

JMarkov Reference Manual

Germán Riaño, Julio Góez, Juan Fernando Pérez y Andrés Sarmiento

October 9, 2006

Contents

Introduction	5
Class Hierarchy	6
1 Package jmarkov	9
1.1 Classes	10
1.1.1 Class DebugReporter	10
1.1.2 Class GeomProcess	13
1.1.3 Class GeomRelState	23
1.1.4 Class GeomState	26
1.1.5 Class MarkovProcess	29
1.1.6 Class MarkovProcess.Status	53
1.1.7 Class SimpleMarkovProcess	55
2 Package jmarkov.basic	59
2.1 Interfaces	61
2.1.1 Interface Actions	61
2.1.2 Interface Events	62
2.1.3 Interface JMarkovElement	63
2.1.4 Interface PropertiesElement	65
2.1.5 Interface States	67
2.1.6 Interface Transitions	69
2.2 Classes	71
2.2.1 Class Action	71
2.2.2 Class ActionsSet	73
2.2.3 Class DecisionRule	75
2.2.4 Class Event	78
2.2.5 Class EventsSet	80
2.2.6 Class Policy	82
2.2.7 Class PropertiesAction	85
2.2.8 Class PropertiesEvent	88
2.2.9 Class PropertiesState	91
2.2.10 Class Solution	95
2.2.11 Class State	96
2.2.12 Class StateC	100
2.2.13 Class StateEvent	102
2.2.14 Class StatesSet	104
2.2.15 Class Transition	108
2.2.16 Class TransitionsSet	110
2.2.17 Class ValueFunction	113

3	Package jmarkov.basic.exceptions	116
3.1	Exceptions	117
3.1.1	Class NonStochasticException	117
3.1.2	Class NotUnichainException	118
3.1.3	Class SolverException	120
3.1.4	Class StructureException	121
4	Package jmarkov.jmdp	122
4.1	Classes	123
4.1.1	Class CT2DTConverter	123
4.1.2	Class CTMDP	126
4.1.3	Class CTMDPEv	131
4.1.4	Class CTMDPEvA	135
4.1.5	Class DTMDP	139
4.1.6	Class DTMDPEv	143
4.1.7	Class DTMDPEvA	147
4.1.8	Class FiniteDP	151
4.1.9	Class FiniteMDP	153
4.1.10	Class FiniteMDPEv	157
4.1.11	Class InfiniteMDP	161
4.1.12	Class MDP	165
4.1.13	Class StochasticShortestPath	170
5	Package jmarkov.jmdp.solvers	172
5.1	Interfaces	174
5.1.1	Interface LPSolver	174
5.1.2	Interface MpsLpSolver	176
5.2	Classes	177
5.2.1	Class AbstractAverageSolver	177
5.2.2	Class AbstractDiscountedSolver	179
5.2.3	Class AbstractFiniteSolver	182
5.2.4	Class AbstractInfiniteSolver	184
5.2.5	Class AbstractTotalSolver	186
5.2.6	Class FiniteSolver	188
5.2.7	Class LPBCLAverageSolver	191
5.2.8	Class LPBCLDiscountedSolver	194
5.2.9	Class MpsLpAverageSolver	198
5.2.10	Class MpsLpDiscountedSolver	201
5.2.11	Class PolicyIterationSolver	205
5.2.12	Class ProbabilitySolver	210
5.2.13	Class RelativeValueIterationSolver	213
5.2.14	Class Solver	216
5.2.15	Class StochasticShortestPathSolver	221
5.2.16	Class ValueIterationSolver	224
6	Package jmarkov.solvers	229
6.1	Classes	230
6.1.1	Class GeometricSolver	230
6.1.2	Class GeometrixSolver	232
6.1.3	Class JamaSolver	234

6.1.4	<i>Class</i> JamaTransientSolver	236
6.1.5	<i>Class</i> MtjLogRedSolver	238
6.1.6	<i>Class</i> MtjSolver	240
6.1.7	<i>Class</i> MtjSolver.EnumPrecond	244
6.1.8	<i>Class</i> MtjSolver.EnumSolver	246
6.1.9	<i>Class</i> Solver	248
6.1.10	<i>Class</i> SteadyStateSolver	250
6.1.11	<i>Class</i> TransientSolver	252
7	Package jphase	254
7.1	Interfaces	256
7.1.1	<i>Interface</i> ContPhaseVar	256
7.1.2	<i>Interface</i> DiscPhaseVar	261
7.1.3	<i>Interface</i> PhaseVar	265
7.2	Classes	270
7.2.1	<i>Class</i> AbstractContPhaseVar	270
7.2.2	<i>Class</i> AbstractDiscPhaseVar	276
7.2.3	<i>Class</i> DenseContPhaseVar	282
7.2.4	<i>Class</i> DenseDiscPhaseVar	287
7.2.5	<i>Class</i> ErlangCoxianVar	291
7.2.6	<i>Class</i> HyperErlangVar	296
7.2.7	<i>Class</i> MarkovMatrix	301
7.2.8	<i>Class</i> MatrixUtils	307
7.2.9	<i>Class</i> PhaseVarSet	323
7.2.10	<i>Class</i> Poly	328
7.2.11	<i>Class</i> SparseContPhaseVar	330
7.2.12	<i>Class</i> SparseDiscPhaseVar	333
7.2.13	<i>Class</i> SuperErlang	336
7.2.14	<i>Class</i> Term	342
7.2.15	<i>Class</i> Utils	348
8	Package jphase.fit	351
8.1	Interfaces	353
8.1.1	<i>Interface</i> PhaseFitter	353
8.2	Classes	354
8.2.1	<i>Class</i> ContPhaseFitter	354
8.2.2	<i>Class</i> DiscPhaseFitter	356
8.2.3	<i>Class</i> EMHyperErlangFit	358
8.2.4	<i>Class</i> EMHyperExpoFit	361
8.2.5	<i>Class</i> EMPhaseFit	363
8.2.6	<i>Class</i> FitterUtils	365
8.2.7	<i>Class</i> MLContPhaseFitter	367
8.2.8	<i>Class</i> MLDiscPhaseFitter	368
8.2.9	<i>Class</i> MomentsACPH2Fit	369
8.2.10	<i>Class</i> MomentsACPHFit	371
8.2.11	<i>Class</i> MomentsADPH2Fit	373
8.2.12	<i>Class</i> MomentsContPhaseFitter	375
8.2.13	<i>Class</i> MomentsDiscPhaseFitter	377
8.2.14	<i>Class</i> MomentsECCompleteFit	379
8.2.15	<i>Class</i> MomentsECPositiveFit	381

9	Package <code>jphase.generator</code>	383
9.1	Classes	384
9.1.1	<i>Class</i> GeneratorUtils	384
9.1.2	<i>Class</i> NeutsContPHGenerator	387
9.1.3	<i>Class</i> NeutsDiscPHGenerator	389
9.1.4	<i>Class</i> PhaseGenerator	391

Introduction

jMarkov is a series of packages designed to model and optimize stochastic models. It consists of various modules:

- JMarkov allows the user to create any size Markov Models by defining the rules of the system. This is achieved extending a given class called SimpleMarkovProcess (see 1.1.7, page 55) , and implementing three functions that describe the dynamics of the system. The user does not need to know the details of the implementation, but rather to describe the dynamics of the system in terms of the states that the system can be and the events that can alter the current state. The user can have freedom to override in many ways the default behavior of the system. The basic steps that the user should follow are
 - Define what the states are by implementing the class State (see 2.2.11, page 96) or the class PropertiesState (see 2.2.9, page 91) .
 - Define what the Events are. This can be accomplished either implementing the Event (see 2.2.4, page 78) class.
 - Extend the class MarkovProcess (see 1.1.5, page 29) or SimpleMarkovProcess (see 1.1.7, page 55) , defining the three following items
 - * Define which events can occur when the system is in each state. This is accomplished by implementing the method active (see 1.1.7, page 56) .
 - * Define what are the new states when an event occurs. This is accomplished implementing the method dests (see 1.1.7, page 56) .
 - * Define what is the the rate at which each of these events occur, by implementing the method rate (see 1.1.7, page 56) .
- jQBD allows the user to model quasi-birth and death processes.
- jPhase allows the user to represent and manipulate phase type distributions. See jPhase User's Manual in the file jPhaseManual.pdf (at jPhaseManual.pdf).
- jMDP allows the user to design optimal control for discrete and continuous Markov Chains (Markov Decision Processes) and also deterministic discrete dynamic programs. See jMDP User's Manual in the file jMDPManual.pdf (at jMDPManual.pdf).

This document is a Reference Manual for jMarkov , jQBD, jMDP and jPhase. It is also available on HTML format with the package, and it is available online for easier consultation.

jMarkov is a proyect created at Universidad de los Andes by the COPA (at <http://copa.uniandes.edu.co>) research group.

Class Hierarchy

Classes

- `java.lang.Object`
 - `Jama.Matrix`
 - `jphase.MarkovMatrix` (see 7.2.7, page 301)
 - `java.lang.Enum`
 - `jmarkov.MarkovProcess.Status` (see 1.1.6, page 53)
 - `jmarkov.solvers.MtjSolver.EnumPrecond` (see 6.1.7, page 244)
 - `jmarkov.solvers.MtjSolver.EnumSolver` (see 6.1.8, page 246)
 - `jmarkov.DebugReporter` (see 1.1.1, page 10)
 - `jmarkov.MarkovProcess` (see 1.1.5, page 29)
 - `jmarkov.SimpleMarkovProcess` (see 1.1.7, page 55)
 - `jmarkov.GeomProcess` (see 1.1.2, page 13)
 - `jmarkov.basic.Action` (see 2.2.1, page 71)
 - `jmarkov.basic.PropertiesAction` (see 2.2.7, page 85)
 - `jmarkov.basic.ActionsSet` (see 2.2.2, page 73)
 - `jmarkov.basic.DecisionRule` (see 2.2.3, page 75)
 - `jmarkov.basic.Event` (see 2.2.4, page 78)
 - `jmarkov.basic.PropertiesEvent` (see 2.2.8, page 88)
 - `jmarkov.basic.EventsSet` (see 2.2.5, page 80)
 - `jmarkov.basic.Policy` (see 2.2.6, page 82)
 - `jmarkov.basic.Solution` (see 2.2.10, page 95)
 - `jmarkov.basic.State` (see 2.2.11, page 96)
 - `jmarkov.GeomRelState` (see 1.1.3, page 23)
 - `jmarkov.GeomState` (see 1.1.4, page 26)
 - `jmarkov.basic.PropertiesState` (see 2.2.9, page 91)
 - `jmarkov.basic.StateC` (see 2.2.12, page 100)
 - `jmarkov.basic.StateEvent` (see 2.2.13, page 102)
 - `jmarkov.basic.StatesSet` (see 2.2.14, page 104)
 - `jmarkov.basic.Transition` (see 2.2.15, page 108)
 - `jmarkov.basic.TransitionsSet` (see 2.2.16, page 110)
 - `jmarkov.basic.ValueFunction` (see 2.2.17, page 113)
 - `jmarkov.jmdp.MDP` (see 4.1.12, page 165)
 - `jmarkov.jmdp.FiniteMDP` (see 4.1.9, page 153)
 - `jmarkov.jmdp.FiniteDP` (see 4.1.8, page 151)
 - `jmarkov.jmdp.FiniteMDPEv` (see 4.1.10, page 157)
 - `jmarkov.jmdp.InfiniteMDP` (see 4.1.11, page 161)
 - `jmarkov.jmdp.CTMDP` (see 4.1.2, page 126)
 - `jmarkov.jmdp.CTMDPEv` (see 4.1.3, page 131)
 - `jmarkov.jmdp.CTMDPEvA` (see 4.1.4, page 135)

- `jmarkov.jmdp.DTMDP` (see 4.1.5, page 139)
 - `jmarkov.jmdp.CT2DTConverter` (see 4.1.1, page 123)
 - `jmarkov.jmdp.DTMDPEv` (see 4.1.6, page 143)
 - `jmarkov.jmdp.DTMDPEvA` (see 4.1.7, page 147)
 - `jmarkov.jmdp.StochasticShortestPath` (see 4.1.13, page 170)
- `jmarkov.jmdp.solvers.ProbabilitySolver` (see 5.2.12, page 210)
- `jmarkov.jmdp.solvers.Solver` (see 5.2.14, page 216)
 - `jmarkov.jmdp.solvers.AbstractFiniteSolver` (see 5.2.3, page 182)
 - `jmarkov.jmdp.solvers.FiniteSolver` (see 5.2.6, page 188)
 - `jmarkov.jmdp.solvers.AbstractInfiniteSolver` (see 5.2.4, page 184)
 - `jmarkov.jmdp.solvers.AbstractAverageSolver` (see 5.2.1, page 177)
 - `jmarkov.jmdp.solvers.LPBCLAverageSolver` (see 5.2.7, page 191)
 - `jmarkov.jmdp.solvers.MpsLpAverageSolver` (see 5.2.9, page 198)
 - `jmarkov.jmdp.solvers.RelativeValueIterationSolver` (see 5.2.13, page 213)
 - `jmarkov.jmdp.solvers.AbstractDiscountedSolver` (see 5.2.2, page 179)
 - `jmarkov.jmdp.solvers.LPBCLDiscountedSolver` (see 5.2.8, page 194)
 - `jmarkov.jmdp.solvers.MpsLpDiscountedSolver` (see 5.2.10, page 201)
 - `jmarkov.jmdp.solvers.PolicyIterationSolver` (see 5.2.11, page 205)
 - `jmarkov.jmdp.solvers.ValueIterationSolver` (see 5.2.16, page 224)
 - `jmarkov.jmdp.solvers.AbstractTotalSolver` (see 5.2.5, page 186)
 - `jmarkov.jmdp.solvers.StochasticShortestPathSolver` (see 5.2.15, page 221)
- `jmarkov.solvers.Solver` (see 6.1.9, page 248)
 - `jmarkov.solvers.GeometricSolver` (see 6.1.1, page 230)
 - `jmarkov.solvers.MtjLogRedSolver` (see 6.1.5, page 238)
 - `jmarkov.solvers.GeometrixSolver` (see 6.1.2, page 232)
 - `jmarkov.solvers.SteadyStateSolver` (see 6.1.10, page 250)
 - `jmarkov.solvers.JamaSolver` (see 6.1.3, page 234)
 - `jmarkov.solvers.MtjSolver` (see 6.1.6, page 240)
 - `jmarkov.solvers.TransientSolver` (see 6.1.11, page 252)
 - `jmarkov.solvers.JamaTransientSolver` (see 6.1.4, page 236)
- `jphase.AbstractContPhaseVar` (see 7.2.1, page 270)
 - `jphase.DenseContPhaseVar` (see 7.2.3, page 282)
 - `jphase.ErlangCoxianVar` (see 7.2.5, page 291)
 - `jphase.HyperErlangVar` (see 7.2.6, page 296)
 - `jphase.SparseContPhaseVar` (see 7.2.11, page 330)
- `jphase.AbstractDiscPhaseVar` (see 7.2.2, page 276)
 - `jphase.DenseDiscPhaseVar` (see 7.2.4, page 287)
 - `jphase.SparseDiscPhaseVar` (see 7.2.12, page 333)
- `jphase.MatrixUtils` (see 7.2.8, page 307)
- `jphase.PhaseVarSet` (see 7.2.9, page 323)
- `jphase.Poly` (see 7.2.10, page 328)
- `jphase.SuperErlang` (see 7.2.13, page 336)
- `jphase.Term` (see 7.2.14, page 342)
- `jphase.Utils` (see 7.2.15, page 348)
- `jphase.fit.ContPhaseFitter` (see 8.2.1, page 354)
 - `jphase.fit.MLContPhaseFitter` (see 8.2.7, page 367)
 - `jphase.fit.EMHyperErlangFit` (see 8.2.3, page 358)
 - `jphase.fit.EMHyperExpoFit` (see 8.2.4, page 361)
 - `jphase.fit.EMPhaseFit` (see 8.2.5, page 363)
 - `jphase.fit.MomentsContPhaseFitter` (see 8.2.12, page 375)

- `jphase.fit.MomentsACPH2Fit` (see 8.2.9, page 369)
- `jphase.fit.MomentsACPHFit` (see 8.2.10, page 371)
- `jphase.fit.MomentsECCompleteFit` (see 8.2.14, page 379)
- `jphase.fit.MomentsECPositiveFit` (see 8.2.15, page 381)
- `jphase.fit.DiscPhaseFitter` (see 8.2.2, page 356)
 - `jphase.fit.MLDiscPhaseFitter` (see 8.2.8, page 368)
 - `jphase.fit.MomentsDiscPhaseFitter` (see 8.2.13, page 377)
 - `jphase.fit.MomentsADPH2Fit` (see 8.2.11, page 373)
- `jphase.fit.FitterUtils` (see 8.2.6, page 365)
- `jphase.generator.GeneratorUtils` (see 9.1.1, page 384)
- `jphase.generator.PhaseGenerator` (see 9.1.4, page 391)
 - `jphase.generator.NeutsContPHGenerator` (see 9.1.2, page 387)
 - `jphase.generator.NeutsDiscPHGenerator` (see 9.1.3, page 389)

Interfaces

- `java.lang.Iterable`
 - `jmarkov.basic.Actions` (see 2.1.1, page 61)
 - `jmarkov.basic.Events` (see 2.1.2, page 62)
 - `jmarkov.basic.States` (see 2.1.5, page 67)
- `jmarkov.basic.JMarkovElement` (see 2.1.3, page 63)
 - `jmarkov.basic.PropertiesElement` (see 2.1.4, page 65)
 - `jmarkov.basic.Transitions` (see 2.1.6, page 69)
 - `jphase.PhaseVar` (see 7.1.3, page 265)
 - `jphase.ContPhaseVar` (see 7.1.1, page 256)
 - `jphase.DiscPhaseVar` (see 7.1.2, page 261)
- `jmarkov.jmdp.solvers.LPSolver` (see 5.1.1, page 174)
 - `jmarkov.jmdp.solvers.MpsLpSolver` (see 5.1.2, page 176)
- `jphase.fit.PhaseFitter` (see 8.1.1, page 353)

Exceptions

- `java.lang.Object`
 - `java.lang.Throwable`
 - `java.lang.Exception`
 - `java.lang.RuntimeException`
 - `jmarkov.basic.exceptions.NonStochasticException` (see 3.1.1, page 117)
 - `jmarkov.basic.exceptions.SolverException` (see 3.1.3, page 120)
 - `jmarkov.basic.exceptions.NotUnichainException` (see 3.1.2, page 118)
 - `jmarkov.basic.exceptions.StructureException` (see 3.1.4, page 121)

Chapter 1

Package jmarkov

Package Contents

Page

Classes

DebugReporter	10
<i>A debug reporter is used to report debug Information from a program.</i>	
GeomProcess	13
<i>The class GeomProcess represents a continuos or discrete Quasi Birth and Death process.</i>	
GeomRelState	23
<i>This class is used to build destinations which are relative to a given Geom-State.</i>	
GeomState	26
<i>The actual Geometric model is build using this class.</i>	
MarkovProcess	29
<i>The abstract class SimpleMarkovProcess represents a Continuous or Discrete Time Markov Chain.</i>	
MarkovProcess.Status	53
<i>Status variables</i>	
SimpleMarkovProcess	55

Provides the basic elements to model continuous time Markov chains (CTMC). An application should extend the class (see 1.1.5, page 29) , (see 2.2.11, page 96) and, optionaly the class (see 2.2.4, page 78) . The package also includes facility to model Matrix Geometrix Systems. For overviews, tutorials, examples, guides, and tool documentation, please see:

- JMarkov User's Manual (at ../MarkovPack.pdf).
 - COPA group WEB page. (at <http://copa.uniandes.edu.co>)
-

1.1 Classes

1.1.1 Class DebugReporter

A debug reporter is used to report debug Information from a program. It has an internal integer number, called the Debuglevel, where 0 means no debug information will be reported and 5 a lot of verbose information will be reported. The information is reported to standard output, to a `PrintWriter` (which can be associated with a file), or to a `TextPanel` which can be included in a graphic user interface.

Declaration

```
public class DebugReporter
extends java.lang.Object
```

Constructor summary

DebugReporter(int) Creates a debug reporter that will report to standard I/O.

DebugReporter(PrintWriter) Creates a debug reporter that will send its output to the given `PrintWriter`.

Method summary

debug(int, String) Reports this debug information.

debug(int, String, boolean) Reports this debug information.

debug(int, String, boolean, boolean) Reports this debug information.

getCurLevel()

getDebugLevel()

setCurLevel(int) Sets the debug level, where level=0 means no debug info, level = 5 verbose info.

setDebugLevel(int) Sets the debug level, where level=0 means no debug info, level = 5 verbose info.

Constructors

- *DebugReporter*

```
public DebugReporter( int initDebugLevel )
```

- **Description**

Creates a debug reporter that will report to standard I/O.

- **Parameters**

* `initDebugLevel` – Initial debug level

- *DebugReporter*

```
public DebugReporter( java.io.PrintWriter dbgWt )
```

- **Description**

Creates a debug reporter that will send its output to the given `PrintWriter`.

- **Parameters**

* `dbgWt` – the `PrintWriter` where the debug information will be sent.

Methods

- *debug*
`public void debug(int level, java.lang.String s)`
 - **Description**
Reports this debug information. Newline and indent are true.
 - **Parameters**
 - * **level** – Use level=0 for very important things, level=5 less important.
 - * **s** – The String to report
 - **See also**
 - * `DebugReporter.debug(int,String,boolean,boolean)` (see 1.1.1, page 11)

- *debug*
`public void debug(int level, java.lang.String s, boolean newline)`
 - **Description**
Reports this debug information. Info is indented if newline is selected.
 - **Parameters**
 - * **level** – Use level=0 for very important things, level=5 less important.
 - * **s** – The String to report
 - * **newline** – whether newline is added at the end.
 - **See also**
 - * `DebugReporter.debug(int,String,boolean,boolean)` (see 1.1.1, page 11)

- *debug*
`public void debug(int level, java.lang.String s, boolean newline, boolean indent)`
 - **Description**
Reports this debug information.
 - **Parameters**
 - * **level** – Use level=0 for very important things, level=5 less important.
 - * **s** – The String to report
 - * **newline** – whether newline is added at the end.
 - * **indent** – whwether information should go indented according to debug level.

- *getCurLevel*
`public final int getCurLevel()`
 - **Returns** – Returns the curLevel.

- *getDebugLevel*
`public synchronized int getDebugLevel()`
 - **Returns** – current debug level, where level=0 means no debug info and level = 5 verbose info.

- *setCurLevel*
`public final void setCurLevel(int curLevel)`

- **Description**

Sets the debug level, where level=0 means no debug info, level = 5 verbose info.

- **Parameters**

- * `curLevel` – The `curLevel` to set to.

- *setDebugLevel*

```
public synchronized void setDebugLevel( int level )
```

- **Description**

Sets the debug level, where level=0 means no debug info, level = 5 verbose info.

- **Parameters**

- * `level` – debug level

1.1.2 Class GeomProcess

The class GeomProcess represents a continuous or discrete Quasi Birth and Death process. This class extends the class SimpleMarkovProcess. The class generates the G matrix through the Logarithmic Reduction algorithm. The user should extend State to generate Sub-States.

Declaration

```
public abstract class GeomProcess
extends jmarkov.SimpleMarkovProcess (see 1.1.7, page 55)
```

Version

1.0

Field summary

defaultGeometrixSolver Default Solver
GeometrixSolver Current Solver

Constructor summary

GeomProcess(Sub, EventsSet) Builds a GeomProcess

Method summary

active(GeomState, E) The user cannot extend this method.
active(Sub, int, E) The user must extend this method to determine which events are active.
dests(GeomState, E) Overrides SimpleMarkovProcess' method
dests(Sub, int, E) Determines the destination set of States when events e occurs.
getAMatrices() Returns the matrices of the repeating levels, A0, A1, and A2.
getBMatrices() Returns the matrices B00, B01 and B10.
getBoundaryStates() Returns an array with the States in the boundary level.
getDefaultGeometrixSolver() Returns the default GeometrixSolver.
getEventRate(int)
getExpectedLevel() Returns the Expected Value for the Level.
getGeometrixSolver() The currently defined solver.
getInitialSol() Computes and returns the initial solution [pi(0), pi(1)].
getMOPsMoment(int, int)
getNumBoundaryStates() The Number of States in the boundary level.
getNumTypicalStates() The number of states in the typical levels.
getRmatrix() The R Matrix of the Geometric solution.
getStateClass() This returns the Sub-States class, rather than GeomState.
getStates()
getSteadyState(int) Return an array with the probabilities for the given level.
getSubMatrices(int, int, int, int) This method constructs any A_n or B_{ij} matrix existing in the process and that are necessary to calculate the R matrix.
getTypicalStates() Returns an array with the States in the typical levels.
getVectorPi0() Returns the steady State probabilities for boundary level.
getVectorPi1() Returns the steady State probabilities for level 1.

getVectorPi1Mod() Returns the steady State probabilities for level 1.
isStable() Determines if the system is stable.
matrixRtoArray()
printAll(PrintWriter)
printStates(PrintWriter, int, int)
rate(GeomState, GeomState, E)
rate(Sub, int, Sub, int, E)
reset()
setGeometrixSolver(GeometricSolver) Allows the user to set an alternate solver.
steadyProbabilities() Computes the steady state probabilities for the generated States (up to level 2).

Fields

- protected solvers.GeometricSolver **GeometrixSolver**
 - Current Solver
- protected solvers.GeometricSolver **defaultGeometrixSolver**
 - Default Solver

Constructors

- *GeomProcess*
 public **GeomProcess**(basic.State i0, basic.EventsSet eSet)
 - **Description**
 Builds a GeomProcess
 - **Parameters**
 - * i0 – Initial state. MUST be a boundary state!
 - * eSet – the event set.

Methods

- *active*
 public final boolean **active**(GeomState i, basic.Event e)
 - **Description**
 The user cannot extend this method. GeomProcess determines this based on active
 - **See also**
 - * GeomProcess.active(State,int,Event) (see 1.1.2, page 14)
 - * SimpleMarkovProcess.active(State,Event) (see 1.1.7, page 56)
-
- *active*
 public abstract boolean **active**(basic.State substate, int iLevel, basic.Event e)
 - **Description**
 The user must extend this method to determine which events are active.

– **Parameters**

- * **substate** – the current sub state
- * **iLevel** – Absolute level of current State i. You should test only whether it is 0 (boundary), 1 or greater than 1. Your code should not behave any different if the level is 2, or 3, etc
- * **e** – The event being tested.

– **Returns** – true if this event occurs

• *dests*

```
public final basic.States dests( GeomState i, basic.Event e )
```

– **Description**

Overrides SimpleMarkovProcess' method

– **Parameters**

- * **i** – current state
- * **e** – Event

– **Returns** – the destinations states

• *dests*

```
public abstract GeomRelState[] dests( basic.State i, int iLevel, basic.Event e )
```

– **Description**

Determines the destination set of States when events e occurs. It has to be implemented by the subclass.

– **Parameters**

- * **i** – current State.
- * **iLevel** – absolute level of current State. For QBD this is 0, 1 or 2. Anything above 2 should report the same result.
- * **e** – The Event that occurred.

– **Returns** – The destination States

• *getAMatrices*

```
public no.uib.cipr.matrix.Matrix[] getAMatrices( )
```

– **Description**

Returns the matrices of the repeating levels, A0, A1, and A2. If the model has not been generated it will be.

– **Returns** – [A0, A1, A2]

• *getBMatrices*

```
public no.uib.cipr.matrix.Matrix[] getBMatrices( )
```

– **Description**

Returns the matrices B00, B01 and B10. It causes the generation of the model if it has not been generated.

– **Returns** – an array with {B00,B01,B10} in that order.

• *getBoundaryStates*

```
public basic.State[] getBoundaryStates( )
```


- **Description**
Returns an array with the States in the boundary level.
 - **Returns** – an array with the Sub-States
-

- *getDefaultGeometrixSolver*

protected final solvers.GeometricSolver getDefaultGeometrixSolver()

- **Description**
Returns the default GeometrixSolver.
 - **Returns** – the default GeometrixSolver.
-

- *getEventRate*

public double getEventRate(int eNum) throws
jmarkov.basic.exceptions.NotUnichainException

- **Throws**
* jmarkov.basic.exceptions.NotUnichainException –
 - **See also**
* MarkovProcess.getEventRate(int) (see 1.1.5, page 37)
-

- *getExpectedLevel*

public double getExpectedLevel() throws
jmarkov.basic.exceptions.NotUnichainException

- **Description**
Returns the Expected Value for the Level. That is
 $L = (p_1 + 2p_2 + 3p_3 + \dots) \mathbf{1} = p_1 (\mathbf{I} - \mathbf{R})^{-2} \mathbf{1}$, where $\mathbf{1}$ is a column vector of ones.
 - **Returns** – L.
 - **Throws**
* jmarkov.basic.exceptions.NotUnichainException –
-

- *getGeometrixSolver*

public solvers.GeometricSolver getGeometrixSolver()

- **Description**
The currently defined solver.
 - **Returns** – Returns the GeometrixSolver.
 - **See also**
* solvers.GeometricSolver (see 6.1.1, page 230)
-

- *getInitialSol*

public double[] getInitialSol() throws
jmarkov.basic.exceptions.NotUnichainException

- **Description**
Computes and returns the initial solution [pi(0), pi(1)].
 - **Returns** – an array with the initial solution [pi(0), pi(1)]
 - **Throws**
* jmarkov.basic.exceptions.NotUnichainException –
-

- *getMOPsMoment*

public double **getMOPsMoment**(int mopNum, int m) throws
jmarkov.basic.exceptions.NotUnichainException

- **Throws**

* jmarkov.basic.exceptions.NotUnichainException –

- **See also**

* MarkovProcess.getMOPsMoment(int,int) (see 1.1.5, page 40)

- *getNumBoundaryStates*

public int **getNumBoundaryStates**()

- **Description**

The Number of States in the boundary level.

- **Returns** – the Number of States in the boundary level.

- *getNumTypicalStates*

public int **getNumTypicalStates**()

- **Description**

The number of states in the typical levels.

- **Returns** – the number of states in the typical levels.

- *getRmatrix*

public no.uib.cipr.matrix.Matrix **getRmatrix**() throws
jmarkov.basic.exceptions.NotUnichainException

- **Description**

The R Matrix of the Geometric solution. This matrix solves
 $\mathbf{A}_0 + \mathbf{R}\mathbf{A}_1 + \mathbf{R}^2\mathbf{A}_2 = \mathbf{0}$

- **Returns** – The **R** Matrix of the Geometric solution. If the system is not stable it returns a zero matrix.

- **Throws**

* jmarkov.basic.exceptions.NotUnichainException –

- *getStateClass*

public java.lang.Class **getStateClass**()

- **Description**

This return the Sub-States class, rather than GeomState.

- **See also**

* MarkovProcess.getStateClass() (see 1.1.5, page 42)

- *getStates*

public basic.StatesSet **getStates**()

- **Description copied from MarkovProcess** (see 1.1.5, page 29)

Returns an array with all the States in the model. It generates the model if it has not been generated.

- **Returns** – The States

- *getSteadyState*

`public double[] getSteadyState(int level) throws
jmarkov.basic.exceptions.NotUnichainException`

- **Description**

Return an array with the probabilities for the given level.

- **Parameters**

* `level` –

- **Returns** – probabilities array `pi(k)`.

- **Throws**

* `jmarkov.basic.exceptions.NotUnichainException` –

- *getSubMatrices*

`public no.uib.cipr.matrix.Matrix getSubMatrices(int rowMin, int rowMax,
int colMin, int colMax)`

- **Description**

This method constructs any A_n or B_{ij} matrix existing in the process and that are necessary to calculate R matrix.

- **Parameters**

* `rowMax` – The upper limit row index.

* `rowMin` – The lower limit row index.

* `colMax` – The upper limit column index.

* `colMin` – The lower limit column index.

- **Returns** – any A matrix in the process.

- *getTypicalStates*

`public basic.State[] getTypicalStates()`

- **Description**

Returns an array with the States in the typical levels.

- **Returns** – an array with the Sub-States

- *getVectorPi0*

`public double[] getVectorPi0() throws
jmarkov.basic.exceptions.NotUnichainException`

- **Description**

Returns the steady State probabilities for boundary level.

- **Returns** – the array `pi(0)`

- **Throws**

* `jmarkov.basic.exceptions.NotUnichainException` –

- *getVectorPi1*

`public double[] getVectorPi1() throws
jmarkov.basic.exceptions.NotUnichainException`

- **Description**

Returns the steady State probabilities for level 1.

- **Returns** – the array `pi(1)`

- **Throws**

* jmarkov.basic.exceptions.NotUnchainException –

- *getVectorPi1Mod*

public double[] **getVectorPi1Mod**() throws
jmarkov.basic.exceptions.NotUnchainException

- **Description**

Returns the steady State probabilities for level 1.

- **Returns** – the array $\pi(1)(I-R)^{-1}$

- **Throws**

* jmarkov.basic.exceptions.NotUnchainException –

- *isStable*

public boolean **isStable**()

- **Description**

Determines if the system is stable.

- **Returns** – true if the system is stable.

- *matrixRtoArray*

public double[][] **matrixRtoArray**() throws
jmarkov.basic.exceptions.NotUnchainException

- **Returns** – Matrix R in an array of doubles.

- **Throws**

* jmarkov.basic.exceptions.NotUnchainException –

- *printAll*

public void **printAll**(java.io.PrintWriter out)

- **Description copied from MarkovProcess** (see 1.1.5, page 29)

Prints to the given PrintWriter a summary of the information related to this MarkovChain. The information is the same as as in the method printAll().

- **Parameters**

* out –

- **See also**

* MarkovProcess.toString() (see 1.1.5, page 51)

* MarkovProcess.printAll() (see 1.1.5, page 46)

- *printStates*

public void **printStates**(java.io.PrintWriter out, int width, int
probDecimals)

- **Description copied from MarkovProcess** (see 1.1.5, page 29)

Prints a description of the States and the Equilibrium Probabilities.

- **Parameters**

* out – The writer to write to.

* width – The width of each column.

* probDecimals – The number of decimals for the probabilities.

- *rate*

```
public double rate( GeomState i, GeomState j, basic.Event e )
```
- *rate*

```
public abstract double rate( basic.State i, int ilevel, basic.State j, int jLevel, basic.Event e )
```

 - **Parameters**
 - * *i* – current sub state
 - * *ilevel* – current state’s absolute level
 - * *j* – destination sub state
 - * *jLevel* – destination level
 - * *e* – Event
 - **Returns** – rate of occurrence

- *reset*

```
public synchronized void reset( )
```

 - **Description copied from MarkovProcess** (see 1.1.5, page 29)
Resets the Model. It erases all found states and transition rates. Keeps the initial state and Events set.

- *setGeometrixSolver*

```
public void setGeometrixSolver( solvers.GeometricSolver geometrixSolver )
```

 - **Description**
Allows the user to set an alternate solver.
 - **Parameters**
 - * *geometrixSolver* – The GeometrixSolver to set.
 - **See also**
 - * *solvers.GeometricSolver* (see 6.1.1, page 230)

- *steadyProbabilities*

```
public double[] steadyProbabilities( ) throws jmarkov.basic.exceptions.NotUnichainException
```

 - **Description**
Computes the steady state probabilities for the generated States (up to level 2).
 - **Returns** – (pi(0), pi(1), pi(2)).
 - **Throws**
 - * *jmarkov.basic.exceptions.NotUnichainException* –

Members inherited from class jmarkov.SimpleMarkovProcess (see 1.1.7, page 55)

- `public abstract boolean active(basic.State i, basic.Event e)`
- `public final Transitions activeTransitions(basic.State i, basic.Event e)`
- `public abstract States dests(basic.State i, basic.Event e)`
- `public abstract double rate(basic.State i, basic.State j, basic.Event e)`

Members inherited from class `jmarkov.MarkovProcess` (see 1.1.5, page 29)

- `public abstract Transitions activeTransitions(basic.State i, basic.Event e)`
- `public boolean addMOP(java.lang.String mopName)`
- `public String allToString()`
- `public boolean canGo()`
- `public void clearMOPs()`
- `protected cnt`
- `public void debug(int level, java.lang.String s)`
- `public void debug(int level, java.lang.String s, boolean newline)`
- `public void debug(int level, java.lang.String s, boolean newline, boolean indent)`
- `protected defaultSteadyStateSolver`
- `protected defaultTransientSolver`
- `public String denseMatrixToString()`
- `public String denseMatrixToString(int width, int rateDecimals, boolean printZeros, boolean useGenerator)`
- `public abstract String description()`
- `public String eventRatesToString(int width, int decimals)`
- `public String eventsRatesToString()`
- `protected void finalize() throws java.lang.Throwable`
- `public void generate()`
- `public int getDebugLevel()`
- `public DebugReporter getDebugReporter()`
- `protected final SteadyStateSolver getDefaultSteadyStateSolver()`
- `protected TransientSolver getDefaultTransientSolver()`
- `public Class getEventClass()`
- `public String getEventNames()`
- `public double getEventRate(int eNum) throws basic.exceptions.NotUnichainException`
- `public Event getEvents()`
- `public double getEventsRates() throws basic.exceptions.NotUnichainException`
- `public synchronized double getFinalRate(basic.State i, basic.State j)`
- `public double getGenerator()`
- `protected State getInitialState()`
- `public long getMaxStates()`
- `public int getMOPIndex(java.lang.String name)`
- `public String getMOPNames()`
- `public String getMOPNames(int mopNum)`
- `public double getMOPsAvg() throws basic.exceptions.NotUnichainException`
- `public double getMOPsAvg(int mopNum) throws basic.exceptions.NotUnichainException`
- `public double getMOPsAvg(java.lang.String mopName) throws basic.exceptions.NotUnichainException`
- `public double getMOPsMoment(int m) throws basic.exceptions.NotUnichainException`
- `public double getMOPsMoment(int mopNum, int m) throws basic.exceptions.NotUnichainException`
- `public double getMOPsMoment(java.lang.String mopName, int m) throws basic.exceptions.NotUnichainException`
- `public Matrix getMtjGenerator()`
- `public synchronized Matrix getMtjRates()`
- `public int getNumStates()`
- `public long getProgress()`
- `public synchronized double getRate(basic.State i, basic.State j)`
- `public synchronized double getRates()`
- `public synchronized Transitions getRates(basic.State i)`
- `public Class getStateClass()`
- `public StatesSet getStates()`
- `public StatesSet getStates(boolean causesGeneration)`
- `public MarkovProcess.Status getStatus()`
- `public String getStatusMsg()`
- `public double getSteadyState() throws basic.exceptions.NotUnichainException`
- `public SteadyStateSolver getSteadyStateSolver()`
- `public TransientSolver getTransientSolver()`
- `public synchronized void go()`
- `public synchronized void goStep()`
- `public void hideGUI()`

- protected String hLine(int length)
- public boolean isGenerated()
- public void killGUI()
- public String label()
- public void loadGUI()
- public String MOPsToString()
- public String MOPsToString(int width, int decimals)
- protected name
- public int numMOPs()
- protected String pad(double v, int w)
- protected String pad(double v, int w, boolean right)
- protected String pad(double v, int w, int d)
- protected String pad(double v, int w, int d, boolean right)
- protected String pad(java.lang.String s, int w)
- protected String pad(java.lang.String s, int w, boolean right)
- public void pause()
- public void printAll()
- public void printAll(java.io.PrintWriter out)
- public void printDenseMatrix(java.io.PrintWriter out)
- public void printDenseMatrix(java.io.PrintWriter out, int width, int rateDecimals, boolean printZeros, boolean useGenerator)
- protected void printDenseMatrix(java.io.PrintWriter out, int width, int rateDecimals, boolean printZeros, boolean useGenerator, int[] idx)
- public void printEventsRates(java.io.PrintWriter out)
- public void printEventsRates(java.io.PrintWriter out, int width, int decimals)
- public final void printMOPs()
- public final void printMOPs(java.io.PrintWriter out)
- public int printMOPs(java.io.PrintWriter out, int width, int decimals)
- public void printStates(java.io.PrintWriter out)
- public void printStates(java.io.PrintWriter out, int width, int probDecimals)
- public synchronized void reset()
- protected synchronized void reset(boolean resetEvents)
- public synchronized void resetResults()
- public void setDebugLevel(int level)
- public void setDebugReporter(DebugReporter reporter)
- protected void setEventSet(basic.EventsSet eSet)
- protected void setInitialState(basic.State i0)
- public void setMaxStates(long num)
- public void setMOPs(java.lang.String[] mopNames)
- public void setSteadyStateSolver(solvers.SteadyStateSolver steadyStateSolver)
- public void setTransientSolver(solvers.TransientSolver transientSolver)
- public void showGUI()
- public int statesLableMaxWidth(int width)
- public String statesToString()
- protected theStates
- public String toString()
- protected String vLine()

1.1.3 *Class* **GeomRelState**

This class is used to build destinations which are relative to a given **GeomState**. The user should not extend this class, but rather use it when building destination states.

Declaration

```
public final class GeomRelState
  extends jmarkov.basic.State (see 2.2.11, page 96)
```

Field summary

boundary Whether it is boundary
rLevel Relative Level
subState subState represents the background states in every rLevel.

Constructor summary

GeomRelState(Sub) Creates a boundary **GeomState** with the given relative level rLevel, and subState.
GeomRelState(Sub, int) Creates a Non boundary **GeomState** with the given relative level rLevel, and subState.

Method summary

compareTo(State) Compares **GeomStates** according to rLevel first and then according to the subStates comparator.
computeMOPs(MarkovProcess)
getRelLevel()
getSubState()
isBoundary() This method determines if the State is a boundary state.
isConsistent()
label()

Fields

- protected int **rLevel**
 - Relative Level
- protected boolean **boundary**
 - Whether it is boundary
- protected basic.State **subState**
 - subState represents the background states in every rLevel.

Constructors

- *GeomRelState*
`public GeomRelState(basic.State subState)`
 - **Description**
Creates a boundary `GeomState` with the given relative level `rLevel`, and `subState`.
 - **Parameters**
 - * `subState` –

- *GeomRelState*
`public GeomRelState(basic.State subState, int rLevel)`
 - **Description**
Creates a Non boundary `GeomState` with the given relative level `rLevel`, and `subState`.
 - **Parameters**
 - * `rLevel` –
 - * `subState` –

Methods

- *compareTo*
`public int compareTo(basic.State s)`
 - **Description**
Compares `GeomStates` according to `rLevel` first and then according to the `subStates` comparator.
 - **Parameters**
 - * `s` – state to compare to.

- *computeMOPs*
`public void computeMOPs(MarkovProcess mp)`
 - **See also**
 - * `basic.State.computeMOPs(MarkovProcess)` (see 2.2.11, page 97)

- *getRelLevel*
`public int getRelLevel()`
 - **Returns** – Returns the `rLevel`.

- *getSubState*
`public basic.State getSubState()`
 - **Returns** – Returns the `subState`.

- *isBoundary*
`public boolean isBoundary()`
 - **Description**
This method determines fi the `State` is a boundary state.

– **Returns** – Whether it is Boundary

- *isConsistent*

public boolean **isConsistent**()

– **See also**

* `basic.State.isConsistent()` (see 2.2.11, page 98)

- *label*

public java.lang.String **label**()

– **See also**

* `basic.State.label()` (see 2.2.11, page 98)

Members inherited from class `jmarkov.basic.State` (see 2.2.11, page 96)

- public abstract int **compareTo**(State j)
- public abstract void **computeMOPs**(jmarkov.MarkovProcess model)
- public String **description**()
- public final boolean **equals**(java.lang.Object o)
- public final int **getIndex**()
- public double **getMOP**(int index)
- public final double **getMOP**(java.lang.String mopName, jmarkov.MarkovProcess model)
- public abstract boolean **isConsistent**()
- public abstract String **label**()
- public final int **setMOP**(int index, double value)
- public int **setMOP**(jmarkov.MarkovProcess model, java.lang.String mopName, double value)
- public final String **toString**()

1.1.4 *Class* **GeomState**

The actual Geometric model is build using this class. The user normally does not have to manipulate this class.

Declaration

```
public final class GeomState
extends jmarkov.basic.State (see 2.2.11, page 96)
```

Field summary

level This represents the relative level.

subState subState represnts the background states in every level.

Constructor summary

GeomState(Sub, int) Creates a GeomState with the given level, ans subState.

Method summary

compareTo(State) Compares GeomStates according to level first and then according to the subStates comparator.

computeMOPs(MarkovProcess)

description()

getLevel()

getMOP(int)

getSubState()

isBoundary()

isConsistent()

label()

setMOP(MarkovProcess, String, double)

Fields

- protected int **level**
 - This represents the relative level.
- protected basic.State **subState**
 - subState represnts the background states in every level.

Constructors

- *GeomState*

```
public GeomState( basic.State subState, int level )
```

 - **Description**

Creates a GeomState with the given level, ans subState.
 - **Parameters**

- * `level` –
- * `subState` –

Methods

- *compareTo*
`public int compareTo(basic.State s)`
 - **Description**
Compares GeomStates according to level first and then according to the subStates comparator.
 - **Parameters**
 - * `s` – state to compare to.

- *computeMOPs*
`public void computeMOPs(MarkovProcess mp)`
 - **See also**
 - * `basic.State.computeMOPs(MarkovProcess)` (see 2.2.11, page 97)

- *description*
`public java.lang.String description()`
 - **Description copied from basic.State** (see 2.2.11, page 96)
Returns a String that describes the State. By default it is an empty string, but you should implement it in order to get a meaningful description.
 - **Returns** – A String description of the State

- *getLevel*
`public int getLevel()`
 - **Returns** – Returns the level.

- *getMOP*
`public double getMOP(int index)`
 - **See also**
 - * `basic.State.getMOP(int)` (see 2.2.11, page 98)

- *getSubState*
`public basic.State getSubState()`
 - **Returns** – Returns the subState.

- *isBoundary*
`public boolean isBoundary()`
 - **Returns** – `true` if this state is level 0.

- *isConsistent*
`public boolean isConsistent()`

– See also

* `basic.State.isConsistent()` (see 2.2.11, page 98)

- *label*

`public java.lang.String label()`

– See also

* `basic.State.label()` (see 2.2.11, page 98)

- *setMOP*

`public int setMOP(MarkovProcess mp, java.lang.String mopName, double value)`

– See also

* `basic.State.setMOP(MarkovProcess,String,double)` (see 2.2.11, page 99)

Members inherited from class `jmarkov.basic.State` (see 2.2.11, page 96)

- `public abstract int compareTo(State j)`
- `public abstract void computeMOPs(jmarkov.MarkovProcess model)`
- `public String description()`
- `public final boolean equals(java.lang.Object o)`
- `public final int getIndex()`
- `public double getMOP(int index)`
- `public final double getMOP(java.lang.String mopName, jmarkov.MarkovProcess model)`
- `public abstract boolean isConsistent()`
- `public abstract String label()`
- `public final int setMOP(int index, double value)`
- `public int setMOP(jmarkov.MarkovProcess model, java.lang.String mopName, double value)`
- `public final String toString()`

1.1.5 Class MarkovProcess

The abstract class SimpleMarkovProcess represents a Continuous or Discrete Time Markov Chain. In order to model a particular problem the user has to extend this class. The class can generate the model through the buildRS algorithm. This enables it to generate all states and the transition matrix, from behavior rule given by the user. These rules are determined by implementing the methods, `active`, `dests` and `rate`. The user should also determine how to code the space state. This is accomplished by implementing the State class. A particular implementation of State is provided where each state is coded with k integer properties. Examples are included in this release.

See also

- `SimpleMarkovProcess.dests(State,Event)` (see 1.1.7, page 56)
- `SimpleMarkovProcess.active(State,Event)` (see 1.1.7, page 56)
- `SimpleMarkovProcess.rate(State,State,Event)` (see 1.1.7, page 56)
- `basic.State` (see 2.2.11, page 96)
- `basic.Event` (see 2.2.4, page 78)
- `basic.PropertiesState` (see 2.2.9, page 91)

Declaration

```
public abstract class MarkovProcess
extends java.lang.Object
implements jmarkov.basic.JMarkovElement
```

Version

1.0a

All known subclasses

SimpleMarkovProcess (see 1.1.7, page 55) , GeomProcess (see 1.1.2, page 13)

Field summary

cnt Number of completed states.
defaultSteadyStateSolver Default Transient Solver
defaultTransientSolver Deafault Transient solver
name The name of the model.
theStates Set of fully analyzed States

Constructor summary

MarkovProcess() If a constructor calls this constructor then it MUST call `setEvents` and `setInitialState` afterwards.
MarkovProcess(S, EventsSet) Builds a SimpleMarkovProcess that contains all states reachable from `i0`, and with `E` being the set of all possible events.
MarkovProcess(S, EventsSet, String) Builds a SimpleMarkovProcess that contains all states reachable from `i0`, and with `E` being the set of all possible events.

Method summary

activeTransitions(S, E) The user MUST implement this Function in order to describe the dynamics of the model.

addMOP(String) This method declares the existence of a measure of performance (MOP).

allToString() Returns a String description of the model and solution.

canGo() Allows to stop model execution by graphical user interface.

clearMOPs() Clear all MOPs defined in the system.

debug(int, String) Prints debug information with this importance level

debug(int, String, boolean) Prints debug information with this importance level

debug(int, String, boolean, boolean) Prints debug information with this importance level

denseMatrixToString() Returns a String with a description of the Model: the States and the Transition Matrix.

denseMatrixToString(int, int, boolean, boolean) Returns the Transition Matrix as a String.

description() This method should be implemented by the subclass to give word description of the model.

eventRatesToString(int, int) Return a String as printed by printEventsrates

eventsRatesToString() Return a string as eventsRatesToString, with width 8 and 4 decimals

finalize()

generate() generate() builds the space state and rate matrix using the algorithm BuildsSR.

getDebugLevel()

getDebugReporter() Gets the DebugReporter currently in use.

getDefaultSteadyStateSolver() Returns the default SteadyStateSolver.

getDefaultTransientSolver() The default solver for transient state.

getEventClass() The Class for the Events in the system.

getEventNames() Returns the defined events

getEventRate(int) Return the steadystate rate of occurrence of the Events number eNum.

getEvents() Returns all The events defined in the model.

getEventsRates() Return an array with the steadystate rate of occurrence of all the Events.

getFinalRate(S, S) Gets the total rate from State number i to j.

getGenerator() Returns the infinitesimal generator matrix \mathbf{Q} , in dense format.

getInitialState() returns the initial state.

getMaxStates()

getMOPIndex(String) Gets the index that correspond to this MOP.

getMOPNames() Return all the names of defined MOPs.

getMOPNames(int) Return the names of the i-th MOP.

getMOPsAvg() Returns an array with the average of all the steady state measures of performance.

getMOPsAvg(int) Returns the steady state measures average of the MOP number mopNum.

getMOPsAvg(String) Returns the steady state measures average of the MOP with name mopName.

getMOPsMoment(int) Returns an array with the m-th moment of all the steady

state measures of performance.

getMOPsMoment(int, int) Returns the steady state measures m-th moment of the MOP number mopNum.

getMOPsMoment(String, int) Returns the steady state measures m-th moment of the MOP with name mopName.

getMtjGenerator() The generator **Q** as an MTJ Matrix

getMtjRates() Returns the transition rates matrix **R** in MTJ format.

getNumStates() Return the number of States in the model.

getProgress() Return the number of states processed so far in the current process.

getRate(S, S) Gets the current total rate from i to j.

getRates() Returns the transition rates matrix **R** in dense format.

getRates(S) This method returns a dynamic data structure with the rate from State i to all reachable states.

getStateClass() The Class for the states in this model.

getStates() Returns an array with all the States in the model.

getStates(boolean) Returns an array with the States in the model that have been checked so far.

getStatus() Returns the current status of the model.

getStatusMsg() Returns a String describing the current status of the model.

getSteadyState() Returns the steady state probabilities for this model.

getSteadyStateSolver() The currently defined solver.

getTransientSolver() The currently defined solver for transient state.

go() Runs the model, or resumes execution if it had been suspended.

goStep() Runs the model for a single step.

hideGUI() Hides the Graphic User Interface (GUI) that represent this Markov Chain if one is defined.

hLine(int) Returns an horizontal text line of the given length.

isGenerated()

killGUI() Destroys the Graphic User Interface (GUI) that represent this Markov Chain if one is defined.

label() Returns the name of the model.

loadGUI() Loads the Graphic User Interface (GUI) that represent this Markov Chain.

MOPsToString() Return a String description of all MOPs in steady state (it reports mean and standard deviation).

MOPsToString(int, int) Return a String description of all MOPs in steady state (it reports mean and standard deviation).

numMOPs() Returns the number of defined Measures of performance (MOPs).

pad(double, int) pad generates a string representing the double v, padded with spaces up to width w.

pad(double, int, boolean) pad generates a string representing the double v, padded with spaces up to width w.

pad(double, int, int) pad fills with blanks up to width w.

pad(double, int, int, boolean) pad fills with blanks up to width w

pad(String, int) pad fills with blanks up to width w

pad(String, int, boolean) pad fills with blanks up to width w

pause() Pauses the current execution of the model.

printAll() Prints a description of the Model: the States and the Transition Matrix.

printAll(PrintWriter) Prints to the given PrintWriter a summary of the information related to this MarkovChain.

printDenseMatrix(PrintWriter) Prints a the Transition Matrix.

printDenseMatrix(PrintWriter, int, int, boolean, boolean) Prints a description of the Model using the given PrintWriter: the States and the Transition Matrix.

printDenseMatrix(PrintWriter, int, int, boolean, boolean, int[]) Prints a description of the Model using the given PrintWriter: the States and the Transition Matrix.

printEventsRates(PrintWriter) Prints a table reporting the steadystate occurrence of all events.

printEventsRates(PrintWriter, int, int) Prints a table reporting the steadystate occurrence of all events.

printMOPs() Prints the Measures of performance (MOPS) on standard output.

printMOPs(PrintWriter) Prints a String description of all MOPs in steady state (it reports mean and standard deviation), with a width of 10 and 5 decimal figures.

printMOPs(PrintWriter, int, int) Prints a String description of all MOPs in steady state (it reports mean and standard deviation).

printStats(PrintWriter) Prints a description of the States and the Equilibrium Probabilities.

printStats(PrintWriter, int, int) Prints a description of the States and the Equilibrium Probabilities.

reset() Resets the Model.

reset(boolean) Resets the Model.

resetResults() Resets the result of the model.

setDebugLevel(int) Sets the debug level, where level=0 means no debug info, level = 5 verbose info.

setDebugReporter(DebugReporter) Sets the DebugReporter to use.

setEventSet(EventsSet) Sets the Events set.

setInitialState(S) Sets the initial state.

setMaxStates(long) Sets the maximum number of states to generate.

setMOPs(String[]) Sets the names of all MOPs (measures of performance).

setSteadyStateSolver(SteadyStateSolver) Allows the user to set an alternate solver.

setTransientSolver(TransientSolver) Allows the user to set an alternate solver.

showGUI() Shows the Graphic User Interface (GUI) that represent this Markov Chain.

statesLableMaxWidth(int) Computes the maximum used by the state's labels.

statesToString() Prints a description of the States and the Equilibrium Probabilities.

toString()

vLine() Returns a text vertical line.

Fields

- protected int **cnt**
 - Number of completed states. Use this counter so that GUI updates correctly.
- protected basic.StatesSet **theStates**
 - Set of fully analyzed States

- protected java.lang.String **name**
 - The name of the model. For a long description override `description()`.
 - See also
 - * `MarkovProcess.description()` (see 1.1.5, page 36)
- protected solvers.SteadyStateSolver **defaultSteadyStateSolver**
 - Default Transient Solver
- protected solvers.TransientSolver **defaultTransientSolver**
 - Default Transient solver

Constructors

- *MarkovProcess*
protected **MarkovProcess**()
 - **Description**
If a constructor calls this constructor then it MUST call `setEvents` and `setInitialState` afterwards.

- *MarkovProcess*
public **MarkovProcess**(basic.State **i0**, basic.EventsSet **eSet**)
 - **Description**
Builds a SimpleMarkovProcess that contains all states reachable from `i0`, and with `E` being the set of all possible events.
 - **Parameters**
 - * `i0` – The initial State.
 - * `eSet` – The set of all Events.

- *MarkovProcess*
public **MarkovProcess**(basic.State **i0**, basic.EventsSet **eSet**, java.lang.String **name**)
 - **Description**
Builds a SimpleMarkovProcess that contains all states reachable from `i0`, and with `E` being the set of all possible events.
 - **Parameters**
 - * `i0` – The initial State.
 - * `eSet` – The set of all Events.
 - * `name` – The name of the Model.

Methods

- *activeTransitions*
public abstract basic.Transitions **activeTransitions**(basic.State **i**, basic.Event **e**)

– **Description**

The user MUST implement this Function in order to describe the dynamics of the model. For the current state *i*, and on action *e*, the user has to describe the transtions that can occur. This implies finding all destination states and the rate at which the transtions occur. There is no guarantee that the event is active, so the user should check for this. If the event is not active an empty Transition element should be returned. A typical code for a queuing system should look like this: `public abstract Transitions activeTransitions(MyState i, MyEvent e){ TransitionsSet trans = new TransitionsSet(); case (ARRIVAL) if (i.size() capacity) trans.add(i.doArrival(), arrRate); break; case(DEPARTURE) if (i.size() >=1) trans.add(i.doDeparture, serviceRate); break; } return trans; }`

– **Parameters**

- * *i* – The current State.
- * *e* – The occurring event.

– **Returns** – The transitions that occur at this state when (and if) this events occurs.

– **See also**

- * `basic.Transitions` (see 2.1.6, page 69)
- * `basic.TransitionsSet` (see 2.2.16, page 110)
- * `basic.Transitions` (see 2.1.6, page 69)

• *addMOP*

`public boolean addMOP(java.lang.String mopName)`

– **Description**

This method declares the existance of a measure of performance (MOP). The MOP for every state is calculated in the class that extends the State class.

– **Parameters**

- * *mopName* – The name of the new MOP.

– **Returns** – true if the name already existed.

• *allToString*

`public java.lang.String allToString()`

– **Description**

Retuns a String description of the model and solution.

– **Returns** – a String wit the information of printAll.

– **See also**

- * `MarkovProcess.printAll()` (see 1.1.5, page 46)

• *canGo*

`public boolean canGo()`

– **Description**

Allos to stop model execution by graphica user interface.

– **Returns** – Used to check with the GUI if the user has requested to stop.

• *clearMOPs*

`public void clearMOPs()`

– **Description**

Clear all MOPs defined in the system.

- *debug*

```
public void debug( int level, java.lang.String s )
```

- **Description**

Prints debug information with this importance level

- **Parameters**

- * **level** –
 - * **s** – The message to send.
-

- *debug*

```
public void debug( int level, java.lang.String s, boolean newline )
```

- **Description**

Prints debug information with this importance level

- **Parameters**

- * **level** – The level of importance (0=show always, 5= show on debug level is 5).
 - * **s** – The message
 - * **newline** – Whether a new line should be written.
-

- *debug*

```
public void debug( int level, java.lang.String s, boolean newline, boolean indent )
```

- **Description**

Prints debug information with this importance level

- **Parameters**

- * **level** – The level of importance (0=show always, 5= show on debug level is 5).
 - * **s** – The string to write.
 - * **newline** – Whether to use a new line.
 - * **indent** – Whether it should indent according to level.
-

- *denseMatrixToString*

```
public java.lang.String denseMatrixToString( )
```

- **Description**

Returns a String with a description of the Model: the States and the Transition Matrix. Its use is not recommended for large models.

- **Returns** – A string with the Matrix

- *denseMatrixToString*

```
public java.lang.String denseMatrixToString( int width, int rateDecimals, boolean printZeros, boolean useGenerator )
```

- **Description**

Returns the Transition Matrix as a String. Its use is not recommended for large models.

- **Parameters**

- * **width** – The width of each column.
- * **rateDecimals** – The number of decimals for the rates.
- * **printZeros** – Whether zeros or blanks should be printed.

- * **useGenerator** – whether the generator matrix **Q**, rather than the rates matrix should be printed.
 - **Returns** – A String description of the rates or generator matrix.

- *description*
 public abstract java.lang.String **description**()
 - **Description**
 This method should be implemented by the subclass to give word description of the model. For example it should say: Queueing system with 2 servers and exponential arrivals with rate 50 per hour.
 - **Returns** – A description of the Model

- *eventRatesToString*
 public java.lang.String **eventRatesToString**(int width, int decimals)
 - **Description**
 Return a String as printed by printEventsrates
 - **Parameters**
 - * **width** – Maximum width for each number
 - * **decimals** – Number of decimals
 - **Returns** – A string with the value of all Rates
 - **See also**
 - * `MarkovProcess.printEventsRates(PrintWriter,int,int)` (see 1.1.5, page 48)

- *eventsRatesToString*
 public java.lang.String **eventsRatesToString**()
 - **Description**
 Return a string as `eventsRatesToString`, with width 8 and 4 decimals
 - **Returns** – A string with the information.
 - **See also**
 - * `MarkovProcess.eventRatesToString(int,int)` (see 1.1.5, page 36)

- *finalize*
 protected void **finalize**() throws java.lang.Throwable

- *generate*
 public void **generate**()
 - **Description**
`generate()` builds the space state and rate matrix using the algorithm BuildsSR. (See Ciardo, G. "Tools for formulating Markov Processes", chapter 2 in Grassman W. "Computational Probability". Kluwer). The states can be collected later with `getStates` and the rates can be accessed in disperse form with the method `getRate(j)` of every state. Alternatively the method `getGenerator()` and `getRates()` access the generator matrix or rate matrix in compact form.

- *getDebugLevel*
 public int **getDebugLevel**()

- **Returns** – current debug level, where level=0 means no debug info and level = 4 verbose info.

- *getDebugReporter*

`public DebugReporter getDebugReporter()`

- **Description**
Gets the DebugReporter currently in use.
- **Returns** – a DebugReporter where debug information is sent.
- **See also**
* `DebugReporter` (see 1.1.1, page 10)

- *getDefaultSteadyStateSolver*

`protected final solvers.SteadyStateSolver getDefaultSteadyStateSolver()`

- **Description**
Returns the default SteadyStateSolver.
- **Returns** – the default SteadyStateSolver.

- *getDefaultTransientSolver*

`protected solvers.TransientSolver getDefaultTransientSolver()`

- **Description**
The default solver for transient state.
- **Returns** – Returns the default transientSolver.
- **See also**
* `solvers.TransientSolver` (see 6.1.11, page 252)

- *getEventClass*

`public java.lang.Class getEventClass()`

- **Description**
The Class for the Events in the system.
- **Returns** – The event Class in the model

- *getEventNames*

`public java.lang.String[] getEventNames()`

- **Description**
Returns the defined events
- **Returns** – an array with the names of all Events

- *getEventRate*

`public double getEventRate(int eNum) throws
jmarkov.basic.exceptions.NotUnichainException`

- **Description**
Return the steadystate rate of occurrence of the Events number eNum.
- **Parameters**
* `eNum` – The even number in the event set.
- **Returns** – an array for all the steady state rates. The order is that of the events set.

- **Throws**
 - * jmarkov.basic.exceptions.NotUnichainException –

- *getEvents*

```
public basic.Event[] getEvents( )
```

 - **Description**

Returns all The events defined in the model.
 - **Returns** – Events array

- *getEventsRates*

```
public double[] getEventsRates( ) throws
jmarkov.basic.exceptions.NotUnichainException
```

 - **Description**

Return an array with the steadystate rate of occurrence of all the Events.
 - **Returns** – an array for all the steady state rates. The order is that of the events set.
 - **Throws**
 - * jmarkov.basic.exceptions.NotUnichainException –

 - *getFinalRate*

```
public synchronized double getFinalRate( basic.State i, basic.State j )
```

 - **Description**

Gets the total rate form State number i to j. The number is relative to the ordered set of all states, therefore this method causes the model to be generated if it has not been generated.
 - **Parameters**
 - * i – origin State
 - * j – destination state
 - **Returns** – total rate
 - **See also**
 - * MarkovProcess.getRate(State,State) (see 1.1.5, page 41)

 - *getGenerator*

```
public double[][] getGenerator( )
```

 - **Description**

Returns the infinitesimal generator matrix **Q** , in dense format. It generates the model if it has not been generated.
 - **Returns** – The generator Matrix.

 - *getInitialState*

```
protected basic.State getInitialState( )
```

 - **Description**

returns the initial state.
 - **Returns** – The initial state.

 - *getMaxStates*

```
public long getMaxStates( )
```

- **Returns** – the Maximum number of states to generate.
-

- *getMOPIndex*

```
public int getMOPIndex( java.lang.String name )
```

- **Description**
Gets the index that correspond to this MOP.
 - **Parameters**
* **name** – MOP name
 - **Returns** – The index of the MOP with this name
-

- *getMOPNames*

```
public java.lang.String[] getMOPNames( )
```

- **Description**
Return all the names of defined MOPs.
 - **Returns** – an array with all the MOP's defined.
-

- *getMOPNames*

```
public java.lang.String getMOPNames( int mopNum )
```

- **Description**
Return the names of the i-th MOP.
 - **Parameters**
* **mopNum** – The number i of the MOP
 - **Returns** – The name of the i-th MOP.
-

- *getMOPsAvg*

```
public double[] getMOPsAvg( ) throws  
jmarkov.basic.exceptions.NotUnichainException
```

- **Description**
Returns an array with the average of all the steady state measures of performance.
The order is the same as in getMOPNames.
 - **Returns** – An array containing the values of all MOPs averages.
 - **Throws**
* `jmarkov.basic.exceptions.NotUnichainException` –
 - **See also**
* `MarkovProcess.getMOPsMoment(int)` (see 1.1.5, page 40)
-

- *getMOPsAvg*

```
public double getMOPsAvg( int mopNum ) throws  
jmarkov.basic.exceptions.NotUnichainException
```

- **Description**
Returns the steady state measures average of the MOP nombre mopNum.
- **Parameters**
* **mopNum** – The Number of the MOP of the which the average is to be computed.
- **Returns** – The long run averagefor this MOP.
- **Throws**
* `jmarkov.basic.exceptions.NotUnichainException` –

- *getMOPsAvg*

`public double getMOPsAvg(java.lang.String mopName)` throws
`jmarkov.basic.exceptions.NotUnichainException`

- **Description**

Returns the steady state measures average of the MOP with name mopName.

- **Parameters**

* mopName – The name whose Average is to be computed.

- **Returns**

The long run average for this MOP.

- **Throws**

* `jmarkov.basic.exceptions.NotUnichainException` –

- *getMOPsMoment*

`public double[] getMOPsMoment(int m)` throws
`jmarkov.basic.exceptions.NotUnichainException`

- **Description**

Returns an array with the m-th moment of all the steady state measures of performance. The order is the same as in `getMOPNames`.

- **Parameters**

* m – the order of the moment desired. m=1 is the expected value.

- **Returns**

An array containing the values of all MOPs m-th moments.

- **Throws**

* `jmarkov.basic.exceptions.NotUnichainException` –

- *getMOPsMoment*

`public double getMOPsMoment(int mopNum, int m)` throws
`jmarkov.basic.exceptions.NotUnichainException`

- **Description**

Returns the steady state measures m-th moment of the MOP number mopNum. m=1 is the long-run expected value, m=2 expected value of the square, etc.

- **Parameters**

* mopNum – The number for the MOP

* m – The value of m.

- **Returns**

The m-th moments for this MOP.

- **Throws**

* `jmarkov.basic.exceptions.NotUnichainException` –

- *getMOPsMoment*

`public double getMOPsMoment(java.lang.String mopName, int m)` throws
`jmarkov.basic.exceptions.NotUnichainException`

- **Description**

Returns the steady state measures m-th moment of the MOP with name mopName. m=1 is the long-run expected value.

- **Parameters**

* mopName – The name of the MOP that is to be computed

* m – Value of the moment

- **Returns** – The m-th moments for this MOP.
 - **Throws**
 - * `jmarkov.basic.exceptions.NotUnichainException` –
-

- *getMtjGenerator*

```
public no.uib.cipr.matrix.Matrix getMtjGenerator( )
```

- **Description**
The generator **Q** as an MTJ Matrix
 - **Returns** – The matrix **Q**.
-

- *getMtjRates*

```
public synchronized no.uib.cipr.matrix.Matrix getMtjRates( )
```

- **Description**
Returns the transition rates matrix **R** in MTJ format. It generates the model if it has not been generated.
 - **Returns** – an `mt.Matrix` with the matrix.
-

- *getNumStates*

```
public int getNumStates( )
```

- **Description**
Return the number of States in the model. If the model has not been generated, then it will be automatically generated.
 - **Returns** – the number of States in the model.
-

- *getProgress*

```
public long getProgress( )
```

- **Description**
Return the number of states processed so far in the current process.
 - **Returns** – the number of states processed so far.
-

- *getRate*

```
public synchronized double getRate( basic.State i, basic.State j )
```

- **Description**
Gets the current total rate from **i** to **j**. **Warning:** If the model has not been generated it returns the current total rate, it does not cause the generation of the model. Run `generate()` first.
 - **Parameters**
 - * **i** – origin State
 - * **j** – destination state
 - **Returns** – total rate
 - **See also**
 - * `MarkovProcess.generate()` (see 1.1.5, page 36)
-

- *getRates*

```
public synchronized double[][] getRates( )
```

– **Description**

Returns the transition rates matrix **R** in dense format. It generates the model if it has not been generated.

– **Returns** – an array with the matrix.

– **See also**

- * `MarkovProcess.getRate(State,State)` (see 1.1.5, page 41)
 - * `MarkovProcess.getRates(State)` (see 1.1.5, page 42)
 - * `MarkovProcess.getMtjRates()` (see 1.1.5, page 41)
-

• *getRates*

```
public synchronized basic.Transitions getRates( basic.State i )
```

– **Description**

This method returns a dynamic data structure with the rate from State i to all reachable states.

– **Parameters**

- * `i` – State

– **Returns** – Rates to reachable states

– **See also**

- * `basic.Transitions` (see 2.1.6, page 69)
-

• *getStateClass*

```
public java.lang.Class getStateClass( )
```

– **Description**

The Class for the states in this model.

– **Returns** – The class for the states

• *getStates*

```
public basic.StatesSet getStates( )
```

– **Description**

Returns an array with all the States in the model. It generates the model if it has not been generated.

