaluateValue, [137](#)
 - evaluateValueState, [138](#)
 - getCol, [138](#)
 - getEntry, [139](#)
 - getEntryByCol, [139](#)
 - getNbElts, [139](#)
 - getStronglyConnectedComponents, [140](#)
 - getTransDistrib, [140](#)
 - markovChain, [143](#)
 - rowSum, [141](#)
 - setEntry, [141](#)
 - sparseMatrix, [134](#)
 - uniformize, [142](#)
 - write, [142](#)
- templateDistribution, [143](#)
 - ccdf, [145](#)
 - cdf, [145](#)
 - copy, [146](#)
 - dLaplace, [146](#)
 - hasMoment, [147](#)
 - iidSample, [147](#)
 - laplace, [148](#)
 - mean, [148](#)
 - moment, [149](#)
 - rate, [149](#)
 - rescale, [150](#)
 - sample, [151](#)
 - templateDistribution, [144](#)
 - toString, [151](#)
 - variance, [151](#)
 - write, [152](#)
- toString

- bernoulliDistribution, 13
- diracDistribution, 31
- discreteDistribution, 47
- Distribution, 56
- exponentialDistribution, 73
- geometricDistribution, 83
- poissonDistribution, 130
- templateDistribution, 151
- transitionStructure, 161
- uniformDiscreteDistribution, 171
- uniformDistribution, 180
- totNbDims
 - marmoteSet, 108
- transitionStructure, 152
 - _type, 163
 - consolidate, 155
 - copy, 155
 - destSize, 155
 - embed, 156
 - evaluateMeasure, 156
 - evaluateValue, 157
 - getCol, 157
 - getEntry, 158
 - getEntryByCol, 158
 - getNbElts, 158
 - getTransDistrib, 159
 - origSize, 159
 - readEntry, 159
 - rowSum, 160
 - setEntry, 160
 - setType, 161
 - setUniformizationRate, 161
 - size, 161
 - toString, 161
 - type, 162
 - uniformizationRate, 162
 - uniformize, 162
 - write, 162
- type
 - transitionStructure, 162
- u_0_1
 - Distribution, 56
- uniformDiscreteDistribution, 163
 - ccdf, 166
 - cdf, 166
 - copy, 166
 - dLaplace, 167
 - getProba, 167
 - hasMoment, 168
 - iidSample, 168
 - laplace, 169
 - mean, 169
 - moment, 169
 - rate, 170
 - rescale, 170
 - sample, 170
 - toString, 171
 - uniformDiscreteDistribution, 165
 - valInf, 171
 - valSup, 171
 - variance, 171
 - write, 172
- uniformDistribution, 172
 - ccdf, 175
 - cdf, 175
 - copy, 176
 - dLaplace, 176
 - hasMoment, 176
 - iidSample, 178
 - laplace, 178
 - mean, 179
 - moment, 179
 - rate, 179
 - rescale, 180
 - sample, 180
 - toString, 180
 - uniformDistribution, 174
 - valInf, 181
 - valSup, 181
 - variance, 181
 - write, 181
- uniformizationRate
 - transitionStructure, 162
- uniformize
 - eventMixture, 67
 - multiDimHomTransition, 120
 - sparseMatrix, 142
 - transitionStructure, 162
- VALUE_TOLERANCE
 - Distribution, 58
- Val
 - LAW_LIST, 85
- valInf
 - uniformDiscreteDistribution, 171
 - uniformDistribution, 181
- valSup
 - uniformDiscreteDistribution, 171
 - uniformDistribution, 181
- value
 - diracDistribution, 31
- values
 - discreteDistribution, 47
- variance
 - Distribution, 57
 - poissonDistribution, 130
 - templateDistribution, 151
 - uniformDiscreteDistribution, 171
 - uniformDistribution, 181
- write
 - bernoulliDistribution, 13
 - diracDistribution, 32
 - discreteDistribution, 47
 - Distribution, 57
 - eventMixture, 67
 - exponentialDistribution, 73

geometricDistribution, [83](#)
multiDimHomTransition, [120](#)
poissonDistribution, [131](#)
sparseMatrix, [142](#)
templateDistribution, [152](#)
transitionStructure, [162](#)
uniformDiscreteDistribution, [172](#)
uniformDistribution, [181](#)