– **Returns** – The States

• *getStates*

```
public basic.StatesSet getStates( boolean causesGeneration )
```

– **Description**

Returns an array with the States in the model that have been checked so far. If `generate` is true it generates the model if it has not been generated. If no states have been generated it returns null.

– **Parameters**

- * `causesGeneration` – whether the model should be generated.

– **Returns** – an array with the states found and checked so far.

• *getStatus*

```
public MarkovProcess.Status getStatus( )
```

- **Description**
Returns the current status of the model.
 - **Returns** – One of the constants IDLE, RUNNING, GENERATED, SUSPENDED, ERROR
-
- *getStatusMsg*
public java.lang.String getStatusMsg()
 - **Description**
Returns a String describing the current status of the model.
 - **Returns** – a String describing the current status of the model.
-
- *getSteadyState*
public double[] getSteadyState() throws
jmarkov.basic.exceptions.NotUnichainException
 - **Description**
Returns the steady state probabilities for this model. That is, it solves the balance equations. It returns an array of zeros if there is no unique solution.
 - **Returns** – An array with the steady-state probabilities.
 - **Throws**
* jmarkov.basic.exceptions.NotUnichainException –
-
- *getSteadyStateSolver*
public solvers.SteadyStateSolver getSteadyStateSolver()
 - **Description**
The currently defined solver.
 - **Returns** – Returns the steadyStateSolver.
 - **See also**
* solvers.SteadyStateSolver (see 6.1.10, page 250)
-
- *getTransientSolver*
public solvers.TransientSolver getTransientSolver()
 - **Description**
The currently defined solver for transient state.
 - **Returns** – Returns the transientSolver.
 - **See also**
* solvers.TransientSolver (see 6.1.11, page 252)
-
- *go*
public synchronized void go()
 - **Description**
Runs the model, or resumes execution if it had been suspended. Its use is intended for Graphical User Interface(GUI). A standard user should use generate() instead.
 - **See also**
* MarkovProcess.generate() (see 1.1.5, page 36)
-

- *goStep*
`public synchronized void goStep()`
 - **Description**
Runs the model for a single step. Its use is intended for Graphical User Interface(GUI).
A standard user should use `generate()` instead.
 - **See also**
 - * `MarkovProcess.generate()` (see 1.1.5, page 36)
 - * `MarkovProcess.go()` (see 1.1.5, page 43)

- *hideGUI*
`public void hideGUI()`
 - **Description**
Hides the Graphic User Interface (GUI) that represent this Markov Chain if one is defined.

- *hLine*
`protected java.lang.String hLine(int length)`
 - **Description**
Retuns an horizontal text line of the given length.
 - **Parameters**
 - * `length` –
 - **Returns** – a horizontal line.

- *isGenerated*
`public boolean isGenerated()`
 - **Returns** – true if the model has been completely generated

- *killGUI*
`public void killGUI()`
 - **Description**
Destroys the Graphic User Interface (GUI) that represent this Markov Chain if one is defined.

- *label*
`public java.lang.String label()`
 - **Description**
Returns the name of the model.
 - **Returns** – The Name of this model.

- *loadGUI*
`public void loadGUI()`
 - **Description**
Loads the Graphic User Interface (GUI) that represent this Markov Chain.

- *MOPsToString*
`public java.lang.String MOPsToString()`

– **Description**

Return a String description of all MOPs in steady state (it reports mean and standard deviation).

– **Returns** – A String description of all MOPs.

• *MOPsToString*

`public java.lang.String MOPsToString(int width, int decimals)`

– **Description**

Return a String description of all MOPs in steady state (it reports mean and standard deviation).

– **Parameters**

* **width** – the columns width

* **decimals** – the number of decimals to use.

– **Returns** – String with a table representing MOPs names, means and standard deviations.

• *numMOPs*

`public int numMOPs()`

– **Description**

Returns the number of defined Measures of performance (MOPs).

– **Returns** – the number of MOPs defined so far.

• *pad*

`protected java.lang.String pad(double v, int w)`

– **Description**

pad generates a string representing the double v, padded with spaces up to width w. *

– **Parameters**

* **v** – The double to print.

* **w** – The width to pad up to.

• *pad*

`protected java.lang.String pad(double v, int w, boolean right)`

– **Description**

pad generates a string representing the double v, padded with spaces up to width w.

– **Parameters**

* **v** – The double to print to print.

* **w** – The width to pad up to.

* **right** – whether the string should be eligned to the right.

• *pad*

`protected java.lang.String pad(double v, int w, int d)`

– **Description**

pad fills with blanks up to width w. Alignment is to the right.

– **Parameters**

* **v** – The number to print.

* **w** – The width to pad up to.

* **d** – number of decimals.

- *pad*

```
protected java.lang.String pad( double v, int w, int d, boolean right )
```

- **Description**

pad fills with blanks up to width w

- **Parameters**

- * *v* – The number to print.
 - * *w* – The width to pad up to.
 - * *d* – number of decimals.
 - * *right* – whether the string should be eligned to the right.
-

- *pad*

```
protected java.lang.String pad( java.lang.String s, int w )
```

- **Description**

pad fills with blanks up to width w

- **Parameters**

- * *s* – The String to print.
 - * *w* – The width to pad up to.
-

- *pad*

```
protected java.lang.String pad( java.lang.String s, int w, boolean right )
```

- **Description**

pad fills with blanks up to width w

- **Parameters**

- * *s* – The String to print.
 - * *w* – The width to pad up to.
 - * *right* – whether the string should be eligned to the right.
-

- *pause*

```
public void pause( )
```

- **Description**

Pauses the current execution of the model. This is called by the GUI.

- *printAll*

```
public void printAll( )
```

- **Description**

Prints a description of the Model: the States and the Transition Matrix. If the model has less than 100 states it shows all states and transition matrix and steady state probabilities. Otherwise only the description, measures of performance and events rates are shown.

- **See also**

- * `MarkovProcess.allToString()` (see 1.1.5, page 34)
-

- *printAll*

```
public void printAll( java.io.PrintWriter out )
```

- **Description**

Prints to the given `PrintWriter` a summary of the information related to this `MarkovChain`. The information is the same as as in the method `printAll()`.

- **Parameters**

- * `out` –

- **See also**

- * `MarkovProcess.toString()` (see 1.1.5, page 51)

- * `MarkovProcess.printAll()` (see 1.1.5, page 46)

- *printDenseMatrix*

```
public void printDenseMatrix( java.io.PrintWriter out )
```

- **Description**

Prints a the Transition Matrix. It will use default values for width and decimals. Its use is not recommended for large models.

- **Parameters**

- * `out` – The writer to write to.

- *printDenseMatrix*

```
public void printDenseMatrix( java.io.PrintWriter out, int width, int
rateDecimals, boolean printZeros, boolean useGenerator )
```

- **Description**

Prints a description of the Model using the given `PrintWriter`: the States and the Transition Matrix. Its use is not recommended for large models.

- **Parameters**

- * `out` – The writer to write to.

- * `width` – The width of each column.

- * `rateDecimals` – The number of decimals for the rates.

- * `printZeros` – Whether zeros or blanks should be printed.

- * `useGenerator` – whether the generator matrix **Q**, rather than the rates matrix should be printed.

- *printDenseMatrix*

```
protected void printDenseMatrix( java.io.PrintWriter out, int width, int
rateDecimals, boolean printZeros, boolean useGenerator, int[] idx )
```

- **Description**

Prints a description of the Model using the given `PrintWriter`: the States and the Transition Matrix. Its use is not recommended for large models.

- **Parameters**

- * `width` – The width of each column.

- * `rateDecimals` – The number of decimals for the rates.

- * `printZeros` – Whether zeros or blanks should be printed.

- * `useGenerator` – whether the generator matrix **Q**, rather than the rates matrix should be printed.

- * `idx` – the indices of separators.

- *printEventsRates*

```
public void printEventsRates( java.io.PrintWriter out )
```

– **Description**

Prints a table reporting the steadystate occurrence of all events.

– **Parameters**

* **out** – where the table will be printed.

• *printEventsRates*

```
public void printEventsRates( java.io.PrintWriter out, int width, int
decimals )
```

– **Description**

Prints a table reporting the steadystate occurrence of all events.

– **Parameters**

* **out** – where the table will be printed.

* **width** – The column width

* **decimals** – The number of decimals to use.

• *printMOPs*

```
public final void printMOPs( )
```

– **Description**

Prints the Measures of performance (MOPS) on standard output.

– **See also**

* `MarkovProcess.printMOPs(PrintWriter)` (see 1.1.5, page 48)

* `MarkovProcess.printMOPs(PrintWriter,int,int)` (see 1.1.5, page 48)

• *printMOPs*

```
public final void printMOPs( java.io.PrintWriter out )
```

– **Description**

Prints a String description of all MOPs in steady state (it reports mean and standard deviation), with a width of 10 and 5 decimal figures.

– **Parameters**

* **out** – The printer where the MOPS will be printed.

– **See also**

* `MarkovProcess.printMOPs()` (see 1.1.5, page 48)

* `MarkovProcess.printMOPs(PrintWriter,int,int)` (see 1.1.5, page 48)

• *printMOPs*

```
public int printMOPs( java.io.PrintWriter out, int width, int decimals )
```

– **Description**

Prints a String description of all MOPs in steady state (it reports mean and standard deviation). You can override this method to print your own MOPs. You can call it in the first line like this `public void printMOPs(PrintWriter out, int width, int decimals) { int namesWidth = super.printMOPs(out,width, decimals); // your own code here: out.println(pad("Another MOP", namesWidth, false) + pad(Value, width, decimals)); }`

– **Parameters**

* **out** – The printer where the MOPS will be printed.

* **width** – the columns width

* **decimals** – the number of decimals to use.

- **Returns** – The max width among the declared MOPs. You can use this to align nicely your own MOPs.
 - **See also**
 - * `MarkovProcess.printMOPs(PrintWriter)` (see 1.1.5, page 48)
 - * `MarkovProcess.printMOPs(PrintWriter,int,int)` (see 1.1.5, page 48)
-

- *printStates*

```
public void printStates( java.io.PrintWriter out )
```

- **Description**
Prints a description of the States and the Equilibrium Probabilities.
 - **Parameters**
 - * `out` – The writer to write to.
-

- *printStates*

```
public void printStates( java.io.PrintWriter out, int width, int
probDecimals )
```

- **Description**
Prints a description of the States and the Equilibrium Probabilities.
 - **Parameters**
 - * `out` – The writer to write to.
 - * `width` – The width of each column.
 - * `probDecimals` – The number of decimals for the probabilities.
-

- *reset*

```
public synchronized void reset( )
```

- **Description**
Resets the Model. It erases all found states and transition rates. Keeps the initial state and Events set.
-

- *reset*

```
protected synchronized void reset( boolean resetEvents )
```

- **Description**
Resets the Model. It erases all found states and transition rates.
 - **Parameters**
 - * `resetEvents` – whether the Events are deleted. WARNING! if this is true you must call `setEventSet`.
 - **See also**
 - * `MarkovProcess.setInitialState(State)` (see 1.1.5, page 50)
 - * `MarkovProcess.setEventSet(basic.EventsSet)` (see 1.1.5, page 50)
-

- *resetResults*

```
public synchronized void resetResults( )
```

- **Description**
Resets the result of the model. If it has been generated this method keeps the Graph, but erases steady state, and transient probabilities.
-

- *setDebugLevel*

`public void setDebugLevel(int level)`

- **Description**

Sets the debug level, where level=0 means no debug info, level = 5 verbose info.

- **Parameters**

* `level` – New debug level

- *setDebugReporter*

`public void setDebugReporter(DebugReporter reporter)`

- **Description**

Sets the DebugReporter to use.

- **Parameters**

* `reporter` – The reporter that will capture the debug information.

- **See also**

* `DebugReporter` (see 1.1.1, page 10)

- *setEventSet*

`protected void setEventSet(basic.EventsSet eSet)`

- **Description**

Sets the Events set. It causes the model to be reset. This method should be called only in a constructor.

- **Parameters**

* `eSet` – the Events set.

- *setInitialState*

`protected void setInitialState(basic.State i0)`

- **Description**

Sets the initial state. It causes the model to be reset. This method should be called only in a constructor.

- **Parameters**

* `i0` – The initial state.

- *setMaxStates*

`public void setMaxStates(long num)`

- **Description**

Sets the maximum number of states to generate. Increase this only if you are sure that your model has a big number of states. Your current License may prevent you from setting this number.

- **Parameters**

* `num` – Maximum Number of States to generate.

- *setMOPs*

`public void setMOPs(java.lang.String[] mopNames)`

- **Description**

Sets the names of all MOPs (measures of performance).

– **Parameters**

* mopNames –

• *setSteadyStateSolver*

```
public void setSteadyStateSolver( solvers.SteadyStateSolver
steadyStateSolver )
```

– **Description**

Allows the user to set an alternate solver.

– **Parameters**

* steadyStateSolver – The steadyStateSolver to set.

– **See also**

* solvers.SteadyStateSolver (see 6.1.10, page 250)

• *setTransientSolver*

```
public void setTransientSolver( solvers.TransientSolver transientSolver )
```

– **Description**

Allows the user to set an alternate solver.

– **Parameters**

* transientSolver – The transientSolver to set.

– **See also**

* solvers.TransientSolver (see 6.1.11, page 252)

• *showGUI*

```
public void showGUI( )
```

– **Description**

Shows the Graphic User Interface (GUI) that represent this Markov Chain. It loads one if none has been defined.

• *statesLableMaxWidth*

```
public int statesLableMaxWidth( int width )
```

– **Description**

Computes the maximum used by the state's labels.

– **Parameters**

* width – minimum width acceptable.

– **Returns** – Max(width, max label).

• *statesToString*

```
public java.lang.String statesToString( )
```

– **Description**

Prints a description of the States and the Equilibrium Probabilities.

– **Returns** – A String description of the States.

• *toString*

```
public java.lang.String toString( )
```

• *vLine*

```
protected java.lang.String vLine( )
```

- **Description**
Returns a text vertical line.
- **Returns** – vertical line.

1.1.6 Class MarkovProcess.Status

Status variables

Declaration

```
public static final class MarkovProcess.Status  
extends java.lang.Enum
```

Field summary

ERROR Model execution generated an error
GENERATED Model has been generated
IDLE Model has not been generated
NoModel No Model loaded.
RUNNING Model is being generated
SUSPENDED Model execution has been suspended
WRITING Model generated, writing matrix

Method summary

valueOf(String)
values()

Fields

- public static final MarkovProcess.Status **IDLE**
 - Model has not been generated
- public static final MarkovProcess.Status **RUNNING**
 - Model is being generated
- public static final MarkovProcess.Status **GENERATED**
 - Model has been generated
- public static final MarkovProcess.Status **SUSPENDED**
 - Model execution has been suspended
- public static final MarkovProcess.Status **WRITING**
 - Model generated, writing matrix
- public static final MarkovProcess.Status **ERROR**
 - Model execution generated an error
- public static final MarkovProcess.Status **NoModel**
 - No Model loaded. (used by GUI)

Methods

- *valueOf*
public static MarkovProcess.Status valueOf(java.lang.String name)
- *values*
public static final MarkovProcess.Status[] values()

Members inherited from class java.lang.Enum

- protected final Object clone() throws CloneNotSupportedException
- public final int compareTo(Enum arg0)
- public final boolean equals(Object arg0)
- public final Class getDeclaringClass()
- public final int hashCode()
- public final String name()
- public final int ordinal()
- public String toString()
- public static Enum valueOf(Class arg0, String arg1)

1.1.7 Class SimpleMarkovProcess

Declaration

```
public abstract class SimpleMarkovProcess
extends jmarkov.MarkovProcess (see 1.1.5, page 29)
```

All known subclasses

GeomProcess (see 1.1.2, page 13)

Constructor summary

```
SimpleMarkovProcess()
SimpleMarkovProcess(S, EventsSet)
SimpleMarkovProcess(S, EventsSet, String)
```

Method summary

```
active(S, E) Determines if event e is active when the system is in state i.
activeTransitions(S, E) This method calls active, dests and rate to create the set of
transitions.
dests(S, E) Determines the destination set of States when events e occurs.
rate(S, S, E) Returns the rate to go from State i to j when Event e occurs.
```

Constructors

- *SimpleMarkovProcess*

```
public SimpleMarkovProcess( )
```

- *SimpleMarkovProcess*

```
public SimpleMarkovProcess( basic.State i0, basic.EventsSet eSet )
```

 - Parameters
 - * i0 –
 - * eSet –

- *SimpleMarkovProcess*

```
public SimpleMarkovProcess( basic.State i0, basic.EventsSet eSet,
java.lang.String name )
```

 - Parameters
 - * i0 –
 - * eSet –
 - * name –

Methods

- *active*

```
public abstract boolean active( basic.State i, basic.Event e )
```

- **Description**

Determines if event e is active when the system is in state i. It has to be implemented by a subclass.

- **Parameters**

- * i – The current State
- * e – The current Event.

- **Returns** – True if the Event is Active.

- *activeTransitions*

```
public final basic.Transitions activeTransitions( basic.State i, basic.Event e )
```

- **Description**

This method calls active, dests and rate to create the set of transitions. The user cannot override this method and would rarely call it.

- **See also**

- * MarkovProcess.activeTransitions(State,Event) (see 1.1.5, page 33)
-

- *dests*

```
public abstract basic.States dests( basic.State i, basic.Event e )
```

- **Description**

Determines the destination set of States when events e occurs. It has to be implemented by the subclass.

- **Parameters**

- * i – current State.
- * e – The Event that occurred.

- **Returns** – The destination States

- *rate*

```
public abstract double rate( basic.State i, basic.State j, basic.Event e )
```

- **Description**

Returns the rate to go from State i to j when Event e occurs. This is called only if Event e is indeed active and j is a valid destination.

- **Parameters**

- * i – Current state
- * j – Destination State
- * e – The occurring event

- **Returns** – The rate at which the system goes from i to j when e occurs.

- **See also**

- * SimpleMarkovProcess.dests(State,Event) (see 1.1.7, page 56)

Members inherited from class `jmarkov.MarkovProcess` (see 1.1.5, page 29)

- `public abstract Transitions activeTransitions(basic.State i, basic.Event e)`
- `public boolean addMOP(java.lang.String mopName)`
- `public String allToString()`
- `public boolean canGo()`
- `public void clearMOPs()`
- `protected cnt`
- `public void debug(int level, java.lang.String s)`
- `public void debug(int level, java.lang.String s, boolean newline)`
- `public void debug(int level, java.lang.String s, boolean newline, boolean indent)`
- `protected defaultSteadyStateSolver`
- `protected defaultTransientSolver`
- `public String denseMatrixToString()`
- `public String denseMatrixToString(int width, int rateDecimals, boolean printZeros, boolean useGenerator)`
- `public abstract String description()`
- `public String eventRatesToString(int width, int decimals)`
- `public String eventsRatesToString()`
- `protected void finalize() throws java.lang.Throwable`
- `public void generate()`
- `public int getDebugLevel()`
- `public DebugReporter getDebugReporter()`
- `protected final SteadyStateSolver getDefaultSteadyStateSolver()`
- `protected TransientSolver getDefaultTransientSolver()`
- `public Class getEventClass()`
- `public String getEventNames()`
- `public double getEventRate(int eNum) throws basic.exceptions.NotUnichainException`
- `public Event getEvents()`
- `public double getEventsRates() throws basic.exceptions.NotUnichainException`
- `public synchronized double getFinalRate(basic.State i, basic.State j)`
- `public double getGenerator()`
- `protected State getInitialState()`
- `public long getMaxStates()`
- `public int getMOPIndex(java.lang.String name)`
- `public String getMOPNames()`
- `public String getMOPNames(int mopNum)`
- `public double getMOPsAvg() throws basic.exceptions.NotUnichainException`
- `public double getMOPsAvg(int mopNum) throws basic.exceptions.NotUnichainException`
- `public double getMOPsAvg(java.lang.String mopName) throws basic.exceptions.NotUnichainException`
- `public double getMOPsMoment(int m) throws basic.exceptions.NotUnichainException`
- `public double getMOPsMoment(int mopNum, int m) throws basic.exceptions.NotUnichainException`
- `public double getMOPsMoment(java.lang.String mopName, int m) throws basic.exceptions.NotUnichainException`
- `public Matrix getMtjGenerator()`
- `public synchronized Matrix getMtjRates()`
- `public int getNumStates()`
- `public long getProgress()`
- `public synchronized double getRate(basic.State i, basic.State j)`
- `public synchronized double getRates()`
- `public synchronized Transitions getRates(basic.State i)`
- `public Class getStateClass()`
- `public StatesSet getStates()`
- `public StatesSet getStates(boolean causesGeneration)`
- `public MarkovProcess.Status getStatus()`
- `public String getStatusMsg()`
- `public double getSteadyState() throws basic.exceptions.NotUnichainException`
- `public SteadyStateSolver getSteadyStateSolver()`
- `public TransientSolver getTransientSolver()`
- `public synchronized void go()`
- `public synchronized void goStep()`
- `public void hideGUI()`

- protected String hLine(int length)
- public boolean isGenerated()
- public void killGUI()
- public String label()
- public void loadGUI()
- public String MOPsToString()
- public String MOPsToString(int width, int decimals)
- protected name
- public int numMOPs()
- protected String pad(double v, int w)
- protected String pad(double v, int w, boolean right)
- protected String pad(double v, int w, int d)
- protected String pad(double v, int w, int d, boolean right)
- protected String pad(java.lang.String s, int w)
- protected String pad(java.lang.String s, int w, boolean right)
- public void pause()
- public void printAll()
- public void printAll(java.io.PrintWriter out)
- public void printDenseMatrix(java.io.PrintWriter out)
- public void printDenseMatrix(java.io.PrintWriter out, int width, int rateDecimals, boolean printZeros, boolean useGenerator)
- protected void printDenseMatrix(java.io.PrintWriter out, int width, int rateDecimals, boolean printZeros, boolean useGenerator, int[] idx)
- public void printEventsRates(java.io.PrintWriter out)
- public void printEventsRates(java.io.PrintWriter out, int width, int decimals)
- public final void printMOPs()
- public final void printMOPs(java.io.PrintWriter out)
- public int printMOPs(java.io.PrintWriter out, int width, int decimals)
- public void printStates(java.io.PrintWriter out)
- public void printStates(java.io.PrintWriter out, int width, int probDecimals)
- public synchronized void reset()
- protected synchronized void reset(boolean resetEvents)
- public synchronized void resetResults()
- public void setDebugLevel(int level)
- public void setDebugReporter(DebugReporter reporter)
- protected void setEventSet(basic.EventsSet eSet)
- protected void setInitialState(basic.State i0)
- public void setMaxStates(long num)
- public void setMOPs(java.lang.String[] mopNames)
- public void setSteadyStateSolver(solvers.SteadyStateSolver steadyStateSolver)
- public void setTransientSolver(solvers.TransientSolver transientSolver)
- public void showGUI()
- public int statesLableMaxWidth(int width)
- public String statesToString()
- protected theStates
- public String toString()
- protected String vLine()

Chapter 2

Package jmarkov.basic

Package Contents

Page

Interfaces

Actions	61
<i>This interface represents a set of objects Action.</i>	
Events	62
<i>This class represents a set of objects Event.</i>	
JMarkovElement	63
<i>All the elements in JMarkov implement this interface, so they can be easily described in the interface.</i>	
PropertiesElement	65
<i>This interface is a wrapper for elements (States, Actions Events) that can be represented by an array of integers.</i>	
States	67
<i>This interface represents a set of objects State.</i>	
Transitions	69

Classes

Action	71
<i>This class represents a single Action in Markov Decision Process (MDP).</i>	
ActionsSet	73
<i>This class represents a set of objects Action.</i>	
DecisionRule	75
<i>This class represents a deterministic decision rule which assigns an action to every state.</i>	
Event	78
<i>The class Event allows the user to define the implementation of the Events that can alter the States of the Markov Chain.</i>	
EventsSet	80
<i>This class represent a set of Events.</i>	
Policy	82
<i>Policy is a set of "Decision Rules".</i>	
PropertiesAction	85
<i>This class is an easy way to use a Action that is represented by an integer valued array.</i>	
PropertiesEvent	88

	<i>This class is an easy way to use an event that is represented by an array of int.</i>	
PropertiesState	<i>The states are characterized by an array of integer-valued properties, whose meaning will change from implementation to implementation.</i>	91
Solution	<i>This class represents the joint information of a value function and a policy which summarizes the solution to a problem.</i>	95
State	<i>The Class State represent a state in a MarkovProcess or MDP.</i>	96
StateC	<i>State to model shortest path problems.</i>	100
StateEvent	<i>This class represents a state compounded of a state and an event.</i>	102
StatesSet	<i>This class represent a set of States.</i>	104
Transition	<i>This class represent a transition to a given state.</i>	108
TransitionsSet		110
ValueFunction	<i>This structure matches each state with a double number representing its value function, or in some cases the steady state probabilities.</i>	113

This package contains basic elements such as State, Event, Action that are used in jMarkov and jMDP.

2.1 Interfaces

2.1.1 Interface Actions

This interface represents a set of objects Action. The user must choose his own data structure and define the constructors. It's recommended to use the Set structure to avoid repeated actions. The ActionsSet class extends this class and explores the goodnesses of Collections. It is recommended to use that class instead of this one for beginner users.

See also

- `java.lang.Iterable`
- `ActionsSet` (see 2.2.2, page 73)

Declaration

```
public interface Actions
implements java.lang.Iterable
```

All known subclasses

ActionsSet (see 2.2.2, page 73)

All classes known to implement interface

ActionsSet (see 2.2.2, page 73)

Method summary

- iterator()** This function must be implemented.
- size()** Returns the number of elements.

Methods

- *iterator*
`java.util.Iterator iterator()`
 - **Description**
This function must be implemented. Must return an iterator over the Actions.
- *size*
`int size()`
 - **Description**
Returns the number of elements.
 - **Returns** – the number of elements.

2.1.2 Interface Events

This class represents a set of objects Event. The user must choose his own data structure and define the constructors. For an easy way to declare and use a set of events see “c>EventsCollection“c>, which is an extension of Events.

Declaration

```
public interface Events
implements java.lang.Iterable
```

All known subclasses

EventsSet (see 2.2.5, page 80)

All classes known to implement interface

EventsSet (see 2.2.5, page 80)

Method summary

add(E) This method adds an object to the set of events.
iterator() This function must be implemented.
size() Returns the number of elements.

Methods

- *add*
 boolean add(Event s)
 - **Description**
 This method adds an object to the set of events.
 - **Parameters**
 * s – object to be added.
 - **Returns** – True if the set did not contained this element.

- *iterator*
 java.util.Iterator iterator()
 - **Description**
 This function must be implemented. Must return an iterator over the events.

- *size*
 int size()
 - **Description**
 Returns the number of elements.
 - **Returns** – the number of Event elements.

2.1.3 Interface JMarkovElement

All the elements in JMarkov implement this interface, so they can be easily described in the interface. It is recommended that the method `toString()` is implemented as final, and calling `label()`.

Declaration

```
public interface JMarkovElement
```

All known subclasses

SimpleMarkovProcess (see 1.1.7, page 55) , MarkovProcess (see 1.1.5, page 29) , GeomState (see 1.1.4, page 26) , GeomRelState (see 1.1.3, page 23) , GeomProcess (see 1.1.2, page 13) , ValueFunction (see 2.2.17, page 113) , TransitionsSet (see 2.2.16, page 110) , Transitions (see 2.1.6, page 69) , Transition (see 2.2.15, page 108) , StateEvent (see 2.2.13, page 102) , StateC (see 2.2.12, page 100) , State (see 2.2.11, page 96) , PropertiesState (see 2.2.9, page 91) , PropertiesEvent (see 2.2.8, page 88) , PropertiesElement (see 2.1.4, page 65) , PropertiesAction (see 2.2.7, page 85) , Event (see 2.2.4, page 78) , DecisionRule (see 2.2.3, page 75) , Action (see 2.2.1, page 71) , ValueIterationSolver (see 5.2.16, page 224) , StochasticShortestPathSolver (see 5.2.15, page 221) , Solver (see 5.2.14, page 216) , RelativeValueIterationSolver (see 5.2.13, page 213) , PolicyIterationSolver (see 5.2.11, page 205) , MpsLpDiscountedSolver (see 5.2.10, page 201) , MpsLpAverageSolver (see 5.2.9, page 198) , LPBCLDiscountedSolver (see 5.2.8, page 194) , LPBCLAverageSolver (see 5.2.7, page 191) , FiniteSolver (see 5.2.6, page 188) , AbstractTotalSolver (see 5.2.5, page 186) , AbstractInfiniteSolver (see 5.2.4, page 184) , AbstractFiniteSolver (see 5.2.3, page 182) , AbstractDiscountedSolver (see 5.2.2, page 179) , AbstractAverageSolver (see 5.2.1, page 177) , TransientSolver (see 6.1.11, page 252) , SteadyStateSolver (see 6.1.10, page 250) , Solver (see 6.1.9, page 248) , MtjSolver (see 6.1.6, page 240) , MtjLogRedSolver (see 6.1.5, page 238) , JamaTransientSolver (see 6.1.4, page 236) , JamaSolver (see 6.1.3, page 234) , GeometrixSolver (see 6.1.2, page 232) , GeometricSolver (see 6.1.1, page 230) , SparseDiscPhaseVar (see 7.2.12, page 333) , SparseContPhaseVar (see 7.2.11, page 330) , PhaseVar (see 7.1.3, page 265) , HyperErlangVar (see 7.2.6, page 296) , ErlangCoxianVar (see 7.2.5, page 291) , DiscPhaseVar (see 7.1.2, page 261) , DenseDiscPhaseVar (see 7.2.4, page 287) , DenseContPhaseVar (see 7.2.3, page 282) , ContPhaseVar (see 7.1.1, page 256) , AbstractDiscPhaseVar (see 7.2.2, page 276) , AbstractContPhaseVar (see 7.2.1, page 270)

All known subinterfaces

Transitions (see 2.1.6, page 69) , PropertiesElement (see 2.1.4, page 65) , PhaseVar (see 7.1.3, page 265)

All classes known to implement interface

MarkovProcess (see 1.1.5, page 29) , ValueFunction (see 2.2.17, page 113) , TransitionsSet (see 2.2.16, page 110) , Transition (see 2.2.15, page 108) , State (see 2.2.11, page 96) , Event (see 2.2.4, page 78) , DecisionRule (see 2.2.3, page 75) , Action (see 2.2.1, page 71) , Solver (see 5.2.14, page 216) , Solver (see 6.1.9, page 248)

Method summary

description() This method return a complete verbal description of this element.
equals(Object) Returns true if these two elements are equal.

label() This method returns a short String used in the user interface to describe this element.

toString() This method returns a short String used in the user interface to describe this element.

Methods

- *description*

java.lang.String **description**()

- **Description**

This method return a complete verbal describtion of this element. This description may contain multiple text rows.

- **Returns** – A String describing this element.

- **See also**

* JMarkovElement.label() (see 2.1.3, page 64)

- *equals*

boolean **equals**(java.lang.Object e)

- **Description**

Returns true if these two elements are equal. If this element implementa a compareTo() method it is recommended that this method returns compareTo(o)==0.

- **Parameters**

* e – The Object to compare to.

- **Returns** – True if the elements are equal.

- **See also**

* java.lang.Object.equals(Object)

- *label*

java.lang.String **label**()

- **Description**

This method returns a short String used in the user interface to describe this element.

- **Returns** – A String label.

- **See also**

* JMarkovElement.description() (see 2.1.3, page 64)

- *toString*

java.lang.String **toString**()

- **Description**

This method returns a short String used in the user interface to describe this element. It is highly recommended that every class calls label(), using the following code:

```
public final String toString() {
    return label();
}
```

- **Returns** – A String label.

- **See also**

* JMarkovElement.label() (see 2.1.3, page 64)

2.1.4 Interface PropertiesElement

This interface is a wrapper for elements (States, Actions Events) that can be represented by an array of integers. Known implementations include PropertiesState, PropertiesEvent and PropertiesAction. Basic methods are provided to access the internal array.

See also

- PropertiesState (see 2.2.9, page 91)
- PropertiesAction (see 2.2.7, page 85)
- PropertiesEvent (see 2.2.8, page 88)

Declaration

```
public interface PropertiesElement
implements JMarkovElement
```

All known subclasses

PropertiesState (see 2.2.9, page 91) , PropertiesEvent (see 2.2.8, page 88) , PropertiesAction (see 2.2.7, page 85)

All classes known to implement interface

PropertiesState (see 2.2.9, page 91) , PropertiesEvent (see 2.2.8, page 88) , PropertiesAction (see 2.2.7, page 85)

Method summary

clone()
getNumProps() Returns the number of properties in the array that characterize this element.
getProperties() Gets the array of properties.
getProperty(int) Gets the value of this property at this index.

Methods

- *clone*
PropertiesElement clone()

- *getNumProps*
int getNumProps()
 - **Description**
Returns the number of properties in the array that characterize this element.
 - **Returns** – The number of properties.

- *getProperties*
int[] getProperties()
 - **Description**
Gets the array of properties.

- **Returns** – Returns the properties array.
-

- *getProperty*

int **getProperty**(**int** **index**)

- **Description**
Gets the value of this property at this index.
- **Parameters**
* **index** –
- **Returns** – the property at the given index

2.1.5 Interface States

This interface represents a set of objects State. The user must choose his own data structure and define the constructors, or provide a mechanism to generate the states on-the-fly. A convenience class, StatesSet, is provided if the states are to be stored.

See also

- **StatesSet** (see 2.2.14, page 104)

Declaration

```
public interface States
implements java.lang.Iterable
```

All known subclasses

StatesSet (see 2.2.14, page 104)

All classes known to implement interface

StatesSet (see 2.2.14, page 104)

Method summary

- isClosed()** The set is closed if all elements have been added.
- iterator()** This function must be implemented.
- numerateStates()** This method numerates all states and returns the number of states found.
- size()** Returns the number of elements.

Methods

- *isClosed*
boolean isClosed()
 - **Description**
The set is closed if all elements have been added.
 - **Returns** – true if the set is closed.

- *iterator*
java.util.Iterator iterator()
 - **Description**
This function must be implemented. Must return an iterator over the states.

- *numerateStates*
int numerateStates()
 - **Description**
This method numerates all states and returns the number of states found. After this method is called it is illegal to add more states to the set.

- **Returns** – The number of states.
-

- *size*

`int size()`

- **Description**
Returns the number of elements.
- **Returns** – the number of State elements.

2.1.6 Interface Transitions

Declaration

```
public interface Transitions
implements JMarkovElement, java.lang.Iterable
```

All known subclasses

TransitionsSet (see 2.2.16, page 110)

All classes known to implement interface

TransitionsSet (see 2.2.16, page 110)

Method summary

add(S, double) Adds a ne transition to the given state
add(Transition)
addRate(S, double) Adds the given rate to the transition to this state.
getRate(S) Gets the rate for this state.
size() Returns the number of Transtions represented by this object.

Methods

- *add*
 boolean add(State state, double rate)
 - **Description**
 Adds a ne transition to the given state
 - **Parameters**
 - * **state** – State the transition goes to
 - * **rate** – The rate at which this transition occurs.
 - **Returns** – true if the state was already on the set.
- *add*
 boolean add(Transition t)
 - **Parameters**
 - * **t** – The trenasition to add.
 - **Returns** – true of the element was already in the set.
- *addRate*
 double addRate(State state, double rate)
 - **Description**
 Adds the given rate to the transition to this state.
 - **Parameters**
 - * **state** –
 - * **rate** –

- **Returns** – The old value associated with this state.
-

- *getRate*

`double getRate(State state)`

- **Description**
Gets the rate for this state. It returns 0.0 if this state is not in the Transitions.
 - **Parameters**
 - * `state` –
 - **Returns** – The rate for this state
-

- *size*

`int size()`

- **Description**
Returns the number of Transitions represented by this object.
- **Returns** – The number of Transitions.
- **See also**
 - * `java.util.Set.size()`

2.2 Classes

2.2.1 Class Action

This class represents a single Action in Markov Decision Process (MDP). It implements Comparable in order to be easily organized and searched.

See also

- `java.lang.Comparable`
- `jmarkov.jmdp` (see 4, page 122)

Declaration

```
public abstract class Action
extends java.lang.Object
implements java.lang.Comparable, JMarkovElement
```

All known subclasses

PropertiesAction (see 2.2.7, page 85)

Constructor summary

Action()

Method summary

- description()** The user SHOULD override this method to give a complete description for the action.
- equals(Object)** This method calls compareTo to check if the Action are equal.
- label()** The user MUST override this method to give a (hopefully short) label for the Action.
- toString()** This calls label().

Constructors

- *Action*
`public Action()`

Methods

- *description*
`public java.lang.String description()`
 - **Description**
The user SHOULD override this method to give a complete description for the action.
 - **Returns** – short description of the state.
- *equals*
`public final boolean equals(java.lang.Object o)`

- **Description**

This method calls `compareTo` to check if the Action are equal.

- **See also**

- * `java.lang.Object.equals(Object)`

- *label*

`public abstract java.lang.String label()`

- **Description**

The user **MUST** override this method to give a (hopefully short) label for the Action.

- **Returns** – short description of the Action.

- *toString*

`public final java.lang.String toString()`

- **Description**

This calls `label()`.

2.2.2 Class ActionsSet

This class represents a set of objects Action. It uses the TreeSet structure to avoid repeated actions. This class extends the Actions class.

See also

- `java.util.Collection`
- `Actions` (see 2.1.1, page 61)
- `Action` (see 2.2.1, page 71)

Declaration

```
public class ActionsSet
extends java.lang.Object
implements Actions
```

Constructor summary

- ActionsSet()** Creates an empty set of Actions.
- ActionsSet(A)** Creates a set of Actions from a given Action.
- ActionsSet(A[])** Creates a set of Actions from a given array of Actions.
- ActionsSet(Actions)** Creates a set of Actions from a given set of Actions.
- ActionsSet(Iterable)** Creates a set of actions from any iterable object over actions.

Method summary

- add(A)** This method adds a new action to the set.
- iterator()** This method returns a safe way to walk along the actions in a particular set.
- size()**

Constructors

- *ActionsSet*
 public **ActionsSet**()
 – **Description**
 Creates an empty set of Actions.

- *ActionsSet*
 public **ActionsSet**(Action ac)
 – **Description**
 Creates a set of Actions from a given Action. This constructor organizes the actions in a TreeSet.
 – **Parameters**
 * ac – an Action.

- *ActionsSet*
 public **ActionsSet**(Action[] acArray)

- **Description**

Creates a set of Actions from a given array of Actions. This constructor organizes the actions in a TreeSet.

- **Parameters**

* **acArray** – set of Actions of type Actions.

- *ActionsSet*

public ActionsSet(Actions ac)

- **Description**

Creates a set of Actions from a given set of Actions.

- **Parameters**

* **ac** – set of Actions of type Actions.

- *ActionsSet*

public ActionsSet(java.lang.Iterable actIter)

- **Description**

Creates a set of actions from any iterable object over actions.

- **Parameters**

* **actIter** –

Methods

- *add*

public void add(Action a)

- **Description**

This method adds a new action to the set.

- **Parameters**

* **a** – The action to be added.

- *iterator*

public final java.util.Iterator iterator()

- **Description**

This method returns a safe way to walk along the actions in a particular set. Collections and their implementations (Set, List, and Map) have iterators defined by default.

- **Returns** – iterator over the states.

- *size*

int size()

- **Description copied from Actions** (see 2.1.1, page 61)

Returns the number of elements.

- **Returns** – the number of elements.

2.2.3 Class DecisionRule

This class represents a deterministic decision rule which assigns an action to every state.

Declaration

```
public final class DecisionRule
  extends java.lang.Object
  implements JMarkovElement, java.lang.Iterable
```

Constructor summary

DecisionRule() Creates a new empty decision rule

DecisionRule(DecisionRule) Creates a decision rule from a given one

Method summary

description() Gives the sting representation of this Rule

equals(Object) Determines if the given decision rules are equal.

getAction(S) Gets the prescribed action for the given State.

iterator() Return an iterator over the State-Action pairs.

label()

print() Prints the Rule to the sandard output

print(PrintWriter) Prints the policiy to the given PrintWriter.

print(PrintWriter, String, String) Prints the policiy to the given PrintWriter.

set(S, A) Maps a given action to a given state

size() Returns the amount of states linked to actions in the decision rule.

toString()

Constructors

- *DecisionRule*

```
public DecisionRule( )
```

- **Description**

Creates a new empty decision rule

- *DecisionRule*

```
public DecisionRule( DecisionRule dr )
```

- **Description**

Creates a decision rule from a given one

- **Parameters**

* **dr** – decision rule

Methods

- *description*

```
public java.lang.String description( )
```

- **Description**
Gives the sting representation of this Rule
-

- *equals*

`public boolean equals(java.lang.Object o)`

- **Description**
Determines if the given decision rules are equal.
 - **Parameters**
* o –
 - **Returns** – True, if the decision rules are equal.
-

- *getAction*

`public Action getAction(State s)`

- **Description**
Gets the prescribed action for the given State.
 - **Parameters**
* s – state
 - **Returns** – the action corresponding to the given state
-

- *iterator*

`public java.util.Iterator iterator()`

- **Description**
Return an iterator over the State-Action pairs.
 - **Returns** – iterator over the entries
-

- *label*

`java.lang.String label()`

- **Description copied from JMarkovElement (see 2.1.3, page 63)**
This method returns a short String used in the user interface to describe this element.
 - **Returns** – A String label.
 - **See also**
* JMarkovElement.description() (see 2.1.3, page 64)
-

- *print*

`public void print()`

- **Description**
Prints the Rule to the sandard output
-

- *print*

`public void print(java.io.PrintWriter pw)`

- **Description**
Prints the policiy to the given PrintWriter.
 - **Parameters**
* pw – PrintWriter to use
-

- *print*

```
public void print( java.io.PrintWriter pw, java.lang.String statesFormat,  
java.lang.String actionFormat )
```

- **Description**

Prints the policy to the given PrintWriter.

- **Parameters**

- * **pw** – PrintWriter to use
 - * **statesFormat** – format for the states , for example "%-10S" to have 10 width left aligned states.
 - * **actionFormat** – format for the actions , for example "%-10S" to have 10 width left aligned actions.
-

- *set*

```
public void set( State s, Action a )
```

- **Description**

Maps a given action to a given state

- **Parameters**

- * **s** – state
 - * **a** – action
-

- *size*

```
public int size( )
```

- **Description**

Returns the amount of states linked to actions in the decision rule.

- **Returns** – Amount of states linked to actions in the decision rule.

- *toString*

```
public java.lang.String toString( )
```

2.2.4 Class Event

The class Event allows the user to define the implementation of the Events that can alter the States of the Markov Chain.

Declaration

```
public abstract class Event
extends java.lang.Object
implements java.lang.Comparable, JMarkovElement
```

All known subclasses

PropertiesEvent (see 2.2.8, page 88)

Constructor summary

Event()

Method summary

compareTo(Event) Returns positive if this Event has a higher number then the given event.
description() It is highly recommended that the user overrides it to give a description to be used when reporting the occurrence rates of the events, and GUI.
equals(Object) This method calls compareTo to check if the Action are equal.
getIndex() Gives the position of the Event in the Events set.
getSet() Returns the set of Events to which this event belongs.
label() If this function is not overridden by the user it returns the Event number.
toString()

Constructors

- *Event*
 public **Event**()

Methods

- *compareTo*
 public int **compareTo**(Event ev)
 – **Description**
 Returns positive if this Event has a higher number then the given event.
 – **See also**
 * `java.lang.Comparable.compareTo(Object)`
- *description*
 public java.lang.String **description**()

- **Description**

It is highly recommended that the user overrides it to give a description to be used when reporting the occurrence rates of the events, and GUI.

- **Returns** – a String description

- **See also**

* `Event.label()` (see 2.2.4, page 79)

- *equals*

```
public final boolean equals( java.lang.Object o )
```

- **Description**

This method calls `compareTo` to check if the Action are equal.

- **See also**

* `java.lang.Object.equals(Object)`

- *getIndex*

```
public int getIndex( )
```

- **Description**

Gives the position of the Event in the Events set. Returns -1 if this events has not yet been added to the set.

- **Returns** – The position of the Event in the Events set. Returns -1 if this events has not yet been added to the set.

- *getSet*

```
public EventsSet getSet( )
```

- **Description**

Returns the set of Events to which this event belongs.

- **Returns** – the set to which this event belongs.

- *label*

```
public java.lang.String label( )
```

- **Description**

If this function is not overridden by the user it returns the Event number. The user should override to give a short label description of the Event. It is highly recommended that the user overrides it to give a more descriptive label to be used when reporting the occurrence rates of the events.

- **Returns** – A short string description of the Event.

- **See also**

* `Event.description()` (see 2.2.4, page 78)

- *toString*

```
public java.lang.String toString( )
```


2.2.5 Class EventsSet

This class represent a set of Events. The set should be build at the beggining and should not be changed in any way afterwards.

Declaration

```
public class EventsSet
  extends java.lang.Object
  implements Events
```

Constructor summary

EventsSet() Creates an empty set of Events;
EventsSet(E[]) Creates an empty set of Events;

Method summary

add(E) Adds the Event e to the set.
contains(Event) Returns true if the set contains this Event.
iterator() This method returns a safe way to walk through the events in a particular set.
size()
toEventArray() Returns an array with the Events in the set.
toString()

Constructors

- *EventsSet*
 public **EventsSet**()
 – **Description**
 Creates an empty set of Events;

- *EventsSet*
 public **EventsSet**(Event[] eventArray)
 – **Description**
 Creates an empty set of Events;
 – **Parameters**
 * eventArray – an array representation of the set.

Methods

- *add*
 public boolean **add**(Event e)
 – **Description**
 Adds the Event e to the set.
 – **Parameters**

- * **e** – The event to be added.
 - **Returns** – True if the set did not already contained this event.

- *contains*
`public boolean contains(Event e)`
 - **Description**
Returns true if the set contains this Event.
 - **Parameters**
 - * **e** – The event.
 - **Returns** – whether the set contains this event.

- *iterator*
`public final java.util.Iterator iterator()`
 - **Description**
This method returns a safe way to walk through the events in a particular set. Collections and their implementations (Set, List, and Map) have iterators defined by default.
 - **Returns** – iterator over the events.

- *size*
`int size()`
 - **Description copied from Events** (see 2.1.2, page 62)
Returns the number of elements.
 - **Returns** – the number of Event elements.

- *toEventArray*
`public Event[] toEventArray()`
 - **Description**
Returns an array with the Events in the set.
 - **Returns** – array representation of the set.

- *toString*
`public java.lang.String toString()`

2.2.6 Class Policy

Policy is a set of "Decision Rules". It contains the Decision Rule for every stage t . When the problem has infinite horizon, only one decision rule is stored.

Declaration

```
public final class Policy
extends java.lang.Object
```

Constructor summary

Policy(DecisionRule) Creates a stationary policy with the given decision rule
Policy(int) Creates a set with the given horizon.

Method summary

getAction(S, int) Gets the action to be taken in state i at this stage t
getDecisionRule()
getDecisionRule(int) Returns the decision rule for stage t
getHorizon() Return the time horizon for this Policy.
print() Prints the policy to the standard output
print(PrintWriter) Prints the policy to the given PrintWriter.
setDecisionRule(DecisionRule) Sets a unique decision rule for the policy, for infinite horizon problems.
setDecisionRule(DecisionRule, int) Sets a decision rule for stage t in the policy
toString() Gives the string representation of this Policy

Constructors

- *Policy*
 public **Policy(DecisionRule d)**
 - **Description**
 Creates a stationary policy with the given decision rule
 - **Parameters**
 * **d** – The rule

- *Policy*
 public **Policy(int stages)**
 - **Description**
 Creates a set with the given horizon.
 - **Parameters**
 * **stages** – The number of stages

Methods

- *getAction*
 public Action **getAction**(State **i**, int **t**)
 – **Description**
 Gets the action to be taken in state **i** at this stage **t**
 – **Parameters**
 * **i** – The state
 * **t** – The stage (time) at which action is to be taken.
 – **Returns** – The action.

- *getDecisionRule*
 public DecisionRule **getDecisionRule**()
 – **Returns** – the unique decision rule, for infinite horizon problems.

- *getDecisionRule*
 public DecisionRule **getDecisionRule**(int **t**)
 – **Description**
 Returns the decision rule for stage **t**
 – **Parameters**
 * **t** – stage.
 – **Returns** – The decision rule for stage **t**.

- *getHorizon*
 public int **getHorizon**()
 – **Description**
 Return the time horizon for this Policy.
 – **Returns** – last stage where actions can be taken

- *print*
 public void **print**()
 – **Description**
 Prints the policy to the standard output

- *print*
 public void **print**(java.io.PrintWriter **pw**)
 – **Description**
 Prints the policy to the given PrintWriter.
 – **Parameters**
 * **pw** – print writer

- *setDecisionRule*
 public void **setDecisionRule**(DecisionRule **pol**)
 – **Description**
 Sets a unique decision rule for the policy, for infinite horizon problems.

- **Parameters**

- * `pol` –

- *setDecisionRule*

```
public void setDecisionRule( DecisionRule dr, int t )
```

- **Description**

- Sets a decision rule for stage `t` in the policy

- **Parameters**

- * `dr` – decision rule

- * `t` – stage

- *toString*

```
public java.lang.String toString( )
```

- **Description**

- Gives the sting representation of this Policy

2.2.7 Class PropertiesAction

This class is an easy way to use a Action that is represented by an integer valued array.

Declaration

```
public class PropertiesAction
extends jmarkov.basic.Action (see 2.2.1, page 71)
implements PropertiesElement
```

Constructor summary

PropertiesAction(int) Creates an Action Object wit an array of the given size.

PropertiesAction(int[]) Builds an object with the given array.

Method summary

clone()

compareTo(Action)

compareTo(PropertiesAction)

getNumProps() Returns the number of properties in the array that characterize this element.

getProperties() Gets thae array of properties.

getProperty(int) Gets the value of this property.

label()

setProperty(int, int) Sets the value of the property at the given index

Constructors

- *PropertiesAction*
 public **PropertiesAction(int size)**
 - **Description**
 Creates an Action Object wit an array of the given size.
 - **Parameters**
 * size –
- *PropertiesAction*
 public **PropertiesAction(int[] properties)**
 - **Description**
 Builds an object with the given array.
 - **Parameters**
 * properties –

Methods

- *clone*
 protected native java.lang.Object **clone()** throws
java.lang.CloneNotSupportedException

- *compareTo*

```
public final int compareTo( Action a )
```
- *compareTo*

```
public final int compareTo( PropertiesAction a )
```

 - **Parameters**
 - * **a** – The action array to compare to
 - **Returns** – +1, -1 or 0.
 - **See also**
 - * `java.lang.Comparable.compareTo(Object)`

- *getNumProps*

```
public int getNumProps( )
```

 - **Description**
Returns the number of properties in the array that characterize this element.
 - **Returns** – The number of properties.

- *getProperties*

```
public final int[] getProperties( )
```

 - **Description**
Gets thae array of properties.
 - **Returns** – Returns the properties array.

- *getProperty*

```
public int getProperty( int index )
```

 - **Description**
Gets the value of this property.
 - **Parameters**
 - * **index** –
 - **Returns** – the property at the given index

- *label*

```
public abstract java.lang.String label( )
```

 - **Description copied from Action** (see 2.2.1, page 71)
The user MUST override this method to give a (hopefully short) label for the Action.
 - **Returns** – short description of the Action.

- *setProperty*

```
protected void setProperty( int index, int value )
```

 - **Description**
Sets the value of the property at the given index
 - **Parameters**
 - * **index** –
 - * **value** –

Members inherited from class `jmarkov.basic.Action` (see 2.2.1, page 71)

- `public String description()`
- `public final boolean equals(java.lang.Object o)`
- `public abstract String label()`
- `public final String toString()`

2.2.8 Class PropertiesEvent

This class is an easy way to use an event that is represented by an array of int.

Declaration

```
public class PropertiesEvent
extends jmarkov.basic.Event (see 2.2.4, page 78)
implements PropertiesElement
```

Constructor summary

- PropertiesEvent(int)** Creates a new PropertiesEvent with an array of the size indicated filled with zeros.
- PropertiesEvent(int[])** Builds a new event with characteristic array as a parameter.

Method summary

- clone()**
- compareTo(Event)**
- compareTo(PropertiesEvent)** Compares the properties in order.
- equals(PropertiesEvent)**
- getNumProps()** Returns the number of properties in the array that characterize this element.
- getProperties()** Gets the array of properties.
- getProperty(int)** Gets the value of this property.
- label()**
- setProperty(int, int)** Sets the value of the property at the given index

Constructors

- *PropertiesEvent*

```
public PropertiesEvent( int size )
```

 - **Description**
Creates a new PropertiesEvent with an array of the size indicated filled with zeros.
 - **Parameters**
* **size** – size of the characteristic array.

- *PropertiesEvent*

```
public PropertiesEvent( int[] status )
```

 - **Description**
Builds a new event with characteristic array as a parameter.
 - **Parameters**
* **status** – characteristic array of the event.

Methods

- *clone*
protected native java.lang.Object **clone**() throws
java.lang.CloneNotSupportedException
- *compareTo*
public int **compareTo**(Event ev)
 - **Description copied from Event** (see 2.2.4, page 78)
Returns positive if this Event has a higher number then the given event.
 - **See also**
* java.lang.Comparable.compareTo(Object)
- *compareTo*
public final int **compareTo**(PropertiesEvent e1)
 - **Description**
Compares the properties in order.
 - **Parameters**
* e1 – The PropertiesEvent to compare to.
 - **Returns** – +1, 0 or -1, according to whether this states dominates e1.
- *equals*
public final boolean **equals**(PropertiesEvent e)
 - **Parameters**
* e – teh PropertiesEvent to compare To
 - **Returns** – true if Events are equal
- *getNumProps*
public int **getNumProps**()
 - **Description**
Returns the number of properties in the array that characterize this element.
 - **Returns** – The number of properties.
- *getProperties*
public final int[] **getProperties**()
 - **Description**
Gets the array of properties.
 - **Returns** – Returns the properties array.
- *getProperty*
public int **getProperty**(int index)
 - **Description**
Gets the value of this property.
 - **Parameters**
* index –

- **Returns** – the property at the given index
-

- *label*

`public java.lang.String label()`

- **Description copied from Event** (see 2.2.4, page 78)

If this function is not overridden by the user it returns the Event number. The user should override to give a short label description of the Event. It is highly recommended that the user overrides it to give a more descriptive label to be used when reporting the occurrence rates of the events.

- **Returns** – A short string description of the Event.
- **See also**

* `Event.description()` (see 2.2.4, page 78)

- *setProperty*

`protected void setProperty(int index, int value)`

- **Description**

Sets the value of the property at the given index

- **Parameters**

* `index` –
 * `value` –

Members inherited from class `jmarkov.basic.Event` (see 2.2.4, page 78)

- `public int compareTo(Event ev)`
- `public String description()`
- `public final boolean equals(java.lang.Object o)`
- `public int getIndex()`
- `public EventsSet getSet()`
- `public String label()`
- `public final String toString()`

2.2.9 Class PropertiesState

The states are characterized by an array of integer-valued properties, whose meaning will change from implementation to implementation. The class must be extended.

Declaration

```
public class PropertiesState
extends jmarkov.basic.State (see 2.2.11, page 96)
implements PropertiesElement
```

Version

1.0a

Field summary

prop This array contains the properties that characterize the state.

Constructor summary

PropertiesState(int) Constructs a State characterized by K properties.

PropertiesState(int[]) This creates a PropertiesState with the given array.

PropertiesState(int[], boolean) This creates a PropertiesState with the given array.

PropertiesState(PropertiesState) Constructs a new State by cloning the given State.

Method summary

clone()

compareTo(PropertiesState) Compares according to the internal properties in lexicographic order.

compareTo(State)

computeMOPs(MarkovProcess) By default it computes the long run average for each property.

getNumProps()

getProperties() Crates a copy of the properties array.

getProperty(int)

isConsistent() It is strongly recommended that the user implements this method.

label() Returns a string representation of this state in vector form.

setProperty(int, int) Sets the value of the property at the given index

Fields

- protected final int **prop**
 - This array contains the properties that characterize the state.

Constructors

- *PropertiesState*

public PropertiesState(int K)

- **Description**

Constructs a State characterized by K properties. The original values of all the properties is 0.

- **Parameters**

* K – Number of Properties

- *PropertiesState*

public PropertiesState(int[] properties)

- **Description**

This creates a PropertiesState with the given array. WARNING: the array is NOT internally copied, so it is assumed then NO changes are made to this array after it is given to the constructor.

- **Parameters**

* **properties** – An integer valued array with the properties that characterize this state.

- *PropertiesState*

public PropertiesState(int[] properties, boolean deepCopy)

- **Description**

This creates a PropertiesState with the given array.

- **Parameters**

* **properties** – An integer valued array with the properties that characterize this state.
 * **deepCopy** – true if the constructor should make a deep copy of the array. This causes some overhead but increased security.

- *PropertiesState*

public PropertiesState(PropertiesState s)

- **Description**

Constructs a new State by cloning the given State. If you are extending PropertiesState you may want to include a code like:

```
public YourState clone() {
    return new PropertiesState(this);
}
```

- **Parameters**

* **s** – A given State.

Methods

- *clone*
 public PropertiesElement **clone**()
 – **See also**
 * PropertiesElement.clone() (see 2.1.4, page 65)

- *compareTo*
 public final int **compareTo**(PropertiesState ps)
 – **Description**
 Compares according to the internal properties in lexicographic order.
 – **Parameters**
 * ps – The state to compare to
 – **Returns** – +1,-1,0 depending on relative ordering.
 – **See also**
 * State.compareTo(State) (see 2.2.11, page 97)

- *compareTo*
 public abstract int **compareTo**(State j)
 – **Description copied from State** (see 2.2.11, page 96)
 The method compareTo should be implemented in order to establish a total ordering among the States.
 – **Returns** – A positive integer if this is greater than j, negative if this is less than j and 0 if this == j.
 – **See also**
 * java.lang.Comparable.compareTo(Object)

- *computeMOPs*
 public void **computeMOPs**(jmarkov.MarkovProcess mp)
 – **Description**
 By default it computes the long run average for each property. the user should override this method in order to compute more meaningful measures of performance.
 – **See also**
 * State.computeMOPs(jmarkov.MarkovProcess) (see 2.2.11, page 97)

- *getNumProps*
 public int **getNumProps**()
 – **See also**
 * PropertiesElement.getNumProps() (see 2.1.4, page 65)

- *getProperties*
 public final int[] **getProperties**()
 – **Description**
 Creates a copy of the properties array.

– **See also**

* `PropertiesElement.getProperties()` (see 2.1.4, page 65)

• *getProperty*

`public int getProperty(int index)`

– **See also**

* `PropertiesElement.getProperty(int)` (see 2.1.4, page 66)

• *isConsistent*

`public boolean isConsistent()`

– **Description**

It is strongly recommended that the user implements this method. If left unimplemented this method returns true.

– **See also**

* `State.isConsistent()` (see 2.2.11, page 98)

• *label*

`public java.lang.String label()`

– **Description**

Returns a string representation of this state in vector form. The String will be in the form (p1,p2,...pK). A Class implementing this Class could give a more meaningful description.

• *setProperty*

`protected void setProperty(int index, int value)`

– **Description**

Sets the value of the property at the given index

– **Parameters**

* `index` –
 * `value` –

Members inherited from class `jmarkov.basic.State` (see 2.2.11, page 96)

- `public abstract int compareTo(State j)`
- `public abstract void computeMOPs(jmarkov.MarkovProcess model)`
- `public String description()`
- `public final boolean equals(java.lang.Object o)`
- `public final int getIndex()`
- `public double getMOP(int index)`
- `public final double getMOP(java.lang.String mopName, jmarkov.MarkovProcess model)`
- `public abstract boolean isConsistent()`
- `public abstract String label()`
- `public final int setMOP(int index, double value)`
- `public int setMOP(jmarkov.MarkovProcess model, java.lang.String mopName, double value)`
- `public final String toString()`

2.2.10 Class Solution

This class represents the joint information of a value function and a policy which summarizes the solution to a problem.

Declaration

```
public class Solution
  extends java.lang.Object
```

Constructor summary

Solution(ValueFunction, Policy) Builds a solution given a value function and a policy

Method summary

getPolicy() Returns the Policy.
getValueFunction() Returns the valueFunction.

Constructors

- *Solution*
public **Solution**(ValueFunction valueFunction, Policy policy)
 - **Description**
Builds a solution given a value function and a policy
 - **Parameters**
 - * valueFunction – value function
 - * policy – policy

Methods

- *getPolicy*
public Policy **getPolicy**()
 - **Description**
Returns the Policy.
 - **Returns** – Returns the policy.
- *getValueFunction*
public ValueFunction **getValueFunction**()
 - **Description**
Returns the valueFunction.
 - **Returns** – Returns the valueFunction.

2.2.11 Class State

The Class State represent a state in a MarkovProcess or MDP. The user of the class should establish her own coding convention AND code the compareTo method. If the State can be represented with a vector of integers describing its properties, then it might be easier to implement PropertiesState rather than State.

See also

- `PropertiesState` (see 2.2.9, page 91)

Declaration

```
public abstract class State
extends java.lang.Object
implements java.lang.Comparable, JMarkovElement
```

Version

1.0a

All known subclasses

`GeomState` (see 1.1.4, page 26) , `GeomRelState` (see 1.1.3, page 23) , `StateEvent` (see 2.2.13, page 102) , `StateC` (see 2.2.12, page 100) , `PropertiesState` (see 2.2.9, page 91)

Constructor summary

State()

Method summary

compareTo(State) The method compareTo should be implemented in order to establish a total ordering among the States.

computeMOPs(MarkovProcess) This method should be implemented in order to compute all the measures of performance MOPs.

description() Returns a String that describes the State.

equals(Object) If Object is not State it returns false.

getIndex()

getMOP(int) Gets the value of this MOP.

getMOP(String, MarkovProcess) Gets the value of the MOP with this name, by calling `getMOP(int)`

isConsistent() This method is called when a state is added to a set, if assertions are enabled.

label() Returns a (hopefully short) label that describes the State.

setMOP(int, double) Sets the value of this MOP.

setMOP(MarkovProcess, String, double) Sets the value of the MOP with this name.

toString() Returns the label.

Constructors

- *State*
`public State()`

Methods

- *compareTo*
`public abstract int compareTo(State j)`
 - **Description**
The method compareTo should be implemented in order to establish a total ordering among the States.
 - **Returns** – A positive integer if this is greater than j, negative if this is less than j and 0 if this == j.
 - **See also**
 - * `java.lang.Comparable.compareTo(Object)`

- *computeMOPs*
`public abstract void computeMOPs(jmarkov.MarkovProcess model)`
 - **Description**
This method should be implemented in order to compute all the measures of performance MOPs. Inside it you should issue commands like `setMop("Utilization server 1", x, model);`. * For large models override this method as empty and rather override `getMOP(int)`. Do NOT mix both approaches!!
 - **Parameters**
 - * `model` – The model being solved.
 - **See also**
 - * `State.getMOP(int)` (see 2.2.11, page 98)

- *description*
`public java.lang.String description()`
 - **Description**
Returns a String that describes the State. By default it is an empty string, but you should implement it in order to get a meaningful description.
 - **Returns** – A String description of the State

- *equals*
`public final boolean equals(java.lang.Object o)`
 - **Description**
If Object is not State it returns false. Otherwise `equals := (compareTo(o)==0)`
 - **See also**
 - * `java.lang.Object.equals(Object)`

- *getIndex*
`public final int getIndex()`

- **Returns** – The index in the State set
-

- *getMOP*

```
public double getMOP( int index )
```

- **Description**

Gets the value of this MOP. The value should had been set via the setMOP method. Alternatively, for better performance define the MOP Names when implementing the MarkovProcess class and override this method. To define the names in the constructor call the method setMOPs(String[]). The index is the same as the one used in the array in the aforementioned method.

- **Parameters**

- * *index* –

- **Returns** – The value of this MOP.

- **See also**

- * `jmarkov.MarkovProcess.setMOPs(java.lang.String[])` (see 1.1.5, page 50)

- *getMOP*

```
public final double getMOP( java.lang.String mopName,  
jmarkov.MarkovProcess model )
```

- **Description**

Gets the value of the MOP with this name, by calling `getMOP(int)`

- **Parameters**

- * *mopName* – The name of the MOP.

- * *model* – Model being solved.

- **Returns** – current MOP value

- **See also**

- * `State.getMOP(int)` (see 2.2.11, page 98)

- *isConsistent*

```
public abstract boolean isConsistent( )
```

- **Description**

This method is called when a state is added to a set, if assertions are enabled. You should include code that checks the consistency of the parameters entered. It is very helpful during development. Once assertions are disabled, this will not reduce the speed of your program.

- **Returns** – true if the state is consistent.
-

- *label*

```
public abstract java.lang.String label( )
```

- **Description**

Returns a (hopefully short) label that describes the State. It is used by all print methods and in the GUI.

- **Returns** – A short String label that identifies the state.
-

- *setMOP*

```
public final int setMOP( int index, double value )
```

- **Description**
Sets the value of this MOP.
 - **Parameters**
 - * `index` –
 - * `value` –
 - **Returns** – the index where it was added.
-

- *setMOP*

```
public int setMOP( jmarkov.MarkovProcess model, java.lang.String
mopName, double value )
```

- **Description**
Sets the value of the MOP with this name. If no MOP with this name exists a new one is declared.
 - **Parameters**
 - * `mopName` –
 - * `model` – The model being solved.
 - * `value` –
 - **Returns** – the index where it was added.
-

- *toString*

```
public final java.lang.String toString( )
```

- **Description**
Returns the label.
- **See also**
 - * `State.label()` (see 2.2.11, page 98)

2.2.12 Class StateC

State to model shortest path problems.

Declaration

```
public abstract class StateC
extends jmarkov.basic.State (see 2.2.11, page 96)
```

Constructor summary

StateC() Default constructor
StateC(boolean) General constructor.

Method summary

isTerminal()

Constructors

- *StateC*
 public **StateC**()
 – **Description**
 Default constructor

- *StateC*
 public **StateC**(boolean t)
 – **Description**
 General constructor.
 – **Parameters**
 * t – Whether it is a terminal state or not.

Methods

- *isTerminal*
 public final boolean **isTerminal**()
 – **Returns** – Returns true if this a terminal state.

Members inherited from class jmarkov.basic.State (see 2.2.11, page 96)

- public abstract int **compareTo**(State j)
- public abstract void **computeMOPs**(jmarkov.MarkovProcess model)
- public String **description**()
- public final boolean **equals**(java.lang.Object o)
- public final int **getIndex**()
- public double **getMOP**(int index)
- public final double **getMOP**(java.lang.String mopName, jmarkov.MarkovProcess model)
- public abstract boolean **isConsistent**()

- `public abstract String label()`
- `public final int setMOP(int index, double value)`
- `public int setMOP(jmarkov.MarkovProcess model, java.lang.String mopName, double value)`
- `public final String toString()`

2.2.13 Class StateEvent

This class represents a state compounded of a state and an event. It is used for state expansion for the problems where actions can depend on the event that happens in a transition. Only future events that can occur from the state state should be allowed as events event.

Declaration

```
public class StateEvent
extends jmarkov.basic.State (see 2.2.11, page 96)
```

Constructor summary

StateEvent(S, E) Builds a new state with the event information

Method summary

```
compareTo(State)
computeMOPs(MarkovProcess)
getEvent() Gets the event.
getState() Gets the state.
isConsistent()
label()
```

Constructors

- *StateEvent*
public StateEvent(State state, Event event)
 - **Description**
Builds a new state with the event information
 - **Parameters**
 - * **state** – state
 - * **event** – event

Methods

- *compareTo*
public abstract int compareTo(State j)
 - **Description copied from State** (see 2.2.11, page 96)
The method compareTo should be implemented in order to establish a total ordering among the States.
 - **Returns** – A positive integer if this is greater than j, negative if this is less than j and 0 if this == j.
 - **See also**
 - * `java.lang.Comparable.compareTo(Object)`
- *computeMOPs*
public void computeMOPs(jmarkov.MarkovProcess model)

– See also

* `State.computeMOPs(jmarkov.MarkovProcess)` (see 2.2.11, page 97)

- *getEvent*

`public Event getEvent()`

– **Description**

Gets the event.

– **Returns** – the original event from the state

- *getState*

`public State getState()`

– **Description**

Gets the state.

– **Returns** – the original state from the state

- *isConsistent*

`public boolean isConsistent()`

– See also

* `State.isConsistent()` (see 2.2.11, page 98)

- *label*

`public abstract java.lang.String label()`

– **Description copied from State** (see 2.2.11, page 96)

Returns a (hopefully short) label that describes the State. It is used by all print methods and in the GUI.

– **Returns** – A short String label that identifies the state.

Members inherited from class jmarkov.basic.State (see 2.2.11, page 96)

- `public abstract int compareTo(State j)`
- `public abstract void computeMOPs(jmarkov.MarkovProcess model)`
- `public String description()`
- `public final boolean equals(java.lang.Object o)`
- `public final int getIndex()`
- `public double getMOP(int index)`
- `public final double getMOP(java.lang.String mopName, jmarkov.MarkovProcess model)`
- `public abstract boolean isConsistent()`
- `public abstract String label()`
- `public final int setMOP(int index, double value)`
- `public int setMOP(jmarkov.MarkovProcess model, java.lang.String mopName, double value)`
- `public final String toString()`

2.2.14 Class StatesSet

This class represent a set of States. It is used as a convinence to build multiple destinations in the method `reachable`.

See also

- `jmarkov.jmdp.DTMDP.reachable(State,Action)` (see 4.1.5, page 141)
- `jmarkov.jmdp.CTMDP.reachable(State,Action)` (see 4.1.2, page 128)

Declaration

```
public class StatesSet
  extends java.lang.Object
  implements States
```

Constructor summary

StatesSet() Creates an empty set of States;
StatesSet(Iterable) Creates a set of objects S from the given States.
StatesSet(S) Creates set of States with only this State;
StatesSet(S[]) Creates a set of objects S from a given set of States.
StatesSet(States) Creates a set of objects S from the given States .

Method summary

add(Iterable) Adds the States in the iterator to the set.
add(S) Adds the State s to the set.
add(States) Adds the States in the iterator to the set.
contains(S) Returns true if the set contains this State.
get(S) Returns the element that is equal (according to `equals()`) to the given element.
isClosed()
iterator() This method returns a safe way to walk through the states in a particular set.
numerateStates() This method numerates all states and returns the number of states found.
remove(S) Removes an object from the set.
size()
toStateArray() Returns an array with the States in the set.
toString()

Constructors

- *StatesSet*
 public StatesSet()
 – **Description**
 Creates an empty set of States;
- *StatesSet*
 public StatesSet(java.lang.Iterable states)

- **Description**
Creates a set of objects S from the given States.
 - **Parameters**
 - * **states** – a set of States of type States.
-

- *StatesSet*

```
public StatesSet( State s )
```

- **Description**
Creates set of States with only this State;
 - **Parameters**
 - * **s** – The state to include in the set.
-

- *StatesSet*

```
public StatesSet( State[] states )
```

- **Description**
Creates a set of objects S from a given set of States.
 - **Parameters**
 - * **states** – a set of States of type States.
-

- *StatesSet*

```
public StatesSet( States states )
```

- **Description**
Creates a set of objects S from the given States .
- **Parameters**
 - * **states** – a set of States of type States.

Methods

- *add*

```
public boolean add( java.lang.Iterable states )
```

- **Description**
Adds the States in the iterator to the set.
 - **Parameters**
 - * **states** – a set of States of type States.
 - **Returns** – True if the set did not contain ANY of the elements.
-

- *add*

```
public boolean add( State s )
```

- **Description**
Adds the State s to the set.
 - **Parameters**
 - * **s** – The State to be added.
 - **Returns** – True if the set did not already contained this event.
-

- *add*

```
public boolean add( States states )
```

- **Description**

Adds the States in the iterator to the set.

- **Parameters**

* **states** – a set of States of type States.

- **Returns** – True if the set did not contain ANY of the elements.

- *contains*

```
public boolean contains( State s )
```

- **Description**

Returns true if the set contains this State.

- **Parameters**

* **s** – A State

- **Returns** – true if the state is contained in the set.

- *get*

```
public State get( State state )
```

- **Description**

Returns the element that is equal (according to equals()) to the given element.

- **Parameters**

* **state** – The given state

- **Returns** – The state in the set, or null if it was not defined in the set.

- *isClosed*

```
boolean isClosed( )
```

- **Description copied from States** (see 2.1.5, page 67)

The set is closed if all elements have been added.

- **Returns** – true if the set is closed.

- *iterator*

```
public final java.util.Iterator iterator( )
```

- **Description**

This method returns a safe way to walk through the states in a particular set.

Collections and their implementations (Set, List, and Map) have iterators defined by default.

- **Returns** – iterator over the states.

- *numerateStates*

```
public int numerateStates( )
```

- **Description**

This method numerates all states and returns the number of states found. After this method is called it is illegal to add more states to the set.

- **Returns** – The number of states.

- *remove*

```
public boolean remove( State s )
```

- **Description**

Removes an object from the set.

- **Parameters**

* **s** – The element to remove.

- **Returns** – If the remove was successful (i.e. the element was in the set).

- *size*

```
int size( )
```

- **Description copied from States** (see 2.1.5, page 67)

Returns the number of elements.

- **Returns** – the number of State elements.

- *toArray*

```
public State[] toArray( )
```

- **Description**

Returns an array with the States in the set.

- **Returns** – An array representation of the states.

- *toString*

```
public java.lang.String toString( )
```

2.2.15 Class Transition

This class represent a transition to a given state. It has an associated rate and state.

Declaration

```
public final class Transition
extends java.lang.Object
implements JMarkovElement
```

Constructor summary

Transition(S, double) Basic constructor.

Method summary

```
description()
getRate() Returns the rate.
getState() Returns the state.
label()
toString()
```

Constructors

- *Transition*
 public **Transition**(State state, double rate)
 - **Description**
 Basic constructor.
 - **Parameters**
 - * state –
 - * rate –

Methods

- *description*
 java.lang.String **description**()
 - **Description copied from JMarkovElement** (see 2.1.3, page 63)
 This method return a complete verbal describtion of this element. This description may contain multiple text rows.
 - **Returns** – A String describing this element.
 - **See also**
 - * JMarkovElement.label() (see 2.1.3, page 64)
- *getRate*
 public final double **getRate**()
 - **Description**
 Returns the rate.

- **Returns** – Returns the rate.
-

- *getState*

```
public final State getState( )
```

- **Description**
Returns the state.
 - **Returns** – Returns the state.
-

- *label*

```
java.lang.String label( )
```

- **Description copied from JMarkovElement (see 2.1.3, page 63)**
This method returns a short String used in the user interface to describe this element.
 - **Returns** – A String label.
 - **See also**
* `JMarkovElement.description()` (see 2.1.3, page 64)
-

- *toString*

```
public java.lang.String toString( )
```

2.2.16 Class TransitionsSet

Declaration

```
public final class TransitionsSet
extends java.lang.Object
implements JMarkovElement, Transitions
```

Constructor summary

TransitionsSet() Default Constructor.

Method summary

add(S, double) Adds a transition with the given state and rate.
add(Transition)
add(Transitions) Adds all the given Transtions to the current set.
addRate(S, double) Adds the given rate to the transition to this state.
description()
getRate(S) Gets the rate for this state.
iterator() Returns an iterator used to walk through the Transitions.
label()
size()
toString()

Constructors

- *TransitionsSet*
public TransitionsSet()
 - **Description**
Default Constructor.

Methods

- *add*
public boolean add(State state, double rate)
 - **Description**
Adds a transition with the given state and rate.
 - **See also**
* `Transitions.add(State,double)` (see 2.1.6, page 69)
- *add*
public boolean add(Transition t)
 - **See also**
* `Transitions.add(Transition)` (see 2.1.6, page 69)

- *add*

```
public boolean add( Transitions trans )
```

 - **Description**
Adds all the given Transtions to the current set.
 - **Parameters**
 - * **trans** – A collection of Transitions
 - **Returns** – true if none of the elements was in the set.
 - **See also**
 - * `java.util.Set.addAll(java.util.Collection)`

- *addRate*

```
public double addRate( State state, double rate )
```

 - **Description**
Adds the given rate to the transition to this state.
 - **Parameters**
 - * **state** –
 - * **rate** –
 - **Returns** – The old value associated with this state.

- *description*

```
java.lang.String description( )
```

 - **Description copied from JMarkovElement** (see 2.1.3, page 63)
This method return a complete verbal describtion of this element. This description may contain multiple text rows.
 - **Returns** – A String describing this element.
 - **See also**
 - * `JMarkovElement.label()` (see 2.1.3, page 64)

- *getRate*

```
public double getRate( State state )
```

 - **Description**
Gets the rate for this state. It returns 0.0 if this state is not in the Transitions.
 - **Parameters**
 - * **state** –
 - **Returns** – The rate for this state

- *iterator*

```
public java.util.Iterator iterator( )
```

 - **Description**
Returns an iterator used to walk through the Transitions.
 - **See also**
 - * `java.lang.Iterable.iterator()`

- *label*

```
java.lang.String label( )
```


- **Description copied from JMarkovElement** (see 2.1.3, page 63)
This method returns a short String used in the user interface to describe this element.
 - **Returns** – A String label.
 - **See also**
 - * JMarkovElement.description() (see 2.1.3, page 64)
-

- *size*

`int size()`

- **Description copied from Transitions** (see 2.1.6, page 69)
Returns the number of Transitions represented by this object.
 - **Returns** – The number of Transitions.
 - **See also**
 - * java.util.Set.size()
-

- *toString*

`public java.lang.String toString()`

2.2.17 Class ValueFunction

This structure matches each state with a double number representing its value function, or in some cases the steady state probabilities.

Declaration

```
public class ValueFunction
extends java.lang.Object
implements JMarkovElement
```

Constructor summary

- ValueFunction()** Creates a new empty value function.
- ValueFunction(String)** Creates a new empty value function.
- ValueFunction(ValueFunction)** Creates a value function from another given value function
- ValueFunction(ValueFunction, String)** Creates a value function from another given value function

Method summary

- description()**
- get()** Gets an array with all the values represented in this value function.
- get(S)** Gets the Value associated with this State.
- iterator()** Return an iterator used to walk through the Value Function.
- label()**
- print(PrintWriter)** Prints the Value Function.
- print(PrintWriter, String, String)** Prints the Value function with the given state format , and values format according to the Format String Syntax.
- set(S, double)** Associates a state and a double value
- toString()**

Constructors

- *ValueFunction*
 public **ValueFunction**()
 – **Description**
 Creates a new empty value function.

- *ValueFunction*
 public **ValueFunction**(java.lang.String name)
 – **Description**
 Creates a new empty value function.
 – **Parameters**
 * **name** – The name for the value function

- *ValueFunction*
 public **ValueFunction**(ValueFunction vf)

- **Description**
Creates a value function from another given value function
 - **Parameters**
 - * **vf** – value function
-

- *ValueFunction*

`public ValueFunction(ValueFunction vf, java.lang.String name)`

- **Description**
Creates a value function from another given value function
- **Parameters**
 - * **vf** – value function
 - * **name** – The name for this value function.

Methods

- *description*

`java.lang.String description()`

- **Description copied from JMarkovElement** (see 2.1.3, page 63)
This method return a complete verbal description of this element. This description may contain multiple text rows.
 - **Returns** – A String describing this element.
 - **See also**
 - * `JMarkovElement.label()` (see 2.1.3, page 64)
-

- *get*

`public double[] get()`

- **Description**
Gets an array with all the values represented in this value function.
 - **Returns** – an array with the values
-

- *get*

`public double get(State s)`

- **Description**
Gets the Value associated with this State.
 - **Parameters**
 - * **s** – given state
 - **Returns** – the double value corresponding to the state
-

- *iterator*

`public java.util.Iterator iterator()`

- **Description**
Return an iterator used to walk through the Value Function.
 - **Returns** – iterator over the entries of the map
-

- *label*

```
java.lang.String label( )
```

- **Description** copied from JMarkovElement (see 2.1.3, page 63)

This method returns a short String used in the user interface to describe this element.

- **Returns** – A String label.

- **See also**

* JMarkovElement.description() (see 2.1.3, page 64)

- *print*

```
public void print( java.io.PrintWriter pw )
```

- **Description**

Prints the Value Function. It uses default states and values format.

- **Parameters**

* pw –

- *print*

```
public void print( java.io.PrintWriter pw, java.lang.String statesFormat,
java.lang.String valuesFormat )
```

- **Description**

Prints the Value function with the given state format , and values format according to the Format String Syntax.

- **Parameters**

* pw –

* **statesFormat** – format for the states , for example "%-10S" to have 10 width left aligned states.

* **valuesFormat** – format to use for values. For example us "%6.2" to have 6 width and 2 decimals.

- **See also**

* java.util.Formatter

- *set*

```
public void set( State s, double val )
```

- **Description**

Associates a state and a double value

- **Parameters**

* **s** – state

* **val** – value

- *toString*

```
public java.lang.String toString( )
```

Chapter 3

Package jmarkov.basic.exceptions

Package Contents

Page

This package contains the definition of the Exceptions thrown by jMarkov.

3.1 Exceptions

3.1.1 Class NonStochasticException

This Exception indicates that the transition probability matrix is not stochastic for the state and action computed. The matrix is not stochastic when the sum of the transition probabilities (row) is not 1.0.

Declaration

```
public class NonStochasticException
extends java.lang.RuntimeException
```

Constructor summary

NonStochasticException(String) Default constructor.

Constructors

- *NonStochasticException*
 public **NonStochasticException**(java.lang.String message)
 - **Description**
 Default constructor.
 - **Parameters**
 * message –

Members inherited from class java.lang.RuntimeException

Members inherited from class java.lang.Exception

Members inherited from class java.lang.Throwable

- public synchronized native Throwable **fillInStackTrace**()
- public Throwable **getCause**()
- public String **getLocalizedMessage**()
- public String **getMessage**()
- public StackTraceElement **getStackTrace**()
- public synchronized Throwable **initCause**(Throwable arg0)
- public void **printStackTrace**()
- public void **printStackTrace**(java.io.PrintStream arg0)
- public void **printStackTrace**(java.io.PrintWriter arg0)
- public void **setStackTrace**(StackTraceElement[] arg0)
- public String **toString**()

3.1.2 Class NotUnichainException

This Exception should be thrown by the SteadyStateSolver if it detects that there is not a unique solution to the stationary probabilities. This occurs when there are multiple closed communicating classes in the system, and therefore the corresponding linear system has multiple solutions.

See also

- `jmarkov.solvers.SteadyStateSolver` (see 6.1.10, page 250)

Declaration

```
public class NotUnichainException
extends jmarkov.basic.exceptions.SolverException (see 3.1.3, page 120)
```

Constructor summary

- NotUnichainException(String)** Default constructor.
- NotUnichainException(String, Throwable)** Constructor with cause.

Constructors

- *NotUnichainException*

```
public NotUnichainException( java.lang.String message )
```

 - **Description**
Default constructor.
 - **Parameters**
 - * `message` –
- *NotUnichainException*

```
public NotUnichainException( java.lang.String message, java.lang.Throwable cause )
```

 - **Description**
Constructor with cause.
 - **Parameters**
 - * `message` –
 - * `cause` –

Members inherited from class `jmarkov.basic.exceptions.SolverException` (see 3.1.3, page 120)

Members inherited from class `java.lang.Exception`

Members inherited from class `java.lang.Throwable`

- `public synchronized native Throwable fillInStackTrace()`
- `public Throwable getCause()`
- `public String getLocalizedMessage()`
- `public String getMessage()`
- `public StackTraceElement getStackTrace()`

- `public synchronized Throwable initCause(Throwable arg0)`
- `public void printStackTrace()`
- `public void printStackTrace(java.io.PrintStream arg0)`
- `public void printStackTrace(java.io.PrintWriter arg0)`
- `public void setStackTrace(StackTraceElement[] arg0)`
- `public String toString()`

3.1.3 Class SolverException

This exception is thrown by solve methods.

Declaration

```
public class SolverException
extends java.lang.Exception
```

All known subclasses

StructureException (see 3.1.4, page 121) , NotUnichainException (see 3.1.2, page 118)

Constructor summary

SolverException(String)
SolverException(String, Throwable)

Constructors

- *SolverException*
 public **SolverException**(java.lang.String message)
 – Parameters
 * message –

- *SolverException*
 public **SolverException**(java.lang.String message, java.lang.Throwable cause)
 – Parameters
 * message –
 * cause –

Members inherited from class java.lang.Exception

Members inherited from class java.lang.Throwable

- public synchronized native Throwable fillInStackTrace()
- public Throwable getCause()
- public String getLocalizedMessage()
- public String getMessage()
- public StackTraceElement getStackTrace()
- public synchronized Throwable initCause(Throwable arg0)
- public void printStackTrace()
- public void printStackTrace(java.io.PrintStream arg0)
- public void printStackTrace(java.io.PrintWriter arg0)
- public void setStackTrace(StackTraceElement[] arg0)
- public String toString()

3.1.4 Class StructureException

This exception is produced in shortest path problems if the conditions for convergence are not met.

Declaration

```
public class StructureException
extends jmarkov.basic.exceptions.SolverException (see 3.1.3, page 120)
```

Constructor summary

StructureException(String) Default constructor.

Constructors

- *StructureException*
public StructureException(java.lang.String message)
 - **Description**
Default constructor.
 - **Parameters**
* *message* –

Members inherited from class `jmarkov.basic.exceptions.SolverException` (see 3.1.3, page 120)

Members inherited from class `java.lang.Exception`

Members inherited from class `java.lang.Throwable`

- **public synchronized native Throwable fillInStackTrace()**
- **public Throwable getCause()**
- **public String getLocalizedMessage()**
- **public String getMessage()**
- **public StackTraceElement getStackTrace()**
- **public synchronized Throwable initCause(Throwable arg0)**
- **public void printStackTrace()**
- **public void printStackTrace(java.io.PrintStream arg0)**
- **public void printStackTrace(java.io.PrintWriter arg0)**
- **public void setStackTrace(StackTraceElement[] arg0)**
- **public String toString()**

Chapter 4

Package jmarkov.jmdp

Package Contents

Page

Classes

CT2DTConverter	123
<i>This class formulates a DTMDP equivalent to a CTMDP.</i>	
CTMDP	126
<i>This class represents a continuous time MDP.</i>	
CTMDPEv	131
<i>This class represents an Infinite horizon, continuous time Markov Decision Process with events.</i>	
CTMDPEvA	135
<i>This class represents an Infinite horizon, continuous time Markov Decision Process with events where actions depend on events.</i>	
DTMDP	139
<i>This class represents a discrete time infinite horizon MDP problem.</i>	
DTMDPEv	143
<i>This class represents an infinite horizon, discrete time, Markov Decision Process with events.</i>	
DTMDPEvA	147
<i>This class represents an infinite horizon, discrete time, Markov Decision Process with events, where actions depend on events.</i>	
FiniteDP	151
<i>This class should ONLY be used in FINITE horizon deterministic problems.</i>	
FiniteMDP	153
<i>This class should ONLY be used in FINITE horizon problems.</i>	
FiniteMDPEv	157
<i>This class represents a finite horizon discrete time MDP with events.</i>	
InfiniteMDP	161
<i>This class is a structural class and is.</i>	
MDP	165
<i>This class is the main framework to build a Dynamic Programming Problem.</i>	
StochasticShortestPath	170
<i>This class represents an infinite horizon shortest path problem.</i>	

jMDP is used to solve Markov Decision Processes. See the jMDP manual for details.

4.1 Classes

4.1.1 Class CT2DTConverter

This class formulates a DTMDP equivalent to a CTMDP.

Declaration

```
public class CT2DTConverter
  extends jmarkov.jmdp.DTMDP (see 4.1.5, page 139)
```

Constructor summary

CT2DTConverter(CTMDP) Constructor is not public because it should only be invoked by CTMDP in this same package.

Method summary

exitRate(S, A) This method calculates the exit rate for a given state and action.
feasibleActions(S)
immediateCost(S, A)
prob(S, S, A)
reachable(S, A)

Constructors

- *CT2DTConverter*
public CT2DTConverter(CTMDP problem)
 - **Description**
 Constructor is not public because it should only be invoked by CTMDP in this same package.
 - **Parameters**
 - * **problem** – an infinite horizon continuous time problem

Methods

- *exitRate*
public double exitRate(jmarkov.basic.State i, jmarkov.basic.Action a)
 - **Description**
 This method calculates the exit rate for a given state and action. It sums all rates for all reachable states under that action.
 - **Parameters**
 - * **i** – current state
 - * **a** – current action
 - **Returns** – The soujurn rate for a given state and action

- *feasibleActions*

```
public abstract jmarkov.basic.Actions feasibleActions( jmarkov.basic.State i )
```

- **Description copied from InfiniteMDP** (see 4.1.11, page 161)

Returns the set of actions available at this state.

- **Parameters**

- * *i* – Current State

- **Returns** – set of Actions that can be taken at this state.

- *immediateCost*

```
public abstract double immediateCost( jmarkov.basic.State i,
jmarkov.basic.Action a )
```

- **Description copied from DTMDP** (see 4.1.5, page 139)

Cost incurred when taking action *a* from state *i*

- **Parameters**

- * *i* – Current State

- * *a* – Current Action

- **Returns** – The cost incurred per transition

- *prob*

```
public abstract double prob( jmarkov.basic.State i, jmarkov.basic.State j,
jmarkov.basic.Action a )
```

- **Description copied from DTMDP** (see 4.1.5, page 139)

Probability of going from state *i* to state *j* by taking the action *a*

- **Parameters**

- * *i* – Current state.

- * *j* – Destination State

- * *a* – Action

- **Returns** – The probability.

- *reachable*

```
public abstract jmarkov.basic.States reachable( jmarkov.basic.State i,
jmarkov.basic.Action a )
```

- **Description copied from DTMDP** (see 4.1.5, page 139)

Set of states that can be reached from this state *i*, after taking the action *a*.

- **Parameters**

- * *i* – Current State

- * *a* – Action taken

- **Returns** – The reachable states.

Members inherited from class jmarkov.jmdp.DTMDP (see 4.1.5, page 139)

- protected StatesSet generate()

- public ValueFunction getSteadyStateProbabilities() throws
jmarkov.basic.exceptions.SolverException

- public abstract double immediateCost(jmarkov.basic.State i, jmarkov.basic.Action a)

- protected StatesSet **oneStageReachable**(jmarkov.basic.States **initSet**)
- public abstract double **prob**(jmarkov.basic.State **i**, jmarkov.basic.State **j**, jmarkov.basic.Action **a**)
- public abstract States **reachable**(jmarkov.basic.State **i**, jmarkov.basic.Action **a**)
- public void **setProbabilitySolver**(solvers.ProbabilitySolver **solv**)
- public final Solution **solve**(double **interestRate**) throws jmarkov.basic.exceptions.SolverException

Members inherited from class jmarkov.jmdp.InfiniteMDP (see 4.1.11, page 161)

- protected absorbingStates
- protected explorationTime
- public abstract Actions **feasibleActions**(jmarkov.basic.State **i**)
- protected abstract StatesSet **generate**()
- public StatesSet **getAllStates**()
- protected AbstractAverageSolver **getDefaultAverageSolver**()
- protected AbstractDiscountedSolver **getDefaultDiscountedSolver**(double **interestRate**)
- protected AbstractInfiniteSolver **getDefaultSolver**()
- public final int **getNumStates**()
- public AbstractInfiniteSolver **getSolver**()
- protected hasAbsorbingState
- protected numStates
- protected probability
- protected probabilitySolver
- protected void **setInterestRate**(double **interestRate**)
- protected states

Members inherited from class jmarkov.jmdp.MDP (see 4.1.12, page 165)

- public void **debug**(int **level**, java.lang.String **message**)
- public void **debug**(int **level**, java.lang.String **s**, boolean **newline**)
- public void **debug**(int **level**, java.lang.String **s**, boolean **newline**, boolean **indent**)
- protected finite
- public int **getDebugLevel**()
- protected abstract Solver **getDefaultSolver**()
- public final Policy **getOptimalPolicy**() throws jmarkov.basic.exceptions.SolverException
- public ValueFunction **getOptimalValueFunction**() throws jmarkov.basic.exceptions.SolverException
- public DebugReporter **getReporter**()
- public Solver **getSolver**()
- protected initial
- public final boolean **isFinite**()
- public final boolean **isSolved**()
- public double **operation**(double **present**, double **future**)
- public void **printSolution**()
- public void **printSolution**(java.io.PrintWriter **pw**)
- protected reporter
- public void **setDebugLevel**(int **level**)
- public void **setReporter**(jmarkov.DebugReporter **reporter**)
- public void **setSolver**(solvers.Solver **solver**)
- public final void **solve**() throws jmarkov.basic.exceptions.SolverException

4.1.2 Class CTMDP

This class represents a continuous time MDP. It should ONLY be used in INFINITE horizon Problems. It must be extended in order to represent the appropriate structure for each INFINITE horizon MDP problem. The user must implement at least the functions that have been declared abstract. It is also necessary to create one of the extensions of the class Solver. By default, the program includes PolicyIterationSolver and ValueIterationSolver classes to solve infinite horizon problems. The FiniteSolver class is only for finite horizon problems. To solve the problem follow the instructions in each of the solvers's instructions.

See also

- `solvers.PolicyIterationSolver` (see 5.2.11, page 205)
- `solvers.ValueIterationSolver` (see 5.2.16, page 224)

Declaration

```
public abstract class CTMDP
  extends jmarkov.jmdp.InfiniteMDP (see 4.1.11, page 161)
```

All known subclasses

CTMDPEvA (see 4.1.4, page 135) , CTMDPEv (see 4.1.3, page 131)

Field summary

activeState No earthly idea what this is for..
converter The converter used to map the problem to a DTMDP.
maxRate Tha maxRate used for uniformization

Constructor summary

CTMDP(States) Creates a new continuous time infinite horizon MDP Problem.

Method summary

continuousCost(S, A) Cost incurred continuously in time until the next transition from state i given that action a is taken.
generate()
getAllStates() Complete set of states explored
getMaxRate()
getSteadyStateProbabilities()
lumpCost(S, A) Cost incurred instantaneously in the moment when action a is taken from state i.
oneStageReachable(States) Finds the states reachable in one step.
rate(S, S, A) Rate of going from state i to state j by taking the action a
reachable(S, A) Set of States that can be reached from this state i, after taking the action a.
setConverter(CT2DTConverter) Sets the class in charge of making a DTMDP equivalent to the CTMDP
solve(double) Solves the problem with the given interest rate

Fields

- protected double **maxRate**
 - The maxRate used for uniformization
- protected jmarkov.basic.State **activeState**
 - No earthly idea what this is for..
- protected CT2DTConverter **converter**
 - The converter used to map the problem to a DTMDP.

Constructors

- *CTMDP*
`public CTMDP(jmarkov.basic.States initial)`
 - **Description**
Creates a new continuous time infinite horizon MDP Problem.
 - **Parameters**
 - * `initial` – set of initial states for the exploration algorithm

Methods

- *continuousCost*
`public abstract double continuousCost(jmarkov.basic.State i, jmarkov.basic.Action a)`
 - **Description**
Cost incurred continuously in time until the next transition from state i given that action a is taken.
 - **Parameters**
 - * `i` – State
 - * `a` – Action
 - **Returns** – Rate at which cost is incurred when action a is taken.

- *generate*
`protected abstract jmarkov.basic.StatesSet generate()`
 - **Returns** – The set of states found.

- *getAllStates*
`public jmarkov.basic.StatesSet getAllStates()`
 - **Description**
Complete set of states explored
 - **Returns** – set of states explored

- *getMaxRate*
`public double getMaxRate()`
 - **Returns** – maximum exit rate for all states and all actions

- *getSteadyStateProbabilities*
`public jmarkov.basic.ValueFunction getSteadyStateProbabilities() throws
jmarkov.basic.exceptions.SolverException`
 - **Returns** – The steady state probability for each state
 - **Throws**
 - * `jmarkov.basic.exceptions.SolverException` –

- *lumpCost*
`public abstract double lumpCost(jmarkov.basic.State i, jmarkov.basic.Action
a)`
 - **Description**
Cost incurred instantaneously in the moment when action a is taken from state i.
 - **Parameters**
 - * `i` – State
 - * `a` – Action
 - **Returns** – Lump cost received.

- *oneStageReachable*
`protected jmarkov.basic.StatesSet oneStageReachable(jmarkov.basic.States
initSet)`
 - **Description**
Finds the states reachable in one step.
 - **Parameters**
 - * `initSet` –
 - **Returns** – States reachable from this set

- *rate*
`public abstract double rate(jmarkov.basic.State i, jmarkov.basic.State j,
jmarkov.basic.Action a)`
 - **Description**
Rate of going from state i to state j by taking the action a
 - **Parameters**
 - * `i` – current state
 - * `j` – Destination state.
 - * `a` – Action taken
 - **Returns** – The rate

- *reachable*
`public abstract jmarkov.basic.States reachable(jmarkov.basic.State i,
jmarkov.basic.Action a)`
 - **Description**
Set of States that can be reached from this state i, after taking the action a.

- **Parameters**

- * **i** – current State
- * **a** – action taken

- **Returns** – the reachable states.

- *setConverter*

public void setConverter(CT2DTConverter converter)

- **Description**

Sets the class in charge of making a DTMDP equivalent to the CTMDP

- **Parameters**

- * **converter** – class that makes a DTMDP equivalent to the CTMDP

- *solve*

**public jmarkov.basic.Solution solve(double interestRate) throws
jmarkov.basic.exceptions.SolverException**

- **Description**

Solves the problem with the given interest rate

- **Parameters**

- * **interestRate** – the interest rate parameter to solve the problem.

- **Returns** – The Optimal solution.

- **Throws**

- * **jmarkov.basic.exceptions.SolverException** –

Members inherited from class jmarkov.jmdp.InfiniteMDP (see 4.1.11, page 161)

- **protected absorbingStates**
- **protected explorationTime**
- **public abstract Actions feasibleActions(jmarkov.basic.State i)**
- **protected abstract StatesSet generate()**
- **public StatesSet getAllStates()**
- **protected AbstractAverageSolver getDefaultAverageSolver()**
- **protected AbstractDiscountedSolver getDefaultDiscountedSolver(double interestRate)**
- **protected AbstractInfiniteSolver getDefaultSolver()**
- **public final int getNumStates()**
- **public AbstractInfiniteSolver getSolver()**
- **protected hasAbsorbingState**
- **protected numStates**
- **protected probability**
- **protected probabilitySolver**
- **protected void setInterestRate(double interestRate)**
- **protected states**

Members inherited from class jmarkov.jmdp.MDP (see 4.1.12, page 165)

- public void **debug**(int level, java.lang.String message)
- public void **debug**(int level, java.lang.String s, boolean newline)
- public void **debug**(int level, java.lang.String s, boolean newline, boolean indent)
- protected **finite**
- public int **getDebugLevel**()
- protected abstract Solver **getDefaultSolver**()
- public final Policy **getOptimalPolicy**() throws
jmarkov.basic.exceptions.SolverException
- public ValueFunction **getOptimalValueFunction**() throws
jmarkov.basic.exceptions.SolverException
- public DebugReporter **getReporter**()
- public Solver **getSolver**()
- protected **initial**
- public final boolean **isFinite**()
- public final boolean **isSolved**()
- public double **operation**(double present, double future)
- public void **printSolution**()
- public void **printSolution**(java.io.PrintWriter pw)
- protected **reporter**
- public void **setDebugLevel**(int level)
- public void **setReporter**(jmarkov.DebugReporter reporter)
- public void **setSolver**(solvers.Solver solver)
- public final void **solve**() throws jmarkov.basic.exceptions.SolverException

4.1.3 Class CTMDPEv

This class represents an Infinite horizon, continuous time Markov Decision Process with events. It allows the definition of events that can occur in a given state and this makes the reward and probability definition easier than in the cases where no events are defined.

Declaration

```
public abstract class CTMDPEv
  extends jmarkov.jmdp.CTMDP (see 4.1.2, page 126)
```

Constructor summary

CTMDPEv(States) This constructor builds a continuous time MDP with events.

Method summary

activeEvents(S, A) Set of events that are active from state i given that action a is taken.

continuousCost(S, A)

continuousCost(S, A, E) Reward obtained continuously in time during the sojourn time in state i until an action a is taken and a transition is triggered.

lumpCost(S, A)

lumpCost(S, A, E) Reward instantaneously gained in the moment when action a is taken from state i.

rate(S, S, A)

rate(S, S, A, E) Rate.

reachable(S, A)

reachable(S, A, E) Set of reachable states from state i given that action a is taken and event e occurs.

Constructors

- *CTMDPEv*
 public CTMDPEv(jmarkov.basic.States initial)
 - **Description**
 This constructor builds a continuous time MDP with events.
 - **Parameters**
 * initial – set of initial states for the exploration algorithm

Methods

- *activeEvents*
 public abstract jmarkov.basic.Events activeEvents(jmarkov.basic.State i, jmarkov.basic.Action a)
 - **Description**
 Set of events that are active from state i given that action a is taken.

- **Parameters**
 - * **i** – current state
 - * **a** – action taken
 - **Returns** – set of events that can occur
-

- *continuousCost*

```
public abstract double continuousCost( jmarkov.basic.State i,
jmarkov.basic.Action a )
```

- **Description copied from CTMDP** (see 4.1.2, page 126)
Cost incurred continuously in time until the next transition from state *i* given that action *a* is taken.
 - **Parameters**
 - * **i** – State
 - * **a** – Action
 - **Returns** – Rate at which cost is incurred when action *a* is taken.
-

- *continuousCost*

```
public abstract double continuousCost( jmarkov.basic.State i,
jmarkov.basic.Action a, jmarkov.basic.Event e )
```

- **Description**
Reward obtained continuously in time during the sojourn time in state *i* until an action *a* is taken and a transition is triggered.
 - **Parameters**
 - * **i** – current state
 - * **a** – action taken
 - * **e** – event that occurs
 - **Returns** – instantaneous reward.
-

- *lumpCost*

```
public abstract double lumpCost( jmarkov.basic.State i, jmarkov.basic.Action
a )
```

- **Description copied from CTMDP** (see 4.1.2, page 126)
Cost incurred instantaneously in the moment when action *a* is taken from state *i*.
 - **Parameters**
 - * **i** – State
 - * **a** – Action
 - **Returns** – Lump cost received.
-

- *lumpCost*

```
public abstract double lumpCost( jmarkov.basic.State i, jmarkov.basic.Action
a, jmarkov.basic.Event e )
```

- **Description**
Reward instantaneously gained in the moment when action *a* is taken from state *i*.
- **Parameters**
 - * **i** – current state
 - * **a** – action taken
 - * **e** – event that occurs

- **Returns** – instantaneous reward.
-

- *rate*

```
public abstract double rate( jmarkov.basic.State i, jmarkov.basic.State j,
jmarkov.basic.Action a )
```

- **Description copied from CTMDP** (see 4.1.2, page 126)

Rate of going from state i to state j by taking the action a

- **Parameters**

- * i – current state
- * j – Destination state.
- * a – Action taken

- **Returns** – The rate
-

- *rate*

```
public abstract double rate( jmarkov.basic.State i, jmarkov.basic.State j,
jmarkov.basic.Action a, jmarkov.basic.Event e )
```

- **Description**

Rate. Rate of going of reaching state j given that the current state is i, the action taken is a and the event that occurs is e.

- **Parameters**

- * i – current state
- * j – state to reach
- * a – action taken (given)
- * e – event that occurs (given)

- **Returns** – Rate
-

- *reachable*

```
public abstract jmarkov.basic.States reachable( jmarkov.basic.State i,
jmarkov.basic.Action a )
```

- **Description copied from CTMDP** (see 4.1.2, page 126)

Set of States that can be reached from this state i, after taking the action a.

- **Parameters**

- * i – current State
- * a – action taken

- **Returns** – the reachable states.
-

- *reachable*

```
public abstract jmarkov.basic.States reachable( jmarkov.basic.State i,
jmarkov.basic.Action a, jmarkov.basic.Event e )
```

- **Description**

Set of reachable states from state i given that action a is taken and event e occurs.

- **Parameters**

- * i – current state
- * a – action taken
- * e – event that occurs

- **Returns** – set of reachable states.

Members inherited from class jmarkov.jmdp.CTMDP (see 4.1.2, page 126)

- protected **activeState**
- public abstract double **continuousCost**(jmarkov.basic.State i, jmarkov.basic.Action a)
- protected **converter**
- protected StatesSet **generate**()
- public StatesSet **getAllStates**()
- public double **getMaxRate**()
- public ValueFunction **getSteadyStateProbabilities**() throws jmarkov.basic.exceptions.SolverException
- public abstract double **lumpCost**(jmarkov.basic.State i, jmarkov.basic.Action a)
- protected **maxRate**
- protected StatesSet **oneStageReachable**(jmarkov.basic.States initSet)
- public abstract double **rate**(jmarkov.basic.State i, jmarkov.basic.State j, jmarkov.basic.Action a)
- public abstract States **reachable**(jmarkov.basic.State i, jmarkov.basic.Action a)
- public void **setConverter**(CT2DTConverter converter)
- public Solution **solve**(double interestRate) throws jmarkov.basic.exceptions.SolverException

Members inherited from class jmarkov.jmdp.InfiniteMDP (see 4.1.11, page 161)

- protected **absorbingStates**
- protected **explorationTime**
- public abstract Actions **feasibleActions**(jmarkov.basic.State i)
- protected abstract StatesSet **generate**()
- public StatesSet **getAllStates**()
- protected AbstractAverageSolver **getDefaultAverageSolver**()
- protected AbstractDiscountedSolver **getDefaultDiscountedSolver**(double interestRate)
- protected AbstractInfiniteSolver **getDefaultSolver**()
- public final int **getNumStates**()
- public AbstractInfiniteSolver **getSolver**()
- protected **hasAbsorbingState**
- protected **numStates**
- protected **probability**
- protected **probabilitySolver**
- protected void **setInterestRate**(double interestRate)
- protected **states**

Members inherited from class jmarkov.jmdp.MDP (see 4.1.12, page 165)

- public void **debug**(int level, java.lang.String message)
- public void **debug**(int level, java.lang.String s, boolean newline)
- public void **debug**(int level, java.lang.String s, boolean newline, boolean indent)
- protected **finite**
- public int **getDebugLevel**()
- protected abstract Solver **getDefaultSolver**()
- public final Policy **getOptimalPolicy**() throws jmarkov.basic.exceptions.SolverException
- public ValueFunction **getOptimalValueFunction**() throws jmarkov.basic.exceptions.SolverException
- public DebugReporter **getReporter**()
- public Solver **getSolver**()
- protected **initial**
- public final boolean **isFinite**()
- public final boolean **isSolved**()
- public double **operation**(double present, double future)
- public void **printSolution**()
- public void **printSolution**(java.io.PrintWriter pw)
- protected **reporter**
- public void **setDebugLevel**(int level)
- public void **setReporter**(jmarkov.DebugReporter reporter)
- public void **setSolver**(solvers.Solver solver)
- public final void **solve**() throws jmarkov.basic.exceptions.SolverException

4.1.4 Class CTMDPEvA

This class represents an Infinite horizon, continuous time Markov Decision Process with events where actions depend on events. It allows the definition of events that can occur in a given state and this makes the cost and probability definition easier than in the cases where no events are defined.

Declaration

```
public abstract class CTMDPEvA
  extends jmarkov.jmdp.CTMDP (see 4.1.2, page 126)
```

Field summary

initSet Initail set od States.

Constructor summary

CTMDPEvA(States) Creates a new continuous time infinite horizon MDP Problem with events

Method summary

activeEvents(S) Set of events that are active from state i given that action a is taken.

continuousCost(S, A, E) Reward obtained continuously in time during the sojourn time in state i until an action a is taken and a transition is triggered.

continuousCost(StateEvent, A)

feasibleAct(S) Returns the set of actions available at this state.

feasibleActions(StateEvent)

generate()

lumpCost(S, A, E) Reward instantaneously gained in the moment when action a is taken from state i.

lumpCost(StateEvent, A)

rate(S, S, A, E) Rate.

rate(StateEvent, StateEvent, A)

reachable(StateEvent, A)

reached(S, A, E) Set of reachable states from state i given that action a is taken and event e occurs.

Fields

- protected jmarkov.basic.States **initSet**
 - Initail set od States.

Constructors

- *CTMDPEvA*

```
public CTMDPEvA( jmarkov.basic.States initial )
```


- **Description**
Creates a new continuous time infinite horizon MDP Problem with events
- **Parameters**
 - * **initial** – set of initial states for the exploration algorithm

Methods

- *activeEvents*

```
public abstract jmarkov.basic.Events activeEvents( jmarkov.basic.State i )
```

 - **Description**
Set of events that are active from state i given that action a is taken.
 - **Parameters**
 - * **i** – current state
 - **Returns** – set of events that can occur

- *continuousCost*

```
public abstract double continuousCost( jmarkov.basic.State i,  
jmarkov.basic.Action a, jmarkov.basic.Event e )
```

 - **Description**
Reward obtained continuously in time during the sojourn time in state i until an action a is taken and a transition is triggered.
 - **Parameters**
 - * **i** – current state
 - * **a** – action taken
 - * **e** – event that occurs
 - **Returns** – instantaneous reward.

- *continuousCost*

```
public double continuousCost( jmarkov.basic.StateEvent i,  
jmarkov.basic.Action a )
```

- *feasibleAct*

```
public abstract jmarkov.basic.Actions feasibleAct( jmarkov.basic.State i )
```

 - **Description**
Returns the set of actions available at this state. The user must implement this method.
 - **Parameters**
 - * **i** – current state
 - **Returns** – set of feasible actions

- *feasibleActions*

```
public jmarkov.basic.Actions feasibleActions( jmarkov.basic.StateEvent s )
```

- *generate*

```
protected abstract jmarkov.basic.StatesSet generate( )
```

 - **Returns** – The set of states found.

- *lumpCost*

```
public abstract double lumpCost( jmarkov.basic.State i, jmarkov.basic.Action
a, jmarkov.basic.Event e )
```

- **Description**

Reward instantaneously gained in the moment when action a is taken from state i.

- **Parameters**

- * i – current state
- * a – action taken
- * e – event that occurs

- **Returns** – instantaneous reward.

- *lumpCost*

```
public double lumpCost( jmarkov.basic.StateEvent i, jmarkov.basic.Action a )
```

- *rate*

```
public abstract double rate( jmarkov.basic.State i, jmarkov.basic.State j,
jmarkov.basic.Action a, jmarkov.basic.Event e )
```

- **Description**

Rate. Rate of going of reaching state j given that the current state is i, the action taken is a and the event that occurs is e.

- **Parameters**

- * i – current state
- * j – state to reach
- * a – action taken (given)
- * e – event that occurs (given)

- **Returns** – Rate

- *rate*

```
public double rate( jmarkov.basic.StateEvent i, jmarkov.basic.StateEvent j,
jmarkov.basic.Action a )
```

- *reachable*

```
public jmarkov.basic.States reachable( jmarkov.basic.StateEvent i,
jmarkov.basic.Action a )
```

- *reached*

```
public abstract jmarkov.basic.States reached( jmarkov.basic.State i,
jmarkov.basic.Action a, jmarkov.basic.Event e )
```

- **Description**

Set of reachable states from state i given that action a is taken and event e occurs.

- **Parameters**

- * i – current state
- * a – action taken
- * e – event that occurs

- **Returns** – set of reachable states.

Members inherited from class jmarkov.jmdp.CTMDP (see 4.1.2, page 126)

- protected **activeState**
- public abstract double **continuousCost**(jmarkov.basic.State i, jmarkov.basic.Action a)
- protected **converter**
- protected StatesSet **generate**()
- public StatesSet **getAllStates**()
- public double **getMaxRate**()
- public ValueFunction **getSteadyStateProbabilities**() throws jmarkov.basic.exceptions.SolverException
- public abstract double **lumpCost**(jmarkov.basic.State i, jmarkov.basic.Action a)
- protected **maxRate**
- protected StatesSet **oneStageReachable**(jmarkov.basic.States initSet)
- public abstract double **rate**(jmarkov.basic.State i, jmarkov.basic.State j, jmarkov.basic.Action a)
- public abstract States **reachable**(jmarkov.basic.State i, jmarkov.basic.Action a)
- public void **setConverter**(CT2DTConverter converter)
- public Solution **solve**(double interestRate) throws jmarkov.basic.exceptions.SolverException

Members inherited from class jmarkov.jmdp.InfiniteMDP (see 4.1.11, page 161)

- protected **absorbingStates**
- protected **explorationTime**
- public abstract Actions **feasibleActions**(jmarkov.basic.State i)
- protected abstract StatesSet **generate**()
- public StatesSet **getAllStates**()
- protected AbstractAverageSolver **getDefaultAverageSolver**()
- protected AbstractDiscountedSolver **getDefaultDiscountedSolver**(double interestRate)
- protected AbstractInfiniteSolver **getDefaultSolver**()
- public final int **getNumStates**()
- public AbstractInfiniteSolver **getSolver**()
- protected **hasAbsorbingState**
- protected **numStates**
- protected **probability**
- protected **probabilitySolver**
- protected void **setInterestRate**(double interestRate)
- protected **states**

Members inherited from class jmarkov.jmdp.MDP (see 4.1.12, page 165)

- public void **debug**(int level, java.lang.String message)
- public void **debug**(int level, java.lang.String s, boolean newline)
- public void **debug**(int level, java.lang.String s, boolean newline, boolean indent)
- protected **finite**
- public int **getDebugLevel**()
- protected abstract Solver **getDefaultSolver**()
- public final Policy **getOptimalPolicy**() throws jmarkov.basic.exceptions.SolverException
- public ValueFunction **getOptimalValueFunction**() throws jmarkov.basic.exceptions.SolverException
- public DebugReporter **getReporter**()
- public Solver **getSolver**()
- protected **initial**
- public final boolean **isFinite**()
- public final boolean **isSolved**()
- public double **operation**(double present, double future)
- public void **printSolution**()
- public void **printSolution**(java.io.PrintWriter pw)
- protected **reporter**
- public void **setDebugLevel**(int level)
- public void **setReporter**(jmarkov.DebugReporter reporter)
- public void **setSolver**(solvers.Solver solver)
- public final void **solve**() throws jmarkov.basic.exceptions.SolverException

4.1.5 Class DTMDP

This class represents a discrete time infinite horizon MDP problem. It must be extended in order to represent the appropriate structure for each problem. The user must implement at least the functions that have been declared abstract.

See also

- `solvers.PolicyIterationSolver` (see 5.2.11, page 205)
- `solvers.ValueIterationSolver` (see 5.2.16, page 224)

Declaration

```
public abstract class DTMDP
  extends jmarkov.jmdp.InfiniteMDP (see 4.1.11, page 161)
```

All known subclasses

`StochasticShortestPath` (see 4.1.13, page 170) , `DTMDPEvA` (see 4.1.7, page 147) , `DTMDPEv` (see 4.1.6, page 143) , `CT2DTConverter` (see 4.1.1, page 123)

Constructor summary

DTMDP(S) Creates a new infinite horizon discrete time (MDP) Problem.
DTMDP(States) Creates a new infinite horizon discrete time (MDP) Problem.

Method summary

generate()
getSteadyStateProbabilities()
immediateCost(S, A) Cost incurred when taking action a from state i
oneStageReachable(States) TFinds the states reached in one step.
prob(S, S, A) Probability of going from state i to state j by taking the action a
reachable(S, A) Set of states that can be reached from this state i, after taking the action a.
setProbabilitySolver(ProbabilitySolver)
solve(double) Solves the problem with the given interest rate

Constructors

- *DTMDP*

```
public DTMDP( jmarkov.basic.State initial )
```

 - **Description**
Creates a new infinite horizon discrete time (MDP) Problem.
 - **Parameters**
 - * **initial** – An initial state for the exploration algorithm
- *DTMDP*

```
public DTMDP( jmarkov.basic.States initial )
```

- **Description**
Creates a new infinite horizon discrete time (MDP) Problem.
- **Parameters**
 - * **initial** – set of initial states for the exploration algorithm

Methods

- *generate*
protected abstract jmarkov.basic.StatesSet generate()

– **Returns** – The set of states found.

- *getSteadyStateProbabilities*
public jmarkov.basic.ValueFunction getSteadyStateProbabilities() throws jmarkov.basic.exceptions.SolverException

– **Returns** – a map with the probability for each state.
– **Throws**
 - * jmarkov.basic.exceptions.SolverException –

- *immediateCost*
public abstract double immediateCost(jmarkov.basic.State i, jmarkov.basic.Action a)

– **Description**
Cost incurred when taking action a from state i
– **Parameters**
 - * **i** – Current State
 - * **a** – Current Action
- **Returns** – The cost incurred per transition

- *oneStageReachable*
protected jmarkov.basic.StatesSet oneStageReachable(jmarkov.basic.States initSet)

– **Description**
TFinds the states reached in one step.
– **Parameters**
 - * **initSet** –
- **Returns** – Set of states reached in one step.

- *prob*
public abstract double prob(jmarkov.basic.State i, jmarkov.basic.State j, jmarkov.basic.Action a)

– **Description**
Probability of going from state i to state j by taking the action a
– **Parameters**
 - * **i** – Current state.
 - * **j** – Destination State

- * *a* – Action
 - **Returns** – The probability.

- *reachable*
 public abstract jmarkov.basic.States reachable(jmarkov.basic.State *i*,
 jmarkov.basic.Action *a*)
 - **Description**
 Set of states that can be reached from this state *i*, after taking the action *a*.
 - **Parameters**
 - * *i* – Current State
 - * *a* – Action taken
 - **Returns** – The reachable states.

- *setProbabilitySolver*
 public void setProbabilitySolver(solvers.ProbabilitySolver *sol*)
 - **Parameters**
 - * *sol* – Sets the solver that solves the steady state probabilities.

- *solve*
 public final jmarkov.basic.Solution solve(double **interestRate**) throws
 jmarkov.basic.exceptions.SolverException
 - **Description**
 Solves the problem with the given interest rate
 - **Parameters**
 - * **interestRate** – the interest rate parameter to solve the problem.
 - **Returns** – The solution to the problem.
 - **Throws**
 - * jmarkov.basic.exceptions.SolverException –

Members inherited from class jmarkov.jmdp.InfiniteMDP (see 4.1.11, page 161)

- protected absorbingStates
- protected explorationTime
- public abstract Actions feasibleActions(jmarkov.basic.State *i*)
- protected abstract StatesSet generate()
- public StatesSet getAllStates()
- protected AbstractAverageSolver getDefaultAverageSolver()
- protected AbstractDiscountedSolver getDefaultDiscountedSolver(double **interestRate**)
- protected AbstractInfiniteSolver getDefaultSolver()
- public final int getNumStates()
- public AbstractInfiniteSolver getSolver()
- protected hasAbsorbingState
- protected numStates
- protected probability
- protected probabilitySolver
- protected void setInterestRate(double **interestRate**)
- protected states

Members inherited from class jmarkov.jmdp.MDP (see 4.1.12, page 165)

- public void **debug**(int level, java.lang.String message)
- public void **debug**(int level, java.lang.String s, boolean newline)
- public void **debug**(int level, java.lang.String s, boolean newline, boolean indent)
- protected **finite**
- public int **getDebugLevel**()
- protected abstract Solver **getDefaultSolver**()
- public final Policy **getOptimalPolicy**() throws
jmarkov.basic.exceptions.SolverException
- public ValueFunction **getOptimalValueFunction**() throws
jmarkov.basic.exceptions.SolverException
- public DebugReporter **getReporter**()
- public Solver **getSolver**()
- protected **initial**
- public final boolean **isFinite**()
- public final boolean **isSolved**()
- public double **operation**(double present, double future)
- public void **printSolution**()
- public void **printSolution**(java.io.PrintWriter pw)
- protected **reporter**
- public void **setDebugLevel**(int level)
- public void **setReporter**(jmarkov.DebugReporter reporter)
- public void **setSolver**(solvers.Solver solver)
- public final void **solve**() throws jmarkov.basic.exceptions.SolverException

4.1.6 Class DTMDPEv

This class represents an infinite horizon, discrete time, Markov Decision Process with events. It allows the definition of events that can occur in a given state and this makes the cost and probability definition easier to define than in the cases where no events are defined.

Declaration

```
public abstract class DTMDPEv
extends jmarkov.jmdp.DTMDP (see 4.1.5, page 139)
```

Constructor summary

DTMDPEv(States) Creates a new infinite horizon discrete time (MDP) Problem with events

Method summary

activeEvents(S, A) Set of events that are active from state i given that action a is taken.

immediateCost(S, A)

immediateCost(S, A, E) Cost incurred received when the current state is i, the action taken is a and event e occurs.

prob(S, E) Conditional Event probability.

prob(S, S, A)

prob(S, S, A, E) Conditional destination probability.

reachable(S, A)

reachable(S, A, E) Set of reachable states from state i given that action a is taken and event e occurs.

Constructors

- *DTMDPEv*

```
public DTMDPEv( jmarkov.basic.States initial )
```

- **Description**

Creates a new infinite horizon discrete time (MDP) Problem with events

- **Parameters**

* **initial** – set of initial states for the exploration algorithm

Methods

- *activeEvents*

```
public abstract jmarkov.basic.Events activeEvents( jmarkov.basic.State i,
jmarkov.basic.Action a )
```

- **Description**

Set of events that are active from state i given that action a is taken.

- **Parameters**

- * **i** – current state
 - * **a** – action taken
 - **Returns** – set of events that can occur
-

- *immediateCost*

```
public abstract double immediateCost( jmarkov.basic.State i,
jmarkov.basic.Action a )
```

- **Description copied from DTMDP** (see 4.1.5, page 139)
Cost incurred when taking action a from state i
 - **Parameters**
 - * **i** – Current State
 - * **a** – Current Action
 - **Returns** – The cost incurred per transition
-

- *immediateCost*

```
public abstract double immediateCost( jmarkov.basic.State i,
jmarkov.basic.Action a, jmarkov.basic.Event e )
```

- **Description**
Cost incurred received when the current state is i, the action taken is a and event e occurs.
 - **Parameters**
 - * **i** – current state
 - * **a** – action taken
 - * **e** – event that occurs
 - **Returns** – reward
-

- *prob*

```
public abstract double prob( jmarkov.basic.State i, jmarkov.basic.Event e )
```

- **Description**
Conditional Event probability. Probability that event e occurs given that the current state is i.
 - **Parameters**
 - * **i** – current state
 - * **e** – event that occurs
 - **Returns** – Conditional probability
-

- *prob*

```
public abstract double prob( jmarkov.basic.State i, jmarkov.basic.State j,
jmarkov.basic.Action a )
```

- **Description copied from DTMDP** (see 4.1.5, page 139)
Probability of going from state i to state j by taking the action a
 - **Parameters**
 - * **i** – Current state.
 - * **j** – Destination State
 - * **a** – Action
 - **Returns** – The probability.
-

- *prob*

```
public abstract double prob( jmarkov.basic.State i, jmarkov.basic.State j,
jmarkov.basic.Action a, jmarkov.basic.Event e )
```

- **Description**

Conditional destination probability. Probability of reaching state j given that the current state is i, the action taken is a and the event that occurs is e.

- **Parameters**

- * i – current state
- * j – state to reach
- * a – action taken (given)
- * e – event that occurs (given)

- **Returns** – Conditional probability

- *reachable*

```
public abstract jmarkov.basic.States reachable( jmarkov.basic.State i,
jmarkov.basic.Action a )
```

- **Description copied from DTMDP** (see 4.1.5, page 139)

Set of states that can be reached from this state i, after taking the action a.

- **Parameters**

- * i – Current State
- * a – Action taken

- **Returns** – The reachable states.

- *reachable*

```
public abstract jmarkov.basic.States reachable( jmarkov.basic.State i,
jmarkov.basic.Action a, jmarkov.basic.Event e )
```

- **Description**

Set of reachable states from state i given that action a is taken and event e occurs.

- **Parameters**

- * i – current state
- * a – action taken
- * e – event that occurs

- **Returns** – set of reachable states.

Members inherited from class jmarkov.jmdp.DTMDP (see 4.1.5, page 139)

- protected StatesSet **generate**()
- public ValueFunction **getSteadyStateProbabilities**() throws jmarkov.basic.exceptions.SolverException
- public abstract double **immediateCost**(jmarkov.basic.State i, jmarkov.basic.Action a)
- protected StatesSet **oneStageReachable**(jmarkov.basic.States initSet)
- public abstract double **prob**(jmarkov.basic.State i, jmarkov.basic.State j, jmarkov.basic.Action a)
- public abstract States **reachable**(jmarkov.basic.State i, jmarkov.basic.Action a)
- public void **setProbabilitySolver**(solvers.ProbabilitySolver solv)
- public final Solution **solve**(double interestRate) throws jmarkov.basic.exceptions.SolverException

Members inherited from class jmarkov.jmdp.InfiniteMDP (see 4.1.11, page 161)

-
- protected absorbingStates
 - protected explorationTime
 - public abstract Actions feasibleActions(jmarkov.basic.State i)
 - protected abstract StatesSet generate()
 - public StatesSet getAllStates()
 - protected AbstractAverageSolver getDefaultAverageSolver()
 - protected AbstractDiscountedSolver getDefaultDiscountedSolver(double interestRate)
 - protected AbstractInfiniteSolver getDefaultSolver()
 - public final int getNumStates()
 - public AbstractInfiniteSolver getSolver()
 - protected hasAbsorbingState
 - protected numStates
 - protected probability
 - protected probabilitySolver
 - protected void setInterestRate(double interestRate)
 - protected states

Members inherited from class jmarkov.jmdp.MDP (see 4.1.12, page 165)

-
- public void debug(int level, java.lang.String message)
 - public void debug(int level, java.lang.String s, boolean newline)
 - public void debug(int level, java.lang.String s, boolean newline, boolean indent)
 - protected finite
 - public int getDebugLevel()
 - protected abstract Solver getDefaultSolver()
 - public final Policy getOptimalPolicy() throws jmarkov.basic.exceptions.SolverException
 - public ValueFunction getOptimalValueFunction() throws jmarkov.basic.exceptions.SolverException
 - public DebugReporter getReporter()
 - public Solver getSolver()
 - protected initial
 - public final boolean isFinite()
 - public final boolean isSolved()
 - public double operation(double present, double future)
 - public void printSolution()
 - public void printSolution(java.io.PrintWriter pw)
 - protected reporter
 - public void setDebugLevel(int level)
 - public void setReporter(jmarkov.DebugReporter reporter)
 - public void setSolver(solvers.Solver solver)
 - public final void solve() throws jmarkov.basic.exceptions.SolverException

4.1.7 Class DTMDPEvA

This class represents an infinite horizon, discrete time, Markov Decision Process with events, where actions depend on events. It allows the definition of events that can occur in a given state and this makes the cost and probability definition easier to define than in the cases where no events are defined.

Declaration

```
public abstract class DTMDPEvA
extends jmarkov.jmdp.DTMDP (see 4.1.5, page 139)
```

Constructor summary

DTMDPEvA(States) Creates a new infinite horizon discrete time (MDP) Problem with events

Method summary

activeEvents(S, A) Set of events that are active from state i given that action a is taken.
feasibleAct(S) Returns the set of actions available at this state.
feasibleActions(StateEvent)
immediateCost(S, A, E) Reward received when the current state is i, the action taken is a and event e occurs.
immediateCost(StateEvent, A)
prob(S, E) Conditional event probability.
prob(S, S, A, E) Conditional destination probability.
prob(StateEvent, StateEvent, A)
reachable(S, A, E) Set of reachable states from state i given that action a is taken and event e occurs.
reachable(StateEvent, A)

Constructors

- *DTMDPEvA*
 public **DTMDPEvA**(jmarkov.basic.States **initial**)
 - **Description**
 Creates a new infinite horizon discrete time (MDP) Problem with events
 - **Parameters**
 * **initial** – set of initial states for the exploration algorithm

Methods

- *activeEvents*
 public abstract jmarkov.basic.Events **activeEvents**(jmarkov.basic.State i, jmarkov.basic.Action a)

– **Description**

Set of events that are active from state *i* given that action *a* is taken.

– **Parameters**

- * *i* – current state
- * *a* – action taken

– **Returns** – set of events that can occur

• *feasibleAct*

```
public abstract jmarkov.basic.Actions feasibleAct( jmarkov.basic.State i )
```

– **Description**

Returns the set of actions available at this state. The user must implement this method.

– **Parameters**

- * *i* – current state

– **Returns** – set of feasible actions

• *feasibleActions*

```
public final jmarkov.basic.Actions feasibleActions( jmarkov.basic.StateEvent i )
```

• *immediateCost*

```
public abstract double immediateCost( jmarkov.basic.State i,
jmarkov.basic.Action a, jmarkov.basic.Event e )
```

– **Description**

Reward received when the current state is *i*, the action taken is *a* and event *e* occurs.

– **Parameters**

- * *i* – current state
- * *a* – action taken
- * *e* – event that occurs

– **Returns** – reward

• *immediateCost*

```
public final double immediateCost( jmarkov.basic.StateEvent i,
jmarkov.basic.Action a )
```

• *prob*

```
public abstract double prob( jmarkov.basic.State i, jmarkov.basic.Event e )
```

– **Description**

Conditional event probability. Probability that event *e* occurs given that the current state is *i*.

– **Parameters**

- * *i* – current state
- * *e* – event that occurs

– **Returns** – Conditional probability

• *prob*

```
public abstract double prob( jmarkov.basic.State i, jmarkov.basic.State j,
jmarkov.basic.Action a, jmarkov.basic.Event e )
```

– **Description**

Conditional destination probability. Probability of reaching state *j* given that the current state is *i*, the action taken is *a* and the event that occurs is *e*.

– **Parameters**

- * *i* – current state
- * *j* – state to reach
- * *a* – action taken (given)
- * *e* – event that occurs (given)

– **Returns** – Conditional probability

• *prob*

```
public final double prob( jmarkov.basic.StateEvent i,
    jmarkov.basic.StateEvent j, jmarkov.basic.Action a )
```

• *reachable*

```
public abstract jmarkov.basic.States reachable( jmarkov.basic.State i,
    jmarkov.basic.Action a, jmarkov.basic.Event e )
```

– **Description**

Set of reachable states from state *i* given that action *a* is taken and event *e* occurs.

– **Parameters**

- * *i* – current state
- * *a* – action taken
- * *e* – event that occurs

– **Returns** – set of reachable states.

• *reachable*

```
public final jmarkov.basic.States reachable( jmarkov.basic.StateEvent i,
    jmarkov.basic.Action a )
```

Members inherited from class `jmarkov.jmdp.DTMDP` (see 4.1.5, page 139)

-
- protected `StatesSet generate()`
 - public `ValueFunction getSteadyStateProbabilities()` throws `jmarkov.basic.exceptions.SolverException`
 - public abstract `double immediateCost(jmarkov.basic.State i, jmarkov.basic.Action a)`
 - protected `StatesSet oneStageReachable(jmarkov.basic.States initSet)`
 - public abstract `double prob(jmarkov.basic.State i, jmarkov.basic.State j, jmarkov.basic.Action a)`
 - public abstract `States reachable(jmarkov.basic.State i, jmarkov.basic.Action a)`
 - public void `setProbabilitySolver(solvers.ProbabilitySolver solv)`
 - public final `Solution solve(double interestRate)` throws `jmarkov.basic.exceptions.SolverException`

Members inherited from class `jmarkov.jmdp.InfiniteMDP` (see 4.1.11, page 161)

-
- protected `absorbingStates`
 - protected `explorationTime`
 - public abstract `Actions feasibleActions(jmarkov.basic.State i)`
 - protected abstract `StatesSet generate()`
 - public `StatesSet getAllStates()`
 - protected `AbstractAverageSolver getDefaultAverageSolver()`

- protected AbstractDiscountedSolver getDefaultDiscountedSolver(double interestRate)
- protected AbstractInfiniteSolver getDefaultSolver()
- public final int getNumStates()
- public AbstractInfiniteSolver getSolver()
- protected hasAbsorbingState
- protected numStates
- protected probability
- protected probabilitySolver
- protected void setInterestRate(double interestRate)
- protected states

Members inherited from class jmarkov.jmdp.MDP (see 4.1.12, page 165)

- public void debug(int level, java.lang.String message)
- public void debug(int level, java.lang.String s, boolean newline)
- public void debug(int level, java.lang.String s, boolean newline, boolean indent)
- protected finite
- public int getDebugLevel()
- protected abstract Solver getDefaultSolver()
- public final Policy getOptimalPolicy() throws jmarkov.basic.exceptions.SolverException
- public ValueFunction getOptimalValueFunction() throws jmarkov.basic.exceptions.SolverException
- public DebugReporter getReporter()
- public Solver getSolver()
- protected initial
- public final boolean isFinite()
- public final boolean isSolved()
- public double operation(double present, double future)
- public void printSolution()
- public void printSolution(java.io.PrintWriter pw)
- protected reporter
- public void setDebugLevel(int level)
- public void setReporter(jmarkov.DebugReporter reporter)
- public void setSolver(solvers.Solver solver)
- public final void solve() throws jmarkov.basic.exceptions.SolverException

4.1.8 Class FiniteDP

This class should ONLY be used in FINITE horizon deterministic problems. It must be extended in order to represent the appropriate structure for each FINITE Dynamic Programming problem. The user must implement at least the functions that have been declared abstract. It's also necessary to create one of the extensions of the class Solver. By default, the program includes the FiniteSolver class to solve finite horizon problems. PolicyIterationSolver and ValueIterationSolver are only for infinite horizon problems. To solve the problem follow the instructions in each of the solvers' instructions.

See also

- `solvers.FiniteSolver` (see 5.2.6, page 188)

Declaration

```
public abstract class FiniteDP
  extends jmarkov.jmdp.FiniteMDP (see 4.1.9, page 153)
```

Constructor summary

FiniteDP(States, int) Creates a new FINITE Dynamic Programming (DP) Problem.

Method summary

destination(S, A, int) State where the system will end up if action a is taken from state i at time t.

prob(S, S, A, int) Final function must not be extended or implemented by any user.

reachable(S, A, int) Final function must not be extended by any user.

Constructors

- *FiniteDP*
 public **FiniteDP**(jmarkov.basic.States initial, int lastStage)
 - **Description**
 Creates a new FINITE Dynamic Programming (DP) Problem.
 - **Parameters**
 - * **initial** – initial set of known states.
 - * **lastStage** – number of the last stage.

Methods

- *destination*
 public abstract jmarkov.basic.State **destination**(jmarkov.basic.State i, jmarkov.basic.Action a, int t)
 - **Description**
 State where the system will end up if action a is taken from state i at time t. The user must implement this method.

– **Parameters**

- * *i* – Current state
- * *a* – Current action
- * *t* – Time stage.

– **Returns** – Destination states

• *prob*

```
public final double prob( jmarkov.basic.State i, jmarkov.basic.State j,
jmarkov.basic.Action a, int t )
```

– **Description**

Final function must not be extended or implementes by any user.

• *reachable*

```
public final jmarkov.basic.States reachable( jmarkov.basic.State i,
jmarkov.basic.Action a, int t )
```

– **Description**

Final function must not be extended by any user.

Members inherited from class jmarkov.jmdp.FiniteMDP (see 4.1.9, page 153)

-
- public abstract Actions **feasibleActions**(jmarkov.basic.State i, int t)
 - public double **finalCost**(jmarkov.basic.State i)
 - protected Solver **getDefaultSolver**()
 - public int **getHorizon**()
 - public States **getStates**(int t)
 - protected **horizon**
 - public abstract double **immediateCost**(jmarkov.basic.State i, jmarkov.basic.Action a, int t)
 - public abstract double **prob**(jmarkov.basic.State i, jmarkov.basic.State j, jmarkov.basic.Action a, int t)
 - public abstract States **reachable**(jmarkov.basic.State i, jmarkov.basic.Action a, int t)
 - protected void **setHorizon**(int T)

Members inherited from class jmarkov.jmdp.MDP (see 4.1.12, page 165)

-
- public void **debug**(int level, java.lang.String message)
 - public void **debug**(int level, java.lang.String s, boolean newline)
 - public void **debug**(int level, java.lang.String s, boolean newline, boolean indent)
 - protected **finite**
 - public int **getDebugLevel**()
 - protected abstract Solver **getDefaultSolver**()
 - public final Policy **getOptimalPolicy**() throws jmarkov.basic.exceptions.SolverException
 - public ValueFunction **getOptimalValueFunction**() throws jmarkov.basic.exceptions.SolverException
 - public DebugReporter **getReporter**()
 - public Solver **getSolver**()
 - protected **initial**
 - public final boolean **isFinite**()
 - public final boolean **isSolved**()
 - public double **operation**(double present, double future)
 - public void **printSolution**()
 - public void **printSolution**(java.io.PrintWriter pw)
 - protected **reporter**
 - public void **setDebugLevel**(int level)
 - public void **setReporter**(jmarkov.DebugReporter reporter)
 - public void **setSolver**(solvers.Solver solver)
 - public final void **solve**() throws jmarkov.basic.exceptions.SolverException

4.1.9 Class FiniteMDP

This class should ONLY be used in FINITE horizon problems. It must be extended in order to represent the appropriate structure for each FINITE Dynamic Programming problem. The user must implement at least the functions that have been declared abstract. It's also necessary to create one of the extensions of the class Solver. By default, the program includes the FiniteSolver class to solve finite horizon problems. PolicyIterationSolver and ValueIterationSolver are only for infinite horizon problems. To solve the problem follow the instructions in each of the solvers' instructions.

See also

- `solvers.FiniteSolver` (see 5.2.6, page 188)

Declaration

```
public abstract class FiniteMDP
  extends jmarkov.jmdp.MDP (see 4.1.12, page 165)
```

All known subclasses

FiniteMDPEv (see 4.1.10, page 157) , FiniteDP (see 4.1.8, page 151)

Field summary

horizon Time horizon.

Constructor summary

FiniteMDP(S, int) Creates a finite horizon MDP.

FiniteMDP(States, int) Creates a new FINITE horizon (MDP) Problem.

Method summary

feasibleActions(S, int) Returns the actions available at this state i and at this stage t .

finalCost(S) This function returns the cost incurred if the last stage ends with the system at state i.

getDefaultSolver()

getHorizon() Returns the time lastStage

getStates(int) All the states that are available at stage t.

immediateCost(S, A, int) This function must return the Immediate cost incurred when taking action a from state i

prob(S, S, A, int) This is the probability of going from state i to state j by taking the action a at stage t.

reachable(S, A, int) Set of States that can be reached from this state i, at this stage t, after taking the action a.

setHorizon(int) Sets the time lastStage at which decisions can be taken

Fields

- protected int **horizon**
 - Time horizon. The last stage in the problem.

Constructors

- *FiniteMDP*

```
public FiniteMDP( jmarkov.basic.State initial, int horizon )
```

 - **Description**
Creates a finite horizon MDP.
 - **Parameters**
 - * **initial** – a initial state
 - * **horizon** – horizon.
-
- *FiniteMDP*

```
public FiniteMDP( jmarkov.basic.States initial, int horizon )
```

 - **Description**
Creates a new FINITE horizon (MDP) Problem.
 - **Parameters**
 - * **initial** – set of initial states
 - * **horizon** – last stage at which actions can be taken

Methods

- *feasibleActions*

```
public abstract jmarkov.basic.Actions feasibleActions( jmarkov.basic.State i, int t )
```

 - **Description**
Returns the actions available at this state *i* and at this stage *t*. The user must implement this method.
 - **Parameters**
 - * **i** – Current State
 - * **t** – Time stage
 - **Returns** – Set of feasible actions.
-
- *finalCost*

```
public double finalCost( jmarkov.basic.State i )
```

 - **Description**
This function returns the cost incurred if the last stage ends with the system at state *i*. The user may extend this method.
 - **Parameters**
 - * **i** – Ending state
 - **Returns** – Cost.

-
- *getDefaultSolver*
`protected abstract solvers.Solver getDefaultSolver()`
 - **Description copied from MDP** (see 4.1.12, page 165)
The class that extends MDP must define the default solver to use.
 - **Returns** – the solver to use for this problem.
-
- *getHorizon*
`public int getHorizon()`
 - **Description**
Returns the time lastStage
 - **Returns** – Time horizon
-
- *getStates*
`public jmarkov.basic.States getStates(int t)`
 - **Description**
All the states that are available at stage t.
 - **Parameters**
 - * t – time stage
 - **Returns** – States available at stage t.
-
- *immediateCost*
`public abstract double immediateCost(jmarkov.basic.State i, jmarkov.basic.Action a, int t)`
 - **Description**
This funtion must return the Immediate cost incurred when taking action a from state i
 - **Parameters**
 - * i – Current state
 - * a – Action
 - * t – Current time stage
 - **Returns** – Cost value
-
- *prob*
`public abstract double prob(jmarkov.basic.State i, jmarkov.basic.State j, jmarkov.basic.Action a, int t)`
 - **Description**
This is the probability of going from state i to state j by taking the action a at stage t.
 - **Parameters**
 - * i – Current state
 - * j – Destination state
 - * a – Action taken
 - * t – Current time stage
 - **Returns** – Probability
-
- *reachable*
`public abstract jmarkov.basic.States reachable(jmarkov.basic.State i, jmarkov.basic.Action a, int t)`

– **Description**

Set of States that can be reached from this state *i*, at this stage *t*, after taking the action *a*. The user must implement this method.

– **Parameters**

- * *i* – Current state
- * *a* – Action taken
- * *t* – Time stage

– **Returns** – Set of reachable states.

• *setHorizon*

protected void **setHorizon**(int *T*)

– **Description**

Sets the time lastStage at which decisions can be taken

Members inherited from class jmarkov.jmdp.MDP (see 4.1.12, page 165)

- public void **debug**(int level, java.lang.String message)
- public void **debug**(int level, java.lang.String s, boolean newline)
- public void **debug**(int level, java.lang.String s, boolean newline, boolean indent)
- protected **finite**
- public int **getDebugLevel**()
- protected abstract Solver **getDefaultSolver**()
- public final Policy **getOptimalPolicy**() throws jmarkov.basic.exceptions.SolverException
- public ValueFunction **getOptimalValueFunction**() throws jmarkov.basic.exceptions.SolverException
- public DebugReporter **getReporter**()
- public Solver **getSolver**()
- protected **initial**
- public final boolean **isFinite**()
- public final boolean **isSolved**()
- public double **operation**(double present, double future)
- public void **printSolution**()
- public void **printSolution**(java.io.PrintWriter pw)
- protected **reporter**
- public void **setDebugLevel**(int level)
- public void **setReporter**(jmarkov.DebugReporter reporter)
- public void **setSolver**(solvers.Solver solver)
- public final void **solve**() throws jmarkov.basic.exceptions.SolverException

4.1.10 Class FiniteMDPEv

This class represents a finite horizon discrete time MDP with events.

Declaration

```
public abstract class FiniteMDPEv
extends jmarkov.jmdp.FiniteMDP (see 4.1.9, page 153)
```

Constructor summary

FiniteMDPEv(States, int)

Method summary

activeEvents(S, A, int) Set of events that are active from state i given that action a is taken.
immediateCost(S, A, E, int) Reward received when the current state is i, the action taken is a and event e occurs.
immediateCost(S, A, int)
prob(S, E, int) Conditional probability.
prob(S, S, A, E, int) Conditional probability.
prob(S, S, A, int)
reachable(S, A, E, int) Set of reachable states from state i given that action a is taken and event e occurs.
reachable(S, A, int)

Constructors

- *FiniteMDPEv*

```
public FiniteMDPEv( jmarkov.basic.States initial, int horizon )
```

 - **Parameters**
 - * **initial** –
 - * **horizon** –

Methods

- *activeEvents*

```
public abstract jmarkov.basic.Events activeEvents( jmarkov.basic.State i,
jmarkov.basic.Action a, int t )
```

 - **Description**
Set of events that are active from state i given that action a is taken.
 - **Parameters**
 - * **i** – current state
 - * **a** – action taken
 - * **t** – current stage
 - **Returns** – set of events that can occur
-

- *immediateCost*

```
public abstract double immediateCost( jmarkov.basic.State i,
jmarkov.basic.Action a, jmarkov.basic.Event e, int t )
```

- **Description**

Reward received when the current state is i, the action taken is a and event e occurs.

- **Parameters**

- * i – current state
- * a – action taken
- * e – event that occurs
- * t – current stage

- **Returns** – reward

- *immediateCost*

```
public abstract double immediateCost( jmarkov.basic.State i,
jmarkov.basic.Action a, int t )
```

- **Description copied from FiniteMDP (see 4.1.9, page 153)**

This funtion must return the Immediate cost incurred when taking action a from state i

- **Parameters**

- * i – Current state
- * a – Action
- * t – Current time stage

- **Returns** – Cost value

- *prob*

```
public abstract double prob( jmarkov.basic.State i, jmarkov.basic.Event e,
int t )
```

- **Description**

Conditional probability. Probability that event e occurs given that the current state is i.

- **Parameters**

- * i – current state
- * e – event that occurs
- * t – current stage

- **Returns** – Conditional probability

- *prob*

```
public abstract double prob( jmarkov.basic.State i, jmarkov.basic.State j,
jmarkov.basic.Action a, jmarkov.basic.Event e, int t )
```

- **Description**

Conditional probability. Probability of reaching state j given that the current state is i, the action taken is a and the event that occurs is e.

- **Parameters**

- * i – current state
- * j – state to reach
- * a – action taken (given)
- * e – event that occurs (given)
- * t – current stage

- **Returns** – conditional probability

- *prob*

```
public abstract double prob( jmarkov.basic.State i, jmarkov.basic.State j,
jmarkov.basic.Action a, int t )
```

- **Description copied from FiniteMDP** (see 4.1.9, page 153)

This is the probability of going from state i to state j by taking the action a at stage t.

- **Parameters**

- * i – Current state
- * j – Destination state
- * a – Action taken
- * t – Current time stage

- **Returns** – Probability

- *reachable*

```
public abstract jmarkov.basic.States reachable( jmarkov.basic.State i,
jmarkov.basic.Action a, jmarkov.basic.Event e, int t )
```

- **Description**

Set of reachable states from state i given that action a is taken and event e occurs.

- **Parameters**

- * i – current state
- * a – action taken
- * e – event that occurs
- * t – current stage

- **Returns** – set of reachable states.

- *reachable*

```
public abstract jmarkov.basic.States reachable( jmarkov.basic.State i,
jmarkov.basic.Action a, int t )
```

- **Description copied from FiniteMDP** (see 4.1.9, page 153)

Set of States that can be reached from this state i, at this stage t, after taking the action a. The user must implement this method.

- **Parameters**

- * i – Current state
- * a – Action taken
- * t – Time stage

- **Returns** – Set of reachable states.

Members inherited from class jmarkov.jmdp.FiniteMDP (see 4.1.9, page 153)

- public abstract Actions **feasibleActions**(jmarkov.basic.State i, int t)
- public double **finalCost**(jmarkov.basic.State i)
- protected Solver **getDefaultSolver**()
- public int **getHorizon**()
- public States **getStates**(int t)
- protected **horizon**
- public abstract double **immediateCost**(jmarkov.basic.State i, jmarkov.basic.Action a, int t)
- public abstract double **prob**(jmarkov.basic.State i, jmarkov.basic.State j, jmarkov.basic.Action a, int t)
- public abstract States **reachable**(jmarkov.basic.State i, jmarkov.basic.Action a, int t)
- protected void **setHorizon**(int T)

Members inherited from class jmarkov.jmdp.MDP (see 4.1.12, page 165)

- public void **debug**(int level, java.lang.String message)
- public void **debug**(int level, java.lang.String s, boolean newline)
- public void **debug**(int level, java.lang.String s, boolean newline, boolean indent)
- protected **finite**
- public int **getDebugLevel**()
- protected abstract Solver **getDefaultSolver**()
- public final Policy **getOptimalPolicy**() throws
jmarkov.basic.exceptions.SolverException
- public ValueFunction **getOptimalValueFunction**() throws
jmarkov.basic.exceptions.SolverException
- public DebugReporter **getReporter**()
- public Solver **getSolver**()
- protected **initial**
- public final boolean **isFinite**()
- public final boolean **isSolved**()
- public double **operation**(double present, double future)
- public void **printSolution**()
- public void **printSolution**(java.io.PrintWriter pw)
- protected **reporter**
- public void **setDebugLevel**(int level)
- public void **setReporter**(jmarkov.DebugReporter reporter)
- public void **setSolver**(solvers.Solver solver)
- public final void **solve**() throws jmarkov.basic.exceptions.SolverException

4.1.11 Class InfiniteMDP

This class is a structural class and is. It represents a general Infinite horizon MDP problem. It is extended for discrete and continuous problems.

See also

- `solvers.PolicyIterationSolver` (see 5.2.11, page 205)
- `solvers.ValueIterationSolver` (see 5.2.16, page 224)

Declaration

```
public abstract class InfiniteMDP
extends jmarkov.jmdp.MDP (see 4.1.12, page 165)
```

All known subclasses

StochasticShortestPath (see 4.1.13, page 170) , DTMDPEvA (see 4.1.7, page 147) , DTMDPEv (see 4.1.6, page 143) , DTMDP (see 4.1.5, page 139) , CTMDPEvA (see 4.1.4, page 135) , CTMDPEv (see 4.1.3, page 131) , CTMDP (see 4.1.2, page 126) , CT2DTConverter (see 4.1.1, page 123)

Field summary

absorbingStates Set of absorbing states.
explorationTime Time used to explore the system.
hasAbsorbingState Whether an absorbing state was found
numStates Number of states.
probability The value function
probabilitySolver Idiotic solver that was of course not needed since JMarkov can handle the job
states set of states

Constructor summary

InfiniteMDP(States) Creates a new INFINITE Dynamic Programming (DP) Problem.

Method summary

feasibleActions(S) Returns the set of actions available at this state.
generate()
getAllStates() Complete set of states explored
getDefaultAverageSolver()
getDefaultDiscountedSolver(double)
getDefaultSolver()
getNumStates() Returns the number of states in the model.
getSolver()
setInterestRate(double) Sets the interest rate to be used in the problem solving if the objective is to minimize the discounted cost.

Fields

- protected jmarkov.basic.StatesSet **states**
 - set of states
- protected jmarkov.basic.ValueFunction **probability**
 - The value function
- protected solvers.ProbabilitySolver **probabilitySolver**
 - Idiotic solver that was of course not needed since JMarkov can handle the job
- protected boolean **hasAbsorbingState**
 - Whether an absorbing state was found
- protected jmarkov.basic.States **absorbingStates**
 - Set of absorbing states.
- protected long **explorationTime**
 - Time used to explore the system.
- protected int **numStates**
 - Number of states. Set when calling generate.

Constructors

- *InfiniteMDP*

```
public InfiniteMDP( jmarkov.basic.States initial )
```

 - **Description**
Creates a new INFINITE Dynamic Programming (DP) Problem.
 - **Parameters**
 - * **initial** – set of initial states for the exploration algorithm

Methods

- *feasibleActions*

```
public abstract jmarkov.basic.Actions feasibleActions( jmarkov.basic.State i )
```

 - **Description**
Returns the set of actions available at this state.
 - **Parameters**
 - * **i** – Current State
 - **Returns** – set of Actions that can be taken at this state.
- *generate*

```
protected abstract jmarkov.basic.StatesSet generate( )
```

- **Returns** – The set of states found.
-

- *getAllStates*

`public jmarkov.basic.StatesSet getAllStates()`

- **Description**
Complete set of states explored
 - **Returns** – set of states explored
-

- *getDefaultAverageSolver*

`protected solvers.AbstractAverageSolver getDefaultAverageSolver()`

- **See also**
* `MDP.getDefaultSolver()` (see 4.1.12, page 167)
-

- *getDefaultDiscountedSolver*

`protected solvers.AbstractDiscountedSolver getDefaultDiscountedSolver(
double interestRate)`

- **See also**
* `MDP.getDefaultSolver()` (see 4.1.12, page 167)
-

- *getDefaultSolver*

`protected abstract solvers.Solver getDefaultSolver()`

- **Description copied from MDP** (see 4.1.12, page 165)
The class that extends MDP must define the default solver to use.
 - **Returns** – the solver to use for this problem.
-

- *getNumStates*

`public final int getNumStates()`

- **Description**
Returns the number of states in the model. It causes the model to be generated.
 - **Returns** – The number of states in the system.
-

- *getSolver*

`public solvers.Solver getSolver()`

- **Returns** – Returns the solver.
-

- *setInterestRate*

`protected void setInterestRate(double interestRate)`

- **Description**
Sets the interest rate to be used in the problem solving if the objective is to minimize the discounted cost.
- **Parameters**
* `interestRate` – effective interest rate

Members inherited from class jmarkov.jmdp.MDP (see 4.1.12, page 165)

- public void **debug**(int level, java.lang.String message)
- public void **debug**(int level, java.lang.String s, boolean newline)
- public void **debug**(int level, java.lang.String s, boolean newline, boolean indent)
- protected **finite**
- public int **getDebugLevel**()
- protected abstract Solver **getDefaultSolver**()
- public final Policy **getOptimalPolicy**() throws
jmarkov.basic.exceptions.SolverException
- public ValueFunction **getOptimalValueFunction**() throws
jmarkov.basic.exceptions.SolverException
- public DebugReporter **getReporter**()
- public Solver **getSolver**()
- protected **initial**
- public final boolean **isFinite**()
- public final boolean **isSolved**()
- public double **operation**(double present, double future)
- public void **printSolution**()
- public void **printSolution**(java.io.PrintWriter pw)
- protected **reporter**
- public void **setDebugLevel**(int level)
- public void **setReporter**(jmarkov.DebugReporter reporter)
- public void **setSolver**(solvers.Solver solver)
- public final void **solve**() throws jmarkov.basic.exceptions.SolverException

4.1.12 Class MDP

This class is the main framework to build a Dynamic Programming Problem. It was initially created to work over Markov Decision Problems which imply random probabilities but can easily be worked out for deterministic problems if the probabilities are set to one. This class should not be extended directly on problems. The default package has `FiniteMDP` and `InfiniteMDP` classes that are intended to be extended on problems. See the examples for a clearer reference.

See also

- `FiniteMDP` (see 4.1.9, page 153)
- `DTMDP` (see 4.1.5, page 139)

Declaration

```
public abstract class MDP
extends java.lang.Object
```

All known subclasses

`StochasticShortestPath` (see 4.1.13, page 170) , `InfiniteMDP` (see 4.1.11, page 161) , `FiniteMDPEv` (see 4.1.10, page 157) , `FiniteMDP` (see 4.1.9, page 153) , `FiniteDP` (see 4.1.8, page 151) , `DTMDPEvA` (see 4.1.7, page 147) , `DTMDPEv` (see 4.1.6, page 143) , `DTMDP` (see 4.1.5, page 139) , `CTMDPEvA` (see 4.1.4, page 135) , `CTMDPEv` (see 4.1.3, page 131) , `CTMDP` (see 4.1.2, page 126) , `CT2DTConverter` (see 4.1.1, page 123)

Field summary

finite States whether the problem is a finite horizon problem or not.
initial Set of initial states.
reporter Reporter used for debug information.

Constructor summary

`MDP()`

Method summary

debug(int, String) Prints a message in the reporter.
debug(int, String, boolean) Prints debug information in the reporter.
debug(int, String, boolean, boolean) Prints debug information in the reporter.
getDebugLevel() Gets the current debug level.
getDefaultSolver() The class that extends `MDP` must define the default solver to use.
getOptimalPolicy() Returns the optimal policy.
getOptimalValueFunction() Returns the optimal `ValueFunction`.
getReporter()
getSolver()
isFinite()
isSolved() Indicates if the problems has been solved
operation(double, double) The Operator between present and future costs.
printSolution() Prints the solution to Standard output.

printSolution(PrintWriter) Prints the solution to the given PrintWriter
setDebugLevel(int) Sets the current level
setReporter(DebugReporter)
setSolver(Solver)
solve() Solves the problem.

Fields

- protected boolean **finite**
 - States whether the problem is a finite horizon problem or not.
- protected jmarkov.basic.States **initial**
 - Set of initial states.
- protected jmarkov.DebugReporter **reporter**
 - Reporter used for debug information.

Constructors

- *MDP*
`public MDP()`

Methods

- *debug*
`public void debug(int level, java.lang.String message)`
 - **Description**
Prints a message in the reporter.
 - **Parameters**
 - * `level` – maximum debug level at which to show message
 - * `message` – message
 - **See also**
 - * `jmarkov.DebugReporter` (see 1.1.1, page 10)
-
- *debug*
`public void debug(int level, java.lang.String s, boolean newline)`
 - **Description**
Prints debug information in the reporter.
 - **Parameters**
 - * `level` – the level for the info
 - * `s` – Message
 - * `newline` – true if a new line is to be inserted
 - **See also**
 - * `jmarkov.DebugReporter` (see 1.1.1, page 10)
 - * `jmarkov.DebugReporter.debug(int,String,boolean)` (see 1.1.1, page 11)

- *debug*

```
public void debug( int level, java.lang.String s, boolean newline, boolean
indent )
```

- **Description**

Prints debug information in the reporter.

- **Parameters**

- * **level** – the level for the info
 - * **s** – Message
 - * **newline** – true if a new line is to be inserted
 - * **indent** – true if the info is indented according to level

- **See also**

- * `jmarkov.DebugReporter` (see 1.1.1, page 10)
 - * `jmarkov.DebugReporter.debug(int,String,boolean,boolean)` (see 1.1.1, page 11)
-

- *getDebugLevel*

```
public int getDebugLevel( )
```

- **Description**

Gets the current debug level.

- **Returns** – The current debug level

- **See also**

- * `jmarkov.DebugReporter` (see 1.1.1, page 10)
 - * `jmarkov.DebugReporter.getDebugLevel()` (see 1.1.1, page 11)
-

- *getDefaultSolver*

```
protected abstract solvers.Solver getDefaultSolver( )
```

- **Description**

The class that extends MDP must define the default solver to use.

- **Returns** – the solver to use for this problem.

- *getOptimalPolicy*

```
public final jmarkov.basic.Policy getOptimalPolicy( ) throws
jmarkov.basic.exceptions.SolverException
```

- **Description**

Returns the optimal policy. This function should be called only AFTER solve has been called.

- **Returns** – The optimal policy.

- **Throws**

- * `jmarkov.basic.exceptions.SolverException` – Exception thrown if a solution cannot be found
-

- *getOptimalValueFunction*

```
public jmarkov.basic.ValueFunction getOptimalValueFunction( ) throws
jmarkov.basic.exceptions.SolverException
```

– **Description**

Returns the optimal ValueFunction. This causes the problem to be solved if it has not been solved.

– **Returns** – Returns the valueFunction.

– **Throws**

* `jmarkov.basic.exceptions.SolverException` – This exception is thrown if a solution cannot be found.

• *getReporter*

```
public jmarkov.DebugReporter getReporter( )
```

– **Returns** – Returns the reporter.

• *getSolver*

```
public solvers.Solver getSolver( )
```

– **Returns** – Returns the solver.

• *isFinite*

```
public final boolean isFinite( )
```

– **Returns** – Returns true if the problem's horizon is finite.

• *isSolved*

```
public final boolean isSolved( )
```

– **Description**

Indicates if the problems has been solved

– **Returns** – true if solved

• *operation*

```
public double operation( double present, double future )
```

– **Description**

The Operator between present and future costs. By default is sum, but can be changed by the user, by overriding this method.

– **Parameters**

* **present** – Cost of current transition

* **future** – Cost of future transitions.

– **Returns** – By default it returns present + future.

• *printSolution*

```
public void printSolution( )
```

– **Description**

Prints the solution to Standard output.

• *printSolution*

```
public void printSolution( java.io.PrintWriter pw )
```

– **Description**

Prints the solution to the given PrintWriter

– **Parameters**

- * `pw` – The `PrintWriter` where the solution will be printed. It must have been initialized.

- *setDebugLevel*

`public void setDebugLevel(int level)`

- **Description**

Sets the current level

- **Parameters**

- * `level` – The new level to level

- **See also**

- * `jmarkov.DebugReporter` (see 1.1.1, page 10)
 - * `jmarkov.DebugReporter.setDebugLevel(int)` (see 1.1.1, page 12)
-

- *setReporter*

`public void setReporter(jmarkov.DebugReporter reporter)`

- **Parameters**

- * `reporter` – The reporter to set.
-

- *setSolver*

`public void setSolver(solvers.Solver solver)`

- **Parameters**

- * `solver` – The solver to set.
-

- *solve*

`public final void solve() throws jmarkov.basic.exceptions.SolverException`

- **Description**

Solves the problem. If no solver has been defined, this used the default solver.

- **Throws**

- * `jmarkov.basic.exceptions.SolverException` – This Exception is thrown if a solution cannot be found.

- **See also**

- * `MDP.getDefaultSolver()` (see 4.1.12, page 167)

4.1.13 Class *StochasticShortestPath*

This class represents an infinite horizon shortest path problem.

Declaration

```
public abstract class StochasticShortestPath
extends jmarkov.jmdp.DTMDP (see 4.1.5, page 139)
```

Constructor summary

StochasticShortestPath(States)

Method summary

modifiedProb(S, S, A) This method was specially created to eliminate in a existent graph the self-transition probabilities.

Constructors

- *StochasticShortestPath*
 public **StochasticShortestPath**(jmarkov.basic.States states)
 - **Parameters**
 - * states – Constructor

Methods

- *modifiedProb*
 public double **modifiedProb**(jmarkov.basic.StateC i, jmarkov.basic.StateC j, jmarkov.basic.Action a) throws jmarkov.basic.exceptions.StructureException
 - **Description**
 This method was specially created to eliminate in a existent graph the self-transition probabilities.
 - **Parameters**
 - * i –
 - * j –
 - * a –
 - **Returns** – the modified probability
 - **Throws**
 - * jmarkov.basic.exceptions.StructureException –

Members inherited from class jmarkov.jmdp.DTMDP (see 4.1.5, page 139)

- protected StatesSet **generate**()
- public ValueFunction **getSteadyStateProbabilities**() throws jmarkov.basic.exceptions.SolverException
- public abstract double **immediateCost**(jmarkov.basic.State i, jmarkov.basic.Action a)

- protected StatesSet **oneStageReachable**(jmarkov.basic.States **initSet**)
- public abstract double **prob**(jmarkov.basic.State **i**, jmarkov.basic.State **j**, jmarkov.basic.Action **a**)
- public abstract States **reachable**(jmarkov.basic.State **i**, jmarkov.basic.Action **a**)
- public void **setProbabilitySolver**(solvers.ProbabilitySolver **sol**)
- public final Solution **solve**(double **interestRate**) throws jmarkov.basic.exceptions.SolverException

Members inherited from class jmarkov.jmdp.InfiniteMDP (see 4.1.11, page 161)

- protected absorbingStates
- protected explorationTime
- public abstract Actions feasibleActions(jmarkov.basic.State **i**)
- protected abstract StatesSet generate()
- public StatesSet **getAllStates**()
- protected AbstractAverageSolver **getDefaultAverageSolver**()
- protected AbstractDiscountedSolver **getDefaultDiscountedSolver**(double **interestRate**)
- protected AbstractInfiniteSolver **getDefaultSolver**()
- public final int **getNumStates**()
- public AbstractInfiniteSolver **getSolver**()
- protected hasAbsorbingState
- protected numStates
- protected probability
- protected probabilitySolver
- protected void **setInterestRate**(double **interestRate**)
- protected states

Members inherited from class jmarkov.jmdp.MDP (see 4.1.12, page 165)

- public void **debug**(int **level**, java.lang.String **message**)
- public void **debug**(int **level**, java.lang.String **s**, boolean **newline**)
- public void **debug**(int **level**, java.lang.String **s**, boolean **newline**, boolean **indent**)
- protected finite
- public int **getDebugLevel**()
- protected abstract Solver **getDefaultSolver**()
- public final Policy **getOptimalPolicy**() throws jmarkov.basic.exceptions.SolverException
- public ValueFunction **getOptimalValueFunction**() throws jmarkov.basic.exceptions.SolverException
- public DebugReporter **getReporter**()
- public Solver **getSolver**()
- protected initial
- public final boolean **isFinite**()
- public final boolean **isSolved**()
- public double **operation**(double **present**, double **future**)
- public void **printSolution**()
- public void **printSolution**(java.io.PrintWriter **pw**)
- protected reporter
- public void **setDebugLevel**(int **level**)
- public void **setReporter**(jmarkov.DebugReporter **reporter**)
- public void **setSolver**(solvers.Solver **solver**)
- public final void **solve**() throws jmarkov.basic.exceptions.SolverException

Chapter 5

Package jmarkov.jmdp.solvers

Package Contents

Page

Interfaces

LPSolver	174
MpsLpSolver	176
<i>This interface define the minimum elements for creating a MPS file.</i>	

Classes

AbstractAverageSolver	177
<i>Structural class for average cost solvers to extend.</i>	
AbstractDiscountedSolver	179
<i>This is a structural class that must be extended by classes solving the discounted cost minimization problem.</i>	
AbstractFiniteSolver	182
<i>Structural class for solvers to extend in order to solve finite horizon problems.</i>	
AbstractInfiniteSolver	184
<i>Structural class to be extended by solvers in order to solve infinite horizon problems</i>	
AbstractTotalSolver	186
<i>Structural class to be extended by solvers in order to solve the total cost criteria for an infinite horizon problem</i>	
FiniteSolver	188
<i>This class belongs to the set of default solvers included in the jmdp package.</i>	
LPBCLAverageSolver	191
<i>This solver solves a average-cost infinite horizon MDP by building and solving a linear problem using as interface Xpress BCL.</i>	
LPBCLDiscountedSolver	194
<i>This solver solves a discounted infinite horizon MDP by building and solving a linear problem using as interface Xpress BCL.</i>	
MpsLpAverageSolver	198
<i>This class builds the Dual Linear Program for an average infinite horizon MDP in a MPS file.</i>	
MpsLpDiscountedSolver	201
<i>This class builds a Linear Program for a discounted infinite horizon MDP in a MPS file.</i>	
PolicyIterationSolver	205

<i>This class solves infinite horizon discounted problems using the policy iteration algorithm.</i>	
ProbabilitySolver	210
<i>This class is designed to calculate the long run probabilities of infinite horizon problem.</i>	
RelativeValueIterationSolver	213
<i>This class solves the average cost criteria for infinite horizon problems</i>	
Solver	216
<i>Structural class for every solver.</i>	
StochasticShortestPathSolver	221
<i>This solver gives a solution for the minimization of the total cost criterion for an infinite horizon MDP.</i>	
ValueIterationSolver	224
<i>This class belongs to the set of default solvers included in the jmdp package.</i>	

This package contains the framework of solvers used by jMDP to solve Markov Decision Processes. See the jMDP manual for details.

5.1 Interfaces

5.1.1 Interface LPSolver

Declaration

```
public interface LPSolver
```

All known subclasses

MpsLpSolver (see 5.1.2, page 176) , MpsLpDiscountedSolver (see 5.2.10, page 201) , MpsLpAverageSolver (see 5.2.9, page 198) , LPBCLDiscountedSolver (see 5.2.8, page 194)

All known subinterfaces

MpsLpSolver (see 5.1.2, page 176)

All classes known to implement interface

LPBCLDiscountedSolver (see 5.2.8, page 194)

Method summary

- buildSolution()** The implementator classes should override this class to build the solution after the model has been solved.
- getBuildTime()** Returns the time taken to build and write the MPS file.
- getLpSolveTime()** Return the time taken to solve the LP model.
- getSolBuildTime()** Returns the time needed to build the Solution after the LP was solved.
- solveLP()** The implementator classes should override this class to solve the problem using the mpsFile that has been created.

Methods

- *buildSolution*
 jmarkov.basic.Solution **buildSolution()** throws
 jmarkov.basic.exceptions.SolverException
 - **Description**
 The implementator classes should override this class to build the solution after the model has been solved.
 - **Returns** – The solution to the problem.
 - **Throws**
 * jmarkov.basic.exceptions.SolverException –
- *getBuildTime*
 long **getBuildTime()**
 - **Description**
 Returns the time taken to build and write the MPS file.

– **Returns** – Returns the buildTime.

- *getLpSolveTime*

long **getLpSolveTime**()

– **Description**

Return the time taken to solve the LP model.

– **Returns** – Returns the lpSolveTime.

- *getSolBuildTime*

long **getSolBuildTime**()

– **Description**

Returns the time needed to build the Solution after the LP was solved.

– **Returns** – Returns the solBuildTime.

- *solveLP*

void **solveLP**() throws jmarkov.basic.exceptions.SolverException

– **Description**

The implementator classes should override this class to solve the problem using the mpsFile that has been created.

– **Throws**

* jmarkov.basic.exceptions.SolverException –

5.1.2 Interface MpsLpSolver

This interface define the minimum elements for creating a MPS file.

Declaration

```
public interface MpsLpSolver
implements LPSolver
```

All known subclasses

MpsLpDiscountedSolver (see 5.2.10, page 201) , MpsLpAverageSolver (see 5.2.9, page 198)

All classes known to implement interface

MpsLpDiscountedSolver (see 5.2.10, page 201) , MpsLpAverageSolver (see 5.2.9, page 198)

Method summary

getMpsFile() Returns the MPS file name.
getMpsFileName() Returns the MPS file name.
getWorkingDir() Returns the working directory (where the MPS file is located).

Methods

- *getMpsFile*
`java.io.File getMpsFile()`
 - **Description**
Returns the MPS file name.
 - **Returns** – Returns the MPS generated file.

- *getMpsFileName*
`java.lang.String getMpsFileName()`
 - **Description**
Returns the MPS file name.
 - **Returns** – Returns the fileName.

- *getWorkingDir*
`java.io.File getWorkingDir()`
 - **Description**
Returns the working directory (where the MPS file is located).
 - **Returns** – Returns the MPS File folder.

5.2 Classes

5.2.1 Class AbstractAverageSolver

Structural class for average cost solvers to extend.

Declaration

```
public abstract class AbstractAverageSolver
extends jmarkov.jmdp.solvers.AbstractInfiniteSolver (see 5.2.4, page 184)
```

All known subclasses

RelativeValueIterationSolver (see 5.2.13, page 213) , MpsLpAverageSolver (see 5.2.9, page 198) , LPBCLAverageSolver (see 5.2.7, page 191)

Constructor summary

- AbstractAverageSolver(CTMDP)** Creates a solver for an infinite horizon continuous time MDP
- AbstractAverageSolver(DTMDP)** Creates a solver for an infinite horizon discrete time MDP

Constructors

- *AbstractAverageSolver*
protected **AbstractAverageSolver**(jmarkov.jmdp.CTMDP problem)
 - **Description**
Creates a solver for an infinite horizon continuous time MDP
 - **Parameters**
* problem – continuous time problem
- *AbstractAverageSolver*
protected **AbstractAverageSolver**(jmarkov.jmdp.DTMDP problem)
 - **Description**
Creates a solver for an infinite horizon discrete time MDP
 - **Parameters**
* problem – discrete time problem

Members inherited from class jmarkov.jmdp.solvers.AbstractInfiniteSolver (see 5.2.4, page 184)

- protected DTMDP **getDiscreteProblem**()
- public abstract long **getIterations**()
- public InfiniteMDP **getProblem**()
- public void **printSolution**(java.io.PrintWriter pw)

Members inherited from class jmarkov.jmdp.solvers.Solver (see 5.2.14, page 216)

- public String **description**()
- public final Policy **getOptimalPolicy**() throws jmarkov.basic.exceptions.SolverException
- public final ValueFunction **getOptimalValueFunction**() throws jmarkov.basic.exceptions.SolverException
- public MDP **getProblem**()
- public abstract long **getProcessTime**()
- public final ValueFunction **getValueFunction**()
- public final boolean **isSolved**()
- public abstract String **label**()
- protected **policy**
- protected **printProcessTime**
- public void **printSolution**() throws java.lang.Exception
- public void **printSolution**(java.io.PrintWriter pw)
- protected **printValueFunction**
- protected **problem**
- public void **setPrintProcessTime**(boolean val)
- public void **setPrintValueFunction**(boolean val)
- public abstract Solution **solve**() throws jmarkov.basic.exceptions.SolverException
- protected **solved**
- public final String **toString**()
- protected **valueFunction**

5.2.2 Class AbstractDiscountedSolver

This is a structural class that must be extended by classes solving the dicounted cost minimization problem.

Declaration

```
public abstract class AbstractDiscountedSolver
  extends jmarkov.jmdp.solvers.AbstractInfiniteSolver (see 5.2.4, page 184)
```

All known subclasses

ValueIterationSolver (see 5.2.16, page 224) , PolicyIterationSolver (see 5.2.11, page 205) ,
MpsLpDiscountedSolver (see 5.2.10, page 201) , LPBCLDiscountedSolver (see 5.2.8, page 194)

Field summary

discountFactor The discount factor used to bring to present value from next period.

Constructor summary

AbstractDiscountedSolver(CTMDP, double)
AbstractDiscountedSolver(DTMDP, double)

Method summary

future(S, A, double) Expected value of valueFunction for the current state and a specified action.
future(S, A, double, ValueFunction) Expected value of valueFunction for the current state and a specified action.
getInterestRate() Returns the current value of the discount factor.
setDiscountFactor(double)
setInterestRate(double) Sets a new Interest Rate

Fields

- protected double **discountFactor**
 - The discount factor used to bring to present value from next period.

Constructors

- *AbstractDiscountedSolver*

```
protected AbstractDiscountedSolver( jmarkov.jmdp.CTMDP problem, double
  interestRate )
```

 - **Parameters**
 - * **problem** – The problem associated with this solver.
 - * **interestRate** – The interest rate (nominal compunded continuously). For example, if you measure time in months, and the APR is A, then this rate satisfies $\exp(i/12) = 1 + A$. Therefore $i=12*\ln(1+A)$.

- *AbstractDiscountedSolver*

```
protected AbstractDiscountedSolver( jmarkov.jmdp.DTMDP problem, double
interestRate )
```

- **Parameters**

- * **problem** – The problem associated with this solver.
- * **interestRate** – The interest rate per period

Methods

- *future*

```
protected final double future( jmarkov.basic.State i, jmarkov.basic.Action a,
double discountF )
```

- **Description**

Expected value of valueFunction for the current state and a specified action.

- **Parameters**

- * **i** – This State
- * **a** – Action taken
- * **discountF** – is the rate for discounting from one period to another. It means how much less it would represent to receive one unit of the reward in the next period instead of receiving it in the present period.

- **Returns** – Expected value of valueFunction.

- *future*

```
protected final double future( jmarkov.basic.State i, jmarkov.basic.Action a,
double discountF, jmarkov.basic.ValueFunction vf )
```

- **Description**

Expected value of valueFunction for the current state and a specified action.

- **Parameters**

- * **discountF** – is the rate for discounting from one period to another. It means how much less it would represent to receive one unit of the reward in the next period instead of receiving it in the present period.
-

- *getInterestRate*

```
public double getInterestRate( )
```

- **Description**

Returns the current value of the discount factor.

- **Returns** – The current value of the discount factor.

- *setDiscountFactor*

```
protected final void setDiscountFactor( double discountFactor )
```

- **Parameters**

- * **discountFactor** – The discountFactor to set.
-

- *setInterestRate*

```
public final void setInterestRate( double interestRate )
```

– **Description**

Sets a new Interest Rate

– **Parameters**

* interestRate – set.

Members inherited from class `jmarkov.jmdp.solvers.AbstractInfiniteSolver` (see 5.2.4, page 184)

- protected DTMDP `getDiscreteProblem()`
- public abstract long `getIterations()`
- public InfiniteMDP `getProblem()`
- public void `printSolution(java.io.PrintWriter pw)`

Members inherited from class `jmarkov.jmdp.solvers.Solver` (see 5.2.14, page 216)

- public String `description()`
- public final Policy `getOptimalPolicy()` throws `jmarkov.basic.exceptions.SolverException`
- public final ValueFunction `getOptimalValueFunction()` throws `jmarkov.basic.exceptions.SolverException`
- public MDP `getProblem()`
- public abstract long `getProcessTime()`
- public final ValueFunction `getValueFunction()`
- public final boolean `isSolved()`
- public abstract String `label()`
- protected `policy`
- protected `printProcessTime`
- public void `printSolution()` throws `java.lang.Exception`
- public void `printSolution(java.io.PrintWriter pw)`
- protected `printValueFunction`
- protected `problem`
- public void `setPrintProcessTime(boolean val)`
- public void `setPrintValueFunction(boolean val)`
- public abstract Solution `solve()` throws `jmarkov.basic.exceptions.SolverException`
- protected `solved`
- public final String `toString()`
- protected `valueFunction`

5.2.3 Class AbstractFiniteSolver

Structural class for solvers to extend in order to solve finite horizon problems.

Declaration

```
public abstract class AbstractFiniteSolver
extends jmarkov.jmdp.solvers.Solver (see 5.2.14, page 216)
```

All known subclasses

FiniteSolver (see 5.2.6, page 188)

Constructor summary

AbstractFiniteSolver(FiniteMDP)

Method summary

getProblem() Returns the problem associated wit this solver.

Constructors

- *AbstractFiniteSolver*
 protected **AbstractFiniteSolver**(jmarkov.jmdp.FiniteMDP problem)
 - **Parameters**
 - * problem – finite horizon problem to be solved

Methods

- *getProblem*
 public jmarkov.jmdp.FiniteMDP **getProblem**()
 - **Description**
 Returns the problem associated wit this solver.
 - **Returns** – the problem associated wit this solver.

Members inherited from class jmarkov.jmdp.solvers.Solver (see 5.2.14, page 216)

- public String **description**()
- public final Policy **getOptimalPolicy**() throws jmarkov.basic.exceptions.SolverException
- public final ValueFunction **getOptimalValueFunction**() throws jmarkov.basic.exceptions.SolverException
- public MDP **getProblem**()
- public abstract long **getProcessTime**()
- public final ValueFunction **getValueFunction**()
- public final boolean **isSolved**()
- public abstract String **label**()
- protected policy
- protected **printProcessTime**

- `public void printSolution()` throws `java.lang.Exception`
- `public void printSolution(java.io.PrintWriter pw)`
- `protected printValueFunction`
- `protected problem`
- `public void setPrintProcessTime(boolean val)`
- `public void setPrintValueFunction(boolean val)`
- `public abstract Solution solve()` throws `jmarkov.basic.exceptions.SolverException`
- `protected solved`
- `public final String toString()`
- `protected valueFunction`

5.2.4 Class AbstractInfiniteSolver

Structural class to be extended by solvers in order to solve infinite horizon problems

Declaration

```
public abstract class AbstractInfiniteSolver
  extends jmarkov.jmdp.solvers.Solver (see 5.2.14, page 216)
```

All known subclasses

ValueIterationSolver (see 5.2.16, page 224) , StochasticShortestPathSolver (see 5.2.15, page 221) , RelativeValueIterationSolver (see 5.2.13, page 213) , PolicyIterationSolver (see 5.2.11, page 205) , MpsLpDiscountedSolver (see 5.2.10, page 201) , MpsLpAverageSolver (see 5.2.9, page 198) , LPBCLDiscountedSolver (see 5.2.8, page 194) , LPBCLAverageSolver (see 5.2.7, page 191) , AbstractTotalSolver (see 5.2.5, page 186) , AbstractDiscountedSolver (see 5.2.2, page 179) , AbstractAverageSolver (see 5.2.1, page 177)

Constructor summary

- AbstractInfiniteSolver(CTMDP)** Creates a solver for an infinite horizon continuous time problem
- AbstractInfiniteSolver(DTMDP)** Constructor method for Discrete Time Markov Decision Processes to be solved for discounted cost.

Method summary

- getDiscreteProblem()**
- getIterations()**
- getProblem()** Returns the problem associated with this solver.
- printSolution(PrintWriter)**

Constructors

- *AbstractInfiniteSolver*
 protected **AbstractInfiniteSolver(jmarkov.jmdp.CTMDP problem)**
 - **Description**
Creates a solver for an infinite horizon continuous time problem
 - **Parameters**
 * **problem** – continuous time problem
- *AbstractInfiniteSolver*
 protected **AbstractInfiniteSolver(jmarkov.jmdp.DTMDP problem)**
 - **Description**
Constructor method for Discrete Time Markov Decision Processes to be solved for discounted cost.
 - **Parameters**
 * **problem** – Discrete Time Markov Decision Process of type DTMDP

Methods

- *getDiscreteProblem*
`protected jmarkov.jmdp.DTMDP getDiscreteProblem()`
 – **Returns** – discrete time problem

- *getIterations*
`public abstract long getIterations()`
 – **Returns** – Returns the iterations in the last solve.

- *getProblem*
`public jmarkov.jmdp.InfiniteMDP getProblem()`
 – **Description**
 Returns the problem associated with this solver.
 – **Returns** – the problem associated with this solver.

- *printSolution*
`public void printSolution(java.io.PrintWriter pw)`
 – **Description copied from Solver** (see 5.2.14, page 216)
 Prints the solution on a given PrintWriter.
 – **Parameters**
 * pw –
 – **See also**
 * java.io.PrintWriter

Members inherited from class `jmarkov.jmdp.solvers.Solver` (see 5.2.14, page 216)

- `public String description()`
- `public final Policy getOptimalPolicy() throws jmarkov.basic.exceptions.SolverException`
- `public final ValueFunction getOptimalValueFunction() throws jmarkov.basic.exceptions.SolverException`
- `public MDP getProblem()`
- `public abstract long getProcessTime()`
- `public final ValueFunction getValueFunction()`
- `public final boolean isSolved()`
- `public abstract String label()`
- `protected policy`
- `protected printProcessTime`
- `public void printSolution() throws java.lang.Exception`
- `public void printSolution(java.io.PrintWriter pw)`
- `protected printValueFunction`
- `protected problem`
- `public void setPrintProcessTime(boolean val)`
- `public void setPrintValueFunction(boolean val)`
- `public abstract Solution solve() throws jmarkov.basic.exceptions.SolverException`
- `protected solved`
- `public final String toString()`
- `protected valueFunction`

5.2.5 Class AbstractTotalSolver

Structural class to be extended by solvers in order to solve the total cost criteria for an infinite horizon problem

Declaration

```
public abstract class AbstractTotalSolver
  extends jmarkov.jmdp.solvers.AbstractInfiniteSolver (see 5.2.4, page 184)
```

All known subclasses

StochasticShortestPathSolver (see 5.2.15, page 221)

Constructor summary

AbstractTotalSolver(DTMDP) Creates a solver for a discrete time problem

Constructors

- *AbstractTotalSolver*
 public **AbstractTotalSolver**(jmarkov.jmdp.DTMDP problem)
 - **Description**
 Creates a solver for a discrete time problem
 - **Parameters**
 * problem – discrete time problem

Members inherited from class jmarkov.jmdp.solvers.AbstractInfiniteSolver (see 5.2.4, page 184)

- protected DTMDP **getDiscreteProblem**()
- public abstract long **getIterations**()
- public InfiniteMDP **getProblem**()
- public void **printSolution**(java.io.PrintWriter pw)

Members inherited from class jmarkov.jmdp.solvers.Solver (see 5.2.14, page 216)

- public String **description**()
- public final Policy **getOptimalPolicy**() throws jmarkov.basic.exceptions.SolverException
- public final ValueFunction **getOptimalValueFunction**() throws jmarkov.basic.exceptions.SolverException
- public MDP **getProblem**()
- public abstract long **getProcessTime**()
- public final ValueFunction **getValueFunction**()
- public final boolean **isSolved**()
- public abstract String **label**()
- protected policy
- protected **printProcessTime**
- public void **printSolution**() throws java.lang.Exception
- public void **printSolution**(java.io.PrintWriter pw)
- protected **printValueFunction**
- protected problem

- `public void setPrintProcessTime(boolean val)`
- `public void setPrintValueFunction(boolean val)`
- `public abstract Solution solve()` throws `jmarkov.basic.exceptions.SolverException`
- `protected solved`
- `public final String toString()`
- `protected valueFunction`

5.2.6 Class FiniteSolver

This class belongs to the set of default solvers included in the jmdp package. It extends Solver and should only be used on FINITE horizon problems. The objective function of the solver is to minimize the the total cost. The result is a deterministic optimal policy for the given structure.

Declaration

```
public class FiniteSolver
  extends jmarkov.jmdp.solvers.AbstractFiniteSolver (see 5.2.3, page 182)
```

Constructor summary

FiniteSolver(FiniteMDP) Initialized the solver with the given getProblem().

Method summary

bestPolicy(S) Prints out the policy
future(S, A, int) This method calculates the expected value of valueFunction for the current state i and a specified action a at the given stage t.
getProcessTime()
label()
solve()

Constructors

- *FiniteSolver*
 public **FiniteSolver**(jmarkov.jmdp.FiniteMDP problem)
 - **Description**
 Initialized the solver with the given getProblem().
 - **Parameters**
 - * problem – The problem to be solved.

Methods

- *bestPolicy*
 public java.lang.String **bestPolicy**(jmarkov.basic.State initial) throws jmarkov.basic.exceptions.SolverException
 - **Description**
 Prints out the policy
 - **Parameters**
 - * initial –
 - **Returns** – a string with the optimal policy
 - **Throws**
 - * jmarkov.basic.exceptions.SolverException –
-

- *future*

protected final double **future**(jmarkov.basic.State i, jmarkov.basic.Action a, int t) throws java.lang.NullPointerException

- **Description**

This method calculates the expected value of valueFunction for the current state i and a specified action a at the given stage t.

- **Parameters**

- * i – Current State
- * a – Action taken
- * t – Time stage

- **Returns** – The value.

- **Throws**

- * java.lang.NullPointerException –

- *getProcessTime*

public abstract long **getProcessTime**()

- **Returns** – Returns the processTime of the last solve. Use System.currentTimeMillis() to get the current time.

- *label*

public java.lang.String **label**()

- **See also**

- * Solver.toString() (see 5.2.14, page 220)

- *solve*

public abstract jmarkov.basic.Solution **solve**() throws jmarkov.basic.exceptions.SolverException

- **Description copied from Solver** (see 5.2.14, page 216)

Called to solve the problem. This method MUST write the local variable policy and valueFunction.

- **Returns** – The solution Object taht contains the plicy and value fuenction.

- **Throws**

- * jmarkov.basic.exceptions.SolverException – This exception is thrown if the solver cannot find a solution for some reason.

Members inherited from class jmarkov.jmdp.solvers.AbstractFiniteSolver (see 5.2.3, page 182)

- public FiniteMDP **getProblem**()

Members inherited from class jmarkov.jmdp.solvers.Solver (see 5.2.14, page 216)

- public String **description**()
- public final Policy **getOptimalPolicy**() throws jmarkov.basic.exceptions.SolverException
- public final ValueFunction **getOptimalValueFunction**() throws jmarkov.basic.exceptions.SolverException

- public MDP **getProblem**()
- public abstract long **getProcessTime**()
- public final ValueFunction **getValueFunction**()
- public final boolean **isSolved**()
- public abstract String **label**()
- protected **policy**
- protected **printProcessTime**
- public void **printSolution**() throws java.lang.Exception
- public void **printSolution**(java.io.PrintWriter pw)
- protected **printValueFunction**
- protected **problem**
- public void **setPrintProcessTime**(boolean val)
- public void **setPrintValueFunction**(boolean val)
- public abstract Solution **solve**() throws jmarkov.basic.exceptions.SolverException
- protected **solved**
- public final String **toString**()
- protected **valueFunction**

5.2.7 Class LPBCLAverageSolver

This solver solves a average-cost infinite horizon MDP by building and solving a linear problem using as interface Xpress BCL. It requires the professional version of XpressMP and the JAVA build path must include the xprb.jar library, for running the applications.

Declaration

```
public class LPBCLAverageSolver
  extends jmarkov.jmdp.solvers.AbstractAverageSolver (see 5.2.1, page 177)
```

Field summary

lpSolveTime Used to store the Linear Programming solve time

Constructor summary

LPBCLAverageSolver(DTMDP) The constructor method exclusively receives a problem of the type DTMDP because this solver is only designed to work on infinite discrete horizon problems.

Method summary

getBuildTime()
getIterations()
getLpSolveTime()
getProcessTime()
getSolBuildTime() Returns the time needed to build the Solution after the LP was solved.
label()
solve() Linear Programming Average Solver is a tool that builds the solution based on the MDP's mathematical background given by Puterman and the software provided by XpressMP (BCL libraries).

Fields

- protected long **lpSolveTime**
 - Used to store the Linear Programming solve time

Constructors

- *LPBCLAverageSolver*

```
public LPBCLAverageSolver( jmarkov.jmdp.DTMDP problem )
```

 - **Description**

The constructor method exclusively receives a problem of the type DTMDP because this solver is only designed to work on infinite discrete horizon problems. This solver solves an average DTMDP.
 - **Parameters**
 - * **problem** – the structure of the problem of type DTMDP

Methods

- *getBuildTime*
`public long getBuildTime()`
 – **Returns** – Returns the build time.

- *getIterations*
`public abstract long getIterations()`
 – **Returns** – Returns the iterations in the last solve.

- *getLpSolveTime*
`public long getLpSolveTime()`
 – **Returns** – Returns the Linear Programming Solve Time

- *getProcessTime*
`public abstract long getProcessTime()`
 – **Returns** – Returns the processTime of the last solve. Use `System.currentTimeMillis()` to get the current time.

- *getSolBuildTime*
`public long getSolBuildTime()`
 – **Description**
 Returns the time needed to build the Solution after the LP was solved.
 – **Returns** – Returns the solBuildTime.

- *label*
`public abstract java.lang.String label()`
 – **Description copied from Solver** (see 5.2.14, page 216)
 The sub classes must return the Solver name.
 – **See also**
 * `Solver.toString()` (see 5.2.14, page 220)

- *solve*
`public jmarkov.basic.Solution solve() throws
 jmarkov.basic.exceptions.SolverException`
 – **Description**
 Linear Programming Average Solver is a tool that builds the solution based on the MDP's mathematical background given by Puterman and the software provided by XpressMP (BCL libraries). Is mandatory for the use to have a Xpress professional version.
 – **Throws**
 * `jmarkov.basic.exceptions.SolverException` –

Members inherited from class `jmarkov.jmdp.solvers.AbstractAverageSolver` (see 5.2.1, page 177)

Members inherited from class `jmarkov.jmdp.solvers.AbstractInfiniteSolver` (see 5.2.4, page 184)

-
- protected DTMDP `getDiscreteProblem()`
 - public abstract long `getIterations()`
 - public InfiniteMDP `getProblem()`
 - public void `printSolution(java.io.PrintWriter pw)`

Members inherited from class `jmarkov.jmdp.solvers.Solver` (see 5.2.14, page 216)

-
- public String `description()`
 - public final Policy `getOptimalPolicy()` throws `jmarkov.basic.exceptions.SolverException`
 - public final ValueFunction `getOptimalValueFunction()` throws `jmarkov.basic.exceptions.SolverException`
 - public MDP `getProblem()`
 - public abstract long `getProcessTime()`
 - public final ValueFunction `getValueFunction()`
 - public final boolean `isSolved()`
 - public abstract String `label()`
 - protected policy
 - protected `printProcessTime`
 - public void `printSolution()` throws `java.lang.Exception`
 - public void `printSolution(java.io.PrintWriter pw)`
 - protected `printValueFunction`
 - protected `problem`
 - public void `setPrintProcessTime(boolean val)`
 - public void `setPrintValueFunction(boolean val)`
 - public abstract Solution `solve()` throws `jmarkov.basic.exceptions.SolverException`
 - protected `solved`
 - public final String `toString()`
 - protected `valueFunction`

5.2.8 Class LPBCLDiscountedSolver

This solver solves a discounted infinite horizon MDP by building and solving a linear problem using as interface Xpress BCL. It requires the professional version of XpressMP and the JAVA build path must include the xprb.jar library, for running the applications.

Declaration

```
public class LPBCLDiscountedSolver
  extends jmarkov.jmdp.solvers.AbstractDiscountedSolver (see 5.2.2, page 179)
  implements LPSolver
```

Constructor summary

LPBCLDiscountedSolver(DTMDP, double) The constructor method receives a problem of the type infinite DTMDP and an interest rate that is modified for being used as discount factor.

Method summary

```
buildSolution()
description()
getBuildTime()
getIterations()
getLpSolveTime()
getProcessTime()
getSolBuildTime() Returns the time needed to build the Solution after the LP was
                    solved.
label()
solve()
solveLP()
```

Constructors

- *LPBCLDiscountedSolver*

```
public LPBCLDiscountedSolver( jmarkov.jmdp.DTMDP problem, double
interestRate )
```

- **Description**

The constructor method receives a problem of the type infinite DTMDP and an interest rate that is modified for being used as discount factor. The discount factor and the problem gives necessary information for solving a discounted MDP.

- **Parameters**

- * **problem** – the structure of the problem of type infinite DTMDP.
- * **interestRate** – A rate which is paid for the use of a resource.

Methods

- *buildSolution*

public jmarkov.basic.Solution **buildSolution**() throws
jmarkov.basic.exceptions.SolverException

– **See also**

* LPSolver.buildSolution() (see 5.1.1, page 174)

- *description*

java.lang.String **description**()

– **Description copied from jmarkov.basic.JMarkovElement** (see 2.1.3, page 63)

This method return a complete verbal describtion of this element. This description may contain multiple text rows.

– **Returns** – A String describing this element.

– **See also**

* jmarkov.basic.JMarkovElement.label() (see 2.1.3, page 64)

- *getBuildTime*

long **getBuildTime**()

– **Description copied from LPSolver** (see 5.1.1, page 174)

Returns the time taken to build and write the MPS file.

– **Returns** – Returns the buildTime.

- *getIterations*

public abstract long **getIterations**()

– **Returns** – Returns the iterations in the last solve.

- *getLpSolveTime*

long **getLpSolveTime**()

– **Description copied from LPSolver** (see 5.1.1, page 174)

Return the time taken to solve the LP model.

– **Returns** – Returns the lpSolveTime.

- *getProcessTime*

public abstract long **getProcessTime**()

– **Returns** – Returns the processTime of the last solve. Use System.currentTimeMillis() to get the current time.

- *getSolBuildTime*

public long **getSolBuildTime**()

– **Description**

Returns the time needed to build the Solution after the LP was solved.

– **Returns** – Returns the solBuildTime.

- *label*

public abstract java.lang.String **label**()

- **Description copied from Solver** (see 5.2.14, page 216)

The sub classes must return the Solver name.

- **See also**

* `Solver.toString()` (see 5.2.14, page 220)

- *solve*

```
public abstract jmarkov.basic.Solution solve( ) throws
jmarkov.basic.exceptions.SolverException
```

- **Description copied from Solver** (see 5.2.14, page 216)

Called to solve the problem. This method MUST write the local variable policy and valueFunction.

- **Returns** – The solution Object taht contains the plicy and value fuenction.

- **Throws**

* `jmarkov.basic.exceptions.SolverException` – This exception is thrown if the solver cannot find a solution for some reason.

- *solveLP*

```
public void solveLP( ) throws jmarkov.basic.exceptions.SolverException
```

- **See also**

* `LPsolver.solveLP()` (see 5.1.1, page 175)

Members inherited from class `jmarkov.jmdp.solvers.AbstractDiscountedSolver` (see 5.2.2, page 179)

- protected `discountFactor`
- protected final double `future(jmarkov.basic.State i, jmarkov.basic.Action a, double discountF)`
- protected final double `future(jmarkov.basic.State i, jmarkov.basic.Action a, double discountF, jmarkov.basic.ValueFunction vf)`
- public double `getInterestRate()`
- protected final void `setDiscountFactor(double discountFactor)`
- public final void `setInterestRate(double interestRate)`

Members inherited from class `jmarkov.jmdp.solvers.AbstractInfiniteSolver` (see 5.2.4, page 184)

- protected DTMDP `getDiscreteProblem()`
- public abstract long `getIterations()`
- public InfiniteMDP `getProblem()`
- public void `printSolution(java.io.PrintWriter pw)`

Members inherited from class `jmarkov.jmdp.solvers.Solver` (see 5.2.14, page 216)

- public String `description()`
- public final Policy `getOptimalPolicy()` throws `jmarkov.basic.exceptions.SolverException`
- public final ValueFunction `getOptimalValueFunction()` throws `jmarkov.basic.exceptions.SolverException`
- public MDP `getProblem()`
- public abstract long `getProcessTime()`

- public final ValueFunction **getValueFunction**()
- public final boolean **isSolved**()
- public abstract String **label**()
- protected **policy**
- protected **printProcessTime**
- public void **printSolution**() throws java.lang.Exception
- public void **printSolution**(java.io.PrintWriter **pw**)
- protected **printValueFunction**
- protected **problem**
- public void **setPrintProcessTime**(boolean **val**)
- public void **setPrintValueFunction**(boolean **val**)
- public abstract Solution **solve**() throws jmarkov.basic.exceptions.SolverException
- protected **solved**
- public final String **toString**()
- protected **valueFunction**

5.2.9 Class MpsLpAverageSolver

This class builds the Dual Linear Program for an average infinite horizon MDP in a MPS file.

Declaration

```
public abstract class MpsLpAverageSolver
extends jmarkov.jmdp.solvers.AbstractAverageSolver (see 5.2.1, page 177)
implements MpsLpSolver
```

Constructor summary

- MpsLpAverageSolver(DTMDP)** This constructor creates a solver for this problem.
- MpsLpAverageSolver(DTMDP, String, String)** The constructor method receives a problem of the type infinite DTMDP, the working directory where the MPS file will be stored, and the name that the user wants for the MPS File.

Method summary

```
getBuildTime()
getDiscSolver()
getIterations()
getLpSolveTime()
getMpsFile()
getMpsFileName()
getProcessTime()
getSolBuildTime()
getWorkingDir()
solve()
```

Constructors

- *MpsLpAverageSolver*

```
public MpsLpAverageSolver( jmarkov.jmdp.DTMDP problem )
```

 - **Description**
This constructor creates a solver for this problem. The created mps file is stored in a temp folder.
 - **Parameters**
 - * **problem** – The structure of the problem of type infinite DTMDP.
-
- *MpsLpAverageSolver*

```
public MpsLpAverageSolver( jmarkov.jmdp.DTMDP problem, java.lang.String
workingDir, java.lang.String fileName )
```

 - **Description**
The constructor method receives a problem of the type infinite DTMDP, the working directory where the MPS file will be stored, and the name that the user wants for the MPS File.

– **Parameters**

- * **problem** – The problem to be solved.
- * **workingDir** – Where the file will be created.
- * **fileName** – Label for the MPS File.

Methods

- *getBuildTime*
 public long **getBuildTime**()

- *getDiscSolver*
 protected MpsLpDiscountedSolver **getDiscSolver**()

 – **Returns** – Returns the discSolver.

- *getIterations*
 public abstract long **getIterations**()

 – **Returns** – Returns the iterations in the last solve.

- *getLpSolveTime*
 public long **getLpSolveTime**()

- *getMpsFile*
 java.io.File **getMpsFile**()

 – **Description copied from MpsLpSolver** (see 5.1.2, page 176)
 Returns the MPS file name.
 – **Returns** – Returns the MPS generated file.

- *getMpsFileName*
 java.lang.String **getMpsFileName**()

 – **Description copied from MpsLpSolver** (see 5.1.2, page 176)
 Returns the MPS file name.
 – **Returns** – Returns the fileName.

- *getProcessTime*
 public abstract long **getProcessTime**()

 – **Returns** – Returns the processTime of the last solve. Use
 System.currentTimeMillis() to get the current time.

- *getSolBuildTime*
 public long **getSolBuildTime**()

- *getWorkingDir*
 public final java.io.File **getWorkingDir**()

 – **See also**
 * MpsLpSolver.getWorkingDir() (see 5.1.2, page 176)

- *solve*

```
public abstract jmarkov.basic.Solution solve( ) throws
jmarkov.basic.exceptions.SolverException
```

 - **Description copied from Solver** (see 5.2.14, page 216)
Called to solve the problem. This method MUST write the local variable policy and valueFunction.
 - **Returns** – The solution Object taht contains the plicy and value fuenction.
 - **Throws**
 - * jmarkov.basic.exceptions.SolverException – This exception is thrown if the solver cannot find a solution for some reason.

Members inherited from class jmarkov.jmdp.solvers.AbstractAverageSolver (see 5.2.1, page 177)

Members inherited from class jmarkov.jmdp.solvers.AbstractInfiniteSolver (see 5.2.4, page 184)

-
- protected DTMDP `getDiscreteProblem()`
 - public abstract long `getIterations()`
 - public InfiniteMDP `getProblem()`
 - public void `printSolution(java.io.PrintWriter pw)`

Members inherited from class jmarkov.jmdp.solvers.Solver (see 5.2.14, page 216)

-
- public String `description()`
 - public final Policy `getOptimalPolicy()` throws jmarkov.basic.exceptions.SolverException
 - public final ValueFunction `getOptimalValueFunction()` throws jmarkov.basic.exceptions.SolverException
 - public MDP `getProblem()`
 - public abstract long `getProcessTime()`
 - public final ValueFunction `getValueFunction()`
 - public final boolean `isSolved()`
 - public abstract String `label()`
 - protected `policy`
 - protected `printProcessTime`
 - public void `printSolution()` throws java.lang.Exception
 - public void `printSolution(java.io.PrintWriter pw)`
 - protected `printValueFunction`
 - protected `problem`
 - public void `setPrintProcessTime(boolean val)`
 - public void `setPrintValueFunction(boolean val)`
 - public abstract Solution `solve()` throws jmarkov.basic.exceptions.SolverException
 - protected `solved`
 - public final String `toString()`
 - protected `valueFunction`

5.2.10 Class MpsLpDiscountedSolver

This class builds a Linear Program for a discounted infinite horizon MDP in a MPS file. A extending class must code solveLP method in order to solve the problem.

See also

- `MpsLpDiscountedSolver.solveLP()` (see 5.2.10, page 204)

Declaration

```
public abstract class MpsLpDiscountedSolver
extends jmarkov.jmdp.solvers.AbstractDiscountedSolver (see 5.2.2, page 179)
implements MpsLpSolver
```

Constructor summary

- MpsLpDiscountedSolver(DTMDP, double)** This is the default constructor for MpsLpDiscountedSolver class, and defines the label MDP for the MPS File.
- MpsLpDiscountedSolver(DTMDP, double, boolean)** The constructor is used by the partner average solver.
- MpsLpDiscountedSolver(DTMDP, double, String, String)** The constructor method exclusively receives a problem of the type infinite DTMDP , an interest rate that is modified for being used as discount factor and the name that the user wants for the MPS File.
- MpsLpDiscountedSolver(DTMDP, double, String, String, boolean)** The constructor is used by the partenr average solver.

Method summary

- buildSolution()** The implementator classes should override this class to build the solution after the model has been solved.
- getBuildTime()**
- getLpSolveTime()**
- getMpsFile()**
- getMpsFileName()**
- getProcessTime()**
- getSolBuildTime()**
- getWorkingDir()** Returns the working directory (where the MPS file is located)
- isAvg()**
- solve()**
- solveLP()** The implementator classes should override this class to solve the problem using the mpsFile that has been created.

Constructors

- *MpsLpDiscountedSolver*

```
public MpsLpDiscountedSolver( jmarkov.jmdp.DTMDP problem, double
interestRate )
```

- **Description**

This is the default constructor for MpsLpDiscountedSolver class, and defines the label MDP for the MPS File. The constructor method exclusively receives a problem of the type infinite DTMDP , an interest rate that is modified for being used as discount factor.

- **Parameters**

- * **problem** – The structure of the problem of type infinite DTMDP.
- * **interestRate** – A rate which is paid for the use of a resource.

- *MpsLpDiscountedSolver*

```
protected MpsLpDiscountedSolver( jmarkov.jmdp.DTMDP problem, double
interestRate, boolean isAverage )
```

- **Description**

The constructor is used by the partner average solver.

- **Parameters**

- * **problem** – The structure of the problem of type infinite DTMDP.
- * **interestRate** – A rate which is paid for the use of a resource.
- * **isAverage** – True if an average model is being built.

- *MpsLpDiscountedSolver*

```
public MpsLpDiscountedSolver( jmarkov.jmdp.DTMDP problem, double
interestRate, java.lang.String workingDir, java.lang.String fileName )
```

- **Description**

The constructor method exclusively receives a problem of the type infinite DTMDP , an interest rate that is modified for being used as discount factor and the name that the user wants for the MPS File.

- **Parameters**

- * **problem** – The structure of the problem of type infinite DTMDP.
- * **interestRate** – A rate which is paid for the use of a resource.
- * **workingDir** – Where the MPS file will be created.
- * **fileName** – Name for the MPS File (with no path).

- *MpsLpDiscountedSolver*

```
protected MpsLpDiscountedSolver( jmarkov.jmdp.DTMDP problem, double
interestRate, java.lang.String workingDir, java.lang.String fileName,
boolean isAverage )
```

- **Description**

The constructor is used by the partenr average solver.

- **Parameters**

- * **problem** – The structure of the problem of type infinite DTMDP.
- * **interestRate** – A rate which is paid for the use of a resource.
- * **workingDir** – Where the MPS file will be created.
- * **fileName** – Name for the MPS File (with no path).
- * **isAverage** – True if an average model is being built.

Methods

- *buildSolution*
`public abstract jmarkov.basic.Solution buildSolution() throws
jmarkov.basic.exceptions.SolverException`
 - **Description**
The implementator classes should override this class to build the solution after the model has been solved.
 - **Returns** – The solution to the problem.

- *getBuildTime*
`public final long getBuildTime()`

- *getLpSolveTime*
`public final long getLpSolveTime()`

- *getMpsFile*
`java.io.File getMpsFile()`
 - **Description copied from MpsLpSolver** (see 5.1.2, page 176)
Returns the MPS file name.
 - **Returns** – Returns the MPS generated file.

- *getMpsFileName*
`java.lang.String getMpsFileName()`
 - **Description copied from MpsLpSolver** (see 5.1.2, page 176)
Returns the MPS file name.
 - **Returns** – Returns the fileName.

- *getProcessTime*
`public abstract long getProcessTime()`
 - **Returns** – Returns the processTime of the last solve. Use `System.currentTimeMillis()` to get the current time.

- *getSolBuildTime*
`public final long getSolBuildTime()`

- *getWorkingDir*
`public final java.io.File getWorkingDir()`
 - **Description**
Returns the working directory (where the MPS file is located)
 - **Returns** – Returns the MPS File folder.

- *isAvg*
`public final boolean isAvg()`
 - **Returns** – Returns the true if an Average problem is being solved.

- *solve*
`public abstract jmarkov.basic.Solution solve() throws
jmarkov.basic.exceptions.SolverException`

- **Description copied from Solver** (see 5.2.14, page 216)
Called to solve the problem. This method MUST write the local variable policy and valueFunction.
- **Returns** – The solution Object taht contains the plicy and value fuenction.
- **Throws**
 - * jmarkov.basic.exceptions.SolverException – This exception is thrown if the solver cannot find a solution for some reason.

- *solveLP*

```
public abstract void solveLP( ) throws
jmarkov.basic.exceptions.SolverException
```

- **Description**
The implementator classes should override this class to solve the problem using the mpsFile that has been created.

Members inherited from class jmarkov.jmdp.solvers.AbstractDiscountedSolver (see 5.2.2, page 179)

- protected **discountFactor**
- protected final double **future**(jmarkov.basic.State i, jmarkov.basic.Action a, double **discountF**)
- protected final double **future**(jmarkov.basic.State i, jmarkov.basic.Action a, double **discountF**, jmarkov.basic.ValueFunction vf)
- public double **getInterestRate**()
- protected final void **setDiscountFactor**(double **discountFactor**)
- public final void **setInterestRate**(double **interestRate**)

Members inherited from class jmarkov.jmdp.solvers.AbstractInfiniteSolver (see 5.2.4, page 184)

- protected DTMDP **getDiscreteProblem**()
- public abstract long **getIterations**()
- public InfiniteMDP **getProblem**()
- public void **printSolution**(java.io.PrintWriter pw)

Members inherited from class jmarkov.jmdp.solvers.Solver (see 5.2.14, page 216)

- public String **description**()
- public final Policy **getOptimalPolicy**() throws jmarkov.basic.exceptions.SolverException
- public final ValueFunction **getOptimalValueFunction**() throws jmarkov.basic.exceptions.SolverException
- public MDP **getProblem**()
- public abstract long **getProcessTime**()
- public final ValueFunction **getValueFunction**()
- public final boolean **isSolved**()
- public abstract String **label**()
- protected **policy**
- protected **printProcessTime**
- public void **printSolution**() throws java.lang.Exception
- public void **printSolution**(java.io.PrintWriter pw)
- protected **printValueFunction**
- protected **problem**
- public void **setPrintProcessTime**(boolean val)
- public void **setPrintValueFunction**(boolean val)
- public abstract Solution **solve**() throws jmarkov.basic.exceptions.SolverException
- protected **solved**
- public final String **toString**()
- protected **valueFunction**

5.2.11 Class PolicyIterationSolver

This class solves infinite horizon discounted problems using the policy iteration algorithm. It extends Solver and should only be used on infinite horizon problems. The objective function the solver uses is the discounted cost. The result is a deterministic optimal policy for the given structure. Policy Iteration is a solver method this is always convergent in a finite number of iterations. The algorithm has to solve a linear system of equations as big as the amount of states. When there are too many states, it is recommendable to use other solvers, or using the modified policy iteration (by using the second constructor). Policy Iteration is a solver method this is always convergent in a finite number of iterations. The algorithm has to solve a linear system of equations as big as the amount of states. When there are too many states, it is recommendable to use other solvers. The advantage of using Policy Iteration is that the result is the true optimal solution and not an approximation as in other common methods. The method starts with a policy. It solves the system of linear equations for the value functions for that policy. With this values it looks for a better policy. It then solves the value functions again and looks for a better policy. If this policy is equal to the last policy tried, it stops, in any other case it keeps improving the policy and updating the value functions.

Declaration

```
public class PolicyIterationSolver
  extends jmarkov.jmdp.solvers.AbstractDiscountedSolver (see 5.2.2, page 179)
```

Field summary

iterations Used to store the number of iterations
processTime Used to store process time

Constructor summary

PolicyIterationSolver(DTMDP, double) The constructor method exclusively receives a problem of the type InfiniteMDP because this solver is only designed to work on infinite horizon problems.
PolicyIterationSolver(DTMDP, double, boolean) The constructor method exclusively receives a problem of the type InfiniteMDP because this solver is only designed to work on infinite horizon problems.

Method summary

description()
getIncreasingFactor()
getInitialIterations()
getIterations()
getProcessTime()
label()
setIncreasingFactor(double) Sets the increasing factor of the maximum iterations of the Modified policy iteration method.
setInitialIterations(int) Sets maximum iterations for the first run of the modified policy iteration.
setModifiedPolicy(boolean) Activates the modified policy iteration algorithm.
solve()

solveMatrix() This method is used by the PolicyIterationSolver to solve the linear system of equations to determine the value functions of each state for a given policy.

solveMatrixModified(DecisionRule) This method is used by the PolicyIterationSolver to solve the linear system of equations to determine the value functions of each state for a given policy.

Fields

- protected long **iterations**
 - Used to store the number of iterations
- protected long **processTime**
 - Used to store process time

Constructors

- *PolicyIterationSolver*

```
public PolicyIterationSolver( jmarkov.jmdp.DTMDP problem, double
discountFactor )
```

- **Description**

The constructor method exclusively receives a problem of the type InfiniteMDP because this solver is only designed to work on infinite horizon problems. This solver solves the discounted objective function problem.

- **Parameters**

- * **problem** – the structure of the problem of type InfiniteMDP
- * **discountFactor** – represents how much less is the reward received in the next period instead of receiving it in the present period.

-
- *PolicyIterationSolver*

```
public PolicyIterationSolver( jmarkov.jmdp.DTMDP problem, double
discountFactor, boolean setModifiedPolicy )
```

- **Description**

The constructor method exclusively receives a problem of the type InfiniteMDP because this solver is only designed to work on infinite horizon problems. This solver solves the discounted objective function problem.

- **Parameters**

- * **problem** – the structure of the problem of type InfiniteMDP
- * **discountFactor** – represents how much less is the reward received in the next period instead of receiving it in the present period.
- * **setModifiedPolicy** –

Methods

- *description*
`java.lang.String description()`
 - **Description copied from `jmarkov.basic.JMarkovElement`** (see 2.1.3, page 63)
This method return a complete verbal description of this element. This description may contain multiple text rows.
 - **Returns** – A String describing this element.
 - **See also**
 - * `jmarkov.basic.JMarkovElement.label()` (see 2.1.3, page 64)

- *getIncreasingFactor*
`public double getIncreasingFactor()`
 - **Returns** – increasing factor of the maximum iterations.

- *getInitialIterations*
`public double getInitialIterations()`
 - **Returns** – initial maximum iterations of the modified policy iteration algorithm.

- *getIterations*
`public abstract long getIterations()`
 - **Returns** – Returns the iterations in the last solve.

- *getProcessTime*
`public abstract long getProcessTime()`
 - **Returns** – Returns the processTime of the last solve. Use `System.currentTimeMillis()` to get the current time.

- *label*
`public abstract java.lang.String label()`
 - **Description copied from `Solver`** (see 5.2.14, page 216)
The sub classes must return the Solver name.
 - **See also**
 - * `Solver.toString()` (see 5.2.14, page 220)

- *setIncreasingFactor*
`public void setIncreasingFactor(double increasingFactor)`
 - **Description**
Sets the increasing factor of the maximum iterations of the Modified policy iteration method. The first iterations are a vague aproximation to the real value functions and need not be exhaustive. But the last iterations must refine the value functions in order to get better precision. The increasing factor determines how many iteratinos are to be done in each iteration. Faster growth will be more precise but computationally more expensive.
 - **Parameters**

* **increasingFactor** – greater than 1. Determines max iterations growth.

- *setInitialIterations*

public void **setInitialIterations**(int **initialIterations**)

- **Description**

Sets maximum iterations for the first run of the modified policy iteration.

- **Parameters**

* **initialIterations** –

- *setModifiedPolicy*

public void **setModifiedPolicy**(boolean **val**)

- **Description**

Activates the modified policy iteration algorithm.

- **Parameters**

* **val** – True if the modified policy iteration is to be used.

- *solve*

public abstract jmarkov.basic.Solution **solve**() throws
jmarkov.basic.exceptions.SolverException

- **Description copied from Solver** (see 5.2.14, page 216)

Called to solve the problem. This method MUST write the local variable policy and valueFunction.

- **Returns** – The solution Object that contains the policy and value function.

- **Throws**

* jmarkov.basic.exceptions.SolverException – This exception is thrown if the solver cannot find a solution for some reason.

- *solveMatrix*

protected jmarkov.basic.ValueFunction **solveMatrix**() throws
jmarkov.basic.exceptions.SolverException

- **Description**

This method is used by the PolicyIterationSolver to solve the linear system of equations to determine the value functions of each state for a given policy.

- **Returns** – a DenseVector (type defined in the JMP package documentation) with the value functions for each state. The index for each state are the same ones determined in the localStates ArrayList

- **Throws**

* jmarkov.basic.exceptions.SolverException –

- *solveMatrixModified*

protected jmarkov.basic.ValueFunction **solveMatrixModified**(
jmarkov.basic.DecisionRule **localDecisionRule**)

- **Description**

This method is used by the PolicyIterationSolver to solve the linear system of equations to determine the value functions of each state for a given policy.

- **Returns** – a DenseVector (type defined in the JMP package documentation) with the value functions for each state. The index for each state are the same ones determined in the localStates ArrayList declared as static.

Members inherited from class `jmarkov.jmdp.solvers.AbstractDiscountedSolver` (see 5.2.2, page 179)

- protected `discountFactor`
- protected final double `future(jmarkov.basic.State i, jmarkov.basic.Action a, double discountF)`
- protected final double `future(jmarkov.basic.State i, jmarkov.basic.Action a, double discountF, jmarkov.basic.ValueFunction vf)`
- public double `getInterestRate()`
- protected final void `setDiscountFactor(double discountFactor)`
- public final void `setInterestRate(double interestRate)`

Members inherited from class `jmarkov.jmdp.solvers.AbstractInfiniteSolver` (see 5.2.4, page 184)

- protected DTMDP `getDiscreteProblem()`
- public abstract long `getIterations()`
- public InfiniteMDP `getProblem()`
- public void `printSolution(java.io.PrintWriter pw)`

Members inherited from class `jmarkov.jmdp.solvers.Solver` (see 5.2.14, page 216)

- public String `description()`
- public final Policy `getOptimalPolicy()` throws `jmarkov.basic.exceptions.SolverException`
- public final ValueFunction `getOptimalValueFunction()` throws `jmarkov.basic.exceptions.SolverException`
- public MDP `getProblem()`
- public abstract long `getProcessTime()`
- public final ValueFunction `getValueFunction()`
- public final boolean `isSolved()`
- public abstract String `label()`
- protected `policy`
- protected `printProcessTime`
- public void `printSolution()` throws `java.lang.Exception`
- public void `printSolution(java.io.PrintWriter pw)`
- protected `printValueFunction`
- protected `problem`
- public void `setPrintProcessTime(boolean val)`
- public void `setPrintValueFunction(boolean val)`
- public abstract Solution `solve()` throws `jmarkov.basic.exceptions.SolverException`
- protected `solved`
- public final String `toString()`
- protected `valueFunction`

5.2.12 Class ProbabilitySolver

This class is designed to calculate the long run probabilities of infinite horizon problem. It uses Jacobi and Power methods for sparse matrixes. Of course this class is not needed, and it is here for idiotic reasons and the extreme arrogance of Mr. Sarmiento. To do this jmdp should pass the problem to JMarkov

Declaration

```
public class ProbabilitySolver
extends java.lang.Object
```

Constructor summary

- ProbabilitySolver(CTMDP)** Initializes a new solver for continuous chains and solves the probabilities for the optimal policy.
- ProbabilitySolver(CTMDP, DecisionRule)** Initializes a new solver for continuous chains and solves the probabilities for a particular decision rule.
- ProbabilitySolver(DTMDP)** Initializes a new solver for discrete chains and solves the probabilities for the optimal policy.
- ProbabilitySolver(DTMDP, DecisionRule)** Initializes a new solver for discrete chains

Method summary

- getProbability()**
- isSolved()**
- setGaussSeidel(boolean)** The GaussSeidel modification of the ValueIteration method is a change that is guaranteed to have a performance at least as good as the methods without the modifications.
- setJacobi(boolean)**
- solve()** Solves the probabilities

Constructors

- *ProbabilitySolver*

```
public ProbabilitySolver( jmarkov.jmdp.CTMDP problem ) throws
jmarkov.basic.exceptions.SolverException
```

 - **Description**
Initializes a new solver for continuous chains and solves the probabilities for the optimal policy.
 - **Parameters**
 - * **problem** – continuous time , infinite horizon problem
 - **Throws**
 - * **jmarkov.basic.exceptions.SolverException** –
- *ProbabilitySolver*

```
public ProbabilitySolver( jmarkov.jmdp.CTMDP problem,
jmarkov.basic.DecisionRule dr )
```

- **Description**

Initializes a new solver for continuous chains and solves the probabilities for a particular decision rule.

- **Parameters**

- * **problem** – continuous time , infinite horizon problem
- * **dr** –

- *ProbabilitySolver*

```
public ProbabilitySolver( jmarkov.jmdp.DTMDP problem ) throws
jmarkov.basic.exceptions.SolverException
```

- **Description**

Initializes a new solver for discrete chains and solves the probabilities for the optimal policy.

- **Parameters**

- * **problem** – discrete time , infinite horizon problem

- **Throws**

- * jmarkov.basic.exceptions.SolverException –

- *ProbabilitySolver*

```
public ProbabilitySolver( jmarkov.jmdp.DTMDP problem,
jmarkov.basic.DecisionRule dr )
```

- **Description**

Initializes a new solver for discrete chains

- **Parameters**

- * **problem** – discrete time , infinite horizon problem
- * **dr** – decision rule to be evaluated

Methods

- *getProbability*

```
public jmarkov.basic.ValueFunction getProbability( )
```

- **Returns** – Returns the probability.

- *isSolved*

```
public boolean isSolved( )
```

- **Returns** – true if the probabilities were calculated.

- *setGaussSeidel*

```
public void setGaussSeidel( boolean val )
```

- **Description**

The GaussSeidel modification of the ValueIteration method is a change that is guaranteed to have a performance at least as good as the methods without the modifications. In many problems, specially the ones with many states, the modification can imply a significant improvement. By default it set to true. It provides no significant improvement if used jointly with the ErrorBounds modification.

- **Parameters**

* `val` – sets whether or not the GaussSeidel modification will be used.

- *setJacobi*

```
public void setJacobi( boolean val )
```

- **Parameters**

- * `val` – true to use jacobi methods

- *solve*

```
public void solve( )
```

- **Description**

- Solves the probabilities

5.2.13 Class RelativeValueIterationSolver

This class solves the average cost criteria for infinite horizon problems

Declaration

```
public class RelativeValueIterationSolver
  extends jmarkov.jmdp.solvers.AbstractAverageSolver (see 5.2.1, page 177)
```

Constructor summary

- RelativeValueIterationSolver(CTMDP)** Creates a new solver for a continuous time, infinite horizon problem.
- RelativeValueIterationSolver(CTMDP, double)** Creates a new solver for a continuous time, infinite horizon problem to be solved with the modified relative value iteration method.
- RelativeValueIterationSolver(DTMDP)** The constructor method exclusively receives a discrete time infinite horizon problem of the type DTMDP.
- RelativeValueIterationSolver(DTMDP, double)** Creates a new solver for the given discrete time, infinite horizon problem.

Method summary

```
getIterations()
getProcessTime()
label()
setFactor(double) Sets the factor for the modified relative value iteration method.
setPrintValueFunction(boolean)
solve()
```

Constructors

- *RelativeValueIterationSolver*
 public **RelativeValueIterationSolver**(jmarkov.jmdp.CTMDP problem)
 - **Description**
Creates a new solver for a continuous time, infinite horizon problem.
 - **Parameters**
 * problem – continuous time, infinite horizon problem
- *RelativeValueIterationSolver*
 public **RelativeValueIterationSolver**(jmarkov.jmdp.CTMDP problem, double factor)
 - **Description**
Creates a new solver for a continuous time, infinite horizon problem to be solved with the modified relative value iteration method.
 - **Parameters**
 * problem – continuous time, infinite horizon problem
 * factor –

- *RelativeValueIterationSolver*

public RelativeValueIterationSolver(jmarkov.jmdp.DTMDP problem)

- **Description**

The constructor method exclusively receives a discrete time infinite horizon problem of the type DTMDP.

- **Parameters**

* **problem** – the structure of the problem of type InfiniteMDP

- *RelativeValueIterationSolver*

public RelativeValueIterationSolver(jmarkov.jmdp.DTMDP problem, double factor)

- **Description**

Creates a new solver for the given discrete time, infinite horizon problem. It uses the modified relative value iteration method. For details, consult the User's Manual. The factor helps avoiding periodicity in the chain.

- **Parameters**

* **problem** – problem
 * **factor** – factor

Methods

- *getIterations*

public abstract long getIterations()

- **Returns** – Returns the iterations in the last solve.

- *getProcessTime*

public abstract long getProcessTime()

- **Returns** – Returns the processTime of the last solve. Use `System.currentTimeMillis()` to get the current time.

- *label*

public abstract java.lang.String label()

- **Description copied from Solver** (see 5.2.14, page 216)

The sub classes must return the Solver name.

- **See also**

* `Solver.toString()` (see 5.2.14, page 220)

- *setFactor*

public void setFactor(double factor)

- **Description**

Sets the factor for the modified relative value iteration method.

- **Parameters**

* **factor** – A number between 0 and 1.

- *setPrintValueFunction*

```
public void setPrintValueFunction( boolean val )
```

- **Description copied from Solver** (see 5.2.14, page 216)
Option to print the final value function for each state. It is set to false by default.
 - **Parameters**
* val – True if the value function is to be reported.
-

- *solve*

```
public abstract jmarkov.basic.Solution solve( ) throws  
jmarkov.basic.exceptions.SolverException
```

- **Description copied from Solver** (see 5.2.14, page 216)
Called to solve the problem. This method MUST write the local variable policy and valueFunction.
- **Returns** – The solution Object taht contains the plicy and value fuenction.
- **Throws**
* jmarkov.basic.exceptions.SolverException – This exception is thrown if the solver cannot find a solution for some reason.

Members inherited from class jmarkov.jmdp.solvers.AbstractAverageSolver (see 5.2.1, page 177)

Members inherited from class jmarkov.jmdp.solvers.AbstractInfiniteSolver (see 5.2.4, page 184)

- protected DTMDP getDiscreteProblem()
- public abstract long getIterations()
- public InfiniteMDP getProblem()
- public void printSolution(java.io.PrintWriter pw)

Members inherited from class jmarkov.jmdp.solvers.Solver (see 5.2.14, page 216)

- public String description()
- public final Policy getOptimalPolicy() throws jmarkov.basic.exceptions.SolverException
- public final ValueFunction getOptimalValueFunction() throws jmarkov.basic.exceptions.SolverException
- public MDP getProblem()
- public abstract long getProcessTime()
- public final ValueFunction getValueFunction()
- public final boolean isSolved()
- public abstract String label()
- protected policy
- protected printProcessTime
- public void printSolution() throws java.lang.Exception
- public void printSolution(java.io.PrintWriter pw)
- protected printValueFunction
- protected problem
- public void setPrintProcessTime(boolean val)
- public void setPrintValueFunction(boolean val)
- public abstract Solution solve() throws jmarkov.basic.exceptions.SolverException
- protected solved
- public final String toString()
- protected valueFunction

5.2.14 Class Solver

Structural class for every solver. Any solver that a user implements must extend this class.

Declaration

```
public abstract class Solver
  extends java.lang.Object
  implements jmarkov.basic.JMarkovElement
```

All known subclasses

ValueIterationSolver (see 5.2.16, page 224) , StochasticShortestPathSolver (see 5.2.15, page 221) , RelativeValueIterationSolver (see 5.2.13, page 213) , PolicyIterationSolver (see 5.2.11, page 205) , MpsLpDiscountedSolver (see 5.2.10, page 201) , MpsLpAverageSolver (see 5.2.9, page 198) , LPBCLDiscountedSolver (see 5.2.8, page 194) , LPBCLAverageSolver (see 5.2.7, page 191) , FiniteSolver (see 5.2.6, page 188) , AbstractTotalSolver (see 5.2.5, page 186) , AbstractInfiniteSolver (see 5.2.4, page 184) , AbstractFiniteSolver (see 5.2.3, page 182) , AbstractDiscountedSolver (see 5.2.2, page 179) , AbstractAverageSolver (see 5.2.1, page 177)

Field summary

policy The policy Object.
printProcessTime True if the process time is to be reported
printValueFunction true if the value function is to be reported
problem The problem to be solved
solved Marker to indicate that the problem has been solved
valueFunction The value function, to be written by the solver

Constructor summary

Solver(MDP) Default constructor.

Method summary

description()
getOptimalPolicy() Gets the optimal policy.
getOptimalValueFunction() Gets the optimal ValueFunction.
getProblem() Returns the problem associated with this solver.
getProcessTime()
getValueFunction() If the problem is solved, it will return the optimal value function.
isSolved() Tells whether the problem has been solved.
label() The sub classes must return the Solver name.
printSolution() Prints the solution in the default PrintWriter (System.out)
printSolution(PrintWriter) Prints the solution on a given PrintWriter.
setPrintProcessTime(boolean) Option to print the time spent solving the problem.
setPrintValueFunction(boolean) Option to print the final value function for each state.
solve() Called to solve the problem.
toString() This calls label().

Fields

- protected jmarkov.basic.Policy **policy**
 - The policy Object. This should be written by the solver.
- protected jmarkov.basic.ValueFunction **valueFunction**
 - The value function, to be written by the solver
- protected boolean **solved**
 - Marker to indicate that the problem has been solved
- protected jmarkov.jmdp.MDP **problem**
 - The problem to be solved
- protected boolean **printProcessTime**
 - True if the process time is to be reported
- protected boolean **printValueFunction**
 - true if the value function is to be reported

Constructors

- *Solver*
protected **Solver**(jmarkov.jmdp.MDP **problem**)
 - **Description**
Default constructor. Receives the problem to solve. All sub classes MUST call this constructor.
 - **Parameters**
* **problem** – to be solved.

Methods

- *description*
java.lang.String **description**()
 - **Description copied from jmarkov.basic.JMarkovElement** (see 2.1.3, page 63)
This method return a complete verbal description of this element. This description may contain multiple text rows.
 - **Returns** – A String describing this element.
 - **See also**
* jmarkov.basic.JMarkovElement.label() (see 2.1.3, page 64)
- *getOptimalPolicy*
public final jmarkov.basic.Policy **getOptimalPolicy**() throws jmarkov.basic.exceptions.SolverException

- **Description**
Gets the optimal policy. It solves the problem if it has not been solved.
- **Returns** – the optimal Policy.
- **Throws**
* jmarkov.basic.exceptions.SolverException –
- **See also**
* jmarkov.basic.Policy (see 2.2.6, page 82)

- *getOptimalValueFunction*
public final jmarkov.basic.ValueFunction **getOptimalValueFunction**() throws
jmarkov.basic.exceptions.SolverException
 - **Description**
Gets the optimal ValueFunction.
 - **Returns** – the optimal ValueFunction.
 - **Throws**
* jmarkov.basic.exceptions.SolverException –
 - **See also**
* jmarkov.basic.ValueFunction (see 2.2.17, page 113)

- *getProblem*
public jmarkov.jmdp.MDP **getProblem**()
 - **Description**
Returns the problem associated with this solver.
 - **Returns** – the problem associated with this solver.

- *getProcessTime*
public abstract long **getProcessTime**()
 - **Returns** – Returns the processTime of the last solve. Use
System.currentTimeMillis() to get the current time.

- *getValueFunction*
public final jmarkov.basic.ValueFunction **getValueFunction**()
 - **Description**
If the problem is solved, it will return the optimal value function. Otherwise it returns
the current valueFunction
 - **Returns** – the value function in the solver.

- *isSolved*
public final boolean **isSolved**()
 - **Description**
Tells whether the problem has been solved.
 - **Returns** – true if the problem has been solved

- *label*
public abstract java.lang.String **label**()

- **Description**

The sub classes must return the Solver name.

- **See also**

* `Solver.toString()` (see 5.2.14, page 220)

- *printSolution*

`public void printSolution() throws java.lang.Exception`

- **Description**

Prints the solution in the default PrintWriter (System.out)

- **Throws**

* `java.lang.Exception` –

- *printSolution*

`public void printSolution(java.io.PrintWriter pw)`

- **Description**

Prints the solution on a given PrintWriter.

- **Parameters**

* `pw` –

- **See also**

* `java.io.PrintWriter`

- *setPrintProcessTime*

`public void setPrintProcessTime(boolean val)`

- **Description**

Option to print the time spent solving the problem. It is set to false by default.

- **Parameters**

* `val` – True if the Process time is to be reported, false otherwise.

- *setPrintValueFunction*

`public void setPrintValueFunction(boolean val)`

- **Description**

Option to print the final value function for each state. It is set to false by default.

- **Parameters**

* `val` – True if the value function is to be reported.

- *solve*

`public abstract jmarkov.basic.Solution solve() throws
jmarkov.basic.exceptions.SolverException`

- **Description**

Called to solve the problem. This method MUST write the local variable policy and valueFunction.

- **Returns** – The solution Object that contains the policy and value function.

- **Throws**

* `jmarkov.basic.exceptions.SolverException` – This exception is thrown if the solver cannot find a solution for some reason.

- *toString*
public final java.lang.String **toString**()
 - **Description**
This calls label().
 - **See also**
 - * **Solver.toString()** (see 5.2.14, page 220)

5.2.15 Class StochasticShortestPathSolver

This solver gives a solution for the minimization of the total cost criterion for an infinite horizon MDP.

Declaration

```
public class StochasticShortestPathSolver
  extends jmarkov.jmdp.solvers.AbstractTotalSolver (see 5.2.5, page 186)
```

Constructor summary

StochasticShortestPathSolver(StochasticShortestPath) Default constructor.

Method summary

bestAction(S) Sets the best action to take in state i, in the variable bestAction.
description()
future(S, A) This method calculates the expected value of valueFunction for the current state and a specified action.
getIterations()
getProblem()
getProcessTime()
label()
solve()

Constructors

- *StochasticShortestPathSolver*
public StochasticShortestPathSolver(jmarkov.jmdp.StochasticShortestPath problem)
 - **Description**
 Default constructor.
 - **Parameters**
 - * **problem** – the structure of the problem of type StochasticShortestPath

Methods

- *bestAction*
protected double bestAction(jmarkov.basic.StateC i) throws jmarkov.basic.exceptions.StructureException
 - **Description**
 Sets the best action to take in state i, in the variable bestAction. Note that in this case StochasticShortestPathProblem Bertsekas expose a transformation for the graph which modify the immediate reward function and the transition probability, only to make a graph without self-transition states. This will increase the finite termination probability for the algorithm.

- **Parameters**
 - * *i* – state for which the best action is being determined
- **Returns** – the new ValueFunction for this state.
- **Throws**
 - * jmarkov.basic.exceptions.StructureException –

- *description*

```
public java.lang.String description( )
```

 - **See also**
 - * java.lang.Object.toString()

- *future*

```
public final double future( jmarkov.basic.StateC i, jmarkov.basic.Action a )
throws jmarkov.basic.exceptions.StructureException
```

 - **Description**

This method calculates the expected value of valueFunction for the current state and a specified action.
 - **Parameters**
 - * *i* – Current State
 - * *a* – Action taken
 - **Returns** – Future value from this state.
 - **Throws**
 - * jmarkov.basic.exceptions.StructureException –

- *getIterations*

```
public abstract long getIterations( )
```

 - **Returns** – Returns the iterations in the last solve.

- *getProblem*

```
public jmarkov.jmdp.StochasticShortestPath getProblem( )
```

 - **See also**
 - * Solver.getProblem() (see 5.2.14, page 218)

- *getProcessTime*

```
public abstract long getProcessTime( )
```

 - **Returns** – Returns the processTime of the last solve. Use System.currentTimeMillis() to get the current time.

- *label*

```
public java.lang.String label( )
```

 - **See also**
 - * Solver.label() (see 5.2.14, page 218)

- *solve*

```
public abstract jmarkov.basic.Solution solve( ) throws
jmarkov.basic.exceptions.SolverException
```

- **Description copied from Solver** (see 5.2.14, page 216)
Called to solve the problem. This method MUST write the local variable policy and valueFunction.
- **Returns** – The solution Object taht contains the plicy and value fuenction.
- **Throws**
 - * jmarkov.basic.exceptions.SolverException – This exception is thrown if the solver cannot find a solution for some reason.

Members inherited from class jmarkov.jmdp.solvers.AbstractTotalSolver (see 5.2.5, page 186)

Members inherited from class jmarkov.jmdp.solvers.AbstractInfiniteSolver (see 5.2.4, page 184)

-
- protected DTMDP getDiscreteProblem()
 - public abstract long getIterations()
 - public InfiniteMDP getProblem()
 - public void printSolution(java.io.PrintWriter pw)

Members inherited from class jmarkov.jmdp.solvers.Solver (see 5.2.14, page 216)

-
- public String description()
 - public final Policy getOptimalPolicy() throws jmarkov.basic.exceptions.SolverException
 - public final ValueFunction getOptimalValueFunction() throws jmarkov.basic.exceptions.SolverException
 - public MDP getProblem()
 - public abstract long getProcessTime()
 - public final ValueFunction getValueFunction()
 - public final boolean isSolved()
 - public abstract String label()
 - protected policy
 - protected printProcessTime
 - public void printSolution() throws java.lang.Exception
 - public void printSolution(java.io.PrintWriter pw)
 - protected printValueFunction
 - protected problem
 - public void setPrintProcessTime(boolean val)
 - public void setPrintValueFunction(boolean val)
 - public abstract Solution solve() throws jmarkov.basic.exceptions.SolverException
 - protected solved
 - public final String toString()
 - protected valueFunction

5.2.16 Class ValueIterationSolver

This class belongs to the set of default solvers included in the jmdp package. It extends Solver and should only be used on INFINITE horizon problems. The objective is to be able to return an optimal policy given a problem structure.

Declaration

```
public class ValueIterationSolver
  extends jmarkov.jmdp.solvers.AbstractDiscountedSolver (see 5.2.2, page 179)
```

Field summary

iterations Used to store the number of iterations
processTime stores the process time

Constructor summary

ValueIterationSolver(CTMDP, double) Default Constructor for continuous time problems.
ValueIterationSolver(DTMDP, double) Default Constructor for Discrete time problems.

Method summary

bestAction(S) Find the minimal value function for this state and sets the best action to take in state i, in the variable bestAction.
computeNoErrorBounds() Computes an iteration of the Value Iteration Algorithm without the use of error bounds.
computeWithErrorBounds() Computes an iteration of the Value Iteration Algorithm with the use of error bounds.
description()
getEpsilon()
getIterations()
getProcessTime()
init() Initializes the valueFunction for all the states.
isAverage()
label()
setEpsilon(double) Value Iteration is a solver method this is theoretically convergent only after infinite iterations.
solve() Solves the problem.
useErrorBounds(boolean) The ErrorBounds modification to the ValueIteration method is a change that is guaranteed to have a performance at least as good as the method without the modifications.
useGaussSeidel(boolean) The GaussSeidel modification of the ValueIteration method is a change that is guaranteed to have a performance at least as good as the method without the modifications.
usesErrorBounds()
usesGaussSeidel()

Fields

- protected long **processTime**
 - stores the process time
- protected long **iterations**
 - Used to store the number of iterations

Constructors

- *ValueIterationSolver*

```
public ValueIterationSolver( jmarkov.jmdp.CTMDP problem, double interestRate )
```

 - **Description**
Default Constructor for continuous time problems.
 - **Parameters**
 - * **problem** – the structure of the problem of type CTMDP
 - * **interestRate** – represents how much less is the reward received in the next period instead of receiving it in the present period.
-
- *ValueIterationSolver*

```
public ValueIterationSolver( jmarkov.jmdp.DTMDP problem, double interestRate )
```

 - **Description**
Default Constructor for Discrete time problems.
 - **Parameters**
 - * **problem** – the structure of the problem of type DTMDP
 - * **interestRate** – represents how much less is the reward received in the next period instead of receiving it in the present period.

Methods

- *bestAction*

```
protected double bestAction( jmarkov.basic.State i )
```

 - **Description**
Find the minimal value function for this state and sets the best action to take in state i, in the variable bestAction.
 - **Parameters**
 - * **i** – state for which the best action is being determined
 - **Returns** – the new ValueFunction for this state.
-
- *computeNoErrorBounds*

```
protected double computeNoErrorBounds( )
```

- **Description**
Computes an iteration of the Value Iteration Algorithm without the use of error bounds.
- **Returns** – maximum change in value function due to this iteration.

- *computeWithErrorBounds*
protected double **computeWithErrorBounds**()
 - **Description**
Computes an iteration of the Value Iteration Algorithm with the use of error bounds.
 - **Returns** – maximum change in value function due to this iteration.

- *description*
java.lang.String **description**()
 - **Description copied from jmarkov.basic.JMarkovElement** (see 2.1.3, page 63)
This method return a complete verbal description of this element. This description may contain multiple text rows.
 - **Returns** – A String describing this element.
 - **See also**
* jmarkov.basic.JMarkovElement.label() (see 2.1.3, page 64)

- *getEpsilon*
public final double **getEpsilon**()
 - **Returns** – Returns the epsilon.

- *getIterations*
public abstract long **getIterations**()
 - **Returns** – Returns the iterations in the last solve.

- *getProcessTime*
public abstract long **getProcessTime**()
 - **Returns** – Returns the processTime of the last solve. Use System.currentTimeMillis() to get the current time.

- *init*
protected void **init**()
 - **Description**
Initializes the valueFunction for all the states.

- *isAverage*
public final boolean **isAverage**()
 - **Returns** – Returns the isAverage.

- *label*
public abstract java.lang.String **label**()
 - **Description copied from Solver** (see 5.2.14, page 216)
The sub classes must return the Solver name.

– **See also**

* `Solver.toString()` (see 5.2.14, page 220)

• *setEpsilon*

`public synchronized void setEpsilon(double epsilon)`

– **Description**

Value Iteration is a solver method this is theoretically convergent only after infinite iterations. Because of the practical impossibility to do this, the solver is designed to stop when the difference between iterations is as much as epsilon. The smaller epsilon is, the closer the result will be to the actual optimum but it will take a longer time to solve the problem. The default value of epsilon is 0.0001.

– **Parameters**

* `epsilon` – maximum difference between iterations.

• *solve*

`public jmarkov.basic.Solution solve()`

– **Description**

Solves the problem.

– **Returns** – returns a Solution with the optimal policy and value function.

• *useErrorBounds*

`public synchronized void useErrorBounds(boolean val)`

– **Description**

The ErrorBounds modification to the ValueIteration method is a change that is guaranteed to have a performance at least as good as the method without the modifications. In many problems, specially the ones with many states, the modification can imply a significant improvement. This method modifies the iterations and the stopping criterion. It builds upper and lower bounds for the optimal in each iteration and stops when the bounds are only delta apart or less ignoring where the actual valueFunction is. The bounds converge faster than the actual valueFunction. By default it is set to false.

– **Parameters**

* `val` – sets whether or not to use the ErrorBounds modification.

• *useGaussSeidel*

`public synchronized void useGaussSeidel(boolean val)`

– **Description**

The GaussSeidel modification of the ValueIteration method is a change that is guaranteed to have a performance at least as good as the method without the modifications. In many problems, specially the ones with many states, the modification can imply a significant improvement. By default it is set to true. It provides no significant improvement if used jointly with the ErrorBounds modification.

– **Parameters**

* `val` – sets whether or not the GaussSeidel modification will be used.

– **See also**

* `ValueIterationSolver.useErrorBounds(boolean)` (see 5.2.16, page 227)

- *usesErrorBounds*
`public final boolean usesErrorBounds()`
 – **Returns** – Returns true if uses Error Bounds.

- *usesGaussSeidel*
`public final boolean usesGaussSeidel()`
 – **Returns** – Returns true if Gauss Seidel is active.

Members inherited from class `jmarkov.jmdp.solvers.AbstractDiscountedSolver` (see 5.2.2, page 179)

- `protected discountFactor`
- `protected final double future(jmarkov.basic.State i, jmarkov.basic.Action a, double discountF)`
- `protected final double future(jmarkov.basic.State i, jmarkov.basic.Action a, double discountF, jmarkov.basic.ValueFunction vf)`
- `public double getInterestRate()`
- `protected final void setDiscountFactor(double discountFactor)`
- `public final void setInterestRate(double interestRate)`

Members inherited from class `jmarkov.jmdp.solvers.AbstractInfiniteSolver` (see 5.2.4, page 184)

- `protected DTMDP getDiscreteProblem()`
- `public abstract long getIterations()`
- `public InfiniteMDP getProblem()`
- `public void printSolution(java.io.PrintWriter pw)`

Members inherited from class `jmarkov.jmdp.solvers.Solver` (see 5.2.14, page 216)

- `public String description()`
- `public final Policy getOptimalPolicy()` throws `jmarkov.basic.exceptions.SolverException`
- `public final ValueFunction getOptimalValueFunction()` throws `jmarkov.basic.exceptions.SolverException`
- `public MDP getProblem()`
- `public abstract long getProcessTime()`
- `public final ValueFunction getValueFunction()`
- `public final boolean isSolved()`
- `public abstract String label()`
- `protected policy`
- `protected printProcessTime`
- `public void printSolution()` throws `java.lang.Exception`
- `public void printSolution(java.io.PrintWriter pw)`
- `protected printValueFunction`
- `protected problem`
- `public void setPrintProcessTime(boolean val)`
- `public void setPrintValueFunction(boolean val)`
- `public abstract Solution solve()` throws `jmarkov.basic.exceptions.SolverException`
- `protected solved`
- `public final String toString()`
- `protected valueFunction`

Chapter 6

Package jmarkov.solvers

<i>Package Contents</i>	<i>Page</i>
<hr/>	
Classes	
GeometricSolver	230
GeometrixSolver	232
JamaSolver	234
<i>Solver implementation for steady state, using JAMA</i>	
JamaTransientSolver	236
<i>This class calculate the transient probabilities.</i>	
MtjLogRedSolver	238
MtjSolver	240
<i>This class uses MTJ to solve Steady State probabilities.</i>	
MtjSolver.EnumPrecond	244
<i>This is the list of preconditioner offered in MTJ.</i>	
MtjSolver.EnumSolver	246
<i>This is the list of solvers provided by MTJ.</i>	
Solver	248
<i>This abstract class has to be extended in order to implement solvers for Steady State and Transient probabilities.</i>	
SteadyStateSolver	250
<i>An abstract clas for steady state solver.</i>	
TransientSolver	252
<i>An abstract class for Transient solvers.</i>	

Provides classes for customizing a solver used by JMarkov to solve transient and steady state probabilities in different models. The bacis classes are @code SteadyStateSolver and @TransientSolver. Implementation of these classes is provided, that rely on Java Matrix Package (Jama) (at <http://math.nist.gov/javanumerics/jama/>) and Matrix Toolkits for Java (MTJ) (at <https://mtj.dev.java.net/>). For overviews, tutorials, examples, guides, and tool documentation, please see:

Copa Group WEB page. (at <http://copa.uniandes.edu.co>)

6.1 Classes

6.1.1 Class GeometricSolver

Declaration

```
public abstract class GeometricSolver
extends jmarkov.solvers.Solver (see 6.1.9, page 248)
```

All known subclasses

MtjLogRedSolver (see 6.1.5, page 238)

Constructor summary

GeometricSolver(MarkovProcess) Builds a Geometrix Solver with the given SimpleMarkovProcess.

Method summary

getRmatrix() This process should be extended in order to compute the R matrix of the QBD.

Constructors

- *GeometricSolver*
 public **GeometricSolver**(jmarkov.MarkovProcess mp)
 - **Description**
 Builds a Geometrix Solver with the given SimpleMarkovProcess.
 - **Parameters**
 * mp – The Markov Process for which the steady state probabilities are sought.

Methods

- *getRmatrix*
 public abstract double[][] **getRmatrix**() throws
 jmarkov.basic.exceptions.NotUnichainException
 - **Description**
 This process should be extended in order to compute the R matrix of the QBD. The user can get information of the SimpleMarkovProcess associated with this solver though the methods `getRates()`, `getGenerator`, and `getRate(State,State)`
 - **Returns** – a Matrix with the R matrix for the given QBD.
 - **Throws**
 * jmarkov.basic.exceptions.NotUnichainException –
 - **See also**
 * jmarkov.MarkovProcess.getGenerator() (see 1.1.5, page 38)
 * jmarkov.MarkovProcess.getRates() (see 1.1.5, page 41)
 * jmarkov.MarkovProcess.getRate(State,State) (see 1.1.5, page 41)

Members inherited from class `jmarkov.solvers.Solver` (see 6.1.9, page 248)

- `public final MarkovProcess getMP()`
- `public abstract String label()`
- `protected mp`
- `public final String toString()`

6.1.2 Class GeometrixSolver

Declaration

```
public abstract class GeometrixSolver
extends jmarkov.solvers.Solver (see 6.1.9, page 248)
```

Constructor summary

GeometrixSolver(MarkovProcess) Builds a Geometrix Solver with the given SimpleMarkovProcess.

Method summary

getRmatrix() This process should be extended in order to compute the R matrix of the QBD.

Constructors

- *GeometrixSolver*
 public **GeometrixSolver**(jmarkov.MarkovProcess mp)
 - **Description**
 Builds a Geometrix Solver with the given SimpleMarkovProcess.
 - **Parameters**
 * mp – The Markov Process for which the steady state probabilities are sought.

Methods

- *getRmatrix*
 public abstract double[][] **getRmatrix**() throws
 jmarkov.basic.exceptions.NotUnichainException
 - **Description**
 This process should be extended in order to compute the R matrix of the QBD. The user can get information of the SimpleMarkovProcess associated with this solver though the methods `getRates()`, `getGenerator`, and `getRate(State,State)`
 - **Returns** – a Matrix with the R matrix for the given QBD.
 - **Throws**
 * jmarkov.basic.exceptions.NotUnichainException –
 - **See also**
 * jmarkov.MarkovProcess.getGenerator() (see 1.1.5, page 38)
 * jmarkov.MarkovProcess.getRates() (see 1.1.5, page 41)
 * jmarkov.MarkovProcess.getRate(State,State) (see 1.1.5, page 41)

Members inherited from class `jmarkov.solvers.Solver` (see 6.1.9, page 248)

- `public final MarkovProcess getMP()`
- `public abstract String label()`
- `protected mp`
- `public final String toString()`

6.1.3 Class JamaSolver

Solver implementation for steady state, using JAMA

Declaration

```
public final class JamaSolver
extends jmarkov.solvers.SteadyStateSolver (see 6.1.10, page 250)
```

Constructor summary

JamaSolver(MarkovProcess)

Method summary

description()
getSteadyState() It find the steady state probabilities.
label()

Constructors

- *JamaSolver*
 public **JamaSolver**(jmarkov.MarkovProcess mp)
 – **Parameters**
 * mp –

Methods

- *description*
 public java.lang.String **description**()
- *getSteadyState*
 public double[] **getSteadyState**() throws
 jmarkov.basic.exceptions.NotUnichainException
 – **Description**
 It find the steady state probabilities. If no Solution is found an array of 0's is returned
- *label*
 public java.lang.String **label**()
 – **See also**
 * Solver.label() (see 6.1.9, page 249)

Members inherited from class jmarkov.solvers.SteadyStateSolver (see 6.1.10, page 250)

- public abstract double **getSteadyState**() throws
 jmarkov.basic.exceptions.NotUnichainException

Members inherited from class `jmarkov.solvers.Solver` (see 6.1.9, page 248)

- `public final MarkovProcess getMP()`
- `public abstract String label()`
- `protected mp`
- `public final String toString()`

6.1.4 Class JamaTransientSolver

This class calculate the transient probabilities. It uses uniformization, and basically it is a wrapper for the 'expUnif' methods in jphase.MarkovMatrix.

See also

- jphase.MarkovMatrix (see 7.2.7, page 301)
- jmarkov.MarkovProcess (see 1.1.5, page 29)

Declaration

```
public class JamaTransientSolver
  extends jmarkov.solvers.TransientSolver (see 6.1.11, page 252)
```

Constructor summary

JamaTransientSolver(MarkovProcess) Default constructor

Method summary

```
description()
getTransientProbs(double[], State)
getTransientProbs(double, State)
getTransientProbs(int, double, State)
label()
```

Constructors

- *JamaTransientSolver*
 public **JamaTransientSolver**(jmarkov.MarkovProcess mp)
 - **Description**
 Default constructor
 - **Parameters**
 * mp – the Markov Process

Methods

- *description*
 public java.lang.String **description**()
 - *getTransientProbs*
 public double[][] **getTransientProbs**(double[] times, jmarkov.basic.State i0)
 - **See also**
 - * TransientSolver.getTransientProbs(double[],State) (see 6.1.11, page 252)
 - * jphase.MarkovMatrix.expUnif(double[],Matrix,Matrix) (see 7.2.7, page 303)
-

- *getTransientProbs*

```
public double[] getTransientProbs( double time, jmarkov.basic.State i0 )
```

– See also

- * TransientSolver.getTransientProbs(double,State) (see 6.1.11, page 253)
 - * jphase.MarkovMatrix.expUnif(double,Matrix,Matrix) (see 7.2.7, page 303)
-

- *getTransientProbs*

```
public double[][] getTransientProbs( int NumberPoints, double delta,
jmarkov.basic.State i0 )
```

– See also

- * TransientSolver.getTransientProbs(int,double,State) (see 6.1.11, page 253)
 - * jphase.MarkovMatrix.expUnif(int,double,Matrix,Matrix) (see 7.2.7, page 303)
-

- *label*

```
public java.lang.String label( )
```

– See also

- * Solver.label() (see 6.1.9, page 249)

Members inherited from class jmarkov.solvers.TransientSolver (see 6.1.11, page 252)

- public double getTransientProbs(double[] times, jmarkov.basic.State i0)
- public abstract double getTransientProbs(double time, jmarkov.basic.State i0)
- public double getTransientProbs(int NumberPoints, double delta, jmarkov.basic.State i0)

Members inherited from class jmarkov.solvers.Solver (see 6.1.9, page 248)

- public final MarkovProcess getMP()
- public abstract String label()
- protected mp
- public final String toString()

6.1.5 Class MtjLogRedSolver

Declaration

```
public class MtjLogRedSolver
extends jmarkov.solvers.GeometricSolver (see 6.1.1, page 230)
```

Constructor summary

MtjLogRedSolver(GeomProcess)

Method summary

description()
getRmatrix()
label()

Constructors

- *MtjLogRedSolver*
public MtjLogRedSolver(jmarkov.GeomProcess mp)
 - **Parameters**
 - * mp – The given QBD

Methods

- *description*
public java.lang.String description()
- *getRmatrix*
public abstract double[][] getRmatrix() throws
jmarkov.basic.exceptions.NotUnichainException
 - **Description copied from GeometricSolver** (see 6.1.1, page 230)
This process should be extended in order to compute the R matrix of the QBD. The user can get information of the SimpleMarkovProcess associated with this solver though the methods `getRates()`, `getGenerator`, and `getRate(State,State)`
 - **Returns** – a Matrix with the R matrix for the given QBD.
 - **Throws**
 - * `jmarkov.basic.exceptions.NotUnichainException` –
 - **See also**
 - * `jmarkov.MarkovProcess.getGenerator()` (see 1.1.5, page 38)
 - * `jmarkov.MarkovProcess.getRates()` (see 1.1.5, page 41)
 - * `jmarkov.MarkovProcess.getRate(State,State)` (see 1.1.5, page 41)
- *label*
public java.lang.String label()
 - **See also**
 - * `Solver.label()` (see 6.1.9, page 249)

Members inherited from class `jmarkov.solvers.GeometricSolver` (see 6.1.1, page 230)

- `public abstract double getRmatrix()` throws `jmarkov.basic.exceptions.NotUnichainException`

Members inherited from class `jmarkov.solvers.Solver` (see 6.1.9, page 248)

- `public final MarkovProcess getMP()`
- `public abstract String label()`
- `protected mp`
- `public final String toString()`

6.1.6 Class MtjSolver

This class uses MTJ to solve Steady State probabilities.

Declaration

```
public class MtjSolver
extends jmarkov.solvers.SteadyStateSolver (see 6.1.10, page 250)
```

Constructor summary

MtjSolver(MarkovProcess) Default constructor.
MtjSolver(MarkovProcess, MtjSolver.EnumSolver) Construct a solver for the given SimpleMarkovProcess.
MtjSolver(MarkovProcess, MtjSolver.EnumSolver, boolean) Construct a solver for the given SimpleMarkovProcess.

Method summary

description()
getCurrentIterSolver()
getCurrentPreConditioner()
getGenerator() Returns the Generator matrix.
getGenMatrix()
getIterativeSolver(Vector)
getIterativeSolver(Vector, MtjSolver.EnumSolver)
getProcessTime()
getSteadyState()
isTryOthers()
label()
setCurrentIterSolver(MtjSolver.EnumSolver) Sets the solver to use.
setCurrentPreConditioner(MtjSolver.EnumPrecond)
setIterSolver(MtjSolver.EnumSolver, boolean) Sets the solver to use.
setTryOthers(boolean) Sets whether the solver shall try other solvers when it fails.

Constructors

- *MtjSolver*

```
public MtjSolver( jmarkov.MarkovProcess mp )
```

 - **Description**
Default constructor. Uses as default solver BiCGStab first. If it fails, it tries other solvers.
 - **Parameters**
 - * mp –

- *MtjSolver*

```
public MtjSolver( jmarkov.MarkovProcess mp, MtjSolver.EnumSolver solver )
```

- **Description**

Construct a solver for the given SimpleMarkovProcess. It uses the given solver enumeration to determine what solver to use.

- **Parameters**

- * **mp** – The Markov process solved
- * **solver** – The solver to use.

- *MtjSolver*

```
public MtjSolver( jmarkov.MarkovProcess mp, MtjSolver.EnumSolver solver,
boolean tryOthers )
```

- **Description**

Construct a solver for the given SimpleMarkovProcess. It uses the given solver enumeration to determine what solver to use. If it fails it tries other solvers.

- **Parameters**

- * **mp** – the Markov Process to solve.
- * **solver** – the iterative solver form the enumeration IterSolver.
- * **tryOthers** – whether a different solver should be tried if the first one fails.

Methods

- *description*

```
public java.lang.String description( )
```

- *getCurrentIterSolver*

```
public MtjSolver.EnumSolver getCurrentIterSolver( )
```

- **Returns** – Returns the currentIterSolver.

- *getCurrentPreConditioner*

```
public MtjSolver.EnumPrecond getCurrentPreConditioner( )
```

- **Returns** – Returns the currentPreConditioner.

- *getGenerator*

```
public no.uib.cipr.matrix.Matrix getGenerator( )
```

- **Description**

Returns the Generator matrix.

- **Returns** – the Generator Matrix **G**.

- *getGenMatrix*

```
public final no.uib.cipr.matrix.Matrix getGenMatrix( )
```

- **Returns** – Returns the generator Matrix.

- *getIterativeSolver*

```
public no.uib.cipr.matrix.sparse.IterativeSolver getIterativeSolver(
no.uib.cipr.matrix.Vector pi )
```

- **Parameters**

- * **pi** – The probability vector.

– **Returns** – Returns the iteartiveSolver.

- *getIterativeSolver*

```
public no.uib.cipr.matrix.sparse.IterativeSolver getIterativeSolver(
    no.uib.cipr.matrix.Vector pi0, MtjSolver.EnumSolver solver )
```

– **Parameters**

- * pi0 – Initial guess value.
- * solver – The solver used.

– **Returns** – Returns the iterativeSolver.

- *getProcessTime*

```
public final long getProcessTime( )
```

– **Returns** – Returns the Process Time of the last solved problem.

- *getSteadyState*

```
public abstract double[] getSteadyState( ) throws
    jmarkov.basic.exceptions.NotUnichainException
```

– **Description copied from SteadyStateSolver** (see 6.1.10, page 250)

This process should be extended in order to compute the steady State probabilities of the MarkovChain. The user can get information of the SimpleMarkovProcess associated with this solver though the methods `getRates()`, `getGenerator`, and `getRate(State,State)`

– **Returns** – an array with the Steady state probabilities for the given problem.

– **Throws**

- * `jmarkov.basic.exceptions.NotUnichainException` –

– **See also**

- * `jmarkov.MarkovProcess.getGenerator()` (see 1.1.5, page 38)
 - * `jmarkov.MarkovProcess.getRates()` (see 1.1.5, page 41)
 - * `jmarkov.MarkovProcess.getRate(State,State)` (see 1.1.5, page 41)
-

- *isTryOthers*

```
public final boolean isTryOthers( )
```

– **Returns** – Returns true if the solver shall try other solvers when it fails.

- *label*

```
public java.lang.String label( )
```

– **See also**

- * `Solver.label()` (see 6.1.9, page 249)
-

- *setCurrentIterSolver*

```
public void setCurrentIterSolver( MtjSolver.EnumSolver iterSolver )
```

– **Description**

Sets the solver to use. It will not try other solvers if this one fails.

– **Parameters**

- * iterSolver –
-

- *setCurrentPreConditioner*

```
public void setCurrentPreConditioner( MtjSolver.EnumPrecond preConditioner
)
```

- **Parameters**

- * `preConditioner` – The currentPreConditioner to set.

- *setIterSolver*

```
public void setIterSolver( MtjSolver.EnumSolver iterSolver, boolean tryOthers
)
```

- **Description**

- Sets the solver to use.

- **Parameters**

- * `iterSolver` – The currentIterSolver to set.
 - * `tryOthers` – whether other solvers should be tried if this fails.

- *setTryOthers*

```
public final void setTryOthers( boolean tryOthers )
```

- **Description**

- Sets whether the solver shall try other solvers when it fails.

- **Parameters**

- * `tryOthers` – true if the solver shall try other solvers when it fails.

Members inherited from class `jmarkov.solvers.SteadyStateSolver` (see 6.1.10, page 250)

- `public abstract double getSteadyState() throws jmarkov.basic.exceptions.NotUnichainException`

Members inherited from class `jmarkov.solvers.Solver` (see 6.1.9, page 248)

- `public final MarkovProcess getMP()`
- `public abstract String label()`
- `protected mp`
- `public final String toString()`

6.1.7 Class MtjSolver.EnumPrecond

This is the list of preconditioner offered in MTJ.

Declaration

```
public static final class MtjSolver.EnumPrecond
extends java.lang.Enum
```

Field summary

DIA
IDEN
ILU
SSOR

Method summary

label() Returns a label with the solver name.
valueOf(String)
values()

Serializable Fields

- private java.lang.String **name**

Fields

- public static final MtjSolver.EnumPrecond **IDEN**
- public static final MtjSolver.EnumPrecond **SSOR**
- public static final MtjSolver.EnumPrecond **DIA**
- public static final MtjSolver.EnumPrecond **ILU**

Methods

- *label*
 public java.lang.String **label**()
 – **Description**
 Returns a label with the solver name.
 – **Returns** – The solver name
 – **See also**
 * **Solver.label()** (see 6.1.9, page 249)
-

- *valueOf*
public static MtjSolver.EnumPrecond valueOf(java.lang.String name)
- *values*
public static final MtjSolver.EnumPrecond[] values()

Members inherited from class java.lang.Enum

- protected final Object clone() throws CloneNotSupportedException
- public final int compareTo(Enum arg0)
- public final boolean equals(Object arg0)
- public final Class getDeclaringClass()
- public final int hashCode()
- public final String name()
- public final int ordinal()
- public String toString()
- public static Enum valueOf(Class arg0, String arg1)

6.1.8 Class MtjSolver.EnumSolver

This is the list of solvers provided by MTJ.

Declaration

```
public static final class MtjSolver.EnumSolver
extends java.lang.Enum
```

Field summary

BiCG
BiCGstab
CGS
GMRES
QMR

Method summary

getName()
valueOf(String)
values()

Serializable Fields

- private java.lang.String **name**

Fields

- public static final MtjSolver.EnumSolver **BiCG**
- public static final MtjSolver.EnumSolver **BiCGstab**
- public static final MtjSolver.EnumSolver **CGS**
- public static final MtjSolver.EnumSolver **GMRES**
- public static final MtjSolver.EnumSolver **QMR**

Methods

- *getName*
 public java.lang.String **getName()**
 – **Returns** – The name.
- *valueOf*
 public static MtjSolver.EnumSolver **valueOf(java.lang.String name)**
- *values*
 public static final MtjSolver.EnumSolver[] **values()**

Members inherited from class `java.lang.Enum`

- `protected final Object clone()` throws `CloneNotSupportedException`
- `public final int compareTo(Enum arg0)`
- `public final boolean equals(Object arg0)`
- `public final Class getDeclaringClass()`
- `public final int hashCode()`
- `public final String name()`
- `public final int ordinal()`
- `public String toString()`
- `public static Enum valueOf(Class arg0, String arg1)`

6.1.9 Class Solver

This abstract class has to be extended in order to implement solvers for Steady State and Transient probabilities. Most users do not need to implement this class since a default solver is provided.

See also

- **SteadyStateSolver** (see 6.1.10, page 250)
- **TransientSolver** (see 6.1.11, page 252)
- **JamaSolver** (see 6.1.3, page 234)

Declaration

```
public abstract class Solver
  extends java.lang.Object
  implements jmarkov.basic.JMarkovElement
```

All known subclasses

TransientSolver (see 6.1.11, page 252) , SteadyStateSolver (see 6.1.10, page 250) , MtjSolver (see 6.1.6, page 240) , MtjLogRedSolver (see 6.1.5, page 238) , JamaTransientSolver (see 6.1.4, page 236) , JamaSolver (see 6.1.3, page 234) , GeometrixSolver (see 6.1.2, page 232) , GeometricSolver (see 6.1.1, page 230)

Field summary

mp The Markovprocess being solved

Constructor summary

Solver(MarkovProcess) Build a solver for the given SimpleMarkovProcess

Method summary

getMP() Returns the Markov process currently being solved by this solver.
label() The name of this solver.
toString() Return the name of the Solver.

Fields

- protected jmarkov.MarkovProcess **mp**
 - The Markovprocess being solved

Constructors

- *Solver*

```
public Solver( jmarkov.MarkovProcess mp )
```

 - **Description**
Build a solver for the given SimpleMarkovProcess

- **Parameters**

- * `mp` – Markov Process to be solved.

Methods

- *getMP*

- `public final jmarkov.MarkovProcess getMP()`

- **Description**

- Returns the Markov process currently being solved by this solver.

- **Returns** – the current Markov Process associated with this solver.

- *label*

- `public abstract java.lang.String label()`

- **Description**

- The name of this solver. This should be implemented by the extending classes..

- *toString*

- `public final java.lang.String toString()`

- **Description**

- Return the name of the Solver.

- **See also**

- * `Solver.label()` (see 6.1.9, page 249)

6.1.10 Class SteadyStateSolver

An abstract class for steady state solver. Any solver for steady state should extend this class.

Declaration

```
public abstract class SteadyStateSolver
extends jmarkov.solvers.Solver (see 6.1.9, page 248)
```

All known subclasses

MtjSolver (see 6.1.6, page 240) , JamaSolver (see 6.1.3, page 234)

Constructor summary

SteadyStateSolver(MarkovProcess) Builds a Steady State Solver with the given SimpleMarkovProcess.

Method summary

getSteadyState() This process should be extended in order to compute the steady State probabilities of the MarkovChain.

Constructors

- *SteadyStateSolver*
public SteadyStateSolver(jmarkov.MarkovProcess mp)
 - **Description**
Builds a Steady State Solver with the given SimpleMarkovProcess.
 - **Parameters**
 - * mp – The Markov Process for which the steady state probabilities are sought.

Methods

- *getSteadyState*
public abstract double[] getSteadyState() throws jmarkov.basic.exceptions.NotUnichainException
 - **Description**
This process should be extended in order to compute the steady State probabilities of the MarkovChain. The user can get information of the SimpleMarkovProcess associated with this solver through the methods `getRates()`, `getGenerator`, and `getRate(State,State)`
 - **Returns** – an array with the Steady state probabilities for the given problem.
 - **Throws**
 - * `jmarkov.basic.exceptions.NotUnichainException` –
 - **See also**
 - * `jmarkov.MarkovProcess.getGenerator()` (see 1.1.5, page 38)
 - * `jmarkov.MarkovProcess.getRates()` (see 1.1.5, page 41)
 - * `jmarkov.MarkovProcess.getRate(State,State)` (see 1.1.5, page 41)

Members inherited from class `jmarkov.solvers.Solver` (see 6.1.9, page 248)

- `public final MarkovProcess getMP()`
- `public abstract String label()`
- `protected mp`
- `public final String toString()`

6.1.11 Class TransientSolver

An abstract class for Transient solvers. At least the method `getTransientProbs(double time, State i0)` has to be implemented. All others call this method, but the user can provide more efficient implementations.

See also

- `TransientSolver.getTransientProbs(double, State)` (see 6.1.11, page 253)

Declaration

```
public abstract class TransientSolver
extends jmarkov.solvers.Solver (see 6.1.9, page 248)
```

All known subclasses

JamaTransientSolver (see 6.1.4, page 236)

Constructor summary

TransientSolver(MarkovProcess) Build a solver with the associated Markov Process.

Method summary

getTransientProbs(double[], State) Computes the steady state probabilities at this given times, assuming the Markov Chain starts in the given state `i0`.
getTransientProbs(double, State) Computes the steady state probabilities at this given time, assuming the Markov Chain starts in the given state `i0`.
getTransientProbs(int, double, State) Computes the steady state probabilities at times `delta`, `2delta`, `3delta`,..., assuming the Markov Chain starts in the given state `i0`.

Constructors

- *TransientSolver*

```
public TransientSolver( jmarkov.MarkovProcess mp )
```

 - **Description**
Build a solver with the associated Markov Process.
 - **Parameters**
 - * `mp` –

Methods

- *getTransientProbs*

```
public double[] [] getTransientProbs( double[] times, jmarkov.basic.State i0 )
```

– **Description**

Computes the steady state probabilities at this given times, assuming the Markov Chain starts in the given state i0.

– **Parameters**

- * **times** – An array with the times at which the probabilities are to be evaluated.
- * **i0** – The initial state (at time t=0).

– **Returns** – probabilities array for each state. The (i,j) entry on the returned state represents the steady state probability for state i at time times[j].

• *getTransientProbs*

```
public abstract double[] getTransientProbs( double time, jmarkov.basic.State i0 )
```

– **Description**

Computes the steady state probabilities at this given time, assuming the Markov Chain starts in the given state i0.

– **Parameters**

- * **time** –
- * **i0** – Initial State.

– **Returns** – probabilities array

• *getTransientProbs*

```
public double[][] getTransientProbs( int NumberPoints, double delta, jmarkov.basic.State i0 )
```

– **Description**

Computes the steady state probabilities at times delta, 2delta, 3delta,..., assuming the Markov Chain starts in the given state i0.

– **Parameters**

- * **NumberPoints** –
- * **delta** – the time gap between measurements.
- * **i0** – Initial state.

– **Returns** – probabilities array for each state. The (i,j) entry on the returned state represents the steady state probability for state i at time j * delta.

Members inherited from class jmarkov.solvers.Solver (see 6.1.9, page 248)

- public final MarkovProcess getMP()
- public abstract String label()
- protected mp
- public final String toString()

Chapter 7

Package jphase

<i>Package Contents</i>	<i>Page</i>
<hr/>	
Interfaces	
ContPhaseVar	256
DiscPhaseVar	261
PhaseVar	265
<i>This interface defines the behaviour that any Phase-Type distribution should have</i>	
Classes	
AbstractContPhaseVar	270
AbstractDiscPhaseVar	276
DenseContPhaseVar	282
<i>This class allows the creation and manipulation of Continuous Phase-type distributions represented by dense matrices.</i>	
DenseDiscPhaseVar	287
<i>This class allows the creation and manipulation of Discrete Phase-type distributions represented by dense matrices.</i>	
ErlangCoxianVar	291
<i>Phase-Type representation of an ErlangCoxian distribution as defined by Osogami and Harchol in "Closed form solutions for mapping general distributions to quasi-minimal PH distributions", 2005.</i>	
HyperErlangVar	296
MarkovMatrix	301
MatrixUtils	307
PhaseVarSet	323
Poly	328
<i>This class represents a polynomial.</i>	
SparseContPhaseVar	330

SparseDiscPhaseVar	333
SuperErlang	336
Term	342
<i>This class describes a basic term in a super-Erlang function It has the form: $\alpha \lambda^n * t^{(n-1)} * \exp(-m*\lambda) / (n-1)!$ Or $\alpha * R(n,\lambda)$ where $R(n,\lambda)$ is an Erlang pdf.</i>	
Utils	348

This package provides capabilities for modeling Phase type distributions. Phase type distributions are a very general family of distribution, which can be shown to be dense. That means that any distribution for a positive random variable can be approximated with a Phase type distribution. Phase type distributions are also useful because of numerous closure properties. For overviews, tutorials, examples, guides, and tool documentation, please see:
Copa Group WEB page. (at <http://copa.uniandes.edu.co>)

7.1 Interfaces

7.1.1 Interface ContPhaseVar

Declaration

```
public interface ContPhaseVar
implements PhaseVar
```

All known subclasses

SparseContPhaseVar (see 7.2.11, page 330) , HyperErlangVar (see 7.2.6, page 296) , ErlangCoxianVar (see 7.2.5, page 291) , DenseContPhaseVar (see 7.2.3, page 282) , AbstractContPhaseVar (see 7.2.1, page 270)

All classes known to implement interface

AbstractContPhaseVar (see 7.2.1, page 270)

Method summary

copy() Creates a deep copy of the original Phase-Type Variable
eqResidualTime() Computes the Equilibrium Residual Distribution
max(ContPhaseVar) Returns the maximum between the variable B and the original: $\text{res} = \max(A, B)$
max(ContPhaseVar, ContPhaseVar) Returns the maximum between the variable B and the original: $\text{res} = \max(A, B)$
min(ContPhaseVar) Returns the minimum between the variable B and the original: $\text{res} = \min(A, B)$
min(ContPhaseVar, ContPhaseVar) Returns the minimum between the variable B and the original: $\text{res} = \min(A, B)$
mix(double, ContPhaseVar) Computes the distribution of the mix: $\text{res} = A * p + B * (1 - p)$
mix(double, ContPhaseVar, ContPhaseVar) Computes the distribution of the mix: $\text{res} = A * p + B * (1 - p)$
newVar(int) Creates a new variable of the same class of the original Continuous Phase-Type Variable
pdf(double) Evaluates the probability density function at x
pdf(int, double) Evaluates the Probability Density Function at n values of x, starting with x=0, step delta
residualTime(double) Computes the Residual Time Distribution
residualVar(double) Computes the variable $(X - a)_+$, i.e.
sum(ContPhaseVar) Computes the sum of variables: $\text{res} = A + B$
sum(ContPhaseVar, ContPhaseVar) Computes the sum of variables: $\text{res} = A + B$
sumGeom(double) Returns the sum of a geometric number of independent copies of this variable
sumPH(DiscPhaseVar) Returns the sum of a Phase number of Continuous Phase-type distributions
sumPH(DiscPhaseVar, ContPhaseVar) Returns the sum of a Phase number of Continuous Phase-type distributions
times(double) Returns a Phase continuous variable that is the original one times c

toString()

waitingQ(double) Computes the distribution of the waiting time in queue

Methods

- *copy*

ContPhaseVar **copy**()

- **Description**

Creates a deep copy of the original Phase-Type Variable

- **Returns** – A deep copy of the original Phase-Type Variable

- *eqResidualTime*

ContPhaseVar **eqResidualTime**()

- **Description**

Computes the Equilibrium Residual Distribution

- **Returns** – $F_0(x) = \text{integ}(0, t, (1 - F(t))) / E(X)$

- *max*

ContPhaseVar **max**(ContPhaseVar **B**)

- **Description**

Returns the maximum between the variable B and the original: $\text{res} = \max(A, B)$

- **Parameters**

- * **B** – Variable to compare with the original

- **Returns** – $\text{res} = \max(A, B)$

- *max*

ContPhaseVar **max**(ContPhaseVar **B**, ContPhaseVar **res**)

- **Description**

Returns the maximum between the variable B and the original: $\text{res} = \max(A, B)$

- **Parameters**

- * **B** – Variable to compare with the original

- * **res** – Variable to store the resulting distribution

- **Returns** – $\text{res} = \max(A, B)$

- *min*

ContPhaseVar **min**(ContPhaseVar **B**)

- **Description**

Returns the minimum between the variable B and the original: $\text{res} = \min(A, B)$

- **Parameters**

- * **B** – Variable to compare with the original

- **Returns** – $\text{res} = \min(A, B)$

- *min*

ContPhaseVar **min**(ContPhaseVar **B**, ContPhaseVar **res**)

– **Description**

Returns the minimum between the variable B and the original: $\text{res} = \min(A, B)$

– **Parameters**

- * B – Variable to compare with the original
- * **res** – Variable to store the resulting distribution

– **Returns** – $\text{res} = \min(A, B)$

• *mix*

`ContPhaseVar mix(double p, ContPhaseVar B)`

– **Description**

Computes the distribution of the mix: $\text{res} = A * p + B * (1 - p)$

– **Parameters**

- * B – Variable to mix with the original
- * p – Portion of this variable in the mix ($0 \leq p \leq 1$)

– **Returns** – Distribution of the mix: $\text{res} = A * p + B * (1 - p)$

• *mix*

`ContPhaseVar mix(double p, ContPhaseVar B, ContPhaseVar res)`

– **Description**

Computes the distribution of the mix: $\text{res} = A * p + B * (1 - p)$

– **Parameters**

- * B – Variable to mix with the original
- * p – Portion of this variable in the mix ($0 \leq p \leq 1$)
- * **res** – Variable to store the resulting distribution with the same number of phases of the original distribution

– **Returns** – Distribution of the mix: $\text{res} = A * p + B * (1 - p)$

• *newVar*

`ContPhaseVar newVar(int n)`

– **Description**

Creates a new variable of the same class of the original Continuous Phase-Type Variable

– **Parameters**

- * n – number of Phases of the new Variable

– **Returns** – A new variable of the same class of the original Continuous Phase-Type Variable

• *pdf*

`double pdf(double x)`

– **Description**

Evaluates the probability density function at x

– **Parameters**

- * x – Evaluation point

– **Returns** – Probability density function at x

• *pdf*

`double[] pdf(int n, double delta)`

– **Description**

Evaluates the Probability Density Function at n values of x , starting with $x=0$, step δ

– **Parameters**

- * **n** – number of evaluation points
- * **delta** – distance between evaluation points

– **Returns** – Evaluation of the probability density function at $x = 0, \delta, 2\delta, \dots, (n-1)\delta$

• *residualTime*

ContPhaseVar residualTime(double x)

– **Description**

Computes the Residual Time Distribution

– **Parameters**

- * **x** – evaluation point

– **Returns** – Distribution of $P(X - \tau = x \mid X > \tau)$

• *residualVar*

ContPhaseVar residualVar(double a)

– **Description**

Computes the variable $(X-a)^+$, i.e. the distribution takes the value of the original distribution if it is greater or equal to a . Otherwise, it is equal to null.

– **Parameters**

- * **a** – Parameter for determining loss variable

– **Returns** – Phase Variable that describes $(X-a)^+$

• *sum*

ContPhaseVar sum(ContPhaseVar B)

– **Description**

Computes the sum of variables: $res = A + B$

– **Parameters**

- * **B** – Variable to sum to the original

– **Returns** – Sum of Variables: $res = A + B$

• *sum*

ContPhaseVar sum(ContPhaseVar B, ContPhaseVar res)

– **Description**

Computes the sum of variables: $res = A + B$

– **Parameters**

- * **B** – Variable to sum to the original
- * **res** – Variable to store the result

– **Returns** – Sum of Variables: $res = A + B$

• *sumGeom*

ContPhaseVar sumGeom(double p)

– **Description**

Returns the sum of a geometric number of independent copies of this variable

– **Parameters**

* **p** – Parameter of the geometric variable

– **Returns** – Sum of a geometric number of independent copies of this variable

• *sumPH*

ContPhaseVar **sumPH**(DiscPhaseVar **B**)

– **Description**

Returns the sum of a Phase number of Continuous Phase-type distributions

– **Parameters**

* **B** – Discrete-Phase Type Distribution that determines the number of Continuous Phase-Type Distributions to sum

– **Returns** – Sum of a Phase number of Continuous Phase-type distributions

• *sumPH*

ContPhaseVar **sumPH**(DiscPhaseVar **B**, ContPhaseVar **res**)

– **Description**

Returns the sum of a Phase number of Continuous Phase-type distributions

– **Parameters**

* **B** – Discrete-Phase Type Distribution that determines the number of Continuous Phase-Type Distributions to sum

* **res** – Continuous Phase Variable to store the resulting distribution

– **Returns** – Sum of a Phase number of Continuous Phase-type distributions

• *times*

ContPhaseVar **times**(double **c**)

– **Description**

Returns a Phase continuous variable that is the original one times **c**

– **Parameters**

* **c** – Scale factor to be applied to the original Phase continuous distribution

– **Returns** – Phase continuous variable that is the original one times **c**

• *toString*

public java.lang.String **toString**()

• *waitingQ*

ContPhaseVar **waitingQ**(double **rho**)

– **Description**

Computes the distribution of the waiting time in queue

– **Parameters**

* **rho** – Server utilization

– **Returns** – Phase Variable that describes the waiting time in Queue

7.1.2 Interface DiscPhaseVar

Declaration

```
public interface DiscPhaseVar
implements PhaseVar
```

All known subclasses

SparseDiscPhaseVar (see 7.2.12, page 333) , DenseDiscPhaseVar (see 7.2.4, page 287) ,
AbstractDiscPhaseVar (see 7.2.2, page 276)

All classes known to implement interface

DenseDiscPhaseVar (see 7.2.4, page 287) , AbstractDiscPhaseVar (see 7.2.2, page 276)

Method summary

copy() Creates a deep copy of the original Phase-Type Variable
max(DiscPhaseVar) Returns the maximum between the variable B and the original:
 $\text{res} = \max(A, B)$
max(DiscPhaseVar, DiscPhaseVar) Returns the maximum between the variable
 B and the original: $\text{res} = \max(A, B)$
min(DiscPhaseVar) Returns the minimum between the variable B and the original:
 $\text{res} = \min(A, B)$
min(DiscPhaseVar, DiscPhaseVar) Returns the minimum between the variable B
 and the original: $\text{res} = \min(A, B)$
mix(double, DiscPhaseVar) Computes the distribution of the mix: $\text{res} = A \cdot p + B \cdot (1-p)$
mix(double, DiscPhaseVar, DiscPhaseVar) Computes the distribution of the
 mix: $\text{res} = A \cdot p + B \cdot (1-p)$
newVar(int) Creates a new variable of the same class of the original Discrete
 Phase-Type Variable
pmf(int) Evaluates the probability mass function at k
pmf(int, int) Evaluates the probability Mass Function at n values of x, from zero to
 n times delta
sum(DiscPhaseVar) Computes the sum of variables: $\text{res} = A + B$
sum(DiscPhaseVar, DiscPhaseVar) Computes the sum of variables: $\text{res} = A + B$
sumGeom(double) Returns the sum of a geometric number of independent copies of
 this variable
sumPH(DiscPhaseVar) Returns the sum of a Phase number of Discrete Phase-type
 distributions
sumPH(DiscPhaseVar, DiscPhaseVar) Returns the sum of a Phase number of
 Discrete Phase-type distributions
toString()

Methods

- *copy*
DiscPhaseVar copy()

– **Description**

Creates a deep copy of the original Phase-Type Variable

– **Returns** – A deep copy of the original Phase-Type Variable

• *max*

DiscPhaseVar **max**(DiscPhaseVar **B**)

– **Description**

Returns the maximum between the variable B and the original: $\text{res} = \max(A, B)$

– **Parameters**

* **B** – Variable to compare with the original

– **Returns** – $\text{res} = \max(A, B)$

• *max*

DiscPhaseVar **max**(DiscPhaseVar **B**, DiscPhaseVar **res**)

– **Description**

Returns the maximum between the variable B and the original: $\text{res} = \max(A, B)$

– **Parameters**

* **B** – Variable to compare with the original

* **res** – Variable to store the resulting distribution

– **Returns** – $\text{res} = \max(A, B)$

• *min*

DiscPhaseVar **min**(DiscPhaseVar **B**)

– **Description**

Returns the minimum between the variable B and the original: $\text{res} = \min(A, B)$

– **Parameters**

* **B** – Variable to compare with the original

– **Returns** – $\text{res} = \min(A, B)$

• *min*

DiscPhaseVar **min**(DiscPhaseVar **B**, DiscPhaseVar **res**)

– **Description**

Returns the minimum between the variable B and the original: $\text{res} = \min(A, B)$

– **Parameters**

* **B** – Variable to compare with the original

* **res** – Variable to store the resulting distribution

– **Returns** – $\text{res} = \min(A, B)$

• *mix*

DiscPhaseVar **mix**(double **p**, DiscPhaseVar **B**)

– **Description**

Computes the distribution of the mix: $\text{res} = A * p + B * (1 - p)$

– **Parameters**

* **B** – Variable to mix with the original

* **p** – Portion of this variable in the mix ($0 \leq p \leq 1$) with the same number of phases of the original distribution

- **Returns** – Distribution of the mix: $\text{res} = A * p + B * (1 - p)$
-

- *mix*

DiscPhaseVar **mix**(double **p**, DiscPhaseVar **B**, DiscPhaseVar **res**)

- **Description**

Computes the distribution of the mix: $\text{res} = A * p + B * (1 - p)$

- **Parameters**

- * **B** – Variable to mix with the original
- * **p** – Portion of this variable in the mix ($0 \leq p \leq 1$)
- * **res** – Variable to store the resulting distribution with the same number of phases of the original distribution

- **Returns** – Distribution of the mix: $\text{res} = A * p + B * (1 - p)$
-

- *newVar*

DiscPhaseVar **newVar**(int **n**)

- **Description**

Creates a new variable of the same class of the original Discrete Phase-Type Variable

- **Parameters**

- * **n** – number of Phases of the new Variable

- **Returns** – A new variable of the same class of the original Discrete Phase-Type Variable
-

- *pmf*

double **pmf**(int **k**)

- **Description**

Evaluates the probability mass function at **k**

- **Parameters**

- * **k** – Evaluation point

- **Returns** – Evaluation of the probability mass function at **k**
-

- *pmf*

double[] **pmf**(int **n**, int **delta**)

- **Description**

Evaluates the probability Mass Function at **n** values of **x**, from zero to **n** times **delta**

- **Parameters**

- * **n** – number of evaluation points
- * **delta** – distance between evaluation points

- **Returns** – Evaluation of the survival Function at $x = 0, d, 2d, \dots, (n-1)d$
-

- *sum*

DiscPhaseVar **sum**(DiscPhaseVar **B**)

- **Description**

Computes the sum of variables: $\text{res} = A + B$

- **Parameters**

- * **B** – Variable to sum to the original

- **Returns** – Sum of Variables: $\text{res} = A + B$

- *sum*

DiscPhaseVar **sum**(DiscPhaseVar **B**, DiscPhaseVar **res**)

- **Description**

Computes the sum of variables: $\text{res} = A + B$

- **Parameters**

- * **B** – Variable to sum to the original
- * **res** – Variable to store the result

- **Returns** – Sum of Variables: $\text{res} = A + B$

- *sumGeom*

DiscPhaseVar **sumGeom**(double **p**)

- **Description**

Returns the sum of a geometric number of independent copies of this variable

- **Parameters**

- * **p** – Parameter of the geometric variable

- **Returns** – Sum of a geometric number of independent copies of this variable

- *sumPH*

DiscPhaseVar **sumPH**(DiscPhaseVar **B**)

- **Description**

Returns the sum of a Phase number of Discrete Phase-type distributions

- **Parameters**

- * **B** – Discrete-Phase Type Distribution that determines the number of Discrete Phase-Type Distributions to sum

- **Returns** – Sum of a Phase number of Discrete Phase-type distributions

- *sumPH*

DiscPhaseVar **sumPH**(DiscPhaseVar **B**, DiscPhaseVar **res**)

- **Description**

Returns the sum of a Phase number of Discrete Phase-type distributions

- **Parameters**

- * **B** – Discrete-Phase Type Distribution that determines the number of Discrete Phase-Type Distributions to sum
- * **res** – Discrete Phase Variable to store the resulting distribution

- **Returns** – Sum of a Phase number of Discrete Phase-type distributions

- *toString*

public java.lang.String **toString**()

7.1.3 Interface PhaseVar

This interface defines the behaviour that any Phase-Type distribution should have

Declaration

```
public interface PhaseVar
implements jmarkov.basic.JMarkovElement
```

Version

0.1

All known subclasses

SparseDiscPhaseVar (see 7.2.12, page 333) , SparseContPhaseVar (see 7.2.11, page 330) , HyperErlangVar (see 7.2.6, page 296) , ErlangCoxianVar (see 7.2.5, page 291) , DiscPhaseVar (see 7.1.2, page 261) , DenseDiscPhaseVar (see 7.2.4, page 287) , DenseContPhaseVar (see 7.2.3, page 282) , ContPhaseVar (see 7.1.1, page 256) , AbstractDiscPhaseVar (see 7.2.2, page 276) , AbstractContPhaseVar (see 7.2.1, page 270)

All known subinterfaces

DiscPhaseVar (see 7.1.2, page 261) , ContPhaseVar (see 7.1.1, page 256)

All classes known to implement interface

HyperErlangVar (see 7.2.6, page 296) , ErlangCoxianVar (see 7.2.5, page 291)

Method summary

cdf(double) Evaluates the cumulative density function at x
cdf(int, double) Evaluates the Cumulative Density Function at n values of x, starting with x=0, step delta
copy() Creates a deep copy of the original Phase-Type Variable
CV() Computes the Coefficient of Variation of the Phase variable
expectedValue() Computes the Expected Value of the Phase variable
getMat0() Returns the exit vector from the transient states to absorption
getMat0Array() Returns the exit vector in double[] format
getMatrix() Returns the transition matrix of the Phase-Type Distribution
getMatrixArray() Returns the transition matrix in double format
getNumPhases() Returns the number of Phases of the Phase distribution
getVec0() Returns the probability mass at zero (alpha_0)
getVector() Returns the initial probability mass vector
getVectorArray() Returns the initial probability mass vector in double[] format
lossFunction1(double) Evaluates the loss function of order 1 at x
lossFunction2(double) Evaluates the loss function of order 2 at x
median() Computes the median of the distribution
moment(int) Computes the k-th Moment of the Phase variable
prob(double, double) Computes the probability that this variable takes a value between a and b
quantil(double) Computes the quantile q of the distribution, such that $F(q) = p$

setMatrix(Matrix) Rate Matrix = A
setVector(Vector) Initial Probability vector = alpha
stdDeviation() Computes the Standard deviation of the Phase variable
survival(double) Evaluates the survival function at x
survival(int, double) Evaluates the Survival Function at n values of x, starting with x=0, step delta
variance() Computes the Variance of the Phase variable

Methods

- *cdf*
double **cdf**(double x)
 - **Description**
Evaluates the cumulative density function at x
 - **Parameters**
* x – Evaluation point
 - **Returns** – Cumulative density function at x

- *cdf*
double[] **cdf**(int n, double delta)
 - **Description**
Evaluates the Cumulative Density Function at n values of x, starting with x=0, step delta
 - **Parameters**
* n – number of evaluation points
* delta – distance between evaluation points
 - **Returns** – Evaluation of the survival Function at x = 0,d,2d,...,(n-1)d

- *copy*
PhaseVar **copy**()
 - **Description**
Creates a deep copy of the original Phase-Type Variable
 - **Returns** – A deep copy of the original Phase-Type Variable

- *CV*
double **CV**()
 - **Description**
Computes the Coefficient of Variation of the Phase variable
 - **Returns** – Coefficient of Variation of the Phase variable

- *expectedValue*
double **expectedValue**()
 - **Description**
Computes the Expected Value of the Phase variable
 - **Returns** – Expected Value of the Phase variable

- *getMat0*
`no.uib.cipr.matrix.Vector getMat0()`
 - **Description**
Returns the exit vector from the transient states to absorption
 - **Returns** – Exit vector from the transient states to absorption

- *getMat0Array*
`double[] getMat0Array()`
 - **Description**
Returns the exit vector in double[] format
 - **Returns** – Exit vector from the transient states to absorption in double[] format

- *getMatrix*
`no.uib.cipr.matrix.Matrix getMatrix()`
 - **Description**
Returns the transition matrix of the Phase-Type Distribution
 - **Returns** – Transition matrix for transient states of the Phase-Type Distribution

- *getMatrixArray*
`double[][] getMatrixArray()`
 - **Description**
Returns the transition matrix in double format
 - **Returns** – Transition matrix for transient states of the Phase-Type Distribution in double[][] format

- *getNumPhases*
`int getNumPhases()`
 - **Description**
Returns the number of Phases of the Phase distribution
 - **Returns** – Number of Phases of the Phase distribution

- *getVec0*
`double getVec0()`
 - **Description**
Returns the probability mass at zero (alpha.0)
 - **Returns** – Probability mass at zero (alpha.0)

- *getVector*
`no.uib.cipr.matrix.Vector getVector()`
 - **Description**
Returns the initial probability mass vector
 - **Returns** – Initial probability mass vector

- *getVectorArray*
`double[] getVectorArray()`

– **Description**

Returns the initial probability mass vector in double[] format

– **Returns** – Initial probability mass vector in double[] format

• *lossFunction1*

double **lossFunction1**(double x)

– **Description**

Evaluates the loss function of order 1 at x

– **Parameters**

* x – Evaluation point

– **Returns** – Evaluation of the loss function of order 1

• *lossFunction2*

double **lossFunction2**(double x)

– **Description**

Evaluates the loss function of order 2 at x

– **Parameters**

* x – Evaluation point

– **Returns** – Evaluation of the loss function of order 2

• *median*

double **median**()

– **Description**

Computes the median of the distribution

– **Returns** – The median of the distribution

• *moment*

double **moment**(int k)

– **Description**

Computes the k-th Moment of the Phase variable

– **Parameters**

* k – Moment

– **Returns** – k-th Moment of the Phase variable

• *prob*

double **prob**(double a, double b)

– **Description**

Computes the probability that this variable takes a value between a and b

– **Parameters**

* a – inferior limit

* b – superior limit

– **Returns** – Probability that this variable takes a value between a and b

• *quantil*

double **quantil**(double p)

- **Description**
Computes the quantile q of the distribution, such that $F(q) = p$
- **Parameters**
* p – probability such that $F(q) = p$
- **Returns** – The quantile q of the distribution, such that $F(q) = p$

- *setMatrix*
void **setMatrix**(no.uib.cipr.matrix.Matrix **A**)
 - **Description**
Rate Matrix = A
 - **Parameters**
* A – Transition matrix for transient states of the Phase-Type Distribution

- *setVector*
void **setVector**(no.uib.cipr.matrix.Vector **alpha**)
 - **Description**
Initial Probability vector = α
 - **Parameters**
* α – Initial probability mass vector

- *stdDeviation*
double **stdDeviation**()
 - **Description**
Computes the Standard deviation of the Phase variable
 - **Returns** – Standard deviation of the Phase variable

- *survival*
double **survival**(double **x**)
 - **Description**
Evaluates the survival function at x
 - **Parameters**
* x – Evaluation point
 - **Returns** – Evaluation of the survival Function at $x = 1 - F(x) = P(X > x)$

- *survival*
double[] **survival**(int **n**, double **delta**)
 - **Description**
Evaluates the Survival Function at n values of x , starting with $x=0$, step δ
 - **Parameters**
* n – number of evaluation points
* δ – distance between evaluation points
 - **Returns** – Evaluation of the survival Function at $x = 0, \delta, 2\delta, \dots, (n-1)\delta$

- *variance*
double **variance**()
 - **Description**
Computes the Variance of the Phase variable
 - **Returns** – Variance of the Phase variable

7.2 Classes

7.2.1 *Class* AbstractContPhaseVar

Declaration

```
public abstract class AbstractContPhaseVar
  extends java.lang.Object
  implements ContPhaseVar
```

Version

0.1 Abstract class that defines the behaviour of Continuous Phase-Type Distributions

All known subclasses

SparseContPhaseVar (see 7.2.11, page 330) , HyperErlangVar (see 7.2.6, page 296) , ErlangCoxianVar (see 7.2.5, page 291) , DenseContPhaseVar (see 7.2.3, page 282)

Constructor summary

AbstractContPhaseVar()

Method summary

cdf(double)
cdf(int, double)
CV()
description()
eqResidualTime()
expectedValue()
getMat0()
getMat0Array()
getMatrixArray()
getNumPhases()
getVec0()
getVectorArray()
label()
lossFunction1(double)
lossFunction2(double)
max(ContPhaseVar)
max(ContPhaseVar, ContPhaseVar)
median()
min(ContPhaseVar)
min(ContPhaseVar, ContPhaseVar)
mix(double, ContPhaseVar)
mix(double, ContPhaseVar, ContPhaseVar)
moment(int)
pdf(double)
pdf(int, double)
prob(double, double)

```

quantil(double)
residualTime(double)
residualVar(double)
stdDeviation()
sum(ContPhaseVar)
sum(ContPhaseVar, ContPhaseVar)
sumGeom(double)
sumPH(DiscPhaseVar)
sumPH(DiscPhaseVar, ContPhaseVar)
survival(double)
survival(int, double)
times(double)
toString()
variance()
waitingQ(double)

```

Constructors

- *AbstractContPhaseVar*

```
public AbstractContPhaseVar( )
```

Methods

- *cdf*

```
public double cdf( double x )
```

 - See also

```
* PhaseVar.cdf(double) (see 7.1.3, page 266)
```
- *cdf*

```
public double[] cdf( int n, double delta )
```

 - See also

```
* PhaseVar.cdf(int,double) (see 7.1.3, page 266)
```
- *CV*

```
public double CV( )
```

 - See also

```
* PhaseVar.CV() (see 7.1.3, page 266)
```
- *description*

```
public java.lang.String description( )
```
- *eqResidualTime*

```
public ContPhaseVar eqResidualTime( )
```

 - See also

```
* ContPhaseVar.eqResidualTime() (see 7.1.1, page 257)
```


- *expectedValue*
`public double expectedValue()`
 - See also
 - * `PhaseVar.expectedValue()` (see 7.1.3, page 266)

- *getMat0*
`public no.uib.cipr.matrix.Vector getMat0()`
 - See also
 - * `PhaseVar.getMat0()` (see 7.1.3, page 267)

- *getMat0Array*
`public double[] getMat0Array()`
 - See also
 - * `PhaseVar.getMat0Array()` (see 7.1.3, page 267)

- *getMatrixArray*
`public double[][] getMatrixArray()`
 - See also
 - * `PhaseVar.getMatrixArray()` (see 7.1.3, page 267)

- *getNumPhases*
`public int getNumPhases()`
 - See also
 - * `PhaseVar.getNumPhases()` (see 7.1.3, page 267)

- *getVec0*
`public double getVec0()`
 - See also
 - * `PhaseVar.getVec0()` (see 7.1.3, page 267)

- *getVectorArray*
`public double[] getVectorArray()`
 - See also
 - * `PhaseVar.getVectorArray()` (see 7.1.3, page 267)

- *label*
`public java.lang.String label()`

- *lossFunction1*
`public double lossFunction1(double x)`
 - See also
 - * `PhaseVar.lossFunction1(double)` (see 7.1.3, page 268)

- *lossFunction2*

```
public double lossFunction2( double x )
```

– See also

* PhaseVar.lossFunction2(double) (see 7.1.3, page 268)

- *max*

```
public ContPhaseVar max( ContPhaseVar v2 )
```

– See also

* ContPhaseVar.max(ContPhaseVar) (see 7.1.1, page 257)

- *max*

```
public ContPhaseVar max( ContPhaseVar v2, ContPhaseVar res )
```

– See also

* ContPhaseVar.max(ContPhaseVar,ContPhaseVar) (see 7.1.1, page 257)

- *median*

```
public double median( )
```

– See also

* PhaseVar.median() (see 7.1.3, page 268)

- *min*

```
public ContPhaseVar min( ContPhaseVar v2 )
```

– See also

* ContPhaseVar.min(ContPhaseVar) (see 7.1.1, page 257)

- *min*

```
public ContPhaseVar min( ContPhaseVar v2, ContPhaseVar res )
```

– See also

* ContPhaseVar.min(ContPhaseVar,ContPhaseVar) (see 7.1.1, page 257)

- *mix*

```
public ContPhaseVar mix( double p, ContPhaseVar v2 )
```

– See also

* ContPhaseVar.mix(double,ContPhaseVar) (see 7.1.1, page 258)

- *mix*

```
public ContPhaseVar mix( double p, ContPhaseVar v2, ContPhaseVar res )
```

– See also

* ContPhaseVar.mix(double,ContPhaseVar,ContPhaseVar) (see 7.1.1, page 258)

- *moment*

```
public double moment( int k )
```

– See also

- * PhaseVar.moment(int) (see 7.1.3, page 268)

- *pdf*
public double pdf(double x)
 - See also
 - * ContPhaseVar.pdf(double) (see 7.1.1, page 258)

- *pdf*
public double[] pdf(int n, double delta)
 - See also
 - * ContPhaseVar.pdf(int,double) (see 7.1.1, page 258)

- *prob*
public double prob(double a, double b)
 - See also
 - * PhaseVar.prob(double,double) (see 7.1.3, page 268)

- *quantil*
public double quantil(double p)
 - See also
 - * PhaseVar.quantil(double) (see 7.1.3, page 268)

- *residualTime*
public ContPhaseVar residualTime(double x)
 - See also
 - * ContPhaseVar.residualTime(double) (see 7.1.1, page 259)

- *residualVar*
public ContPhaseVar residualVar(double a)
 - See also
 - * ContPhaseVar.residualVar(double) (see 7.1.1, page 259)

- *stdDeviation*
public double stdDeviation()
 - See also
 - * PhaseVar.stdDeviation() (see 7.1.3, page 269)

- *sum*
public ContPhaseVar sum(ContPhaseVar v2)
 - See also
 - * ContPhaseVar.sum(ContPhaseVar) (see 7.1.1, page 259)

- *sum*
public ContPhaseVar sum(ContPhaseVar v2, ContPhaseVar res)

- See also
 - * ContPhaseVar.sum(ContPhaseVar,ContPhaseVar) (see 7.1.1, page 259)

- *sumGeom*

```
public ContPhaseVar sumGeom( double p )
```

 - See also
 - * ContPhaseVar.sumGeom(double) (see 7.1.1, page 259)

- *sumPH*

```
public ContPhaseVar sumPH( DiscPhaseVar v2 )
```

 - See also
 - * ContPhaseVar.sumPH(DiscPhaseVar) (see 7.1.1, page 260)

- *sumPH*

```
public ContPhaseVar sumPH( DiscPhaseVar B, ContPhaseVar res )
```

 - See also
 - * ContPhaseVar.sumPH(DiscPhaseVar,ContPhaseVar) (see 7.1.1, page 260)

- *survival*

```
public double survival( double x )
```

 - See also
 - * PhaseVar.survival(double) (see 7.1.3, page 269)

- *survival*

```
public double[] survival( int n, double delta )
```

 - See also
 - * PhaseVar.survival(int,double) (see 7.1.3, page 269)

- *times*

```
public ContPhaseVar times( double c )
```

 - See also
 - * ContPhaseVar.times(double) (see 7.1.1, page 260)

- *toString*

```
public final java.lang.String toString( )
```

 - See also
 - * ContPhaseVar.toString() (see 7.1.1, page 260)

- *variance*

```
public double variance( )
```

 - See also
 - * PhaseVar.variance() (see 7.1.3, page 269)

- *waitingQ*

```
public ContPhaseVar waitingQ( double rho )
```

 - See also
 - * ContPhaseVar.waitingQ(double) (see 7.1.1, page 260)

7.2.2 Class AbstractDiscPhaseVar

Declaration

```
public abstract class AbstractDiscPhaseVar
  extends java.lang.Object
  implements DiscPhaseVar
```

Version

0.1 Abstract class that defines the behaviour of the Discrete Phase-Type distributions

All known subclasses

SparseDiscPhaseVar (see 7.2.12, page 333) , DenseDiscPhaseVar (see 7.2.4, page 287)

Constructor summary

AbstractDiscPhaseVar()

Method summary

cdf(double)
cdf(int, double)
CV()
description()
expectedValue()
getMat0()
getMat0Array()
getMatrixArray()
getNumPhases()
getVec0()
getVectorArray()
label()
lossFunction1(double)
lossFunction2(double)
max(DiscPhaseVar)
max(DiscPhaseVar, DiscPhaseVar)
median()
min(DiscPhaseVar)
min(DiscPhaseVar, DiscPhaseVar)
mix(double, DiscPhaseVar)
mix(double, DiscPhaseVar, DiscPhaseVar)
moment(int)
pmf(int)
pmf(int, int)
prob(double, double)
quantil(double)
stdDeviation()
sum(DiscPhaseVar)
sum(DiscPhaseVar, DiscPhaseVar)

```

sumGeom(double)
sumPH(DiscPhaseVar)
sumPH(DiscPhaseVar, DiscPhaseVar)
survival(double)
survival(int, double)
toString()
variance()

```

Constructors

- *AbstractDiscPhaseVar*

```
public AbstractDiscPhaseVar( )
```

Methods

- *cdf*

```
public double cdf( double x )
```

 - See also

```
* PhaseVar.cdf(double) (see 7.1.3, page 266)
```
- *cdf*

```
public double[] cdf( int n, double delta )
```

 - See also

```
* PhaseVar.cdf(int,double) (see 7.1.3, page 266)
```
- *CV*

```
public double CV( )
```

 - See also

```
* PhaseVar.CV() (see 7.1.3, page 266)
```
- *description*

```
public java.lang.String description( )
```
- *expectedValue*

```
public double expectedValue( )
```

 - See also

```
* PhaseVar.expectedValue() (see 7.1.3, page 266)
```
- *getMat0*

```
public no.uib.cipr.matrix.Vector getMat0( )
```

 - See also

```
* PhaseVar.getMat0() (see 7.1.3, page 267)
```
- *getMat0Array*

```
public double[] getMat0Array( )
```

- See also
 - * `PhaseVar.getMat0Array()` (see 7.1.3, page 267)

- *getMatrixArray*
`public double[][] getMatrixArray()`
 - See also
 - * `PhaseVar.getMatrixArray()` (see 7.1.3, page 267)

- *getNumPhases*
`public int getNumPhases()`
 - See also
 - * `PhaseVar.getNumPhases()` (see 7.1.3, page 267)

- *getVec0*
`public double getVec0()`
 - See also
 - * `PhaseVar.getVec0()` (see 7.1.3, page 267)

- *getVectorArray*
`public double[] getVectorArray()`
 - See also
 - * `PhaseVar.getVectorArray()` (see 7.1.3, page 267)

- *label*
`public java.lang.String label()`

- *lossFunction1*
`public double lossFunction1(double x)`
 - See also
 - * `PhaseVar.lossFunction1(double)` (see 7.1.3, page 268)

- *lossFunction2*
`public double lossFunction2(double x)`
 - See also
 - * `PhaseVar.lossFunction2(double)` (see 7.1.3, page 268)

- *max*
`public DiscPhaseVar max(DiscPhaseVar v2)`
 - See also
 - * `DiscPhaseVar.max(DiscPhaseVar)` (see 7.1.2, page 262)

- *max*
`public DiscPhaseVar max(DiscPhaseVar v2, DiscPhaseVar res)`
 - See also

* DiscPhaseVar.max(DiscPhaseVar,DiscPhaseVar) (see 7.1.2, page 262)

- *median*

public double median()

– See also

* PhaseVar.median() (see 7.1.3, page 268)

- *min*

public DiscPhaseVar min(DiscPhaseVar v2)

– See also

* DiscPhaseVar.min(DiscPhaseVar) (see 7.1.2, page 262)

- *min*

public DiscPhaseVar min(DiscPhaseVar v2, DiscPhaseVar res)

– See also

* DiscPhaseVar.min(DiscPhaseVar,DiscPhaseVar) (see 7.1.2, page 262)

- *mix*

public DiscPhaseVar mix(double p, DiscPhaseVar v2)

– See also

* DiscPhaseVar.mix(double,DiscPhaseVar) (see 7.1.2, page 262)

- *mix*

public DiscPhaseVar mix(double p, DiscPhaseVar v2, DiscPhaseVar res)

– See also

* DiscPhaseVar.mix(double,DiscPhaseVar,DiscPhaseVar) (see 7.1.2, page 263)

- *moment*

public double moment(int k)

– See also

* PhaseVar.moment(int) (see 7.1.3, page 268)

- *pmf*

public double pmf(int k)

– See also

* DiscPhaseVar.pmf(int) (see 7.1.2, page 263)

- *pmf*

public double[] pmf(int n, int delta)

– See also

* DiscPhaseVar.pmf(int,int) (see 7.1.2, page 263)

- *prob*

public double prob(double a, double b)

- See also
 - * PhaseVar.prob(double,double) (see 7.1.3, page 268)

- *quantil*

```
public double quantil( double p )
```

 - See also
 - * PhaseVar.quantil(double) (see 7.1.3, page 268)

- *stdDeviation*

```
public double stdDeviation( )
```

 - See also
 - * PhaseVar.stdDeviation() (see 7.1.3, page 269)

- *sum*

```
public DiscPhaseVar sum( DiscPhaseVar v2 )
```

 - See also
 - * DiscPhaseVar.sum(DiscPhaseVar) (see 7.1.2, page 263)

- *sum*

```
public DiscPhaseVar sum( DiscPhaseVar v2, DiscPhaseVar res )
```

 - See also
 - * DiscPhaseVar.sum(DiscPhaseVar,DiscPhaseVar) (see 7.1.2, page 264)

- *sumGeom*

```
public DiscPhaseVar sumGeom( double p )
```

 - See also
 - * DiscPhaseVar.sumGeom(double) (see 7.1.2, page 264)

- *sumPH*

```
public DiscPhaseVar sumPH( DiscPhaseVar v2 )
```

 - See also
 - * DiscPhaseVar.sumPH(DiscPhaseVar) (see 7.1.2, page 264)

- *sumPH*

```
public DiscPhaseVar sumPH( DiscPhaseVar B, DiscPhaseVar res )
```

 - See also
 - * DiscPhaseVar.sumPH(DiscPhaseVar,DiscPhaseVar) (see 7.1.2, page 264)

- *survival*

```
public double survival( double x )
```

 - See also
 - * PhaseVar.survival(double) (see 7.1.3, page 269)

- *survival*
public double[] survival(int n, double delta)

– See also
 * PhaseVar.survival(int,double) (see 7.1.3, page 269)

- *toString*
public final java.lang.String toString()

– See also
 * DiscPhaseVar.toString() (see 7.1.2, page 264)

- *variance*
public double variance()

– See also
 * PhaseVar.variance() (see 7.1.3, page 269)

7.2.3 Class DenseContPhaseVar

This class allows the creation and manipulation of Continuous Phase-type distributions represented by dense matrices.

Declaration

```
public class DenseContPhaseVar
  extends jphase.AbstractContPhaseVar (see 7.2.1, page 270)
```

Version

0.1 This class allows the creation and manipulation of Continuous Phase-type distributions represented by dense matrices.

Field summary

A Rate Matrix
alpha Initial Probability distribution vector

Constructor summary

DenseContPhaseVar() Constructs an empty continuous Phase-type Distribution with dense representation
DenseContPhaseVar(DenseVector, DenseMatrix) Constructs a continuous Phase-type Distribution with dense representation
DenseContPhaseVar(double[], double[][]) Construcs a continuous Phase-type Distribution with dense representation
DenseContPhaseVar(int) Constructs an empty Continuous Phase-type Distribution of size n with dense representation
DenseContPhaseVar(Vector, Matrix) Constructs a continuous Phase-type Distribution with dense representation

Method summary

copy()
Coxian(int, double[], double[]) Construcs a Phase-Type representation of a Coxian distribution with n phases
Erlang(double, int) Constructs a Phase-Type representation of an Erlang distribution with rate lambda and n exponential phases
ErlangCoxian(int, double, double, double, double, double) Construcs a Phase-Type representation of an ErlangCoxian distribution as defined by Osogami and Harchol in "Closed form solutions for mapping general distributions to quasi-minimal PH distributions", 2005.
expo(double) Constructs a Phase-Type representation of an Exponential distribution with rate lambda
getMatrix()
getVector()
HyperErlang(HyperErlangVar) Construcs a Phase-Type representation of a Hyper-Erlang distribution from a Dense representation of the same distribution

HyperErlang(int, double[], int[], double[]) Constructs a Phase-Type representation of a Hyper-Erlang distribution with k erlang branches, its k rates and n number of phases per branch

HyperExpo(double[], double[]) Constructs a Phase Distribution that represents a HyperExponential distribution with the especified parameters

newVar(int)

setMatrix(Matrix)

setVector(Vector)

Fields

- protected no.uib.cipr.matrix.DenseMatrix **A**
 - Rate Matrix
- protected no.uib.cipr.matrix.DenseVector **alpha**
 - Initial Probability distribution vector

Constructors

- *DenseContPhaseVar*

```
public DenseContPhaseVar( )
```

 - **Description**

Constructs an empty continuous Phase-type Distribution with dense representation
- *DenseContPhaseVar*

```
public DenseContPhaseVar( no.uib.cipr.matrix.DenseVector alpha,
no.uib.cipr.matrix.DenseMatrix A )
```

 - **Description**

Constructs a continuous Phase-type Distribution with dense representation
 - **Parameters**
 - * **A** – rate matrix
 - * **alpha** – initial probability distribution vector
- *DenseContPhaseVar*

```
public DenseContPhaseVar( double[] alpha, double[][] A )
```

 - **Description**

Constructs a continuous Phase-type Distribution with dense representation
 - **Parameters**
 - * **A** – rate matrix
 - * **alpha** – initial probability distribution vector
- *DenseContPhaseVar*

```
public DenseContPhaseVar( int n )
```

 - **Description**

Constructs an empty Continuous Phase-type Distribution of size n with dense representation

– **Parameters**

* **n** – size of the Continuous Phase-type Distribution

• *DenseContPhaseVar*

```
public DenseContPhaseVar( no.uib.cipr.matrix.Vector alpha,
no.uib.cipr.matrix.Matrix A )
```

– **Description**

Constructs a continuous Phase-type Distribution with dense representation

– **Parameters**

* **A** – rate matrix

* **alpha** – initial probability distribution vector

Methods

• *copy*

```
public ContPhaseVar copy( )
```

• *Coxian*

```
public static DenseContPhaseVar Coxian( int n, double[] lambdas, double[]
probs )
```

– **Description**

Constructs a Phase-Type representation of a Coxian distribution with **n** phases

– **Parameters**

* **n** – number of phases

* **lambdas** – exponential rates of each phase

* **probs** – probability of going to the next phase (no absorption) in each phase except the last one.

– **Returns** – Dense Continuous Phase-Type Distribution

• *Erlang*

```
public static DenseContPhaseVar Erlang( double lambda, int n )
```

– **Description**

Constructs a Phase-Type representation of an Erlang distribution with rate **lambda** and **n** exponential phases

– **Parameters**

* **lambda** – exponential rate in each phase

* **n** – number of exponential phases

– **Returns** – Dense Continuous Phase-Type Distribution

• *ErlangCoxian*

```
public static DenseContPhaseVar ErlangCoxian( int n, double p, double
lambdaY, double lambdaX1, double lambdaX2, double px )
```

– **Description**

Constructs a Phase-Type representation of an ErlangCoxian distribution as defined by Osogami and Harchol in "Closed form solutions for mapping general distributions to quasi-minimal PH distributions", 2005.

– **Parameters**

- * **n** – total number of phases (Erlang degree: n-2)
- * **p** – probability of having a positive elapse time in the distribution. 1-p: mass probability at zero
- * **lambdaY** – rate of the Erlang distribution
- * **lambdaX1** – rate of the first stage of the Coxian distribution
- * **lambdaX2** – rate of the second stage of the Coxian distribution
- * **px** – probability of going from the first to the second stage in the Coxian distribution. 1-p: probability of absorption at the first stage of the Coxian distribution

– **Returns** – Dense Continuous Phase-Type Distribution

- *expo*

`public static DenseContPhaseVar expo(double lambda)`

– **Description**

Constructs a Phase-Type representation of an Exponential distribution with rate lambda

– **Parameters**

- * **lambda** – exponential distribution rate

– **Returns** – Dense Continuous Phase-Type Distribution

- *getMatrix*

`public no.uib.cipr.matrix.Matrix getMatrix()`

- *getVector*

`public no.uib.cipr.matrix.Vector getVector()`

- *HyperErlang*

`public static DenseContPhaseVar HyperErlang(HyperErlangVar var)`

– **Description**

Constructs a Phase-Type representation of a Hyper-Erlang distribution from a Dense representation of the same distribution

– **Parameters**

- * **var** – HyperErlang variable from which the Dense Continuous variable must be constructed

– **Returns** – Dense Continuous Phase-Type Distribution

- *HyperErlang*

`public static DenseContPhaseVar HyperErlang(int k, double[] lambdas, int[] n, double[] probs)`

– **Description**

Constructs a Phase-Type representation of a Hyper-Erlang distribution with k erlang branches, its k rates and n number of phases per branch

– **Parameters**

- * **k** – number of Erlang branches
- * **lambdas** – exponential rate of each phase in each branch
- * **n** – number of exponential phases in each branch
- * **probs** – probability distribution of taking each one of the branches

– **Returns** – Dense Continuous Phase-Type Distribution

- *HyperExpo*

```
public static DenseContPhaseVar HyperExpo( double[] lambdas, double[]
probs )
```

- **Description**

Constructs a Phase Distribution that represents a HyperExponential distribution with the specified parameters

- **Parameters**

- * **lambdas** – each one of the exponential rates
- * **probs** – initial probability vector

- **Returns** – Dense Continuous Phase-Type Distribution

- *newVar*

```
public ContPhaseVar newVar( int n )
```

- *setMatrix*

```
public void setMatrix( no.uib.cipr.matrix.Matrix A )
```

- *setVector*

```
public void setVector( no.uib.cipr.matrix.Vector alpha )
```

Members inherited from class `jphase.AbstractContPhaseVar` (see 7.2.1, page 270)

- `public double cdf(double x)`
- `public double cdf(int n, double delta)`
- `public double CV()`
- `public String description()`
- `public ContPhaseVar eqResidualTime()`
- `public double expectedValue()`
- `public Vector getMat0()`
- `public double getMat0Array()`
- `public double getMatrixArray()`
- `public int getNumPhases()`
- `public double getVec0()`
- `public double getVectorArray()`
- `public String label()`
- `public double lossFunction1(double x)`
- `public double lossFunction2(double x)`
- `public ContPhaseVar max(ContPhaseVar v2)`
- `public ContPhaseVar max(ContPhaseVar v2, ContPhaseVar res)`
- `public double median()`
- `public ContPhaseVar min(ContPhaseVar v2)`
- `public ContPhaseVar min(ContPhaseVar v2, ContPhaseVar res)`
- `public ContPhaseVar mix(double p, ContPhaseVar v2)`
- `public ContPhaseVar mix(double p, ContPhaseVar v2, ContPhaseVar res)`
- `public double moment(int k)`
- `public double pdf(double x)`
- `public double pdf(int n, double delta)`
- `public double prob(double a, double b)`
- `public double quantil(double p)`
- `public ContPhaseVar residualTime(double x)`
- `public ContPhaseVar residualVar(double a)`
- `public double stdDeviation()`
- `public ContPhaseVar sum(ContPhaseVar v2)`
- `public ContPhaseVar sum(ContPhaseVar v2, ContPhaseVar res)`
- `public ContPhaseVar sumGeom(double p)`
- `public ContPhaseVar sumPH(DiscPhaseVar v2)`
- `public ContPhaseVar sumPH(DiscPhaseVar B, ContPhaseVar res)`
- `public double survival(double x)`
- `public double survival(int n, double delta)`
- `public ContPhaseVar times(double c)`
- `public final String toString()`
- `public double variance()`
- `public ContPhaseVar waitingQ(double rho)`

7.2.4 Class DenseDiscPhaseVar

This class allows the creation and manipulation of Discrete Phase-type distributions represented by dense matrices.

Declaration

```
public class DenseDiscPhaseVar
  extends jphase.AbstractDiscPhaseVar (see 7.2.2, page 276)
  implements DiscPhaseVar
```

Version

0.1 This class allows the creation and manipulation of Discrete Phase-type distributions represented by dense matrices.

Field summary

A Probability Transition Matrix
alpha Initial Probability distribution vector

Constructor summary

DenseDiscPhaseVar() Constructs an empty Discrete Phase-type Distribution with dense representation
DenseDiscPhaseVar(DenseVector, DenseMatrix) Constructs a Discrete Phase-type Distribution with dense representation
DenseDiscPhaseVar(double[], double[][]) Constructs a Discrete Phase-type Distribution with dense representation
DenseDiscPhaseVar(int) Constructs an empty Discrete Phase-type Distribution of size n with dense representation
DenseDiscPhaseVar(Vector, Matrix) Constructs a Discrete Phase-type Distribution with dense representation

Method summary

copy()
Geom(double) Discrete Phase distribution that represents a geometric distribution with probability of success p
getMatrix()
getVector()
NegativeBinomial(double, int) Discrete Phase Distribution that represents a Negative Binomial distribution with parameters p and r
newVar(int)
setMatrix(Matrix)
setVector(Vector)

Fields

- protected no.uib.cipr.matrix.DenseMatrix **A**

- Probability Transition Matrix
- protected no.uib.cipr.matrix.DenseVector **alpha**
 - Initial Probability distribution vector

Constructors

- *DenseDiscPhaseVar*
 public **DenseDiscPhaseVar**()
 – **Description**
 Constructs an empty Discrete Phase-type Distribution with dense representation

- *DenseDiscPhaseVar*
 public **DenseDiscPhaseVar**(no.uib.cipr.matrix.DenseVector **alpha**,
 no.uib.cipr.matrix.DenseMatrix **A**)
 – **Description**
 Constructs a Discrete Phase-type Distribution with dense representation
 – **Parameters**
 - * **A** – transition probability matrix
 - * **alpha** – initial probability distribution vector

- *DenseDiscPhaseVar*
 public **DenseDiscPhaseVar**(double[] **alpha**, double[][] **A**)
 – **Description**
 Constructs a Discrete Phase-type Distribution with dense representation
 – **Parameters**
 - * **A** – transition probability matrix
 - * **alpha** – initial probability distribution vector

- *DenseDiscPhaseVar*
 public **DenseDiscPhaseVar**(int **n**)
 – **Description**
 Constructs an empty Discrete Phase-type Distribution of size n with dense representation
 – **Parameters**
 - * **n** – size of the Discrete Phase-type Distribution

- *DenseDiscPhaseVar*
 public **DenseDiscPhaseVar**(no.uib.cipr.matrix.Vector **alpha**,
 no.uib.cipr.matrix.Matrix **A**)
 – **Description**
 Constructs a Discrete Phase-type Distribution with dense representation
 – **Parameters**
 - * **A** – transition probability matrix
 - * **alpha** – initial probability distribution vector

Methods

- *copy*
`public DiscPhaseVar copy()`
 - **See also**
`* ContPhaseVar.copy()` (see 7.1.1, page 257)

- *Geom*
`public static DenseDiscPhaseVar Geom(double p)`
 - **Description**
Discrete Phase distribution that represents a geometric distribution with probability of success p
 - **Parameters**
`* p` – probability of succes
 - **Returns** – Dense Discrete Phase-Type Distribution

- *getMatrix*
`public no.uib.cipr.matrix.Matrix getMatrix()`
 - **See also**
`* PhaseVar.getMatrix()` (see 7.1.3, page 267)

- *getVector*
`public no.uib.cipr.matrix.Vector getVector()`
 - **See also**
`* PhaseVar.getVector()` (see 7.1.3, page 267)

- *NegativeBinomial*
`public static DenseDiscPhaseVar NegativeBinomial(double p, int r)`
 - **Description**
Discrete Phase Distribution thata represents a Negative Binomial distribution with parameters p and r
 - **Parameters**
`* p` – probability of success in one trial
`* r` – number of successes until absorption
 - **Returns** – Dense Discrete Phase-Type distribution

- *newVar*
`DiscPhaseVar newVar(int n)`
 - **Description copied from DiscPhaseVar** (see 7.1.2, page 261)
Creates a new variable of the same class of the original Discrete Phase-Type Variable
 - **Parameters**
`* n` – number of Phases of the new Variable
 - **Returns** – A new variable of the same class of the original Discrete Phase-Type Variable

- *setMatrix*

```
public void setMatrix( no.uib.cipr.matrix.Matrix A )
```

– See also

* `PhaseVar.setMatrix(Matrix)` (see 7.1.3, page 269)

- *setVector*

```
public void setVector( no.uib.cipr.matrix.Vector alpha )
```

– See also

* `PhaseVar.setVector(Vector)` (see 7.1.3, page 269)

Members inherited from class `jphase.AbstractDiscPhaseVar` (see 7.2.2, page 276)

- `public double cdf(double x)`
- `public double cdf(int n, double delta)`
- `public double CV()`
- `public String description()`
- `public double expectedValue()`
- `public Vector getMat0()`
- `public double getMat0Array()`
- `public double getMatrixArray()`
- `public int getNumPhases()`
- `public double getVec0()`
- `public double getVectorArray()`
- `public String label()`
- `public double lossFunction1(double x)`
- `public double lossFunction2(double x)`
- `public DiscPhaseVar max(DiscPhaseVar v2)`
- `public DiscPhaseVar max(DiscPhaseVar v2, DiscPhaseVar res)`
- `public double median()`
- `public DiscPhaseVar min(DiscPhaseVar v2)`
- `public DiscPhaseVar min(DiscPhaseVar v2, DiscPhaseVar res)`
- `public DiscPhaseVar mix(double p, DiscPhaseVar v2)`
- `public DiscPhaseVar mix(double p, DiscPhaseVar v2, DiscPhaseVar res)`
- `public double moment(int k)`
- `public double pmf(int k)`
- `public double pmf(int n, int delta)`
- `public double prob(double a, double b)`
- `public double quantil(double p)`
- `public double stdDeviation()`
- `public DiscPhaseVar sum(DiscPhaseVar v2)`
- `public DiscPhaseVar sum(DiscPhaseVar v2, DiscPhaseVar res)`
- `public DiscPhaseVar sumGeom(double p)`
- `public DiscPhaseVar sumPH(DiscPhaseVar v2)`
- `public DiscPhaseVar sumPH(DiscPhaseVar B, DiscPhaseVar res)`
- `public double survival(double x)`
- `public double survival(int n, double delta)`
- `public final String toString()`
- `public double variance()`

7.2.5 Class ErlangCoxianVar

Phase-Type representation of an ErlangCoxian distribution as defined by Osogami and Harchol in "Closed form solutions for mapping general distributions to quasi-minimal PH distributions", 2005.

Declaration

```
public class ErlangCoxianVar
extends jphase.AbstractContPhaseVar (see 7.2.1, page 270)
implements PhaseVar
```

Constructor summary

- ErlangCoxianVar()** Constructor of an Erlang Coxian variable in dense representation.
- ErlangCoxianVar(int)** Constructor of a ErlangCoxian variable in dense representation
- ErlangCoxianVar(int, double, double, double, double, double)** Constructor of a ErlangCoxian variable in dense representation

Method summary

- convoExpo(double)** Creates a Dense Continuous Phase Variable that represents the convolution of the original ErlangCoxian distribution and an exponential phase with rate lambda
- copy()**
- description()**
- getLambdaX1()**
- getLambdaX2()**
- getLambdaY()**
- getMatrix()**
- getN()**
- getP()**
- getPx()**
- getVector()**
- mixtureExpo(double, double)** Creates a Dense Continuous Phase Variable that represents the mixture of the original ErlangCoxian distribution (p) and an exponential phase with rate lambda (1-p)
- newVar(int)**
- setLambdaX1(double)**
- setLambdaX2(double)**
- setLambdaY(double)**
- setMatrix(Matrix)**
- setN(int)**
- setP(double)**
- setPx(double)**
- setVector(Vector)**

Constructors

- *ErlangCoxianVar*

public ErlangCoxianVar()

- **Description**

Constructor of an Erlang Coxian variable in dense representation. As default it has just one phase in the Erlang that is taken with probability one. This phase and those of the Coxian distribution has rate 1 per time unit.

- *ErlangCoxianVar*

public ErlangCoxianVar(int n)

- **Description**

Constructor of a ErlangCoxian variable in dense representation

- **Parameters**

* **n** – total number of phases (Erlang degree: n-2) absorption at the first stage of the Coxian distribution

- *ErlangCoxianVar*

public ErlangCoxianVar(int n, double p, double lambdaY, double lambdaX1, double lambdaX2, double px)

- **Description**

Constructor of a ErlangCoxian variable in dense representation

- **Parameters**

* **n** – total number of phases (Erlang degree: n-2)
 * **p** – probability of having a positive elapse time in the distribution. 1-p: mass probability at zero
 * **lambdaY** – rate of the Erlang distribution
 * **lambdaX1** – rate of the first stage of the Coxian distribution
 * **lambdaX2** – rate of the second stage of the Coxian distribution
 * **px** – probability of going from the first to the second stage in the Coxian distribution. 1-p: probability of absorption at the first stage of the Coxian distribution

Methods

- *convoExpo*

public PhaseVar convoExpo(double lambda)

- **Description**

Creates a Dense Continuous Phase Variable that represents the convolution of the original ErlangCoxian distribution and an exponential phase with rate lambda

- **Parameters**

* **lambda** – rate of the exponential phase to be included in the convolution

- **Returns** – Dense Continuous Phase Variable that represents the convolution of the original ErlangCoxian distribution and an exponential phase with rate lambda

- *copy*
PhaseVar copy()
 - **Description copied from PhaseVar** (see 7.1.3, page 265)
Creates a deep copy of the original Phase-Type Variable
 - **Returns** – A deep copy of the original Phase-Type Variable

- *description*
public java.lang.String description()

- *getLambdaX1*
public double getLambdaX1()
 - **Returns** – Rate of the first stage at the Coxian distribution

- *getLambdaX2*
public double getLambdaX2()
 - **Returns** – Rate of the secons stage at the Coxian distribution

- *getLambdaY*
public double getLambdaY()
 - **Returns** – Erlang distribution rate

- *getMatrix*
no.uib.cipr.matrix.Matrix getMatrix()
 - **Description copied from PhaseVar** (see 7.1.3, page 265)
Returns the transition matrix of the Phase-Type Distribution
 - **Returns** – Transition matrix for transient states of the Phase-Type Distribution

- *getN*
public int getN()
 - **Returns** – Total number of phases

- *getP*
public double getP()
 - **Returns** – Probability of having a positive elapse time in the distribution. 1-p: mass probability at zero

- *getPx*
public double getPx()
 - **Returns** – Probability of going from the first to the second stage in the Coxian distribution. 1-p: probability of absorption at the first stage if the Coxian distribution

- *getVector*
no.uib.cipr.matrix.Vector getVector()
 - **Description copied from PhaseVar** (see 7.1.3, page 265)
Returns the initial probability mass vector
 - **Returns** – Initial probability mass vector

- *mixtureExpo*

```
public PhaseVar mixtureExpo( double lambda, double p )
```

- **Description**

Creates a Dense Continuous Phase Variable that represents the mixture of the original ErlangCoxian distribution (p) and an exponential phase with rate lambda (1-p)

- **Parameters**

- * **lambda** – rate of the exponential phase to be included in the mixture
 - * **p** – probability mass of the ErlangCoxian distribution in the mixture

- **Returns** – Dense Continuous Phase Variable that represents the mixture of the original ErlangCoxian distribution (p) and an exponential phase with rate lambda (1-p)

- *newVar*

```
public ContPhaseVar newVar( int n )
```

- *setLambdaX1*

```
public void setLambdaX1( double lambdaX1 )
```

- **Parameters**

- * **lambdaX1** – Rate of the first stage at the Coxian distribution
-

- *setLambdaX2*

```
public void setLambdaX2( double lambdaX2 )
```

- **Parameters**

- * **lambdaX2** – Rate of the second stage at the Coxian distribution
-

- *setLambdaY*

```
public void setLambdaY( double lambdaY )
```

- **Parameters**

- * **lambdaY** – Erlang distribution rate
-

- *setMatrix*

```
void setMatrix( no.uib.cipr.matrix.Matrix A )
```

- **Description copied from PhaseVar** (see 7.1.3, page 265)

Rate Matrix = A

- **Parameters**

- * **A** – Transition matrix for transient states of the Phase-Type Distribution
-

- *setN*

```
public void setN( int n )
```

- **Parameters**

- * **n** – Total number of phases
-

- *setP*

```
public void setP( double p )
```

- **Parameters**

* *p* – Probability of having a positive elapse time in the distribution. 1-*p*: mass probability at zero

- *setPx*

```
public void setPx( double px )
```

- **Parameters**

* *px* – Probability of going from the first to the second stage in the Coxian distribution. 1-*p*: probability of absorption at the first stage if the Coxian distribution

- *setVector*

```
void setVector( no.uib.cipr.matrix.Vector alpha )
```

- **Description copied from PhaseVar** (see 7.1.3, page 265)

Initial Probability vector = *alpha*

- **Parameters**

* *alpha* – Initial probability mass vector

Members inherited from class jphase.AbstractContPhaseVar (see 7.2.1, page 270)

- `public double cdf(double x)`
- `public double cdf(int n, double delta)`
- `public double CV()`
- `public String description()`
- `public ContPhaseVar eqResidualTime()`
- `public double expectedValue()`
- `public Vector getMat0()`
- `public double getMat0Array()`
- `public double getMatrixArray()`
- `public int getNumPhases()`
- `public double getVec0()`
- `public double getVectorArray()`
- `public String label()`
- `public double lossFunction1(double x)`
- `public double lossFunction2(double x)`
- `public ContPhaseVar max(ContPhaseVar v2)`
- `public ContPhaseVar max(ContPhaseVar v2, ContPhaseVar res)`
- `public double median()`
- `public ContPhaseVar min(ContPhaseVar v2)`
- `public ContPhaseVar min(ContPhaseVar v2, ContPhaseVar res)`
- `public ContPhaseVar mix(double p, ContPhaseVar v2)`
- `public ContPhaseVar mix(double p, ContPhaseVar v2, ContPhaseVar res)`
- `public double moment(int k)`
- `public double pdf(double x)`
- `public double pdf(int n, double delta)`
- `public double prob(double a, double b)`
- `public double quantil(double p)`
- `public ContPhaseVar residualTime(double x)`
- `public ContPhaseVar residualVar(double a)`
- `public double stdDeviation()`
- `public ContPhaseVar sum(ContPhaseVar v2)`
- `public ContPhaseVar sum(ContPhaseVar v2, ContPhaseVar res)`
- `public ContPhaseVar sumGeom(double p)`
- `public ContPhaseVar sumPH(DiscPhaseVar v2)`
- `public ContPhaseVar sumPH(DiscPhaseVar B, ContPhaseVar res)`
- `public double survival(double x)`
- `public double survival(int n, double delta)`
- `public ContPhaseVar times(double c)`
- `public final String toString()`
- `public double variance()`
- `public ContPhaseVar waitingQ(double rho)`

7.2.6 Class HyperErlangVar

Declaration

```
public class HyperErlangVar
  extends jphase.AbstractContPhaseVar (see 7.2.1, page 270)
  implements PhaseVar
```

Version

0.1 This class allows the creation and manipulation of HyperErlang distributions. The associated matrix has dense representation

Constructor summary

HyperErlangVar() Constructor of a Hyper Erlang variable in dense representation.

HyperErlangVar(int) Constructor of a Hyper Erlang variable with n phases in dense representation

HyperErlangVar(int[], double[], double[], boolean) Constructor of a Hyper Erlang variable in dense representation

HyperErlangVar(int, int, int[], double[], double[], boolean) Constructor of a Hyper Erlang variable in dense representation

Method summary

```
cdf(double)
cdf(int, double)
copy()
description()
expectedValue()
getAlphas()
getLambdas()
getM()
getMatrix()
getN()
getR()
getVector()
moment(int)
newVar(int)
pdf(double)
pdf(int, double)
setAlphas(double[])
setLambdas(double[])
setM(int)
setMatrix(Matrix)
setN(int)
setR(int[])
setVector(Vector)
```

Constructors

- *HyperErlangVar*
public HyperErlangVar()
 - **Description**
 Constructor of a Hyper Erlang variable in dense representation. As default it has just one branch that is taken with probability one. the unique branch has one phase with rate 1 per time unit.

- *HyperErlangVar*
public HyperErlangVar(int n)
 - **Description**
 Constructor of a Hyper Erlang variable with n phases in dense representation
 - **Parameters**
 - * **n** – Total number of phases

- *HyperErlangVar*
public HyperErlangVar(int[] r, double[] alphas, double[] lambdas, boolean deep)
 - **Description**
 Constructor of a Hyper Erlang variable in dense representation
 - **Parameters**
 - * **r** – Number of phases in each branch
 - * **alphas** – Probability associated to each branch
 - * **lambdas** – Rate associated to each branch
 - * **deep** – True if this is a deep constructor, false if not

- *HyperErlangVar*
public HyperErlangVar(int N, int M, int[] r, double[] alphas, double[] lambdas, boolean deep)
 - **Description**
 Constructor of a Hyper Erlang variable in dense representation
 - **Parameters**
 - * **N** – Total number of phases
 - * **M** – Number of branches
 - * **r** – Number of phases in each branch
 - * **alphas** – Probability associated to each branch
 - * **lambdas** – Rate associated to each branch
 - * **deep** – True if this is a deep constructor, false if not

Methods

- *cdf*
public double cdf(double x)
 - **See also**
 - * **PhaseVar.cdf(double)** (see 7.1.3, page 266)

-
- *cdf*
`public double[] cdf(int n, double delta)`
 – **See also**
 * `PhaseVar.cdf(int,double)` (see 7.1.3, page 266)

 - *copy*
`PhaseVar copy()`
 – **Description copied from PhaseVar** (see 7.1.3, page 265)
 Creates a deep copy of the original Phase-Type Variable
 – **Returns** – A deep copy of the original Phase-Type Variable

 - *description*
`public java.lang.String description()`
 - *expectedValue*
`public double expectedValue()`
 – **See also**
 * `PhaseVar.expectedValue()` (see 7.1.3, page 266)

 - *getAlphas*
`public double[] getAlphas()`
 – **Returns** – Probability associated to each branch

 - *getLambdas*
`public double[] getLambdas()`
 – **Returns** – Rate associated to each branch

 - *getM*
`public int getM()`
 – **Returns** – Nnumber of branches

 - *getMatrix*
`no.uib.cipr.matrix.Matrix getMatrix()`
 – **Description copied from PhaseVar** (see 7.1.3, page 265)
 Returns the transition matrix of the Phase-Type Distribution
 – **Returns** – Transition matrix for transient states of the Phase-Type Distribution

 - *getN*
`public int getN()`
 – **Returns** – Total number of phases

 - *getR*
`public int[] getR()`
 – **Returns** – Number of phases in each branch

- *getVector*

```
no.uib.cipr.matrix.Vector getVector( )
```

- **Description copied from PhaseVar** (see 7.1.3, page 265)

Returns the initial probability mass vector

- **Returns** – Initial probability mass vector

- *moment*

```
public double moment( int k )
```

- **See also**

* PhaseVar.moment(int) (see 7.1.3, page 268)

- *newVar*

```
public ContPhaseVar newVar( int n )
```

- *pdf*

```
public double pdf( double x )
```

- **See also**

* ContPhaseVar.pdf(double) (see 7.1.1, page 258)

- *pdf*

```
public double[] pdf( int n, double delta )
```

- **See also**

* ContPhaseVar.pdf(int,double) (see 7.1.1, page 258)

- *setAlphas*

```
public void setAlphas( double[] alphas )
```

- **Parameters**

* alphas – Probability associated to each branch to set

- *setLambdas*

```
public void setLambdas( double[] lambdas )
```

- **Parameters**

* lambdas – Rates associated to each branch to set

- *setM*

```
public void setM( int M )
```

- **Parameters**

* M – Number of branches to set

- *setMatrix*

```
void setMatrix( no.uib.cipr.matrix.Matrix A )
```

- **Description copied from PhaseVar** (see 7.1.3, page 265)

Rate Matrix = A

- **Parameters**

* A – Transition matrix for transient states of the Phase-Type Distribution

- *setN*

```
public void setN( int N )
```

- **Parameters**

* N – Total number of phases to set

- *setR*

```
public void setR( int[] r )
```

- **Parameters**

* r – Number of phases in each branch to set

- *setVector*

```
void setVector( no.uib.cipr.matrix.Vector alpha )
```

- **Description copied from PhaseVar (see 7.1.3, page 265)**

Initial Probability vector = alpha

- **Parameters**

* alpha – Initial probability mass vector

Members inherited from class jphase.AbstractContPhaseVar (see 7.2.1, page 270)

- public double **cdf**(double x)
- public double **cdf**(int n, double delta)
- public double **CV**()
- public String **description**()
- public ContPhaseVar **eqResidualTime**()
- public double **expectedValue**()
- public Vector **getMat0**()
- public double **getMat0Array**()
- public double **getMatrixArray**()
- public int **getNumPhases**()
- public double **getVec0**()
- public double **getVectorArray**()
- public String **label**()
- public double **lossFunction1**(double x)
- public double **lossFunction2**(double x)
- public ContPhaseVar **max**(ContPhaseVar v2)
- public ContPhaseVar **max**(ContPhaseVar v2, ContPhaseVar res)
- public double **median**()
- public ContPhaseVar **min**(ContPhaseVar v2)
- public ContPhaseVar **min**(ContPhaseVar v2, ContPhaseVar res)
- public ContPhaseVar **mix**(double p, ContPhaseVar v2)
- public ContPhaseVar **mix**(double p, ContPhaseVar v2, ContPhaseVar res)
- public double **moment**(int k)
- public double **pdf**(double x)
- public double **pdf**(int n, double delta)
- public double **prob**(double a, double b)
- public double **quantil**(double p)
- public ContPhaseVar **residualTime**(double x)
- public ContPhaseVar **residualVar**(double a)
- public double **stdDeviation**()
- public ContPhaseVar **sum**(ContPhaseVar v2)
- public ContPhaseVar **sum**(ContPhaseVar v2, ContPhaseVar res)
- public ContPhaseVar **sumGeom**(double p)
- public ContPhaseVar **sumPH**(DiscPhaseVar v2)
- public ContPhaseVar **sumPH**(DiscPhaseVar B, ContPhaseVar res)
- public double **survival**(double x)
- public double **survival**(int n, double delta)
- public ContPhaseVar **times**(double c)
- public final String **toString**()
- public double **variance**()
- public ContPhaseVar **waitingQ**(double rho)

7.2.7 Class MarkovMatrix

Declaration

```
public class MarkovMatrix
  extends Jama.Matrix
```

Field summary

useUniformization

Constructor summary

MarkovMatrix(double[][])
 MarkovMatrix(Matrix)

Method summary

compExp()
 compLog()
 concatCols(Matrix, Matrix)
 concatRows(Matrix, Matrix)
 exp(double)
 exp(double, Matrix, Matrix)
 exp(int, double, Matrix, Matrix)
 expRunge(int, double, Matrix, Matrix)
 expTimesOnes(double)
 expTimesOnes(double, Matrix)
 expTimesOnes(int, double, Matrix)
 expUnif(double[], Matrix, Matrix) Computes leftMat * exp(A x) * rightMat, for
 all values x.
 expUnif(double[], Matrix, Matrix, int) Computes leftMat * exp(A x) * rightMat,
 for all values x.
 expUnif(double, Matrix, Matrix) Computes leftMat * exp(A x) * rightMat, for
 the value x.
 expUnif(int, double, Matrix, Matrix) Computes leftMat * exp(A x) * rightMat,
 for all values x.
 expUnif(int, double, Matrix, Matrix, int) Computes leftMat * exp(A x) *
 rightMat, for all values x= 0, delta, 2delta, 3delta,....
 identity(int)
 inverse()
 isStochastic() Detrmines if the matrix is stochastic.
 kronecker(Matrix)
 kronecker(Matrix, Matrix)
 kroneckerSum(Matrix)
 kroneckerSum(Matrix, Matrix)
 oldExp(double, Matrix)
 plus(double)
 pow(int)
 power(int) Returns A^k
 readTxt(String)

```

scalar()
size()
solveTranspose(Matrix) Solve  $X^*A = B$ , which is also  $A^*X = B$ 
times(double)
times(Matrix)
timesOne()
toMarkovMatrix(Matrix)
toString()
toStringRTF()
toTxt()
uminus()
Zeros(int, int)

```

Serializable Fields

- private java.util.List **powers**
- private int **maxPower**
- private MarkovMatrix **normal**
- private double **ldaMax**
- MarkovMatrix **aExpStep**

Fields

- public static boolean **useUniformization**

Constructors

- *MarkovMatrix*
public MarkovMatrix(double[] [] mat)
- *MarkovMatrix*
public MarkovMatrix(Jama.Matrix mat)

Methods

- *compExp*
public MarkovMatrix compExp()
- *compLog*
public MarkovMatrix compLog()
- *concatCols*
public static Jama.Matrix concatCols(Jama.Matrix A, Jama.Matrix B)

- *concatRows*

```
public static Jama.Matrix concatRows( Jama.Matrix A, Jama.Matrix B )
```
- *exp*

```
public MarkovMatrix exp( double x )
```
- *exp*

```
public MarkovMatrix exp( double x, Jama.Matrix leftMat, Jama.Matrix  
rightMat )
```
- *exp*

```
public MarkovMatrix[] exp( int n, double delta, Jama.Matrix leftMat,  
Jama.Matrix rightMat )
```
- *expRunge*

```
public MarkovMatrix[] expRunge( int n, double delta, Jama.Matrix leftMat,  
Jama.Matrix rightMat )
```
- *expTimesOnes*

```
public MarkovMatrix expTimesOnes( double x )
```
- *expTimesOnes*

```
public MarkovMatrix expTimesOnes( double x, Jama.Matrix leftMatrix )
```
- *expTimesOnes*

```
public MarkovMatrix[] expTimesOnes( int N, double delta, Jama.Matrix  
leftMatrix )
```
- *expUnif*

```
public MarkovMatrix[] expUnif( double[] times, Jama.Matrix leftMat,  
Jama.Matrix rightMat )
```

 - **Description**
Computes $\text{leftMat} * \exp(A x) * \text{rightMat}$, for all values x . It uses the uniformization algorithm as described in page 60 of Latouche and Ramaswami
- *expUnif*

```
public MarkovMatrix[] expUnif( double[] times, Jama.Matrix leftMat,  
Jama.Matrix rightMat, int truncate )
```

 - **Description**
Computes $\text{leftMat} * \exp(A x) * \text{rightMat}$, for all values x . It uses the uniformization algorithm as described in page 60 of Latouche and Ramaswami
- *expUnif*

```
public MarkovMatrix[] expUnif( double x, Jama.Matrix leftMat, Jama.Matrix  
rightMat )
```

 - **Description**
Computes $\text{leftMat} * \exp(A x) * \text{rightMat}$, for the value x . It uses the uniformization algorithm as described in page 60 of Latouche and Ramaswami
- *expUnif*

```
public MarkovMatrix[] expUnif( int n, double delta, Jama.Matrix leftMat,  
Jama.Matrix rightMat )
```

– **Description**

Computes $\text{leftMat} * \exp(A x) * \text{rightMat}$, for all values x . It uses the uniformization algorithm as described in page 60 of Latouche and Ramaswami

- *expUnif*

```
public MarkovMatrix[] expUnif( int n, double delta, Jama.Matrix leftMat,
Jama.Matrix rightMat, int truncate )
```

– **Description**

Computes $\text{leftMat} * \exp(A x) * \text{rightMat}$, for all values $x = 0, \text{delta}, 2\text{delta}, 3\text{delta}, \dots$. It uses the uniformization algorithm as described in page 60 of Latouche and Ramaswami

- *identity*

```
public static Jama.Matrix identity( int n )
```

- *inverse*

```
public Jama.Matrix inverse( )
```

- *isStochastic*

```
public boolean isStochastic( )
```

– **Description**

Determines if the matrix is stochastic.

– **Returns** – true if the matrix is stochastic.

- *kronecker*

```
public Jama.Matrix kronecker( Jama.Matrix B )
```

- *kronecker*

```
public static Jama.Matrix kronecker( Jama.Matrix A, Jama.Matrix B )
```

- *kroneckerSum*

```
public Jama.Matrix kroneckerSum( Jama.Matrix B )
```

- *kroneckerSum*

```
public static Jama.Matrix kroneckerSum( Jama.Matrix A, Jama.Matrix B )
```

- *oldExp*

```
public MarkovMatrix oldExp( double x, Jama.Matrix rightMat )
```

- *plus*

```
public MarkovMatrix plus( double x )
```

- *pow*

```
public Jama.Matrix pow( int k )
```

- *power*

```
public MarkovMatrix power( int k )
```

– **Description**

Returns A^k

- *readTxt*

```
public static MarkovMatrix readTxt( java.lang.String stg )
```

- *scalar*
public double scalar()
- *size*
public int size()
- *solveTranspose*
public Jama.Matrix solveTranspose(Jama.Matrix B)
 - **Description**
Solve $X \cdot A = B$, which is also $A' \cdot X' = B'$
 - **Parameters**
* B – right hand side
 - **Returns** – solution if A is square, least squares solution otherwise.
- *times*
public Jama.Matrix times(double arg0)
- *times*
public Jama.Matrix times(Jama.Matrix arg0)
- *timesOne*
public MarkovMatrix timesOne()
- *toMarkovMatrix*
public static MarkovMatrix toMarkovMatrix(Jama.Matrix A)
- *toString*
public java.lang.String toString()
- *toStringRTF*
public java.lang.String toStringRTF()
- *toTxt*
public java.lang.String toTxt()
- *uminus*
public Jama.Matrix uminus()
- *Zeros*
public static MarkovMatrix Zeros(int rows, int cols)

Members inherited from class Jama.Matrix

- public Matrix arrayLeftDivide(Matrix arg0)
- public Matrix arrayLeftDivideEquals(Matrix arg0)
- public Matrix arrayRightDivide(Matrix arg0)
- public Matrix arrayRightDivideEquals(Matrix arg0)
- public Matrix arrayTimes(Matrix arg0)
- public Matrix arrayTimesEquals(Matrix arg0)
- public CholeskyDecomposition chol()
- public Object clone()
- public double cond()
- public static Matrix constructWithCopy(double[][] arg0)
- public Matrix copy()
- public double det()
- public EigenvalueDecomposition eig()
- public double get(int arg0, int arg1)
- public double getArray()

- public double `getArrayCopy()`
- public int `getColumnDimension()`
- public double `getColumnPackedCopy()`
- public Matrix `getMatrix(int[] arg0, int[] arg1)`
- public Matrix `getMatrix(int[] arg0, int arg1, int arg2)`
- public Matrix `getMatrix(int arg0, int arg1, int[] arg2)`
- public Matrix `getMatrix(int arg0, int arg1, int arg2, int arg3)`
- public int `getRowDimension()`
- public double `getRowPackedCopy()`
- public static Matrix `identity(int arg0, int arg1)`
- public Matrix `inverse()`
- public LUdecomposition `lu()`
- public Matrix `minus(Matrix arg0)`
- public Matrix `minusEquals(Matrix arg0)`
- public double `norm1()`
- public double `norm2()`
- public double `normF()`
- public double `normInf()`
- public Matrix `plus(Matrix arg0)`
- public Matrix `plusEquals(Matrix arg0)`
- public void `print(int arg0, int arg1)`
- public void `print(java.text.NumberFormat arg0, int arg1)`
- public void `print(java.io.PrintWriter arg0, int arg1, int arg2)`
- public void `print(java.io.PrintWriter arg0, java.text.NumberFormat arg1, int arg2)`
- public QRdecomposition `qr()`
- public static Matrix `random(int arg0, int arg1)`
- public int `rank()`
- public static Matrix `read(java.io.BufferedReader arg0) throws java.io.IOException`
- public void `set(int arg0, int arg1, double arg2)`
- public void `setMatrix(int[] arg0, int[] arg1, Matrix arg2)`
- public void `setMatrix(int[] arg0, int arg1, int arg2, Matrix arg3)`
- public void `setMatrix(int arg0, int arg1, int[] arg2, Matrix arg3)`
- public void `setMatrix(int arg0, int arg1, int arg2, int arg3, Matrix arg4)`
- public Matrix `solve(Matrix arg0)`
- public Matrix `solveTranspose(Matrix arg0)`
- public SingularValueDecomposition `svd()`
- public Matrix `times(double arg0)`
- public Matrix `times(Matrix arg0)`
- public Matrix `timesEquals(double arg0)`
- public double `trace()`
- public Matrix `transpose()`
- public Matrix `uminus()`

7.2.8 Class MatrixUtils

Declaration

```
public class MatrixUtils
extends java.lang.Object
```

Version

0.1 Utilities class for the jphase package

Constructor summary

MatrixUtils()

Method summary

average(double[]) Computes data average

average2(double[]) Computes the second moment of the data

concatCols(DenseMatrix, DenseMatrix) Concatenates the columns of the matrices, keeping the same number of rows in dense format

concatCols(Matrix, Matrix, Matrix) Concatenates the columns of the matrices, keeping the same number of rows in the predefined format

concatQuad(Matrix, Matrix, Matrix, Matrix, Matrix) Concatenates the columns of the left and right upper matrices and the result is concatenated by rows with the concatenation of left and right lower matrices

concatRows(DenseMatrix, DenseMatrix) Concatenates the rows of the matrices, keeping the same number of columns in dense format

concatRows(Matrix, Matrix, Matrix) Concatenates the rows of the matrices, keeping the same number of columns in the predefined format

concatVectors(DenseVector, DenseVector) Concatenates the vectors in dense format

concatVectors(Vector, Vector, Vector) Concatenates the vectors in the predefined format

CV(double[]) Return the Coefficient of Variation of the data trace

distance(double[], double[]) Calculates the distance between two arrays, defined as the maximum euclidean distance between every entry in those arrays

exp(Matrix, double) Returns $\exp(A x)$, for the value x .

exp(Matrix, double, Matrix, Matrix) Returns $\text{leftMat} * \exp(A x) * \text{rightMat}$, for the value x .

exp(Matrix, double, Matrix, Matrix, boolean) Returns $\text{leftMat} * \exp(A x) * \text{rightMat}$, for the value x .

exp(Matrix, double, Vector, Vector) Returns $\text{leftVec} * \exp(A x) * \text{rightVec}$, for the value x .

exp(Matrix, int, double, Matrix, Matrix, boolean) Computes $\text{leftMat} * \exp(A x) * \text{rightMat}$, for all values $x = 0 + i * \text{delta}$, $i=0, \dots, n$.

exp(Matrix, int, double, Vector, Vector, boolean) Returns $\text{leftVec} * \exp(A x) * \text{rightVec}$, for all values $x = 0 + i * \text{delta}$, $i=0, \dots, n$.

expRunge(Matrix, int, double, Matrix, Matrix) Computes $\text{leftMat} * \exp(A x) * \text{rightMat}$, for all values $x = 0 + i * \text{delta}$, $i=0, \dots, n$.

expTimesOnes(Matrix, double) Returns $\exp(A x) * \text{Ones}$, for the value x .

expTimesOnes(Matrix, double, Matrix) Returns $\text{leftMat} * \exp(A x) * \text{Ones}$, for the value x .

expTimesOnes(Matrix, double, Vector) Returns $\text{leftVec} * \exp(A x) * \text{OnesVector}$, for the value x .

expTimesOnes(Matrix, int, double, Matrix) Returns $\text{leftMat} * \exp(A x) * \text{OnesCol}$, for all values $x = 0 + i * \text{delta}$, $i=0, \dots, n$.

expTimesOnes(Matrix, int, double, Vector) Returns $\text{leftVec} * \exp(A x) * \text{OnesVector}$, for all values $x = 0 + i * \text{delta}$, $i=0, \dots, n$.

expUnif(Matrix, double[], Matrix, Matrix) Computes $\text{leftMat} * \exp(A x) * \text{rightMat}$, for all values x in times.

expUnif(Matrix, double[], Matrix, Matrix, int) Computes $\text{leftMat} * \exp(A x) * \text{rightMat}$, for all values x in times.

expUnif(Matrix, double[], Vector, Vector, int) Computes $\text{leftVec} * \exp(A x) * \text{rightVec}$, for all values x in times.

expUnif(Matrix, double, Matrix, Matrix) Computes $\text{leftMat} * \exp(A x) * \text{rightMat}$, for the value x .

expUnif(Matrix, int, double, Matrix, Matrix) Computes $\text{leftMat} * \exp(A x) * \text{rightMat}$, for all values $x = 0 + i * \text{delta}$, $i=0, \dots, n$.

expUnif(Matrix, int, double, Matrix, Matrix, int) Computes $\text{leftMat} * \exp(A x) * \text{rightMat}$, for all values $x = 0 + i * \text{delta}$, $i=0, \dots, n$.

expUnif(Matrix, int, double, Vector, Vector) Computes $\text{leftVec} * \exp(A x) * \text{rightVec}$, for all values $x = 0 + i * \text{delta}$, $i=0, \dots, n$.

expUnif(Matrix, int, double, Vector, Vector, int) Computes $\text{leftVec} * \exp(A x) * \text{rightVec}$, for all values $x = 0 + i * \text{delta}$, $i=0, \dots, n$.

kronecker(Matrix, Matrix) Returns the kronecker product of two matrices in dense format

kronecker(Matrix, Matrix, Matrix) Returns the kronecker product of two matrices in the predefined storage format

kronecker(Matrix, Vector, Matrix) Returns the kronecker product of one matrix with one vector (Matrix x Vector) in dense format

kronecker(Vector, Matrix, Matrix) Returns the kronecker product of one vector with one matrix (Vector x Matrix) in dense format

kroneckerSum(Matrix, Matrix) Returns the kronecker sum of two matrices in dense format

kroneckerSum(Matrix, Matrix, Matrix) Returns the kronecker sum of two matrices and stores it in the predefined format

kroneckerVectors(DenseVector, DenseVector) Returns the kronecker product of two vectors in dense format

kroneckerVectors(Vector, Vector, Vector) Returns the kronecker product of two vectors in the predefined format

matPower(Matrix, int) Computes k power of the matrix A

matPower(Matrix, int, Vector, Vector) Computes the k th power of the matrix A premultiplied by leftVec and postmultiplied by rightVec

multVector(Vector, Vector, Matrix) Computes the product of two vectors $A x B \wedge T$

OnesCol(int) Returns a one-column matrix in dense format with one in every entry

OnesRow(int) Returns a one-row matrix in dense format with one in every entry

OnesVector(int) Returns a DenseVector with one in every entry

OnesVector(Vector) Returns a Vector with one in every entry in the predefined storage format

pow(double, int) Calculates x^n
scalar(Matrix) Returns the value of the position (0,0) in the matrix, if its number of columns is equal to one (1)
sumMatPower(Matrix, int, Vector, Vector) Computes the sum of the first k terms of the succession $T^{(j-1)}$, from $j = 1$
variance(double[]) Return the variance of the data

Constructors

- *MatrixUtils*
`public MatrixUtils()`

Methods

- *average*
`public static double average(double[] datos)`
 - **Description**
Computes data average
 - **Parameters**
 - * `datos` –
 - **Returns** – Data Average

- *average2*
`public static double average2(double[] data)`
 - **Description**
Computes the second moment of the data
 - **Parameters**
 - * `data` – data trace
 - **Returns** – Second data trace moment

- *concatCols*
`public static no.uib.cipr.matrix.DenseMatrix concatCols(no.uib.cipr.matrix.DenseMatrix A, no.uib.cipr.matrix.DenseMatrix B)`
 - **Description**
Concatenates the columns of the matrices, keeping the same number of rows in dense format
 - **Parameters**
 - * `A` – DenseMatrix
 - * `B` – DenseMatrix
 - **Returns** – Resulting matrix from concatenation

- *concatCols*
`public static no.uib.cipr.matrix.Matrix concatCols(no.uib.cipr.matrix.Matrix A, no.uib.cipr.matrix.Matrix B, no.uib.cipr.matrix.Matrix res)`

– **Description**

Concatenates the columns of the matrices, keeping the same number of rows in the predefined format

– **Parameters**

- * **A** – Matrix
- * **B** – Matrix
- * **res** – Resulting matrix

– **Returns** – Resulting matrix from concatenation

• *concatQuad*

```
public static no.uib.cipr.matrix.Matrix concatQuad(
no.uib.cipr.matrix.Matrix leftUp, no.uib.cipr.matrix.Matrix rightUp,
no.uib.cipr.matrix.Matrix leftDown, no.uib.cipr.matrix.Matrix rightDown,
no.uib.cipr.matrix.Matrix res )
```

– **Description**

Concatenates the columns of the left and right upper matrices and the result is concatenated by rows with the concatenation of left and right lower matrices

– **Parameters**

- * **leftUp** – Left upper Matrix
- * **rightUp** – Right upper Matrix
- * **leftDown** – Left lower Matrix
- * **rightDown** – Right lower matrix
- * **res** – Resulting Matrix

– **Returns** – resulting matrix from concatenation

• *concatRows*

```
public static no.uib.cipr.matrix.DenseMatrix concatRows(
no.uib.cipr.matrix.DenseMatrix A, no.uib.cipr.matrix.DenseMatrix B )
```

– **Description**

Concatenates the rows of the matrices, keeping the same number of columns in dense format

– **Parameters**

- * **A** – DenseMatrix
- * **B** – DenseMatrix

– **Returns** – Resulting matrix from concatenation

• *concatRows*

```
public static no.uib.cipr.matrix.Matrix concatRows(
no.uib.cipr.matrix.Matrix A, no.uib.cipr.matrix.Matrix B,
no.uib.cipr.matrix.Matrix res )
```

– **Description**

Concatenates the rows of the matrices, keeping the same number of columns in the predefined format

– **Parameters**

- * **A** – Matrix
- * **B** – Matrix
- * **res** – Resulting matrix

– **Returns** – Resulting matrix from concatenation

- *concatVectors*

```
public static no.uib.cipr.matrix.DenseVector concatVectors(
no.uib.cipr.matrix.DenseVector A, no.uib.cipr.matrix.DenseVector B )
```

- **Description**

Concatenates the vectors in dense format

- **Parameters**

* A – DenseVector

* B – DenseVector

- **Returns** – Resulting DenseVector from concatenation

- *concatVectors*

```
public static no.uib.cipr.matrix.Vector concatVectors(
no.uib.cipr.matrix.Vector A, no.uib.cipr.matrix.Vector B,
no.uib.cipr.matrix.Vector res )
```

- **Description**

Concatenates the vectors in the predefined format

- **Parameters**

* A – Vector

* B – vector

* res – Resulting vector

- **Returns** – Resulting vectormatrix from concatenation

- *CV*

```
public static double CV( double[] data )
```

- **Description**

Return the Coefficient of Variation of the data trace

- **Parameters**

* data – data trace

- **Returns** – Data Coefficient of Variation

- *distance*

```
public static double distance( double[] v1, double[] v2 )
```

- **Description**

Calculates the distance between two arrays, defined as the maximum euclidean distance between every entry in those arrays

- **Parameters**

* v1 – array

* v2 – array

- **Returns** – Distance between two arrays

- *exp*

```
public static no.uib.cipr.matrix.Matrix exp( no.uib.cipr.matrix.Matrix A,
double x )
```

- **Description**

Returns $\exp(A x)$, for the value x. It uses the uniformization algorithm as described in page 60 of Latouche and Ramaswami

– **Parameters**

- * **A** – Matrix
- * **x** – evaluation point

– **Returns** – $\exp(A \ x)$

• *exp*

```
public static no.uib.cipr.matrix.Matrix exp( no.uib.cipr.matrix.Matrix A,
double x, no.uib.cipr.matrix.Matrix leftMat, no.uib.cipr.matrix.Matrix
rightMat )
```

– **Description**

Returns $\text{leftMat} * \exp(A \ x) * \text{rightMat}$, for the value x . It uses the uniformization algorithm as described in page 60 of Latouche and Ramaswami

– **Parameters**

- * **A** – Matrix
- * **x** – evaluation point
- * **leftMat** – Matrix
- * **rightMat** – Matrix

– **Returns** – $\text{leftMat} * \exp(A \ x) * \text{rightMat}$

• *exp*

```
public static no.uib.cipr.matrix.Matrix exp( no.uib.cipr.matrix.Matrix A,
double x, no.uib.cipr.matrix.Matrix leftMat, no.uib.cipr.matrix.Matrix
rightMat, boolean useUniformization )
```

– **Description**

Returns $\text{leftMat} * \exp(A \ x) * \text{rightMat}$, for the value x . It uses the uniformization algorithm or the RungeKutta method

– **Parameters**

- * **A** – Matrix
- * **x** – evaluation point
- * **leftMat** – Matrix
- * **rightMat** – Matrix
- * **useUniformization** – true if the method to use is Uniformization, false if it is RungeKutta

– **Returns** – $\text{leftMat} * \exp(A \ x) * \text{rightMat}$

• *exp*

```
public static double exp( no.uib.cipr.matrix.Matrix A, double x,
no.uib.cipr.matrix.Vector leftVec, no.uib.cipr.matrix.Vector rightVec )
```

– **Description**

Returns $\text{leftVec} * \exp(A \ x) * \text{rightVec}$, for the value x . It uses the uniformization algorithm as described in page 60 of Latouche and Ramaswami

– **Parameters**

- * **A** – Matrix
- * **x** – evaluation point
- * **leftVec** – Vector
- * **rightVec** – Vector

– **Returns** – $\text{leftVec} * \exp(A \ x) * \text{rightVec}$

- *exp*

```
public static no.uib.cipr.matrix.Matrix[] exp( no.uib.cipr.matrix.Matrix A,
int n, double delta, no.uib.cipr.matrix.Matrix leftMat,
no.uib.cipr.matrix.Matrix rightMat, boolean useUniformization )
```

- **Description**

Computes $\text{leftMat} * \exp(A x) * \text{rightMat}$, for all values $x = 0 + i\delta$, $i=0, \dots, n$. It uses the uniformization algorithm or the RungeKutta method

- **Parameters**

- * *A* – Matrix
- * *n* – number of evaluation points
- * *delta* – separation between evaluation points
- * *leftMat* – Matrix
- * *rightMat* – Matrix
- * *useUniformization* – true if the method to use is Uniformization, false if it is RungeKutta

- **Returns** – $\text{leftMat} * \exp(A x) * \text{rightMat}$, for $x = 0 + i\delta$, $i=0, \dots, n$

- *exp*

```
public static double[] exp( no.uib.cipr.matrix.Matrix A, int n, double
delta, no.uib.cipr.matrix.Vector leftVec, no.uib.cipr.matrix.Vector rightVec,
boolean useUniformization )
```

- **Description**

Returns $\text{leftVec} * \exp(A x) * \text{rightVec}$, for all values $x = 0 + i\delta$, $i=0, \dots, n$. It uses the uniformization algorithm or the RungeKutta method

- **Parameters**

- * *A* – Matrix
- * *n* – number of evaluation points
- * *delta* – separation between evaluation points
- * *leftVec* – Vector
- * *rightVec* – Vector
- * *useUniformization* – true if the method to use is Uniformization, false if it is RungeKutta

- **Returns** – $\text{leftVec} * \exp(A x) * \text{rightVec}$

- *expRunge*

```
public static no.uib.cipr.matrix.Matrix[] expRunge(
no.uib.cipr.matrix.Matrix A, int n, double delta, no.uib.cipr.matrix.Matrix
leftMat, no.uib.cipr.matrix.Matrix rightMat )
```

- **Description**

Computes $\text{leftMat} * \exp(A x) * \text{rightMat}$, for all values $x = 0 + i\delta$, $i=0, \dots, n$. It uses the RungeKutta method

- **Parameters**

- * *A* – Matrix
- * *n* – number of evaluation points
- * *delta* – separation between evaluation points
- * *leftMat* – Matrix
- * *rightMat* – Matrix

- **Returns** – $\text{leftMat} * \exp(A x) * \text{rightMat}$, for $x = 0 + i\delta$, $i=0, \dots, n$

- *expTimesOnes*

```
public static no.uib.cipr.matrix.Matrix expTimesOnes(
no.uib.cipr.matrix.Matrix A, double x )
```

- **Description**

Returns $\exp(A x) * \text{Ones}$, for the value x . It uses the uniformization algorithm as described in page 60 of Latouche and Ramaswami

- **Parameters**

- * A – Matrix
- * x – evaluation point

- **Returns** – $\exp(A x) * \text{Ones}$

- *expTimesOnes*

```
public static no.uib.cipr.matrix.Matrix expTimesOnes(
no.uib.cipr.matrix.Matrix A, double x, no.uib.cipr.matrix.Matrix leftMat )
```

- **Description**

Returns $\text{leftMat} * \exp(A x) * \text{Ones}$, for the value x . It uses the uniformization algorithm as described in page 60 of Latouche and Ramaswami

- **Parameters**

- * A – Matrix
- * x – evaluation point
- * leftMat – Matrix

- **Returns** – $\text{leftMat} * \exp(A x) * \text{rightMat}$

- *expTimesOnes*

```
public static double expTimesOnes( no.uib.cipr.matrix.Matrix A, double x,
no.uib.cipr.matrix.Vector leftVec )
```

- **Description**

Returns $\text{leftVec} * \exp(A x) * \text{OnesVector}$, for the value x . It uses the uniformization algorithm as described in page 60 of Latouche and Ramaswami

- **Parameters**

- * A – Matrix
- * x – evaluation point
- * leftVec – Vector

- **Returns** – $\text{leftVec} * \exp(A x) * \text{OnesVector}$

- *expTimesOnes*

```
public static no.uib.cipr.matrix.Matrix[] expTimesOnes(
no.uib.cipr.matrix.Matrix A, int n, double delta, no.uib.cipr.matrix.Matrix
leftMat )
```

- **Description**

Returns $\text{leftMat} * \exp(A x) * \text{OnesCol}$, for all values $x = 0 + i * \text{delta}$, $i=0, \dots, n$. It uses the uniformization algorithm as described in page 60 of Latouche and Ramaswami

- **Parameters**

- * A – Matrix
- * n – number of evaluation points
- * delta – separation between evaluation points
- * leftMat – Matrix

– **Returns** – $\text{leftMat} * \exp(A x) * \text{OnesCol}$, for $x = 0 + i \cdot \text{delta}$, $i=0, \dots, n$

- *expTimesOnes*

```
public static double[] expTimesOnes( no.uib.cipr.matrix.Matrix A, int n,
double delta, no.uib.cipr.matrix.Vector leftVec )
```

– **Description**

Returns $\text{leftVec} * \exp(A x) * \text{OnesVector}$, for all values $x = 0 + i \cdot \text{delta}$, $i=0, \dots, n$. It uses the uniformization algorithm as described in page 60 of Latouche and Ramaswami

– **Parameters**

- * **A** – Matrix
- * **n** – number of evaluation points
- * **delta** – separation between evaluation points
- * **leftVec** – Vector

– **Returns** – $\text{leftVec} * \exp(A x) * \text{OnesVector}$, for $x = 0 + i \cdot \text{delta}$, $i=0, \dots, n$

- *expUnif*

```
public static no.uib.cipr.matrix.Matrix[] expUnif( no.uib.cipr.matrix.Matrix
A, double[] times, no.uib.cipr.matrix.Matrix leftMat,
no.uib.cipr.matrix.Matrix rightMat )
```

– **Description**

Computes $\text{leftMat} * \exp(A x) * \text{rightMat}$, for all values x in **times**. It uses the uniformization algorithm

– **Parameters**

- * **A** – Matrix
- * **times** – evaluation points
- * **leftMat** – Matrix
- * **rightMat** – Matrix

– **Returns** – $\text{leftMat} * \exp(A x) * \text{rightMat}$

- *expUnif*

```
public static no.uib.cipr.matrix.Matrix[] expUnif( no.uib.cipr.matrix.Matrix
A, double[] times, no.uib.cipr.matrix.Matrix leftMat,
no.uib.cipr.matrix.Matrix rightMat, int truncate )
```

– **Description**

Computes $\text{leftMat} * \exp(A x) * \text{rightMat}$, for all values x in **times**. It uses the uniformization algorithm

– **Parameters**

- * **A** – Matrix
- * **times** – evaluation points
- * **leftMat** – Matrix
- * **rightMat** – Matrix
- * **truncate** – upper bound for iterations

– **Returns** – $\text{leftMat} * \exp(A x) * \text{rightMat}$

- *expUnif*

```
public static double[] expUnif( no.uib.cipr.matrix.Matrix A, double[]
times, no.uib.cipr.matrix.Vector leftVec, no.uib.cipr.matrix.Vector rightVec,
int truncate )
```

- **Description**

Computes $\text{leftVec} * \exp(A x) * \text{rightVec}$, for all values x in times. It uses the uniformization algorithm

- **Parameters**

- * **A** – Matrix
- * **times** – evaluation points
- * **leftVec** – Vector
- * **rightVec** – Vector
- * **truncate** – upper bound for iterations

- **Returns** – $\text{leftVec} * \exp(A x) * \text{rightVec}$

- *expUnif*

```
public static no.uib.cipr.matrix.Matrix[] expUnif( no.uib.cipr.matrix.Matrix
A, double x, no.uib.cipr.matrix.Matrix leftMat, no.uib.cipr.matrix.Matrix
rightMat )
```

- **Description**

Computes $\text{leftMat} * \exp(A x) * \text{rightMat}$, for the value x . It uses the uniformization algorithm as described in page 60 of Latouche and Ramaswami

- **Parameters**

- * **A** – Matrix
- * **x** – evaluation point
- * **leftMat** – Matrix
- * **rightMat** – Matrix

- **Returns** – $\text{leftMat} * \exp(A x) * \text{rightMat}$

- *expUnif*

```
public static no.uib.cipr.matrix.Matrix[] expUnif( no.uib.cipr.matrix.Matrix
A, int n, double delta, no.uib.cipr.matrix.Matrix leftMat,
no.uib.cipr.matrix.Matrix rightMat )
```

- **Description**

Computes $\text{leftMat} * \exp(A x) * \text{rightMat}$, for all values $x = 0 + i \cdot \text{delta}$, $i=0, \dots, n$. It uses the uniformization algorithm as described in page 60 of Latouche and Ramaswami

- **Parameters**

- * **A** – Matrix
- * **n** – number of evaluation points
- * **delta** – separation between evaluation points
- * **leftMat** – Matrix
- * **rightMat** – Matrix

- **Returns** – $\text{leftMat} * \exp(A x) * \text{rightMat}$

- *expUnif*

```
public static no.uib.cipr.matrix.Matrix[] expUnif( no.uib.cipr.matrix.Matrix
A, int n, double delta, no.uib.cipr.matrix.Matrix leftMat,
no.uib.cipr.matrix.Matrix rightMat, int truncate )
```

- **Description**

Computes $\text{leftMat} * \exp(A x) * \text{rightMat}$, for all values $x = 0 + i \cdot \text{delta}$, $i=0, \dots, n$. It uses the uniformization algorithm

- **Parameters**

- * **A** – Matrix

- * **n** – num of evaluation points
 - * **delta** – evaluation points separation
 - * **leftMat** – Matrix
 - * **rightMat** – Matrix
 - * **truncate** – upper bound for iterations
 - **Returns** – $\text{leftMat} * \exp(A x) * \text{rightMat}$
-

- *expUnif*

```
public static double[] expUnif( no.uib.cipr.matrix.Matrix A, int n, double
delta, no.uib.cipr.matrix.Vector leftVec, no.uib.cipr.matrix.Vector rightVec
)
```

- **Description**

Computes $\text{leftVec} * \exp(A x) * \text{rightVec}$, for all values $x = 0 + i*\text{delta}$, $i=0,\dots,n$. It uses the uniformization algorithm as described in page 60 of Latouche and Ramaswami

- **Parameters**

- * **A** – Matrix
- * **n** – number of evaluation points
- * **delta** – separation between evaluation points
- * **leftVec** – Vector
- * **rightVec** – Vector

- **Returns** – $\text{leftVec} * \exp(A x) * \text{rightVec}$
-

- *expUnif*

```
public static double[] expUnif( no.uib.cipr.matrix.Matrix A, int n, double
delta, no.uib.cipr.matrix.Vector leftVec, no.uib.cipr.matrix.Vector rightVec,
int truncate )
```

- **Description**

Computes $\text{leftVec} * \exp(A x) * \text{rightVec}$, for all values $x = 0 + i*\text{delta}$, $i=0,\dots,n$. It uses the uniformization algorithm

- **Parameters**

- * **A** – Matrix
- * **n** – num of evaluation points
- * **delta** – evaluation points separation
- * **leftVec** – Vector
- * **rightVec** – Vector
- * **truncate** – upper bound for iterations

- **Returns** – $\text{leftVec} * \exp(A x) * \text{rightVec}$
-

- *kronecker*

```
public static no.uib.cipr.matrix.Matrix kronecker( no.uib.cipr.matrix.Matrix
A, no.uib.cipr.matrix.Matrix B )
```

- **Description**

Returns the kronecker product of two matrices in dense format

- **Parameters**

- * **A** – Matrix
- * **B** – Matrix

- **Returns** – Kronecker Product $A \times B$
-

- *kronecker*

```
public static no.uib.cipr.matrix.Matrix kronecker( no.uib.cipr.matrix.Matrix
A, no.uib.cipr.matrix.Matrix B, no.uib.cipr.matrix.Matrix res )
```

- **Description**

Returns the kronecker product of two matrices in the predefined storage format

- **Parameters**

- * A – Matrix

- * B – Matrix

- * **res** – Matrix such that $\text{res.numCols} = \text{A.numCols} * \text{B.numCols}$ and $\text{res.numRows} = \text{A.numRows} * \text{B.numRows}$

- **Returns** – Kronecker Product $A \times B$

- *kronecker*

```
public static no.uib.cipr.matrix.Matrix kronecker( no.uib.cipr.matrix.Matrix
A, no.uib.cipr.matrix.Vector B, no.uib.cipr.matrix.Matrix res )
```

- **Description**

Returns the kronecker product of one matrix with one vector (Matrix x Vector) in dense format

- **Parameters**

- * A – Matrix

- * B – Vector

- * **res** – Matrix such that $\text{res.numRows} = \text{A.numRows}$ and $\text{res.numCols} = \text{A.numCols} * \text{B.size}$

- **Returns** – Kronecker Product $A \times B$

- *kronecker*

```
public static no.uib.cipr.matrix.Matrix kronecker( no.uib.cipr.matrix.Vector
A, no.uib.cipr.matrix.Matrix B, no.uib.cipr.matrix.Matrix res )
```

- **Description**

Returns the kronecker product of one vector with one matrix (Vector x Matrix) in dense format

- **Parameters**

- * A – Vector

- * B – Matrix

- * **res** – Matrix such that $\text{res.numRows} = \text{A.size} * \text{B.numRows}$ and $\text{res.numCols} = \text{B.numCols}$

- **Returns** – Kronecker Product $A \times B$

- *kroneckerSum*

```
public static no.uib.cipr.matrix.Matrix kroneckerSum(
no.uib.cipr.matrix.Matrix A, no.uib.cipr.matrix.Matrix B )
```

- **Description**

Returns the kronecker sum of two matrices in dense format

- **Parameters**

- * A – Matrix

- * B – Matrix

- **Returns** – Kronecker Sum $A + B$

- *kroneckerSum*

```
public static no.uib.cipr.matrix.Matrix kroneckerSum(
no.uib.cipr.matrix.Matrix A, no.uib.cipr.matrix.Matrix B,
no.uib.cipr.matrix.Matrix res )
```

- **Description**

Returns the kronecker sum of two matrices and stores it in the predefined format

- **Parameters**

- * A – Matrix
 - * B – Matrix
 - * **res** – Result Matrix such that $\text{res.numRows} = \text{A.numRows} * \text{B.numRows}$ and $\text{res.numCols} = \text{A.numCols} * \text{B.numCols}$

- **Returns** – Kronecker Sum $A + B$

- *kroneckerVectors*

```
public static no.uib.cipr.matrix.DenseVector kroneckerVectors(
no.uib.cipr.matrix.DenseVector A, no.uib.cipr.matrix.DenseVector B )
```

- **Description**

Returns the kronecker product of two vectors in dense format

- **Parameters**

- * A – Vector
 - * B – Vector

- **Returns** – Kronecker Product $A \times B$

- *kroneckerVectors*

```
public static no.uib.cipr.matrix.Vector kroneckerVectors(
no.uib.cipr.matrix.Vector A, no.uib.cipr.matrix.Vector B,
no.uib.cipr.matrix.Vector res )
```

- **Description**

Returns the kronecker product of two vectors in the predefined format

- **Parameters**

- * A – Vector
 - * B – Vector
 - * **res** – Vector such that $\text{res.size} = \text{A.size} * \text{B.size}$

- **Returns** – Kronecker Product $A \times B$

- *matPower*

```
public static no.uib.cipr.matrix.Matrix matPower( no.uib.cipr.matrix.Matrix
A, int k )
```

- **Description**

Computes k power of the matrix A

- **Parameters**

- * A – matrix base
 - * k – exponent

- **Returns** – A^k

- *matPower*

```
public static double matPower( no.uib.cipr.matrix.Matrix A, int k,
no.uib.cipr.matrix.Vector leftVec, no.uib.cipr.matrix.Vector rightVec )
```

- **Description**

Computes the kth power of the matrix A premultiplied by leftVec and postmultiplied by rightVec

- **Parameters**

- * A – Matrix base
- * k – exponent
- * leftVec – Vector
- * rightVec – Vector

- **Returns** – leftVec * A^k * rightVec

- *multVector*

```
public static no.uib.cipr.matrix.Matrix multVector(
no.uib.cipr.matrix.Vector A, no.uib.cipr.matrix.Vector B,
no.uib.cipr.matrix.Matrix res )
```

- **Description**

Computes the product of two vectors A x B^T

- **Parameters**

- * A – Vector
- * B – Vector
- * res – Vector to store the resulting matrix

- **Returns** – res = A x B^T

- *OnesCol*

```
public static no.uib.cipr.matrix.DenseMatrix OnesCol( int m )
```

- **Description**

Returns a one-column matrix in dense format with one in every entry

- **Parameters**

- * m – size of the matrix (m, 1)

- **Returns** – One column matrix with one in every entry

- *OnesRow*

```
public static no.uib.cipr.matrix.DenseMatrix OnesRow( int m )
```

- **Description**

Returns a one-row matrix in dense format with one in every entry

- **Parameters**

- * m – size of the matrix (1, m)

- **Returns** – Row matrix with one in every entry

- *OnesVector*

```
public static no.uib.cipr.matrix.DenseVector OnesVector( int m )
```

- **Description**

Returns a DenseVector with one in every entry

- **Parameters**

- * *m* – size of the DenseVector
 - **Returns** – DenseVector with one in every entry
-

- *OnesVector*

```
public static no.uib.cipr.matrix.Vector OnesVector(
no.uib.cipr.matrix.Vector vec )
```

- **Description**
Returns a Vector with one in every entry in the predefined storage format
 - **Parameters**
* *vec* – Vector to be modified
 - **Returns** – Vector with one in every entry
-

- *pow*

```
public static double pow( double x, int n )
```

- **Description**
Calculates x^n
 - **Parameters**
* *x* – base
* *n* – exponent
 - **Returns** – x^n
-

- *scalar*

```
public static double scalar( no.uib.cipr.matrix.Matrix A )
```

- **Description**
Returns the value of the position (0,0) in the matrix, if its number of columns is equal to one (1)
 - **Parameters**
* *A* – matrix
 - **Returns** – Value of the position (0,0) in the matrix
-

- *sumMatPower*

```
public static double sumMatPower( no.uib.cipr.matrix.Matrix A, int k,
no.uib.cipr.matrix.Vector leftVec, no.uib.cipr.matrix.Vector rightVec )
```

- **Description**
Computes the sum of the first *k* terms of the succession $T^{(j-1)}$, from $j = 1$
 - **Parameters**
* *A* – Matrix base
* *k* – maximum exponent
* *leftVec* –
* *rightVec* –
 - **Returns** – $\text{leftVec} * \sum_{j=1}^k T^{(j-1)} * \text{rightVec}$
-

- *variance*

```
public static double variance( double[] data )
```

- **Description**
Return the variance of the data

- **Parameters**
 - * **data** – data trace
- **Returns** – Data Variance

7.2.9 Class PhaseVarSet

Declaration

```
public class PhaseVarSet
  extends java.lang.Object
  implements java.io.Serializable
```

Field summary

fileName
isDirty
name

Constructor summary

PhaseVarSet() Default constructor
PhaseVarSet(PhaseVar[]) Constructs a new set with specified variables
PhaseVarSet(String) Constructs a new set with specified name
PhaseVarSet(String, PhaseVar[]) Constructs a new set with specified name and variables

Method summary

add(PhaseVar)
getMeans() Returns a vector with the means of all elements
indexOfName(String) Returns the index in the det of the variables with the specified name
newUniqueVarName(String) Builds a unique name for a variable from a proposed name
numVars() Returns the number of variables in the set
open(String) Reads a .sed file with the information of a set
openTxt(String) Reads a .txt file with the information of a set
remove(PhaseVar) Remove the specified variable
remove(String) Removes the variable with specified name
save()
save(String)
saveTxt() Saves the set information in a file
saveTxt(String) Saves the set information in a file
toString()
varAt(int) Returns the variable at index i

Serializable Fields

- private java.util.List **vars**
 - Variables in the set
- private java.util.List **names**
 - Names of the variables in the set

- public java.lang.String **name**
- public java.lang.String **fileName**
- public boolean **isDirty**

Fields

- public java.lang.String **name**
- public java.lang.String **fileName**
- public boolean **isDirty**

Constructors

- *PhaseVarSet*
public **PhaseVarSet**()

– **Description**
Default constructor

- *PhaseVarSet*
public **PhaseVarSet**(PhaseVar[] **vars**)

– **Description**
Constructs a new set with specified variables
– **Parameters**
* **vars** – Set variables

- *PhaseVarSet*
public **PhaseVarSet**(java.lang.String **nam**)

– **Description**
Constructs a new set with specified name
– **Parameters**
* **nam** – Set name

- *PhaseVarSet*
public **PhaseVarSet**(java.lang.String **nam**, PhaseVar[] **vars**)

– **Description**
Constructs a new set with specified name and variables
– **Parameters**
* **nam** – Set name
* **vars** – Set variables

Methods

- *add*
`public void add(PhaseVar var)`
 - **Parameters**
 - * `var` –

- *getMeans*
`public double[] getMeans()`
 - **Description**
Returns a vector with the means of all elements
 - **Returns** – Vector with the means of all elements

- *indexOfName*
`public int indexOfName(java.lang.String s)`
 - **Description**
Returns the index in the det of the variables with the specified name
 - **Parameters**
 - * `s` – Name to be evaluated
 - **Returns** – Index in the det of the variables with the specified name

- *newUniqueVarName*
`public java.lang.String newUniqueVarName(java.lang.String proposedName)`
 - **Description**
Builds a unique name for a variable from a proposed name
 - **Parameters**
 - * `proposedName` – proposed name
 - **Returns** – Unique name for a variable from a proposed name

- *numVars*
`public int numVars()`
 - **Description**
Returns the number of variables in the set
 - **Returns** – Number of variables in the set

- *open*
`public static PhaseVarSet open(java.lang.String fileName) throws java.lang.Exception`
 - **Description**
Reads a .sed file with the information of a set
 - **Parameters**
 - * `fileName` – File with the variable set
 - **Returns** – Set of variables in the file

- **Throws**
 - * `java.lang.Exception` – `IOException`

- *openTxt*

```
public static PhaseVarSet openTxt( java.lang.String fileName ) throws
java.lang.Exception
```

 - **Description**

Reads a .txt file with the information of a set
 - **Parameters**
 - * `fileName` – File with the variable set
 - **Returns** – Set of variables in the file
 - **Throws**
 - * `java.lang.Exception` – `IOException`

 - *remove*

```
public void remove( PhaseVar var )
```

 - **Description**

Remove the specified variable
 - **Parameters**
 - * `var` – variable to remove

 - *remove*

```
public int remove( java.lang.String varName )
```

 - **Description**

Removes the variable with specified name
 - **Parameters**
 - * `varName` – name of the variable to remove
 - **Returns** – index of the removed variable

 - *save*

```
public void save( ) throws java.io.IOException
```

 - **Throws**
 - * `java.io.IOException` –

 - *save*

```
public void save( java.lang.String fileName ) throws java.io.IOException
```

 - **Parameters**
 - * `fileName` –
 - **Throws**
 - * `java.io.IOException` –

 - *saveTxt*

```
public boolean saveTxt( ) throws java.io.IOException
```

 - **Description**

Saves the set information in a file
 - **Returns** – True if the file could be saved, false elsewhere

– **Throws**

* java.io.IOException –

• *saveTxt*

public boolean saveTxt(java.lang.String fileName) throws
java.io.IOException

– **Description**

Saves the set information in a file

– **Parameters**

* fileName – File name

– **Returns** – True if the file could be saved, false elsewhere

– **Throws**

* java.io.IOException –

• *toString*

public java.lang.String toString()

• *varAt*

public PhaseVar varAt(int i)

– **Description**

Returns the variable at index i

– **Parameters**

* i – Index of the required variables

– **Returns** – Variable at index i

7.2.10 Class Poly

This class represents a polynomial.

Declaration

```
public class Poly
  extends java.lang.Object
```

Constructor summary

Poly() ‘ * Creates a Polynomial = 0.0.
Poly(double[]) Build a Polynomial with these coefficients

Method summary

addTerm(double, int) Adds this term: $cf * t^n$
evaluate(double) Returns the value of this polynomial at x.

Constructors

- *Poly*
 public **Poly**()
 – **Description**
 ‘ * Creates a Polynomial = 0.0.

- *Poly*
 public **Poly**(double[] **cf**)
 – **Description**
 Build a Polynomial with these coefficients
 – **Parameters**
 * **cf** – Coefficients

Methods

- *addTerm*
 public Poly **addTerm**(double **coeff**, int **n**)
 – **Description**
 Adds this term: $cf * t^n$
 – **Parameters**
 * **coeff** – Coefficient of the term to be added
 * **n** – Exponent of the term to be added
 – **Returns** – Modified Polynomial

- *evaluate*
 public double **evaluate**(double **x**)

- **Description**
Returns the value of this polynomial at x .
- **Parameters**
 - * x – evaluation point
- **Returns** – Value of this polynomial at x .

7.2.11 Class SparseContPhaseVar

Declaration

```
public class SparseContPhaseVar
extends jphase.AbstractContPhaseVar (see 7.2.1, page 270)
```

Version

0.1 This class allows the creation and manipulation of Continuous Phase-type distributions represented by sparse (Flexible Compressed Row) matrices.

Field summary

A Rate Matrix in Sparse representation (CompRowMatrix)
alpha Initial Probability distribution vector

Constructor summary

SparseContPhaseVar(double[], double[][]) Construcs a continuous Phase-type Distribution with sparse representation (CompRowMatrix)
SparseContPhaseVar(int) Construcs an empty Continuous Phase-type Distribution of size n with sparse representation (CompRowMatrix)
SparseContPhaseVar(SparseVector, FlexCompRowMatrix) Construcs a continuous Phase-type Distribution with sparse representation (CompRowMatrix)
SparseContPhaseVar(Vector, Matrix) Construcs a continuous Phase-type Distribution with sparse representation (CompRowMatrix)

Method summary

copy()
getMatrix()
getVector()
newVar(int)
setMatrix(Matrix)
setVector(Vector)

Fields

- protected no.uib.cipr.matrix.sparse.FlexCompRowMatrix **A**
 - Rate Matrix in Sparse representation (CompRowMatrix)
- protected no.uib.cipr.matrix.sparse.SparseVector **alpha**
 - Initial Probability distribution vector

Constructors

- *SparseContPhaseVar*
public SparseContPhaseVar(double[] alpha, double[][] A)
 - **Description**
 Construcs a continuous Phase-type Distribution with sparse representation (CompRowMatrix)
 - **Parameters**
 - * A – rate matrix
 - * alpha – initial probability distribution vector

- *SparseContPhaseVar*
public SparseContPhaseVar(int n)
 - **Description**
 Construcs an empty Continuous Phase-type Distribution of size n with sparse representation (CompRowMatrix)
 - **Parameters**
 - * n – size of the Continuous Phase-type Distribution

- *SparseContPhaseVar*
public SparseContPhaseVar(no.uib.cipr.matrix.sparse.SparseVector alpha, no.uib.cipr.matrix.sparse.FlexCompRowMatrix A)
 - **Description**
 Construcs a continuous Phase-type Distribution with sparse representation (CompRowMatrix)
 - **Parameters**
 - * A – rate matrix
 - * alpha – initial probability distribution vector

- *SparseContPhaseVar*
public SparseContPhaseVar(no.uib.cipr.matrix.Vector alpha, no.uib.cipr.matrix.Matrix A)
 - **Description**
 Construcs a continuous Phase-type Distribution with sparse representation (CompRowMatrix)
 - **Parameters**
 - * A – rate matrix
 - * alpha – initial probability distribution vector

Methods

- *copy*
public ContPhaseVar copy()

- *getMatrix*
public no.uib.cipr.matrix.Matrix getMatrix()

- *getVector*
public no.uib.cipr.matrix.Vector getVector()
- *newVar*
public ContPhaseVar newVar(int n)
- *setMatrix*
public void setMatrix(no.uib.cipr.matrix.Matrix A)
- *setVector*
public void setVector(no.uib.cipr.matrix.Vector alpha)

Members inherited from class `jphase.AbstractContPhaseVar` (see 7.2.1, page 270)

- public double `cdf(double x)`
- public double `cdf(int n, double delta)`
- public double `CV()`
- public String `description()`
- public ContPhaseVar `eqResidualTime()`
- public double `expectedValue()`
- public Vector `getMat0()`
- public double `getMat0Array()`
- public double `getMatrixArray()`
- public int `getNumPhases()`
- public double `getVec0()`
- public double `getVectorArray()`
- public String `label()`
- public double `lossFunction1(double x)`
- public double `lossFunction2(double x)`
- public ContPhaseVar `max(ContPhaseVar v2)`
- public ContPhaseVar `max(ContPhaseVar v2, ContPhaseVar res)`
- public double `median()`
- public ContPhaseVar `min(ContPhaseVar v2)`
- public ContPhaseVar `min(ContPhaseVar v2, ContPhaseVar res)`
- public ContPhaseVar `mix(double p, ContPhaseVar v2)`
- public ContPhaseVar `mix(double p, ContPhaseVar v2, ContPhaseVar res)`
- public double `moment(int k)`
- public double `pdf(double x)`
- public double `pdf(int n, double delta)`
- public double `prob(double a, double b)`
- public double `quantil(double p)`
- public ContPhaseVar `residualTime(double x)`
- public ContPhaseVar `residualVar(double a)`
- public double `stdDeviation()`
- public ContPhaseVar `sum(ContPhaseVar v2)`
- public ContPhaseVar `sum(ContPhaseVar v2, ContPhaseVar res)`
- public ContPhaseVar `sumGeom(double p)`
- public ContPhaseVar `sumPH(DiscPhaseVar v2)`
- public ContPhaseVar `sumPH(DiscPhaseVar B, ContPhaseVar res)`
- public double `survival(double x)`
- public double `survival(int n, double delta)`
- public ContPhaseVar `times(double c)`
- public final String `toString()`
- public double `variance()`
- public ContPhaseVar `waitingQ(double rho)`

7.2.12 Class SparseDiscPhaseVar

Declaration

```
public class SparseDiscPhaseVar
  extends jphase.AbstractDiscPhaseVar (see 7.2.2, page 276)
```

Version

0.1 This class allows the creation and manipulation of Discrete Phase-type distributions represented by sparse (Flexible Compressed Row) matrices.

Field summary

A Transition Matrix in Sparse representation (FlexCompRowMatrix)
alpha Initial Probability distribution vector

Constructor summary

SparseDiscPhaseVar(double[], double[][]) Construcs a discrete Phase-type Distribution with sparse representation (FlexCompRowMatrix)
SparseDiscPhaseVar(int) Construcs an empty Discrete Phase-type Distribution of size n with sparse representation (FlexCompRowMatrix)
SparseDiscPhaseVar(SparseVector, FlexCompRowMatrix) Construcs a discrete Phase-type Distribution with sparse representation (FlexCompRowMatrix)
SparseDiscPhaseVar(Vector, Matrix) Construcs a discrete Phase-type Distribution with sparse representation (FlexCompRowMatrix)

Method summary

copy()
getMatrix()
getVector()
newVar(int)
setMatrix(Matrix)
setVector(Vector)

Fields

- protected no.uib.cipr.matrix.sparse.FlexCompRowMatrix **A**
 - Transition Matrix in Sparse representation (FlexCompRowMatrix)
- protected no.uib.cipr.matrix.sparse.SparseVector **alpha**
 - Initial Probability distribution vector

Constructors

- *SparseDiscPhaseVar*

```
public SparseDiscPhaseVar( double[] alpha, double[] [] A )
```

- **Description**

Construcs a discrete Phase-type Distribution with sparse representation (FlexCompRowMatrix)

- **Parameters**

- * A – transtion probability matrix
 - * alpha – initial probability distribution vector
-

- *SparseDiscPhaseVar*

```
public SparseDiscPhaseVar( int n )
```

- **Description**

Construcs an empty Discrete Phase-type Distribution of size n with sparse representation (FlexCompRowMatrix)

- **Parameters**

- * n – size of the Discrete Phase-type Distribution
-

- *SparseDiscPhaseVar*

```
public SparseDiscPhaseVar( no.uib.cipr.matrix.sparse.SparseVector alpha,  
no.uib.cipr.matrix.sparse.FlexCompRowMatrix A )
```

- **Description**

Construcs a discrete Phase-type Distribution with sparse representation (FlexCompRowMatrix)

- **Parameters**

- * A – transition probability matrix
 - * alpha – initial probability distribution vector
-

- *SparseDiscPhaseVar*

```
public SparseDiscPhaseVar( no.uib.cipr.matrix.Vector alpha,  
no.uib.cipr.matrix.Matrix A )
```

- **Description**

Construcs a discrete Phase-type Distribution with sparse representation (FlexCompRowMatrix)

- **Parameters**

- * A – transition probability matrix
- * alpha – initial probability distribution vector

Methods

- *copy*

```
public DiscPhaseVar copy( )
```

- **See also**

- * ContPhaseVar.copy() (see 7.1.1, page 257)

-
- *getMatrix*
`public no.uib.cipr.matrix.Matrix getMatrix()`
 – See also
 * `PhaseVar.getMatrix()` (see 7.1.3, page 267)

 - *getVector*
`public no.uib.cipr.matrix.Vector getVector()`
 – See also
 * `PhaseVar.getVector()` (see 7.1.3, page 267)

 - *newVar*
`public DiscPhaseVar newVar(int n)`

 - *setMatrix*
`public void setMatrix(no.uib.cipr.matrix.Matrix A)`
 – See also
 * `PhaseVar.setMatrix(Matrix)` (see 7.1.3, page 269)

 - *setVector*
`public void setVector(no.uib.cipr.matrix.Vector alpha)`
 – See also
 * `PhaseVar.setVector(Vector)` (see 7.1.3, page 269)

Members inherited from class `jphase.AbstractDiscPhaseVar` (see 7.2.2, page 276)

- `public double cdf(double x)`
- `public double cdf(int n, double delta)`
- `public double CV()`
- `public String description()`
- `public double expectedValue()`
- `public Vector getMat0()`
- `public double getMat0Array()`
- `public double getMatrixArray()`
- `public int getNumPhases()`
- `public double getVec0()`
- `public double getVectorArray()`
- `public String label()`
- `public double lossFunction1(double x)`
- `public double lossFunction2(double x)`
- `public DiscPhaseVar max(DiscPhaseVar v2)`
- `public DiscPhaseVar max(DiscPhaseVar v2, DiscPhaseVar res)`
- `public double median()`
- `public DiscPhaseVar min(DiscPhaseVar v2)`
- `public DiscPhaseVar min(DiscPhaseVar v2, DiscPhaseVar res)`
- `public DiscPhaseVar mix(double p, DiscPhaseVar v2)`
- `public DiscPhaseVar mix(double p, DiscPhaseVar v2, DiscPhaseVar res)`
- `public double moment(int k)`
- `public double pmf(int k)`
- `public double pmf(int n, int delta)`
- `public double prob(double a, double b)`
- `public double quantil(double p)`
- `public double stdDeviation()`
- `public DiscPhaseVar sum(DiscPhaseVar v2)`
- `public DiscPhaseVar sum(DiscPhaseVar v2, DiscPhaseVar res)`
- `public DiscPhaseVar sumGeom(double p)`
- `public DiscPhaseVar sumPH(DiscPhaseVar v2)`
- `public DiscPhaseVar sumPH(DiscPhaseVar B, DiscPhaseVar res)`
- `public double survival(double x)`
- `public double survival(int n, double delta)`
- `public final String toString()`
- `public double variance()`

7.2.13 Class SuperErlang

Declaration

```
public class SuperErlang
  extends java.lang.Object
  implements java.io.Serializable
```

Field summary

ONE The number one (1)

Constructor summary

SuperErlang() $f(x) = 0.0$
SuperErlang(double, int, double) $f(x) = \text{cf } E(n, \text{lbd})$
SuperErlang(Term) $f(x) = t$

Method summary

addTerm(double, int, double)
addTerm(Term)
clone() Clones this function
convolution(SuperErlang, SuperErlang) Return the convolution of this two functions
defIntegrate() Returns the integral from 0 to infinity of this function.
defIntegrate(double) Returns the definite integral from 0 to x of this function
derive() Returns the derivative at t of this function.
exp()
expand(double) Evaluates $f(t)$ at $a \cdot t$.
integrate() Returns the integral from 0 to t of this function.
integrateCom() Returns the integral from t to infinity of this function.
isZero() Determines whether this function is identically equal to 0
moment(int)
move(double) Evaluates the function at $t+a$
multiply(SuperErlang) Multiply the function f2 with this function
multiply(SuperErlang, SuperErlang) Return the product of this two functions
numTerms() Returns the number of terms.
poly(double, int) Return a monomy $c \cdot t^n$
sum(SuperErlang) Sums the function f2 to this function
sum(SuperErlang, SuperErlang) Return the sum of this two functions
term(int) Returns the i-th term.
times(double) Returns this function times the constant
toString()
toStringE() String representation using the notation $p1E(n1, a2) + p2E(n2, a2) + \dots$
 where $E(n, a) = a^n x^{n-1} e^{-a x} / (n-1)!$, is an Erlang pdf.
toStringP() String representation using the notation $p1P(n1, a2) + p2P(n2, a2) + \dots$
 where $E(n, a) = (a x)^n e^{-a x} / n!$, is a poisson cdf..
toStringRTF() String representation in RTF

Serializable Fields

- private java.util.List **v**

Fields

- public static SuperErlang **ONE**
 - The number one (1)

Constructors

- *SuperErlang*
public **SuperErlang**()
 - **Description**
 $f(x) = 0.0$
- *SuperErlang*
public **SuperErlang**(double **cf**, int **n**, double **lbd**)
 - **Description**
 $f(x) = cf \text{ E}(n, lbd)$
 - **Parameters**
 - * **cf** –
 - * **n** –
 - * **lbd** –
- *SuperErlang*
public **SuperErlang**(Term **trm**)
 - **Description**
 $f(x) = t$
 - **Parameters**
 - * **trm** –

Methods

- *addTerm*
public SuperErlang **addTerm**(double **coeff**, int **power**, double **lmb**)
 - **Parameters**
 - * **coeff** –
 - * **power** –
 - * **lmb** –
 - **Returns** – $f(x) := f(x) + \text{coeff E}(\text{power}, \text{lmb})$

- *addTerm*

```
public SuperErlang addTerm( Term tr )
```

 - **Parameters**
 - * **tr** –
 - **Returns** – $f(x) := f(x) + tr$

- *clone*

```
public java.lang.Object clone( )
```

 - **Description**
Clones this function

- *convolution*

```
public static SuperErlang convolution( SuperErlang f1, SuperErlang f2 )
```

 - **Description**
Return the convolution of this two functions
 - **Parameters**
 - * **f1** –
 - * **f2** –
 - **Returns** – $f1 * f2(x)$

- *defIntegrate*

```
public double defIntegrate( )
```

 - **Description**
Returns the integral from 0 to infinity of this function.
 - **Returns** – integral value

- *defIntegrate*

```
public double defIntegrate( double x )
```

 - **Description**
Returns the definite integral from 0 to x of this function
 - **Parameters**
 - * **x** –
 - **Returns** – $\int_0^x f(t)dt$

- *derive*

```
public SuperErlang derive( )
```

 - **Description**
Returns the derivative at t of this function.
 - **Returns** – $f'(x)$

- *exp*

```
protected double exp( )
```

 - **Returns** – expected value for the given pdf

- *expand*

```
public SuperErlang expand( double a )
```

- **Description**
Evaluates $f(t)$ at $a \cdot t$.
 - **Parameters**
* a –
 - **Returns** – $f(a \cdot x)$
-

- *integrate*

public SuperErlang **integrate**()

- **Description**
Returns the integral from 0 to t of this function.
 - **Returns** – $\int_0^t f(t) dt$
-

- *integrateCom*

public SuperErlang **integrateCom**()

- **Description**
Returns the integral from t to infinity of this function.
 - **Returns** – $\int_t^\infty f(t) dt$
-

- *isZero*

public boolean **isZero**()

- **Description**
Determines whether this function is identically equal to 0
 - **Returns** – true if $f(x) == 0.0$
-

- *moment*

protected double **moment**(int k)

- **Parameters**
* k –
 - **Returns** – k -th moment
-

- *move*

public SuperErlang **move**(double a)

- **Description**
Evaluates the function at $t+a$
 - **Parameters**
* a –
 - **Returns** – $f(t+a)$
-

- *multiply*

public SuperErlang **multiply**(SuperErlang $f2$)

- **Description**
Multiply the function $f2$ with this function
- **Parameters**
* $f2$ –
- **Returns** – $f(x) \times f2(x)$

-
- *multiply*
 public static SuperErlang **multiply**(SuperErlang **f1**, SuperErlang **f2**)
 – **Description**
 Return the product of this two functions
 – **Parameters**
 * **f1** –
 * **f2** –
 – **Returns** – $f1(x) \times f2(x)$

 - *numTerms*
 public int **numTerms**()
 – **Description**
 Returns the number of terms.
 – **Returns** – Number of terms

 - *poly*
 public static SuperErlang **poly**(double **coef**, int **n**)
 – **Description**
 Return a monomy $c t^n$
 – **Parameters**
 * **coef** –
 * **n** –
 – **Returns** – $c t^n$

 - *sum*
 public SuperErlang **sum**(SuperErlang **f2**)
 – **Description**
 Sums the function **f2** to this function
 – **Parameters**
 * **f2** –
 – **Returns** – $f(x) := f(x) + f2(x)$

 - *sum*
 public static SuperErlang **sum**(SuperErlang **f1**, SuperErlang **f2**)
 – **Description**
 Return the sum of this two functions
 – **Parameters**
 * **f1** –
 * **f2** –
 – **Returns** – $f1+f2$

 - *term*
 public Term **term**(int **i**)
 – **Description**
 Returns the i-th term.

– **Parameters**

* *i* – The 0based index.

– **Returns** – The *i*-th term

• *times*

public SuperErlang times(double cons)

– **Description**

Returns this function times the constant

– **Parameters**

* *cons* –

– **Returns** – $\text{cons} * f(x)$

• *toString*

public java.lang.String toString()

• *toStringE*

public java.lang.String toStringE()

– **Description**

String representation using the notation $p1E(n1,a2) + p2E(n2,a2) + \dots$ where $E(n,a) = a^n x^{n-1} e^{-a} / (n-1)!$, is an Erlang pdf.

– **Returns** – Erlang type representation.

• *toStringP*

public java.lang.String toStringP()

– **Description**

String representation using the notation $p1P(n1,a2) + p2P(n2,a2) + \dots$ where $E(n,a) = (a x)^n e^{-a} / n!$, is a poisson cdf..

– **Returns** – Erlang type representation.

• *toStringRTF*

public java.lang.String toStringRTF()

– **Description**

String representation in RTF

– **Returns** – string

7.2.14 Class Term

This class describes a basic term in a super-Erlang function. It has the form: $\alpha \lambda^n t^{n-1} \exp(-\lambda t)$ or $\alpha * R(n, \lambda)$ where $R(n, \lambda)$ is an Erlang pdf.

Declaration

```
public class Term
  extends java.lang.Object
  implements java.io.Serializable
```

Field summary

ONE Number 1.0

Constructor summary

Term(double, int, double) Creates a term $cf * t^p \exp(-\lambda * t)$

Method summary

clone() Clones this term
compareTo(Term) Compares according to lambda, power in that order.
convolution(Term, Term) Return the convolution of this two terms
convolutionUseExp(Term, Term)
defIntegrate() Returns the integral from 0 to infinity
defIntegrate(double) Returns the integral from 0 to x
derive() Returns the derivative at t
equalPowers(Term)
evaluate() Evaluates this term at infinity
evaluate(double) Evaluates this term at the value x
expand(double) Return this term evaluated at t^a
getCoeff() Returns the value of the coefficient
getDegree() Returns the value of the degree
getLambda() Returns the value of lambda
integrate() Returns the integral from 0 to t
integrateCom() Returns the integral from t to infinity
isConstant() Tells if it is constant.
isPTerm() Tells if the term is a polynomial
isZero() Tells if it is identically = 0.
moment(int) Computes the k-th moment
move(double) Return this term evaluated at $t+a$
multiply(Term) Return the product of this term with tr
multiply(Term, Term) Return the product of this two terms
setCoeff(double) Sets the value of the coefficient
setDegree(int) Returns the value of the degree
setLambda(double) Returns the value of the coefficient
toString()
toStringE() Represents the term as a multiple of an Erlang pdf.
toStringP() String representation of the Polynomial
toStringRTF() RTF representation of the term

Serializable Fields

- private double **coeff**
- private int **power**
- private double **lambda**

Fields

- public static Term **ONE**
 - Number 1.0

Constructors

- *Term*
 public **Term**(double **cf**, int **pw**, double **lb**)
 - **Description**
 Creates a term $cf * t^{pw} \exp(-lb * t)$
 - **Parameters**
 - * **cf** – coefficient
 - * **pw** – power
 - * **lb** – lambda

Methods

- *clone*
 public java.lang.Object **clone**()
 - **Description**
 Clones this term
- *compareTo*
 public int **compareTo**(Term **tr**)
 - **Description**
 Compares according to lambda,power in that order.
 - **Parameters**
 - * **tr** – Term to compare to
 - **Returns** – establishes an ordering among terms
- *convolution*
 public static SuperErlang **convolution**(Term **term1**, Term **term2**)
 - **Description**
 Return the convolution of this two terms

- **Parameters**
 - * **term1** – First term
 - * **term2** – Second term
 - **Returns** – SuperErlang representation the convolution of this two terms
-
- *convolutionUseExp*
 public static SuperErlang **convolutionUseExp**(Term **term1**, Term **term2**)
 - **Parameters**
 - * **term1** –
 - * **term2** –
 - **Returns** – Convolution of Term1 and Term2
-
- *defIntegrate*
 public double **defIntegrate**()
 - **Description**
 Returns the integral from 0 to infinity
 - **Returns** – Value of the integral from 0 to infinity
-
- *defIntegrate*
 public double **defIntegrate**(double **x**)
 - **Description**
 Returns the integral from 0 to x
 - **Parameters**
 - * **x** – upper limit of the integral
 - **Returns** – Value of the integral from 0 to x
-
- *derive*
 public SuperErlang **derive**()
 - **Description**
 Returns the derivative at t
 - **Returns** – SuperErlang representation of the derivative at t
-
- *equalPowers*
 public boolean **equalPowers**(Term **tr**)
 - **Parameters**
 - * **tr** –
 - **Returns** – true if both terms have equal powers.
-
- *evaluate*
 public double **evaluate**()
 - **Description**
 Evaluates this term at infinity
 - **Returns** – value of the term at infinity
-
- *evaluate*
 public double **evaluate**(double **x**)

- **Description**
Evaluates this term at the value x
 - **Parameters**
* x – Evaluation point
 - **Returns** – Value of the term at the specified point
-

- *expand*

public Term **expand**(double a)

- **Description**
Return this term evaluated at t*a
 - **Parameters**
* a –
 - **Returns** – This term evaluated at t*a
-

- *getCoeff*

public double **getCoeff**()

- **Description**
Returns the value of the coefficient
 - **Returns** – the coefficient
-

- *getDegree*

public int **getDegree**()

- **Description**
Returns the value of the degree
 - **Returns** – the degree of this term
-

- *getLambda*

public double **getLambda**()

- **Description**
Returns the value of lambda
 - **Returns** – the value of lambda
-

- *integrate*

public SuperErlang **integrate**()

- **Description**
Returns the integral form 0 to t
 - **Returns** – SuperErlang representation of the integral from 0 to t
-

- *integrateCom*

public SuperErlang **integrateCom**()

- **Description**
Returns the integral from t to infinity
 - **Returns** – SuperErlang representation of the integral from t to infinity
-

- *isConstant*

public boolean **isConstant**()

- **Description**
Tells if it is constant.
 - **Returns** – true if this term represents a constant
-

- *isPTerm*

public boolean **isPTerm**()

- **Description**
Tells if the term is a polynomial
 - **Returns** – true if it is a power term.
-

- *isZero*

public boolean **isZero**()

- **Description**
Tells if it is identically = 0.
 - **Returns** – true if this term represents a zero
-

- *moment*

protected double **moment**(int k)

- **Description**
Computes the k-th moment
 - **Parameters**
* k – required moment
 - **Returns** – k-th moment
-

- *move*

public SuperErlang **move**(double a)

- **Description**
Return this term evaluated at t+a
 - **Parameters**
* a –
 - **Returns** – SuperErlang representation of the term evaluated at t+a
-

- *multiply*

public Term **multiply**(Term tr)

- **Description**
Return the product of this term with tr
 - **Parameters**
* tr – Term to multiply
 - **Returns** – Product of this term with tr
-

- *multiply*

public static Term **multiply**(Term t1, Term t2)

- **Description**
Return the product of these two terms
- **Parameters**

- * **t1** – First term to multiply
 - * **t2** – Second term to multiply
 - **Returns** – Product of this two terms
-

- *setCoeff*

```
public void setCoeff( double alpha )
```

- **Description**
Sets the value of the coefficient
 - **Parameters**
* **alpha** – new value
-

- *setDegree*

```
public void setDegree( int n )
```

- **Description**
Returns the value of the degree
 - **Parameters**
* **n** –
-

- *setLambda*

```
public void setLambda( double lbd )
```

- **Description**
Returns the value of the coefficient
 - **Parameters**
* **lbd** – lambda
-

- *toString*

```
public java.lang.String toString( )
```

- *toStringE*

```
public java.lang.String toStringE( )
```

- **Description**
Represents the term as a multiple of an Erlang pdf.
 - **Returns** – aforementioned string
-

- *toStringP*

```
public java.lang.String toStringP( )
```

- **Description**
String representation of the Polynomial
 - **Returns** – String representation of the Polynomial
-

- *toStringRTF*

```
public java.lang.String toStringRTF( )
```

- **Description**
RTF representation fo the term
- **Returns** – a string representing this term

7.2.15 Class Utils

Declaration

```
public class Utils
extends java.lang.Object
```

Constructor summary

Utils()

Method summary

binomial(int, int) Binomial coefficient
distance(double[], double[]) Euclidean norm btween given arrays
fact(int) Factorial function ($n!$).
gammaP(double, double) Incomplete gamma function.
initUpperTriangular(int) Creates storage for un upper triangular matrix.
lnBinomial(int, int) ln Binomial coefficient.
lnFactorial(int) Computes the log of Factorial function
lnGamma(double) Computes the log of gamma function.
lnPermut(int, int) Computes $\ln(n!/(n-k)!)$
permut(int, int) Computes $n!/(n-k)!$
pow(double, int) Power function obtained by multiplying.

Constructors

- *Utils*
 public **Utils()**

Methods

- *binomial*
 public static double **binomial**(int *n*, int *k*)
 – **Description**
 Binomial coefficient
 – **Parameters**
 * *n* –
 * *k* –
 – **Returns** – $n! / k! (n-k)!$
- *distance*
 public static double **distance**(double[] *v1*, double[] *v2*)
 – **Description**
 Euclidean norm btween given arrays
 – **Parameters**
 * *v1* –

-
- * `v2` –
 - **Returns** – Euclidean norm
-
- *fact*
 public static double **fact**(int `n`)
 - **Description**
Factorial function (`n!`).
 - **Parameters**
 - * `n` –
 - **Returns** – `n!`
-
- *gammaP*
 public static double **gammaP**(double `a`, double `x`)
 - **Description**
Incomplete gamma function. ;
 - **Parameters**
 - * `a` – argument
 - * `x` – upper limit
 - **Returns** – $\int_{0, \infty, x} e^{(-t)} t^a / \gamma(a) dt$
-
- *initUpperTriangular*
 public static double[][] **initUpperTriangular**(int `n`)
 - **Description**
Creates storage for an upper triangular matrix.
 - **Parameters**
 - * `n` –
 - **Returns** – An `n`x`n` upper triangular matrix.
-
- *lnBinomial*
 public static double **lnBinomial**(int `n`, int `k`)
 - **Description**
ln Binomial coefficient.
 - **Parameters**
 - * `n` –
 - * `k` –
 - **Returns** – $\ln [n! / k! (n-k)!]$
-
- *lnFactorial*
 public static double **lnFactorial**(int `n`)
 - **Description**
Computes the log of Factorial function
 - **Parameters**
 - * `n` –
 - **Returns** – $\ln (n!)$
-

- *lnGamma*

```
public static double lnGamma( double xx )
```

- **Description**

Computes the log of gamma function.

- **Parameters**

* **xx** – value

- **Returns** – $\ln\Gamma(xx)$

- *lnPermut*

```
public static double lnPermut( int n, int k )
```

- **Description**

Computes $\ln(n!/(n-k)!)$

- **Parameters**

* **n** –

* **k** –

- **Returns** – $\ln(n!/(n-k)!)$

- *permut*

```
public static double permut( int n, int k )
```

- **Description**

Computes $n!/(n-k)!$

- **Parameters**

* **n** –

* **k** –

- **Returns** – Computes $n!/(n-k)!$

- *pow*

```
public static double pow( double x, int n )
```

- **Description**

Power function obtained by multiplying.

- **Parameters**

* **x** –

* **n** –

- **Returns** – x^n

Chapter 8

Package `jphase.fit`

Package Contents

Page

Interfaces

PhaseFitter	353
<i>This class defines the behaviour that any class for fitting data to a Phase-Type distribution should have</i>	

Classes

ContPhaseFitter	354
<i>This class defines the behaviour that any class for fitting data to a Continuous Phase-Type distribution should have</i>	
DiscPhaseFitter	356
<i>This class defines the behaviour that any class for fitting data to a Discrete Phase-Type distribution should have</i>	
EMHyperErlangFit	358
<i>This class implements the Maximum Likelihood method proposed by Thümmeler, Buchholz and Telek in "A novel approach for fitting probability distributions to real trace data with the EM algorithm", 2005.</i>	
EMHyperExpoFit	361
<i>This class implements the Maximum Likelihood method proposed by Khayari, Sadre and Haverkort in "Fitting world-wide web request traces with the EM algorithm", 2003.</i>	
EMPhaseFit	363
<i>This class implements the Maximum Likelihood method proposed by Asmussen, Nerman and Olsson in "Fitting Phase-type Distributions via the EM algorithm", 1996.</i>	
FitterUtils	365
<i>This class contains a set of methods to make some usual calculations for the PhaseFitter classes</i>	
MLContPhaseFitter	367
<i>This class defines the behaviour for a class that implements a maximum likelihood algorithm for fitting data to a Continuous Phase-Type distribution</i>	
MLDiscPhaseFitter	368
<i>This class defines the behaviour for a class that implements a maximum likelihood algorithm for fitting data to a Discrete Phase-Type distribution</i>	
MomentsACPH2Fit	369
<i>This class implements the Matching Moments method proposed by Telek and Heindl in "Matching Moments for Acyclic discrete and continuous Phase-Type distributions of Second order", 2002.</i>	

MomentsACPHFit	371
<i>* This class implements the Matching Moments method proposed by Bobbio, Horvath and Telek in "Matching threee moments with minimal acyclic Phase-Type distributions", 2005.</i>	
MomentsADPH2Fit	373
<i>This class implements the Matching Moments method proposed by Telek and Heindl in "Matching Moments for Acyclic discrete and continuous Phase-Type distributions of Second order", 2002.</i>	
MomentsContPhaseFitter	375
<i>This class defines the behaviour for a class that implements a moment matching algorithm for fitting data to a Continuous Phase-Type distribution</i>	
MomentsDiscPhaseFitter	377
<i>This class defines the behaviour for a class that implements a moment matching algorithm for fitting data to a Discrete Phase-Type distribution</i>	
MomentsECCompleteFit	379
<i>This class implements the Matching Moments method proposed by Osogami and Harchol in "Closed form solutions for mapping general distributions to quasi-minimal PH distributions", 2005.</i>	
MomentsECPositiveFit	381
<i>This class implements the Matching Moments method proposed by Osogami and Harchol in "Closed form solutions for mapping general distributions to quasi-minimal PH distributions", 2005.</i>	

Provides capabilities for fitting Phase type distribution parameters from data. There exists very different algorithms to adjust the parameters of these distributions, which can be divided in two groups: moments based methods and maximum likelihood approaches. This package includes the implementation of some of these algorithms, a framework and a set of utilities for implement new methods. For overviews, tutorials, examples, guides, and tool documentation, please see: Copa Group WEB page. (at <http://copa.uniandes.edu.co>)

8.1 Interfaces

8.1.1 Interface PhaseFitter

This class defines the behaviour that any class for fitting data to a Phase-Type distribution should have

Declaration

```
public interface PhaseFitter
```

All known subclasses

MomentsECPositiveFit (see 8.2.15, page 381) , MomentsECCCompleteFit (see 8.2.14, page 379) , MomentsDiscPhaseFitter (see 8.2.13, page 377) , MomentsContPhaseFitter (see 8.2.12, page 375) , MomentsADPH2Fit (see 8.2.11, page 373) , MomentsACPHFit (see 8.2.10, page 371) , MomentsACPH2Fit (see 8.2.9, page 369) , MLDiscPhaseFitter (see 8.2.8, page 368) , MLContPhaseFitter (see 8.2.7, page 367) , EMPhaseFit (see 8.2.5, page 363) , EMHyperExpoFit (see 8.2.4, page 361) , EMHyperErlangFit (see 8.2.3, page 358) , DiscPhaseFitter (see 8.2.2, page 356) , ContPhaseFitter (see 8.2.1, page 354)

All classes known to implement interface

DiscPhaseFitter (see 8.2.2, page 356) , ContPhaseFitter (see 8.2.1, page 354)

Method summary

fit() Executes the fitting procedure to find the parameter set

Methods

- *fit*
jphase.PhaseVar **fit**()
 - **Description**
Executes the fitting procedure to find the parameter set
 - **Returns** – Phase variable found

8.2 Classes

8.2.1 Class ContPhaseFitter

This class defines the behaviour that any class for fitting data to a Continuous Phase-Type distribution should have

Declaration

```
public abstract class ContPhaseFitter
  extends java.lang.Object
  implements PhaseFitter
```

All known subclasses

MomentsECPositiveFit (see 8.2.15, page 381) , MomentsECCompleteFit (see 8.2.14, page 379) , MomentsContPhaseFitter (see 8.2.12, page 375) , MomentsACPHFit (see 8.2.10, page 371) , MomentsACPH2Fit (see 8.2.9, page 369) , MLContPhaseFitter (see 8.2.7, page 367) , EMPhaseFit (see 8.2.5, page 363) , EMHyperExpoFit (see 8.2.4, page 361) , EMHyperErlangFit (see 8.2.3, page 358)

Field summary

data Non-negative data trace from independent experiments
var Fitted Continuous Phase-Type variable

Constructor summary

ContPhaseFitter(double[])

Method summary

fit()
getLogLikelihood()

Fields

- protected jphase.ContPhaseVar **var**
 - Fitted Continuous Phase-Type variable
- protected double **data**
 - Non-negative data trace from independent experiments

Constructors

- *ContPhaseFitter*
 public **ContPhaseFitter**(double[] data)
 - **Parameters**
 - * data –

Methods

- *fit*
public abstract `jphase.ContPhaseVar fit()`
 - **See also**
 - * `PhaseFitter.fit()` (see 8.1.1, page 353)

- *getLogLikelihood*
public double `getLogLikelihood()`
 - **Returns** – -1 if there is no data associated to the algorithm, 0 if there has not been found a `ContPhaseVar` yet, or the likelihood.

8.2.2 Class DiscPhaseFitter

This class defines the behaviour that any class for fitting data to a Discrete Phase-Type distribution should have

Declaration

```
public abstract class DiscPhaseFitter
  extends java.lang.Object
  implements PhaseFitter
```

All known subclasses

MomentsDiscPhaseFitter (see 8.2.13, page 377) , MomentsADPH2Fit (see 8.2.11, page 373) ,
MLDiscPhaseFitter (see 8.2.8, page 368)

Field summary

data Non-negative data trace from independent experiments
var Fitted Discrete Phase-Type variable

Constructor summary

DiscPhaseFitter(int[])

Method summary

fit()
getLogLikelihood()

Fields

- protected jphase.DiscPhaseVar **var**
 - Fitted Discrete Phase-Type variable
- protected int **data**
 - Non-negative data trace from independent experiments

Constructors

- *DiscPhaseFitter*
public **DiscPhaseFitter**(int[] **data**)
 - **Parameters**
 - * **data** –

Methods

- *fit*
public abstract jphase.DiscPhaseVar **fit**()
 - **See also**
 - * `PhaseFitter.fit()` (see 8.1.1, page 353)
- *getLogLikelihood*
public double **getLogLikelihood**()
 - **Returns** – -1 if there is no data associated to the algorithm, 0 if there has not been found a ContPhaseVar yet, or the likelihood.

8.2.3 Class EMHyperErlangFit

This class implements the Maximum Likelihood method proposed by Thümmeler, Buchholz and Telek in "A novel approach for fitting probability distributions to real trace data with the EM algorithm", 2005. The method matches the likelihood of any distribution to a subclass of Phase-Type distributions known as Hyper-Erlang distributions.

Declaration

```
public class EMHyperErlangFit
extends jphase.fit.MLContPhaseFitter (see 8.2.7, page 367)
```

Field summary

precision Precision for the convergence criterion in the algorithm
precisionCV Precision for the convergence criterion in the coefficient of variance

Constructor summary

EMHyperErlangFit(double[])

Method summary

doFitHyperErlang(double[]) Returns a HyperErlang variable with the best fit
doFitNM(double[], HyperErlangVar) This method returns a completely specified HyperErlang variable, such that it has the best likelihood between all the possible combinations of N phases in M branches
doFitNMR(double[], HyperErlangVar) This method returns a completely specified HyperErlang variable, such that it has the best likelihood after the execution of the EM algorithm for the case where the variable has N phases in M branches, distributed as determined by the vector r
fit() Returns a HyperErlang variable with the best fit, in the form of a Dense Continuous Phase variable

Fields

- public static double **precision**
 - Precision for the convergence criterion in the algorithm
- public static double **precisionCV**
 - Precision for the convergence criterion in the coefficient of variance

Constructors

- *EMHyperErlangFit*
public EMHyperErlangFit(double[] data)
 - **Parameters**
 - * **data** –

Methods

- *doFitHyperErlang*

```
public jphase.HyperErlangVar doFitHyperErlang( double[] data )
```

- **Description**

Returns a HyperErlang variable with the best fit

- **Parameters**

* **data** – non-negative data trace from independent experiments to be fitted

- **Returns** – HyperErlang variable with the best fit

- *doFitNM*

```
public double doFitNM( double[] data, jphase.HyperErlangVar var )
```

- **Description**

This method returns a completely specified HyperErlang variable, such that it has the best likelihood between all the possible combinations of N phases in M branches

- **Parameters**

* **data** – non-negative data trace from independent experiments to be fitted

* **var** – HyperErlang variable with the parameters N and M determined

- **Returns** – Likelihood of the best variable found. The variable is modified with the best parameters found.

- *doFitNMR*

```
public double doFitNMR( double[] data, jphase.HyperErlangVar var )
```

- **Description**

This method returns a completely specified HyperErlang variable, such that it has the best likelihood after the execution of the EM algorithm for the case where the variable has N phases in M branches, distributed as determined by the vector r

- **Parameters**

* **data** – non-negative data trace from independent experiments to be fitted

* **var** – HyperErlang variable with the parameters N, M and r determined

- **Returns** – Likelihood of the best variable found. The variable is modified with the best parameters found.

- *fit*

```
public jphase.DenseContPhaseVar fit( )
```

- **Description**

Returns a HyperErlang variable with the best fit, in the form of a Dense Continuous Phase variable

- **Returns** – HyperErlang variable with the best fit

Members inherited from class jphase.fit.MLContPhaseFitter (see 8.2.7, page 367)

- `public double getLogLikelihood()`

Members inherited from class `jphase.fit.ContPhaseFitter` (see 8.2.1, page 354)

- `protected data`
- `public abstract ContPhaseVar fit()`
- `public double getLogLikelihood()`
- `protected var`

8.2.4 Class EMHyperExpoFit

This class implements the Maximum Likelihood method proposed by Khayari, Sadre and Haverkort in "Fitting world-wide web request traces with the EM algorithm", 2003. The method matches the likelihood of heavy tailed distributions to the class of Hyper-Exponential distributions.

Declaration

```
public class EMHyperExpoFit
extends jphase.fit.MLContPhaseFitter (see 8.2.7, page 367)
```

Field summary

precision Precision for the convergence criterion in the algorithm
precisionParam Precision for the convergence criterion in the coefficient of variance

Constructor summary

EMHyperExpoFit(double[])

Method summary

doFitN(double[]) This method implements the EM algorithm from the data, with the specified number of exponential phases
fit()

Fields

- public static double **precision**
 - Precision for the convergence criterion in the algorithm
- public static double **precisionParam**
 - Precision for the convergence criterion in the coefficient of variance

Constructors

- *EMHyperExpoFit*
 public **EMHyperExpoFit**(double[] data)
 - **Parameters**
 - * data –

Methods

- *doFitN*

```
public double doFitN( double[] data )
```

- **Description**

This method implements the EM algorithm from the data, with the specified number of exponential phases

- **Parameters**

* **data** – non-negative data trace from independent experiments

- **Returns** – The loglikelihood of the solution found. The values of the parameters are stored in the attributes probs and rates

- *fit*

```
public abstract jphase.ContPhaseVar fit( )
```

- **See also**

* `PhaseFitter.fit()` (see 8.1.1, page 353)

Members inherited from class `jphase.fit.MLContPhaseFitter` (see 8.2.7, page 367)

- `public double getLogLikelihood()`

Members inherited from class `jphase.fit.ContPhaseFitter` (see 8.2.1, page 354)

- `protected data`
- `public abstract ContPhaseVar fit()`
- `public double getLogLikelihood()`
- `protected var`

8.2.5 Class EMPhaseFit

This class implements the Maximum Likelihood method proposed by Asmussen, Nerman and Olsson in "Fitting Phase-type Distributions via the EM algorithm", 1996. The method matches the likelihood of any distribution to the entire class of Phase-Type distributions.

Declaration

```
public class EMPhaseFit
  extends jphase.fit.MLContPhaseFitter (see 8.2.7, page 367)
```

Field summary

evalPoints Constant to multiply the size of the data trace to obtain the number of evaluation points.

logPrecision

precision Precision for the convergence criterion in the algorithm

precisionParam Precision for the convergence criterion in the coefficient of variance

Constructor summary

EMPhaseFit(double[])

Method summary

doFitN(double[])
fit()

Fields

- public static double **precision**
 - Precision for the convergence criterion in the algorithm
- public static double **logPrecision**
- public static double **precisionParam**
 - Precision for the convergence criterion in the coefficient of variance
- public static int **evalPoints**
 - Constant to multiply the size of the data trace to obtain the number of evaluation points. The number of evaluation points in the Runge-Kutta algorithm is the size of the data trace times evalPoints

Constructors

- *EMPhaseFit*
 public **EMPhaseFit**(double[] data)
 - **Parameters**
 - * data –

Methods

- *doFitN*
public double doFitN(double[] data)
 - **Parameters**
 - * data –
 - **Returns** – The loglikelihood
-
- *fit*
public abstract jphase.ContPhaseVar fit()
 - **See also**
 - * PhaseFitter.fit() (see 8.1.1, page 353)

Members inherited from class jphase.fit.MLContPhaseFitter (see 8.2.7, page 367)

- public double getLogLikelihood()

Members inherited from class jphase.fit.ContPhaseFitter (see 8.2.1, page 354)

- protected data
- public abstract ContPhaseVar fit()
- public double getLogLikelihood()
- protected var

8.2.6 Class FitterUtils

This class contains a set of methods to make some usual calculations for the PhaseFitter classes

Declaration

```
public class FitterUtils
extends java.lang.Object
```

Constructor summary

FitterUtils()

Method summary

ceil(double, double) Calculates the ceil of a double with a predefined precision
factMomentK(int[], int) Calculates the k-th factorial moment of the data trace
floor(double, double) Calculates the floor of a double with a predefined precision
powerMomentK(double[], int) Calculates the k-th power moment of the data trace
sqrt(double, double) Calculates the square root of a double with a predefined precision

Constructors

- *FitterUtils*
public **FitterUtils**()

Methods

- *ceil*
public static double **ceil**(double x, double epsilon)
 - **Description**
Calculates the ceil of a double with a predefined precision
 - **Parameters**
 - * x –
 - * epsilon – precision
 - **Returns** – The ceil of a double with the predefined precision

- *factMomentK*
public static double **factMomentK**(int[] data, int k)
 - **Description**
Calculates the k-th factorial moment of the data trace
 - **Parameters**
 - * data – data trace
 - * k – factorial moment to be calculated (≥ 1)
 - **Returns** – Data k-th factorial Moment

- *floor*

```
public static double floor( double x, double epsilon )
```

- **Description**

Calculates the floor of a double with a predefined precision

- **Parameters**

- * **x** –

- * **epsilon** – precision

- **Returns** – The floor of a double with the predefined precision

- *powerMomentK*

```
public static double powerMomentK( double[] data, int k )
```

- **Description**

Calculates the k-th power moment of the data trace

- **Parameters**

- * **data** – data trace

- * **k** – power moment to be calculated (≥ 1)

- **Returns** – Data k-th power Moment

- *sqrt*

```
public static double sqrt( double x, double epsilon )
```

- **Description**

Calculates the square root of a double with a predefined precision

- **Parameters**

- * **x** –

- * **epsilon** – precision

- **Returns** – The square root of a double with the predefined precision

8.2.7 Class MLContPhaseFitter

This class defines the behaviour for a class that implements a maximum likelihood algorithm for fitting data to a Continuous Phase-Type distribution

Declaration

```
public abstract class MLContPhaseFitter
extends jphase.fit.ContPhaseFitter (see 8.2.1, page 354)
```

All known subclasses

EMPhaseFit (see 8.2.5, page 363) , EMHyperExpoFit (see 8.2.4, page 361) , EMHyperErlangFit (see 8.2.3, page 358)

Constructor summary

MLContPhaseFitter(double[])

Method summary

getLogLikelihood()

Constructors

- *MLContPhaseFitter*
 public **MLContPhaseFitter**(double[] data)
 – **Parameters**
 * data –

Methods

- *getLogLikelihood*
 public double **getLogLikelihood**()
 – **Returns** – -1 if there is no data associated to the algorithm, 0 if there has not been found a ContPhaseVar yet, or the likelihood.

Members inherited from class jphase.fit.ContPhaseFitter (see 8.2.1, page 354)

- protected data
- public abstract ContPhaseVar fit()
- public double getLogLikelihood()
- protected var

8.2.8 Class MLDiscPhaseFitter

This class defines the behaviour for a class that implements a maximum likelihood algorithm for fitting data to a Discrete Phase-Type distribution

Declaration

```
public abstract class MLDiscPhaseFitter
extends jphase.fit.DiscPhaseFitter (see 8.2.2, page 356)
```

Constructor summary

MLDiscPhaseFitter(int[])

Method summary

getLogLikelihood()

Constructors

- *MLDiscPhaseFitter*
public MLDiscPhaseFitter(int[] data)
 - **Parameters**
 - * data –

Methods

- *getLogLikelihood*
public double getLogLikelihood()
 - **Returns** – -1 if there is no data associated to the algorithm, 0 if there has not been found a ContPhaseVar yet, or the likelihood.

Members inherited from class jphase.fit.DiscPhaseFitter (see 8.2.2, page 356)

- protected data
- public abstract DiscPhaseVar fit()
- public double getLogLikelihood()
- protected var

8.2.9 Class MomentsACPH2Fit

This class implements the Matching Moments method proposed by Telek and Heindl in "Matching Moments for Acyclic discrete and continuous Phase-Type distributions of Second order", 2002. This is for the continuous case.

Declaration

```
public class MomentsACPH2Fit
  extends jphase.fit.MomentsContPhaseFitter (see 8.2.12, page 375)
```

Field summary

precision TODO get precision precision for calculations and convergence criterion

Constructor summary

MomentsACPH2Fit(double[])
MomentsACPH2Fit(double, double, double)

Method summary

fit()

Fields

- public static double **precision**
 - TODO get precision precision for calculations and convergence criterion

Constructors

- *MomentsACPH2Fit*
 public **MomentsACPH2Fit(double[] data)**
 - **Parameters**
 - * **data** – Data to be fitted
- *MomentsACPH2Fit*
 public **MomentsACPH2Fit(double m1, double m2, double m3)**
 - **Parameters**
 - * **m1** –
 - * **m2** –
 - * **m3** –

Methods

- *fit*
public abstract jphase.ContPhaseVar fit()
 - See also
 - * PhaseFitter.fit() (see 8.1.1, page 353)

Members inherited from class jphase.fit.MomentsContPhaseFitter (see 8.2.12, page 375)

- protected m1
- protected m2
- protected m3

Members inherited from class jphase.fit.ContPhaseFitter (see 8.2.1, page 354)

- protected data
- public abstract ContPhaseVar fit()
- public double getLogLikelihood()
- protected var

8.2.10 Class MomentsACPHFit

* This class implements the Matching Moments method proposed by Bobbio, Horvath and Telek in "Matching three moments with minimal acyclic Phase-Type distributions", 2005. The method match the first three moments of any distribution to a subclass of Phase-Type distributions known as Acyclic Phase-Type distributions.

Declaration

```
public class MomentsACPHFit
extends jphase.fit.MomentsContPhaseFitter (see 8.2.12, page 375)
```

Constructor summary

MomentsACPHFit(double[])
MomentsACPHFit(double, double, double)

Method summary

fit() Solve the equation system to get the parameters of the distribution, if the moments are feasible
getSize(double, double) Calculates the minimum number of phases needed to represent the tuple of the normalized moments

Constructors

- *MomentsACPHFit*
public MomentsACPHFit(double[] data)
 – **Parameters**
 * data –

- *MomentsACPHFit*
public MomentsACPHFit(double m1, double m2, double m3)
 – **Parameters**
 * m1 –
 * m2 –
 * m3 –

Methods

- *fit*
public jphase.ContPhaseVar fit()
 – **Description**
 Solve the equation system to get the parameters of the distribution, if the moments are feasible
 – **Returns** – Parameters of the Acyclic Continuous Phase Distribution of order n.
 param[0]=n, param[1]=p, param[2]=lambdaY, param[3]=lambdaX1,
 param[4]=lambdaX2, param[5]=pX

-
- *getSize*
 public int getSize(double **n2**, double **n3**)
 - **Description**
 Calculates the minimum number of phases needed to represent the tuple of the normalized moments
 - **Parameters**
 - * **n2** – Second normalized moment
 - * **n3** – Third normalized moment
 - **Returns** – Minimum number of phases needed to represent the tuple of the normalized moments

Members inherited from class `jphase.fit.MomentsContPhaseFitter` (see 8.2.12, page 375)

- protected **m1**
- protected **m2**
- protected **m3**

Members inherited from class `jphase.fit.ContPhaseFitter` (see 8.2.1, page 354)

- protected **data**
- public abstract `ContPhaseVar` **fit**()
- public double **getLogLikelihood**()
- protected **var**

8.2.11 Class MomentsADPH2Fit

This class implements the Matching Moments method proposed by Telek and Heindl in "Matching Moments for Acyclic discrete and continuous Phase-Type distributions of Second order", 2002. This is for the discrete case.

Declaration

```
public class MomentsADPH2Fit
  extends jphase.fit.MomentsDiscPhaseFitter (see 8.2.13, page 377)
```

Constructor summary

```
MomentsADPH2Fit(double, double, double)
MomentsADPH2Fit(int[])
```

Method summary

```
fit()
```

Constructors

- *MomentsADPH2Fit*
 public MomentsADPH2Fit(double m1, double m2, double m3)
 – See also
 * MomentsDiscPhaseFitter(double,double,double) (see 8.2.13, page 377)

- *MomentsADPH2Fit*
 public MomentsADPH2Fit(int[] data)
 – See also
 * MomentsDiscPhaseFitter(int[]) (see 8.2.13, page 377)

Methods

- *fit*
 public abstract jphase.DiscPhaseVar fit()
 – See also
 * PhaseFitter.fit() (see 8.1.1, page 353)

Members inherited from class jphase.fit.MomentsDiscPhaseFitter (see 8.2.13, page 377)

- protected m1
- protected m2
- protected m3

Members inherited from class `jphase.fit.DiscPhaseFitter` (see 8.2.2, page 356)

- protected `data`
- public abstract `DiscPhaseVar fit()`
- public double `getLogLikelihood()`
- protected `var`

8.2.12 Class MomentsContPhaseFitter

This class defines the behaviour for a class that implements a moment matching algorithm for fitting data to a Continuous Phase-Type distribution

Declaration

```
public abstract class MomentsContPhaseFitter
  extends jphase.fit.ContPhaseFitter (see 8.2.1, page 354)
```

All known subclasses

MomentsECPositiveFit (see 8.2.15, page 381) , MomentsECCompleteFit (see 8.2.14, page 379) ,
MomentsACPHFit (see 8.2.10, page 371) , MomentsACPH2Fit (see 8.2.9, page 369)

Field summary

m1
m2
m3

Constructor summary

MomentsContPhaseFitter(double[])
MomentsContPhaseFitter(double, double, double)

Fields

- protected double **m1**
- protected double **m2**
- protected double **m3**

Constructors

- *MomentsContPhaseFitter*
 public **MomentsContPhaseFitter(double[] data)**
 – **Parameters**
 * data –
- *MomentsContPhaseFitter*
 public **MomentsContPhaseFitter(double m1, double m2, double m3)**
 – **Parameters**
 * m1 –
 * m2 –
 * m3 –

Members inherited from class `jphase.fit.ContPhaseFitter` (see 8.2.1, page 354)

- `protected data`
- `public abstract ContPhaseVar fit()`
- `public double getLogLikelihood()`
- `protected var`

8.2.13 Class MomentsDiscPhaseFitter

This class defines the behaviour for a class that implements a moment matching algorithm for fitting data to a Discrete Phase-Type distribution

Declaration

```
public abstract class MomentsDiscPhaseFitter
  extends jphase.fit.DiscPhaseFitter (see 8.2.2, page 356)
```

All known subclasses

MomentsADPH2Fit (see 8.2.11, page 373)

Field summary

m1
m2
m3

Constructor summary

MomentsDiscPhaseFitter(double, double, double)
MomentsDiscPhaseFitter(int[])

Fields

- protected double **m1**
- protected double **m2**
- protected double **m3**

Constructors

- *MomentsDiscPhaseFitter*
 public **MomentsDiscPhaseFitter(double m1, double m2, double m3)**
 – **Parameters**
 * m1 –
 * m2 –
 * m3 –

- *MomentsDiscPhaseFitter*
 public **MomentsDiscPhaseFitter(int[] data)**
 – **Parameters**
 * data –

Members inherited from class `jphase.fit.DiscPhaseFitter` (see 8.2.2, page 356)

- `protected data`
- `public abstract DiscPhaseVar fit()`
- `public double getLogLikelihood()`
- `protected var`

8.2.14 Class MomentsECCCompleteFit

This class implements the Matching Moments method proposed by Osogami and Harchol in "Closed form solutions for mapping general distributions to quasi-minimal PH distributions", 2005. The method match the first three moments of any distribution to a subclass of Phase-Type distributions known as Erlang-Coxian distributions. This class implements the Complete solution.

Declaration

```
public class MomentsECCCompleteFit
extends jphase.fit.MomentsContPhaseFitter (see 8.2.12, page 375)
```

Constructor summary

MomentsECCCompleteFit(double[])
MomentsECCCompleteFit(double, double, double)

Method summary

fit() Fit a Phase Type distribution from a set of moments with the Complete method described by Osogami et al.
getParam() Solve the equation system to get the parameters of the distribution, if the moments are feasible

Constructors

- *MomentsECCCompleteFit*
 public **MomentsECCCompleteFit**(double[] data)
 – **Parameters**
 * data –

- *MomentsECCCompleteFit*
 public **MomentsECCCompleteFit**(double m1, double m2, double m3)
 – **Parameters**
 * m1 –
 * m2 –
 * m3 –

Methods

- *fit*
 public jphase.ContPhaseVar **fit**()
 – **Description**
 Fit a Phase Type distribution from a set of moments with the Complete method described by Osogami et al.
 – **Returns** – Phase variable found

- *getParam*

public double[] **getParam**()

– **Description**

Solve the equation system to get the parameters of the distribution, if the moments are feasible

– **Returns** – Parameters of the Acyclic Continuous Phase Distribution of order n.

param[0]=n, param[1]=p, param[2]=lambdaY, param[3]=lambdaX1,
param[4]=lambdaX2, param[5]=pX

Members inherited from class jphase.fit.MomentsContPhaseFitter (see 8.2.12, page 375)

- protected **m1**
- protected **m2**
- protected **m3**

Members inherited from class jphase.fit.ContPhaseFitter (see 8.2.1, page 354)

- protected **data**
- public abstract ContPhaseVar **fit**()
- public double **getLogLikelihood**()
- protected **var**

8.2.15 Class MomentsECPositiveFit

This class implements the Matching Moments method proposed by Osogami and Harchol in "Closed form solutions for mapping general distributions to quasi-minimal PH distributions", 2005. The method match the first three moments of any distribution to a subclass of Phase-Type distributions known as Erlang-Coxian distributions. This class implements the Positive solution.

Declaration

```
public class MomentsECPositiveFit
extends jphase.fit.MomentsContPhaseFitter (see 8.2.12, page 375)
```

Constructor summary

```
MomentsECPositiveFit(double[])
MomentsECPositiveFit(double, double, double)
```

Method summary

```
fit() Fit a Phase Type distribution with no mass at zero from a set of moments with
      the Positive method described by Osogami et al.
getParam() Solve the equation system to get the parameters of the distribution, if
      the moments are feasible
```

Constructors

- *MomentsECPositiveFit*

```
public MomentsECPositiveFit( double[] data )
```

 - **Parameters**
 - * data –
- *MomentsECPositiveFit*

```
public MomentsECPositiveFit( double m1, double m2, double m3 )
```

 - **Parameters**
 - * m1 –
 - * m2 –
 - * m3 –

Methods

- *fit*

```
public jphase.ContPhaseVar fit( )
```

 - **Description**
 Fit a Phase Type distribution with no mass at zero from a set of moments with the Positive method described by Osogami et al.
 - **Returns** – Phase variable found

- *getParam*

`public double[] getParam() throws java.lang.IllegalArgumentException`

- **Description**

Solve the equation system to get the parameters of the distribution, if the moments are feasible

- **Returns** – Parameters of the Acyclic Continuous Phase Distribution of order n.

param[0]=n, param[1]=p, param[2]=lambdaY, param[3]=lambdaX1, param[4]=lambdaX2, param[5]=pX. If it is necessary, a new exponential phase with parameter lambda will be aggregated through convolution: param[6]=lambda. If it is necessary, a new exponential phase with parameter lambda will be mixed up with probability 1-pMix: param[6]=lambda, param[7] = pMix.

- **Throws**

* `java.lang.IllegalArgumentException` –

Members inherited from class `jphase.fit.MomentsContPhaseFitter` (see 8.2.12, page 375)

- protected `m1`
- protected `m2`
- protected `m3`

Members inherited from class `jphase.fit.ContPhaseFitter` (see 8.2.1, page 354)

- protected `data`
- public abstract `ContPhaseVar fit()`
- public `double getLogLikelihood()`
- protected `var`

Chapter 9

Package `jphase.generator`

Package Contents

Page

Classes

GeneratorUtils	384
<i>This class contains a set of methods to make some usual calculations for the PhaseGenerator classes</i>	
NeutsContPHGenerator	387
<i>This class implements the algorithm proposed by Neuts and Pagano "generating Random Variates of Phase-Type", 1981.</i>	
NeutsDiscPHGenerator	389
<i>This class implements the algorithm proposed by Neuts and Pagano "generating Random Variates of Phase-Type", 1981.</i>	
PhaseGenerator	391
<i>This abstract class defines the behaviour that any Phase-Type random number generator should have</i>	

Provides capabilities for generating random variates from Phase type distributions. These includes discrete and continuous random number generators. This package includes the implementation of two algorithms based on the alias method, a framework and a set of utilities for implement new methods. For overviews, tutorials, examples, guides, and tool documentation, please see: Copa Group WEB page. (at <http://copa.uniandes.edu.co>)

9.1 Classes

9.1.1 Class GeneratorUtils

This class contains a set of methods to make some usual calculations for the PhaseGenerator classes

Declaration

```
public class GeneratorUtils
  extends java.lang.Object
```

Constructor summary

GeneratorUtils()

Method summary

algorKS(double[], PhaseVar) This method implements the KS algorithm proposed by González, Sahni and Franta in "An efficient algorithm for the Kolmogorov-Smirnov and Lilliefors Tests" in ACM Transactions on Mathematical Software, Vol 3, No.

aliasCut(double[], int[], double[]) This method generates the aliases and cutoff values according to the distribution specified.

aliasCut(Vector, int[], double[]) This method generates the aliases and cutoff values according to the distribution specified.

argmax(double[]) Returns the index of the maximum value in the data

argmin(double[]) Returns the index of the minimum value in the data

erlang(double, int, Random) Returns a random number with Erlang(lambda, r) distribution.

getNumber(double[], int[], double[], Random) Returns a random number with discrete distribution dist in {0,...,n}

sum(double[]) Returns the sum of the elements of the data array

sumAbs(double[]) Returns the sum of the absolut values of the elements of the data array

Constructors

- *GeneratorUtils*
public **GeneratorUtils**()

Methods

- *algorKS*
public static double **algorKS**(double[] data, jphase.PhaseVar var)
– **Description**
This method implements the KS algorithm proposed by González, Sahni and Franta in "An efficient algorithm for the Kolmogorov-Smirnov and Lilliefors Tests" in ACM Transactions on Mathematical Software, Vol 3, No. 1, March 1977, pages 60-64.

– **Parameters**

- * **data** – data trace to be tested
- * **var** – theoretical phase variable to be compared against the trace

– **Returns** – absolute maximum deviation from the data to the phase variable

• *aliasCut*

`public static void aliasCut(double[] dist, int[] alias, double[] cutoff)`

– **Description**

This method generates the aliases and cutoff values according to the distribution specified.

– **Parameters**

- * **dist** – Distribution from which the aliases and cutoff vectors must be generated. Represented by array of doubles.
 - * **alias** – vector of the distribution
 - * **cutoff** – values of the distribution
-

• *aliasCut*

`public static void aliasCut(no.uib.cipr.matrix.Vector dist, int[] alias, double[] cutoff)`

– **Description**

This method generates the aliases and cutoff values according to the distribution specified. This distribution comes in a MTJ Vector and may sum less than one (as in the initial condition vector that defines a Markov process or a Phase-Type distribution). The distribution is then adjusted by adding a position in zero that completes the probability mass to sum to zero.

– **Parameters**

- * **dist** – Distribution from which the aliases and cutoff vectors must be generated. Represented by a MTJ Vector.
 - * **alias** – vector of the distribution
 - * **cutoff** – values of the distribution
-

• *argmax*

`public static int argmax(double[] data)`

– **Description**

Returns the index of the maximum value in the data

– **Parameters**

- * **data** – array of data

– **Returns** – The index of the maximum value in data

• *argmin*

`public static int argmin(double[] data)`

– **Description**

Returns the index of the minimum value in the data

– **Parameters**

- * **data** – array of data

– **Returns** – The index of the minimum value in the data

- *erlang*

```
public static double erlang( double lambda, int r, java.util.Random rand )
```

- **Description**

Returns a random number with Erlang(lambda, r) distribution.

- **Parameters**

- * **lambda** – Erlang rate
 - * **r** – number of phases in the Erlang Distribution
 - * **rand** – Random number source

- **Returns** – random number with Erlang(lambda, r) distribution.

- *getNumber*

```
public static int getNumber( double[] dist, int[] alias, double[] cutoff,  
java.util.Random rand )
```

- **Description**

Returns a random number with discrete distribution dist in {0,...,n}

- **Parameters**

- * **dist** – Discrete distribution in {0,...,n}
 - * **alias** – Aliases of the distributions points
 - * **cutoff** – Cutoff values to generate the random numbers
 - * **rand** – Random type object to use as generator of random numbers

- **Returns** – A random number with discrete distribution dist in {0,...,n}

- *sum*

```
public static double sum( double[] data )
```

- **Description**

Returns the sum of the elements of the data array

- **Parameters**

- * **data** – array of data

- **Returns** – Sum of the elements of the data array

- *sumAbs*

```
public static double sumAbs( double[] data )
```

- **Description**

Returns the sum of the absolut values of the elements of the data array

- **Parameters**

- * **data** – array of data

- **Returns** – Sum of the absolut values of the elements of the data array

9.1.2 Class NeutsContPHGenerator

This class implements the algorithm proposed by Neuts and Pagano "generating Random Variates of Phase-Type", 1981. This is also based in the so called alias method to generate a variate from a discrete distribution. This class implements the algorithm for the continuous case.

Declaration

```
public class NeutsContPHGenerator
  extends jphase.generator.PhaseGenerator (see 9.1.4, page 391)
```

Constructor summary

NeutsContPHGenerator(AbstractContPhaseVar)

Method summary

getRandom()
getRandom(int)
initialize()

Constructors

- *NeutsContPHGenerator*
 public **NeutsContPHGenerator**(jphase.AbstractContPhaseVar var)
 – **Parameters**
 * var –

Methods

- *getRandom*
 public abstract double **getRandom**()
 – **Returns** – A random number that has a probability distribution of Phase-Type
- *getRandom*
 public abstract double[] **getRandom**(int num)
 – **Parameters**
 * num – Number of variates to be generated
 – **Returns** – A vector of random numbers that have a probability distribution of Phase-Type
- *initialize*
 protected abstract void **initialize**()
 – **Description copied from PhaseGenerator** (see 9.1.4, page 391)
 Initialize the cutoff values and the aliases for the initial probability distribution and the transition probability matrix

Members inherited from class `jphase.generator.PhaseGenerator` (see 9.1.4, page 391)

- `public abstract double getRandom()`
- `public abstract double getRandom(int num)`
- `public PhaseVar getVar()`
- `protected abstract void initialize()`
- `protected var`

9.1.3 Class NeutsDiscPHGenerator

This class implements the algorithm proposed by Neuts and Pagano "generating Random Variates of Phase-Type", 1981. This is also based in the so called alias method to generate a variate from a discrete distribution. This class implements the algorithm for the discrete case.

Declaration

```
public class NeutsDiscPHGenerator
  extends jphase.generator.PhaseGenerator (see 9.1.4, page 391)
```

Constructor summary

NeutsDiscPHGenerator(AbstractDiscPhaseVar)

Method summary

getRandom()
getRandom(int)
initialize()

Constructors

- *NeutsDiscPHGenerator*
 public **NeutsDiscPHGenerator**(jphase.AbstractDiscPhaseVar var)
 – **Parameters**
 * var –

Methods

- *getRandom*
 public abstract double **getRandom**()
 – **Returns** – A random number that has a probability distribution of Phase-Type

- *getRandom*
 public abstract double[] **getRandom**(int num)
 – **Parameters**
 * num – Number of variates to be generated
 – **Returns** – A vector of random numbers that have a probability distribution of Phase-Type

- *initialize*
 protected abstract void **initialize**()
 – **Description copied from PhaseGenerator** (see 9.1.4, page 391)
 Initialize the cutoff values and the aliases for the initial probability distribution and the transition probability matrix

Members inherited from class `jphase.generator.PhaseGenerator` (see 9.1.4, page 391)

- `public abstract double getRandom()`
- `public abstract double getRandom(int num)`
- `public PhaseVar getVar()`
- `protected abstract void initialize()`
- `protected var`

9.1.4 Class PhaseGenerator

This abstract class defines the behaviour that any Phase-Type random number generator should have

Declaration

```
public abstract class PhaseGenerator
  extends java.lang.Object
```

All known subclasses

NeutsDiscPHGenerator (see 9.1.3, page 389) , NeutsContPHGenerator (see 9.1.2, page 387)

Field summary

var Phase variable from which the random numbers must be generated

Constructor summary

PhaseGenerator(PhaseVar) Construcs a new PhaseGenerator through its initialization

Method summary

getRandom()
getRandom(int)
getVar()
initialize() Initialize the cutoff values and the aliases for the initial probability distribution and the transition probability matrix

Fields

- protected jphase.PhaseVar **var**
 - Phase variable from which the random numbers must be generated

Constructors

- *PhaseGenerator*

```
public PhaseGenerator( jphase.PhaseVar var )
```

 - **Description**
 Construcs a new PhaseGenerator through its initialization
 - **Parameters**
 - * **var** – variable from which the random numbers must be generated

Methods

- *getRandom*
public abstract double **getRandom**()

– **Returns** – A random number that has a probability distribution of Phase-Type

- *getRandom*
public abstract double[] **getRandom**(int num)

– **Parameters**
 * num – Number of variates to be generated

– **Returns** – A vector of random numbers that have a probability distribution of Phase-Type

- *getVar*
public jphase.PhaseVar **getVar**()

– **Returns** – Phase variable that is being used to generate the random numbers

- *initialize*
protected abstract void **initialize**()

– **Description**
 Initialize the cutoff values and the aliases for the initial probability distribution and the transition probability matrix

Index

- A, 283, 287, 330, 333
- absorbingStates, 162
- AbstractAverageSolver, 177
- AbstractAverageSolver(CTMDP), 177
- AbstractAverageSolver(DTMDP), 177
- AbstractContPhaseVar, 270
- AbstractContPhaseVar(), 271
- AbstractDiscountedSolver, 179
- AbstractDiscountedSolver(CTMDP, double), 179
- AbstractDiscountedSolver(DTMDP, double), 180
- AbstractDiscPhaseVar, 276
- AbstractDiscPhaseVar(), 277
- AbstractFiniteSolver, 182
- AbstractFiniteSolver(FiniteMDP), 182
- AbstractInfiniteSolver, 184
- AbstractInfiniteSolver(CTMDP), 184
- AbstractInfiniteSolver(DTMDP), 184
- AbstractTotalSolver, 186
- AbstractTotalSolver(DTMDP), 186
- Action, 71
- Action(), 71
- Actions*, 61
- ActionsSet, 73
- ActionsSet(), 73
- ActionsSet(A), 73
- ActionsSet(A[]), 73
- ActionsSet(Actions), 74
- ActionsSet(Iterable), 74
- active(GeomState, E), 14
- active(S, E), 56
- active(Sub, int, E), 14
- activeEvents(S), 136
- activeEvents(S, A), 131, 143, 147
- activeEvents(S, A, int), 157
- activeState, 127
- activeTransitions(S, E), 33, 56
- add(A), 74
- add(E), 62, 80
- add(Iterable), 105
- add(PhaseVar), 325
- add(S), 105
- add(S, double), 69, 110
- add(States), 106
- add(Transition), 69, 110
- add(Transitions), 111
- addMOP(String), 34
- addRate(S, double), 69, 111
- addTerm(double, int), 328
- addTerm(double, int, double), 337
- addTerm(Term), 338
- aExpStep, 302
- algorKS(double[], PhaseVar), 384
- aliasCut(double[], int[], double[]), 385
- aliasCut(Vector, int[], double[]), 385
- allToString(), 34
- alpha, 283, 288, 330, 333
- argmax(double[]), 385
- argmin(double[]), 385
- average(double[]), 309
- average2(double[]), 309
- bestAction(S), 221, 225
- bestPolicy(S), 188
- BiCG, 246
- BiCGstab, 246
- binomial(int, int), 348
- boundary, 23
- buildSolution(), 174, 195, 203
- canGo(), 34
- cdf(double), 266, 271, 277, 297
- cdf(int, double), 266, 271, 277, 298
- ceil(double, double), 365
- CGS, 246
- clearMOPs(), 34
- clone(), 65, 85, 89, 93, 338, 343
- cnt, 32
- coeff, 343
- compareTo(Action), 86
- compareTo(Event), 78, 89
- compareTo(PropertiesAction), 86
- compareTo(PropertiesEvent), 89
- compareTo(PropertiesState), 93

- compareTo(State), 24, 27, 93, 97, 102
- compareTo(Term), 343
- compExp(), 302
- compLog(), 302
- computeMOPs(MarkovProcess), 24, 27, 93, 97, 102
- computeNoErrorBounds(), 225
- computeWithErrorBounds(), 226
- concatCols(DenseMatrix, DenseMatrix), 309
- concatCols(Matrix, Matrix), 302
- concatCols(Matrix, Matrix, Matrix), 309
- concatQuad(Matrix, Matrix, Matrix, Matrix, Matrix), 310
- concatRows(DenseMatrix, DenseMatrix), 310
- concatRows(Matrix, Matrix), 303
- concatRows(Matrix, Matrix, Matrix), 310
- concatVectors(DenseVector, DenseVector), 311
- concatVectors(Vector, Vector, Vector), 311
- contains(Event), 81
- contains(S), 106
- continuousCost(S, A), 127, 132
- continuousCost(S, A, E), 132, 136
- continuousCost(StateEvent, A), 136
- ContPhaseFitter, 354
- ContPhaseFitter(double[]), 354
- ContPhaseVar*, 256
- converter, 127
- convoExpo(double), 292
- convolution(SuperErlang, SuperErlang), 338
- convolution(Term, Term), 343
- convolutionUseExp(Term, Term), 344
- copy(), 257, 261, 266, 284, 289, 293, 298, 331, 334
- Coxian(int, double[], double[]), 284
- CT2DTConverter, 123
- CT2DTConverter(CTMDP), 123
- CTMDP, 126
- CTMDP(States), 127
- CTMDPEv, 131
- CTMDPEv(States), 131
- CTMDPEvA, 135
- CTMDPEvA(States), 135
- CV(), 266, 271, 277
- CV(double[]), 311
- data, 354, 356
- debug(int, String), 11, 35, 166
- debug(int, String, boolean), 11, 35, 166
- debug(int, String, boolean, boolean), 11, 35, 167
- DebugReporter, 10
- DebugReporter(int), 10
- DebugReporter(PrintWriter), 10
- DecisionRule, 75
- DecisionRule(), 75
- DecisionRule(DecisionRule), 75
- defaultGeometrixSolver, 14
- defaultSteadyStateSolver, 33
- defaultTransientSolver, 33
- defIntegrate(), 338, 344
- defIntegrate(double), 338, 344
- DenseContPhaseVar, 282
- DenseContPhaseVar(), 283
- DenseContPhaseVar(DenseVector, DenseMatrix), 283
- DenseContPhaseVar(double[], double[][]), 283
- DenseContPhaseVar(int), 283
- DenseContPhaseVar(Vector, Matrix), 284
- DenseDiscPhaseVar, 287
- DenseDiscPhaseVar(), 288
- DenseDiscPhaseVar(DenseVector, DenseMatrix), 288
- DenseDiscPhaseVar(double[], double[][]), 288
- DenseDiscPhaseVar(int), 288
- DenseDiscPhaseVar(Vector, Matrix), 288
- denseMatrixToString(), 35
- denseMatrixToString(int, int, boolean, boolean), 35
- derive(), 338, 344
- description(), 27, 36, 64, 71, 75, 78, 97, 108, 111, 114, 195, 207, 217, 222, 226, 234, 236, 238, 241, 271, 277, 293, 298
- destination(S, A, int), 151
- dests(GeomState, E), 15
- dests(S, E), 56
- dests(Sub, int, E), 15
- DIA, 244
- discountFactor, 179
- DiscPhaseFitter, 356
- DiscPhaseFitter(int[]), 356
- DiscPhaseVar*, 261
- distance(double[], double[]), 311, 348
- doFitHyperErlang(double[]), 359
- doFitN(double[]), 362, 364
- doFitNM(double[], HyperErlangVar), 359
- doFitNMR(double[], HyperErlangVar), 359
- DTMDP, 139
- DTMDP(S), 139
- DTMDP(States), 139
- DTMDPEv, 143

- DTMDPEv(States), 143
- DTMDPEvA, 147
- DTMDPEvA(States), 147
- EMHyperErlangFit, 358
- EMHyperErlangFit(double[]), 358
- EMHyperExpoFit, 361
- EMHyperExpoFit(double[]), 361
- EMPhaseFit, 363
- EMPhaseFit(double[]), 363
- eqResidualTime(), 257, 271
- equalPowers(Term), 344
- equals(Object), 64, 71, 76, 79, 97
- equals(PropertiesEvent), 89
- Erlang(double, int), 284
- erlang(double, int, Random), 386
- ErlangCoxian(int, double, double, double, double, double), 284
- ErlangCoxianVar, 291
- ErlangCoxianVar(), 292
- ErlangCoxianVar(int), 292
- ErlangCoxianVar(int, double, double, double, double, double), 292
- ERROR, 53
- evalPoints, 363
- evaluate(), 344
- evaluate(double), 328, 344
- Event, 78
- Event(), 78
- eventRatesToString(int, int), 36
- Events*, 62
- eventsRatesToString(), 36
- EventsSet, 80
- EventsSet(), 80
- EventsSet(E[]), 80
- exitRate(S, A), 123
- exp(), 338
- exp(double), 303
- exp(double, Matrix, Matrix), 303
- exp(int, double, Matrix, Matrix), 303
- exp(Matrix, double), 311
- exp(Matrix, double, Matrix, Matrix), 312
- exp(Matrix, double, Matrix, Matrix, boolean), 312
- exp(Matrix, double, Vector, Vector), 312
- exp(Matrix, int, double, Matrix, Matrix, boolean), 313
- exp(Matrix, int, double, Vector, Vector, boolean), 313
- expand(double), 338, 345
- expectedValue(), 266, 272, 277, 298
- explorationTime, 162
- expo(double), 285
- expRunge(int, double, Matrix, Matrix), 303
- expRunge(Matrix, int, double, Matrix, Matrix), 313
- expTimesOnes(double), 303
- expTimesOnes(double, Matrix), 303
- expTimesOnes(int, double, Matrix), 303
- expTimesOnes(Matrix, double), 314
- expTimesOnes(Matrix, double, Matrix), 314
- expTimesOnes(Matrix, double, Vector), 314
- expTimesOnes(Matrix, int, double, Matrix), 314
- expTimesOnes(Matrix, int, double, Vector), 315
- expUnif(double, Matrix, Matrix), 303
- expUnif(double[], Matrix, Matrix), 303
- expUnif(double[], Matrix, Matrix, int), 303
- expUnif(int, double, Matrix, Matrix), 303
- expUnif(int, double, Matrix, Matrix, int), 304
- expUnif(Matrix, double, Matrix, Matrix), 316
- expUnif(Matrix, double[], Matrix, Matrix), 315
- expUnif(Matrix, double[], Matrix, Matrix, int), 315
- expUnif(Matrix, double[], Vector, Vector, int), 315
- expUnif(Matrix, int, double, Matrix, Matrix), 316
- expUnif(Matrix, int, double, Matrix, Matrix, int), 316
- expUnif(Matrix, int, double, Vector, Vector), 317
- expUnif(Matrix, int, double, Vector, Vector, int), 317
- fact(int), 349
- factMomentK(int[], int), 365
- feasibleAct(S), 136, 148
- feasibleActions(S), 124, 162
- feasibleActions(S, int), 154
- feasibleActions(StateEvent), 136, 148
- fileName, 324
- finalCost(S), 154
- finalize(), 36
- finite, 166
- FiniteDP, 151
- FiniteDP(States, int), 151
- FiniteMDP, 153
- FiniteMDP(S, int), 154
- FiniteMDP(States, int), 154

FiniteMDPEv, 157
 FiniteMDPEv(States, int), 157
 FiniteSolver, 188
 FiniteSolver(FiniteMDP), 188
 fit(), 353, 355, 357, 359, 362, 364, 370, 371, 373, 379, 381
 FitterUtils, 365
 FitterUtils(), 365
 floor(double, double), 366
 future(S, A), 222
 future(S, A, double), 180
 future(S, A, double, ValueFunction), 180
 future(S, A, int), 189

 gammaP(double, double), 349
 generate(), 36, 127, 136, 140, 162
 GENERATED, 53
 GeneratorUtils, 384
 GeneratorUtils(), 384
 Geom(double), 289
 GeometricSolver, 230
 GeometricSolver(MarkovProcess), 230
 GeometrixSolver, 14, 232
 GeometrixSolver(MarkovProcess), 232
 GeomProcess, 13
 GeomProcess(Sub, EventsSet), 14
 GeomRelState, 23
 GeomRelState(Sub), 24
 GeomRelState(Sub, int), 24
 GeomState, 26
 GeomState(Sub, int), 26
 get(), 114
 get(S), 106, 114
 getAction(S), 76
 getAction(S, int), 83
 getAllStates(), 127, 163
 getAlphas(), 298
 getAMatrices(), 15
 getBMatrices(), 15
 getBoundaryStates(), 15
 getBuildTime(), 174, 192, 195, 199, 203
 getCoeff(), 345
 getCurLevel(), 11
 getCurrentIterSolver(), 241
 getCurrentPreConditioner(), 241
 getDebugLevel(), 11, 36, 167
 getDebugReporter(), 37
 getDecisionRule(), 83
 getDecisionRule(int), 83
 getDefaultAverageSolver(), 163
 getDefaultDiscountedSolver(double), 163
 getDefaultGeometrixSolver(), 16
 getDefaultSolver(), 155, 163, 167
 getDefaultSteadyStateSolver(), 37
 getDefaultTransientSolver(), 37
 getDegree(), 345
 getDiscreteProblem(), 185
 getDiscSolver(), 199
 getEpsilon(), 226
 getEvent(), 103
 getEventClass(), 37
 getEventNames(), 37
 getEventRate(int), 16, 37
 getEvents(), 38
 getEventsRates(), 38
 getExpectedLevel(), 16
 getFinalRate(S, S), 38
 getGenerator(), 38, 241
 getGenMatrix(), 241
 getGeometrixSolver(), 16
 getHorizon(), 83, 155
 getIncreasingFactor(), 207
 getIndex(), 79, 97
 getInitialIterations(), 207
 getInitialSol(), 16
 getInitialState(), 38
 getInterestRate(), 180
 getIteartiveSolver(Vector), 241
 getIterations(), 185, 192, 195, 199, 207, 214, 222, 226
 getIterativeSolver(Vector, MtjSolver.EnumSolver), 242
 getLambda(), 345
 getLambdas(), 298
 getLambdaX1(), 293
 getLambdaX2(), 293
 getLambdaY(), 293
 getLevel(), 27
 getLogLikelihood(), 355, 357, 367, 368
 getLpSolveTime(), 175, 192, 195, 199, 203
 getM(), 298
 getMat0(), 267, 272, 277
 getMat0Array(), 267, 272, 277
 getMatrix(), 267, 285, 289, 293, 298, 331, 335
 getMatrixArray(), 267, 272, 278
 getMaxRate(), 128
 getMaxStates(), 38
 getMeans(), 325
 getMOP(int), 27, 98
 getMOP(String, MarkovProcess), 98

getMOPIndex(String), 39
 getMOPNames(), 39
 getMOPNames(int), 39
 getMOPsAvg(), 39
 getMOPsAvg(int), 39
 getMOPsAvg(String), 40
 getMOPsMoment(int), 40
 getMOPsMoment(int, int), 17, 40
 getMOPsMoment(String, int), 40
 getMP(), 249
 getMpsFile(), 176, 199, 203
 getMpsFileName(), 176, 199, 203
 getMtjGenerator(), 41
 getMtjRates(), 41
 getN(), 293, 298
 getName(), 246
 getNumber(double[], int[], double[], Random), 386
 getNumBoundaryStates(), 17
 getNumPhases(), 267, 272, 278
 getNumProps(), 65, 86, 89, 93
 getNumStates(), 41, 163
 getNumTypicalStates(), 17
 getOptimalPolicy(), 167, 217
 getOptimalValueFunction(), 167, 218
 getP(), 293
 getParam(), 380, 382
 getPolicy(), 95
 getProbability(), 211
 getProblem(), 182, 185, 218, 222
 getProcessTime(), 189, 192, 195, 199, 203, 207, 214, 218, 222, 226, 242
 getProgress(), 41
 getProperties(), 65, 86, 89, 93
 getProperty(int), 66, 86, 89, 94
 getPx(), 293
 getR(), 298
 getRandom(), 387, 389, 392
 getRandom(int), 387, 389, 392
 getRate(), 108
 getRate(S), 70, 111
 getRate(S, S), 41
 getRates(), 41
 getRates(S), 42
 getRelLevel(), 24
 getReporter(), 168
 getRmatrix(), 17, 230, 232, 238
 getSet(), 79
 getSize(double, double), 372
 getSolBuildTime(), 175, 192, 195, 199, 203
 getSolver(), 163, 168
 getState(), 103, 109
 getStateClass(), 17, 42
 getStates(), 17, 42
 getStates(boolean), 42
 getStates(int), 155
 getStatus(), 42
 getStatusMsg(), 43
 getSteadyState(), 43, 234, 242, 250
 getSteadyState(int), 18
 getSteadyStateProbabilities(), 128, 140
 getSteadyStateSolver(), 43
 getSubMatrices(int, int, int, int), 18
 getSubState(), 24, 27
 getTransientProbs(double, State), 237, 253
 getTransientProbs(double[], State), 236, 252
 getTransientProbs(int, double, State), 237, 253
 getTransientSolver(), 43
 getTypicalStates(), 18
 getValueFunction(), 95, 218
 getVar(), 392
 getVec0(), 267, 272, 278
 getVector(), 267, 285, 289, 293, 299, 332, 335
 getVectorArray(), 267, 272, 278
 getVectorPi0(), 18
 getVectorPi1(), 18
 getVectorPi1Mod(), 19
 getWorkingDir(), 176, 199, 203
 GMRES, 246
 go(), 43
 goStep(), 44
 hasAbsorbingState, 162
 hideGUI(), 44
 hLine(int), 44
 horizon, 154
 HyperErlang(HyperErlangVar), 285
 HyperErlang(int, double[], int[], double[]), 285
 HyperErlangVar, 296
 HyperErlangVar(), 297
 HyperErlangVar(int), 297
 HyperErlangVar(int, int, int[], double[], double[], boolean), 297
 HyperErlangVar(int[], double[], double[], boolean), 297
 HyperExpo(double[], double[]), 286
 IDEN, 244
 identity(int), 304
 IDLE, 53

- ILU, 244
- immediateCost(S, A), 124, 140, 144
- immediateCost(S, A, E), 144, 148
- immediateCost(S, A, E, int), 158
- immediateCost(S, A, int), 155, 158
- immediateCost(StateEvent, A), 148
- indexOfName(String), 325
- InfiniteMDP, 161
- InfiniteMDP(States), 162
- init(), 226
- initial, 166
- initialize(), 387, 389, 392
- initSet, 135
- initUpperTriangular(int), 349
- integrate(), 339, 345
- integrateCom(), 339, 345
- inverse(), 304
- isAverage(), 226
- isAvg(), 203
- isBoundary(), 24, 27
- isClosed(), 67, 106
- isConsistent(), 25, 27, 94, 98, 103
- isConstant(), 345
- isDirty, 324
- isFinite(), 168
- isGenerated(), 44
- isPTerm(), 346
- isSolved(), 168, 211, 218
- isStable(), 19
- isStochastic(), 304
- isTerminal(), 100
- isTryOthers(), 242
- isZero(), 339, 346
- iterations, 206, 225
- iterator(), 61, 62, 67, 74, 76, 81, 106, 111, 114
- JamaSolver, 234
- JamaSolver(MarkovProcess), 234
- JamaTransientSolver, 236
- JamaTransientSolver(MarkovProcess), 236
- JMarkovElement*, 63
- killGUI(), 44
- kronecker(Matrix), 304
- kronecker(Matrix, Matrix), 304, 317
- kronecker(Matrix, Matrix, Matrix), 318
- kronecker(Matrix, Vector, Matrix), 318
- kronecker(Vector, Matrix, Matrix), 318
- kroneckerSum(Matrix), 304
- kroneckerSum(Matrix, Matrix), 304, 318
- kroneckerSum(Matrix, Matrix, Matrix), 319
- kroneckerVectors(DenseVector, DenseVector), 319
- kroneckerVectors(Vector, Vector, Vector), 319
- label(), 25, 28, 44, 64, 72, 76, 79, 86, 90, 94, 98, 103, 109, 111, 115, 189, 192, 195, 207, 214, 218, 222, 226, 234, 237, 238, 242, 244, 249, 272, 278
- lambda, 343
- ldaMax, 302
- level, 26
- lnBinomial(int, int), 349
- lnFactorial(int), 349
- lnGamma(double), 350
- lnPermut(int, int), 350
- loadGUI(), 44
- logPrecision, 363
- lossFunction1(double), 268, 272, 278
- lossFunction2(double), 268, 273, 278
- LPBCLAverageSolver, 191
- LPBCLAverageSolver(DTMDP), 191
- LPBCLDiscountedSolver, 194
- LPBCLDiscountedSolver(DTMDP, double), 194
- LPSolver*, 174
- lpSolveTime, 191
- lumpCost(S, A), 128, 132
- lumpCost(S, A, E), 132, 137
- lumpCost(StateEvent, A), 137
- m1, 375, 377
- m2, 375, 377
- m3, 375, 377
- MarkovMatrix, 301
- MarkovMatrix(double[][]), 302
- MarkovMatrix(Matrix), 302
- MarkovProcess, 29
- MarkovProcess(), 33
- MarkovProcess(S, EventsSet), 33
- MarkovProcess(S, EventsSet, String), 33
- MarkovProcess.Status, 53
- matPower(Matrix, int), 319
- matPower(Matrix, int, Vector, Vector), 320
- matrixRtoArray(), 19
- MatrixUtils, 307
- MatrixUtils(), 309
- max(ContPhaseVar), 257, 273
- max(ContPhaseVar, ContPhaseVar), 257, 273
- max(DiscPhaseVar), 262, 278
- max(DiscPhaseVar, DiscPhaseVar), 262, 278

- maxPower, 302
- maxRate, 127
- MDP, 165
- MDP(), 166
- median(), 268, 273, 279
- min(ContPhaseVar), 257, 273
- min(ContPhaseVar, ContPhaseVar), 257, 273
- min(DiscPhaseVar), 262, 279
- min(DiscPhaseVar, DiscPhaseVar), 262, 279
- mix(double, ContPhaseVar), 258, 273
- mix(double, ContPhaseVar, ContPhaseVar), 258, 273
- mix(double, DiscPhaseVar), 262, 279
- mix(double, DiscPhaseVar, DiscPhaseVar), 263, 279
- mixtureExpo(double, double), 294
- MLContPhaseFitter, 367
- MLContPhaseFitter(double[]), 367
- MLDiscPhaseFitter, 368
- MLDiscPhaseFitter(int[]), 368
- modifiedProb(S, S, A), 170
- moment(int), 268, 273, 279, 299, 339, 346
- MomentsACPH2Fit, 369
- MomentsACPH2Fit(double, double, double), 369
- MomentsACPH2Fit(double[]), 369
- MomentsACPHFit, 371
- MomentsACPHFit(double, double, double), 371
- MomentsACPHFit(double[]), 371
- MomentsADPH2Fit, 373
- MomentsADPH2Fit(double, double, double), 373
- MomentsADPH2Fit(int[]), 373
- MomentsContPhaseFitter, 375
- MomentsContPhaseFitter(double, double, double), 375
- MomentsContPhaseFitter(double[]), 375
- MomentsDiscPhaseFitter, 377
- MomentsDiscPhaseFitter(double, double, double), 377
- MomentsDiscPhaseFitter(int[]), 377
- MomentsECCompleteFit, 379
- MomentsECCompleteFit(double, double, double), 379
- MomentsECCompleteFit(double[]), 379
- MomentsECPositiveFit, 381
- MomentsECPositiveFit(double, double, double), 381
- MomentsECPositiveFit(double[]), 381
- MOPsToString(), 44
- MOPsToString(int, int), 45
- move(double), 339, 346
- mp, 248
- MpsLpAverageSolver, 198
- MpsLpAverageSolver(DTMDP), 198
- MpsLpAverageSolver(DTMDP, String, String), 198
- MpsLpDiscountedSolver, 201
- MpsLpDiscountedSolver(DTMDP, double), 201
- MpsLpDiscountedSolver(DTMDP, double, boolean), 202
- MpsLpDiscountedSolver(DTMDP, double, String, String), 202
- MpsLpDiscountedSolver(DTMDP, double, String, String, boolean), 202
- MpsLpSolver*, 176
- MtjLogRedSolver, 238
- MtjLogRedSolver(GeomProcess), 238
- MtjSolver, 240
- MtjSolver(MarkovProcess), 240
- MtjSolver(MarkovProcess, MtjSolver.EnumSolver), 240
- MtjSolver(MarkovProcess, MtjSolver.EnumSolver, boolean), 241
- MtjSolver.EnumPrecond, 244
- MtjSolver.EnumSolver, 246
- multiply(SuperErlang), 339
- multiply(SuperErlang, SuperErlang), 340
- multiply(Term), 346
- multiply(Term, Term), 346
- multVector(Vector, Vector, Matrix), 320
- name, 33, 244, 246, 324
- names, 323
- NegativeBinomial(double, int), 289
- NeutsContPHGenerator, 387
- NeutsContPHGenerator(AbstractContPhaseVar), 387
- NeutsDiscPHGenerator, 389
- NeutsDiscPHGenerator(AbstractDiscPhaseVar), 389
- newUniqueVarName(String), 325
- newVar(int), 258, 263, 286, 289, 294, 299, 332, 335
- NoModel, 53
- NonStochasticException, 117
- NonStochasticException(String), 117
- normal, 302
- NotUnichainException, 118
- NotUnichainException(String), 118

NotUnchainException(String, Throwable), 118
 numerateStates(), 67, 106
 numMOPs(), 45
 numStates, 162
 numTerms(), 340
 numVars(), 325

 oldExp(double, Matrix), 304
 ONE, 337, 343
 OnesCol(int), 320
 OnesRow(int), 320
 oneStageReachable(States), 128, 140
 OnesVector(int), 320
 OnesVector(Vector), 321
 open(String), 325
 openTxt(String), 326
 operation(double, double), 168

 pad(double, int), 45
 pad(double, int, boolean), 45
 pad(double, int, int), 45
 pad(double, int, int, boolean), 46
 pad(String, int), 46
 pad(String, int, boolean), 46
 pause(), 46
 pdf(double), 258, 274, 299
 pdf(int, double), 258, 274, 299
 permut(int, int), 350
PhaseFitter, 353
 PhaseGenerator, 391
 PhaseGenerator(PhaseVar), 391
PhaseVar, 265
 PhaseVarSet, 323
 PhaseVarSet(), 324
 PhaseVarSet(PhaseVar[]), 324
 PhaseVarSet(String), 324
 PhaseVarSet(String, PhaseVar[]), 324
 plus(double), 304
 pmf(int), 263, 279
 pmf(int, int), 263, 279
 Policy, 82
 policy, 217
 Policy(DecisionRule), 82
 Policy(int), 82
 PolicyIterationSolver, 205
 PolicyIterationSolver(DTMDP, double), 206
 PolicyIterationSolver(DTMDP, double,
 boolean), 206
 Poly, 328
 Poly(), 328
 poly(double, int), 340
 Poly(double[]), 328
 pow(double, int), 321, 350
 pow(int), 304
 power, 343
 power(int), 304
 powerMomentK(double[], int), 366
 powers, 302
 precision, 358, 361, 363, 369
 precisionCV, 358
 precisionParam, 361, 363
 print(), 76, 83
 print(PrintWriter), 76, 83, 115
 print(PrintWriter, String, String), 77, 115
 printAll(), 46
 printAll(PrintWriter), 19, 46
 printDenseMatrix(PrintWriter), 47
 printDenseMatrix(PrintWriter, int, int,
 boolean, boolean), 47
 printDenseMatrix(PrintWriter, int, int,
 boolean, boolean, int[]), 47
 printEventsRates(PrintWriter), 47
 printEventsRates(PrintWriter, int, int), 48
 printMOPs(), 48
 printMOPs(PrintWriter), 48
 printMOPs(PrintWriter, int, int), 48
 printProcessTime, 217
 printSolution(), 168, 219
 printSolution(PrintWriter), 168, 185, 219
 printStates(PrintWriter), 49
 printStates(PrintWriter, int, int), 19, 49
 printValueFunction, 217
 prob(double, double), 268, 274, 279
 prob(S, E), 144, 148
 prob(S, E, int), 158
 prob(S, S, A), 124, 140, 144
 prob(S, S, A, E), 145, 148
 prob(S, S, A, E, int), 158
 prob(S, S, A, int), 152, 155, 159
 prob(StateEvent, StateEvent, A), 149
 probability, 162
 ProbabilitySolver, 210
 probabilitySolver, 162
 ProbabilitySolver(CTMDP), 210
 ProbabilitySolver(CTMDP, DecisionRule), 210
 ProbabilitySolver(DTMDP), 211
 ProbabilitySolver(DTMDP, DecisionRule), 211
 problem, 217
 processTime, 206, 225
 prop, 91

PropertiesAction, 85
 PropertiesAction(int), 85
 PropertiesAction(int[]), 85
PropertiesElement, 65
 PropertiesEvent, 88
 PropertiesEvent(int), 88
 PropertiesEvent(int[]), 88
 PropertiesState, 91
 PropertiesState(int), 92
 PropertiesState(int[]), 92
 PropertiesState(int[], boolean), 92
 PropertiesState(PropertiesState), 92

 QMR, 246
 quantil(double), 268, 274, 280

 rate(GeomState, GeomState, E), 20
 rate(S, S, A), 128, 133
 rate(S, S, A, E), 133, 137
 rate(S, S, E), 56
 rate(StateEvent, StateEvent, A), 137
 rate(Sub, int, Sub, int, E), 20
 reachable(S, A), 124, 128, 133, 141, 145
 reachable(S, A, E), 133, 145, 149
 reachable(S, A, E, int), 159
 reachable(S, A, int), 152, 155, 159
 reachable(StateEvent, A), 137, 149
 reached(S, A, E), 137
 readTxt(String), 304
 RelativeValueIterationSolver, 213
 RelativeValueIterationSolver(CTMDP), 213
 RelativeValueIterationSolver(CTMDP, double),
 213
 RelativeValueIterationSolver(DTMDP), 214
 RelativeValueIterationSolver(DTMDP, double),
 214
 remove(PhaseVar), 326
 remove(S), 107
 remove(String), 326
 reporter, 166
 reset(), 20, 49
 reset(boolean), 49
 resetResults(), 49
 residualTime(double), 259, 274
 residualVar(double), 259, 274
 rLevel, 23
 RUNNING, 53

 save(), 326
 save(String), 326
 saveTxt(), 326

 saveTxt(String), 327
 scalar(), 305
 scalar(Matrix), 321
 set(S, A), 77
 set(S, double), 115
 setAlphas(double[]), 299
 setCoeff(double), 347
 setConverter(CT2DTConverter), 129
 setCurLevel(int), 11
 setCurrentIterSolver(MtjSolver.EnumSolver),
 242
 setCurrentPreConditioner(MtjSolver.EnumPrecond),
 243
 setDebugLevel(int), 12, 50, 169
 setDebugReporter(DebugReporter), 50
 setDecisionRule(DecisionRule), 83
 setDecisionRule(DecisionRule, int), 84
 setDegree(int), 347
 setDiscountFactor(double), 180
 setEpsilon(double), 227
 setEventSet(EventsSet), 50
 setFactor(double), 214
 setGaussSeidel(boolean), 211
 setGeometrixSolver(GeometricSolver), 20
 setHorizon(int), 156
 setIncreasingFactor(double), 207
 setInitialIterations(int), 208
 setInitialState(S), 50
 setInterestRate(double), 163, 180
 setIterSolver(MtjSolver.EnumSolver, boolean),
 243
 setJacobi(boolean), 212
 setLambda(double), 347
 setLambdas(double[]), 299
 setLambdaX1(double), 294
 setLambdaX2(double), 294
 setLambdaY(double), 294
 setM(int), 299
 setMatrix(Matrix), 269, 286, 290, 294, 299, 332,
 335
 setMaxStates(long), 50
 setModifiedPolicy(boolean), 208
 setMOP(int, double), 98
 setMOP(MarkovProcess, String, double), 28, 99
 setMOPs(String[]), 50
 setN(int), 294, 300
 setP(double), 294
 setPrintProcessTime(boolean), 219
 setPrintValueFunction(boolean), 215, 219
 setProbabilitySolver(ProbabilitySolver), 141

- setProperty(int, int), 86, 90, 94
- setPx(double), 295
- setR(int[]), 300
- setReporter(DebugReporter), 169
- setSolver(Solver), 169
- setSteadyStateSolver(SteadyStateSolver), 51
- setTransientSolver(TransientSolver), 51
- setTryOthers(boolean), 243
- setVector(Vector), 269, 286, 290, 295, 300, 332, 335
- showGUI(), 51
- SimpleMarkovProcess, 55
- SimpleMarkovProcess(), 55
- SimpleMarkovProcess(S, EventsSet), 55
- SimpleMarkovProcess(S, EventsSet, String), 55
- size(), 61, 62, 68, 70, 74, 77, 81, 107, 112, 305
- Solution, 95
- Solution(ValueFunction, Policy), 95
- solve(), 169, 189, 192, 196, 200, 203, 208, 212, 215, 219, 222, 227
- solve(double), 129, 141
- solved, 217
- solveLP(), 175, 196, 204
- solveMatrix(), 208
- solveMatrixModified(DecisionRule), 208
- Solver, 216, 248
- Solver(MarkovProcess), 248
- Solver(MDP), 217
- SolverException, 120
- SolverException(String), 120
- SolverException(String, Throwable), 120
- solveTranspose(Matrix), 305
- SparseContPhaseVar, 330
- SparseContPhaseVar(double[], double[][]), 331
- SparseContPhaseVar(int), 331
- SparseContPhaseVar(SparseVector, FlexCompRowMatrix), 331
- SparseContPhaseVar(Vector, Matrix), 331
- SparseDiscPhaseVar, 333
- SparseDiscPhaseVar(double[], double[][]), 334
- SparseDiscPhaseVar(int), 334
- SparseDiscPhaseVar(SparseVector, FlexCompRowMatrix), 334
- SparseDiscPhaseVar(Vector, Matrix), 334
- sqrt(double, double), 366
- SSOR, 244
- State, 96
- State(), 97
- StateC, 100
- StateC(), 100
- StateC(boolean), 100
- StateEvent, 102
- StateEvent(S, E), 102
- States, 67
- states, 162
- statesLableMaxWidth(int), 51
- StatesSet, 104
- StatesSet(), 104
- StatesSet(Iterable), 104
- StatesSet(S), 105
- StatesSet(S[]), 105
- StatesSet(States), 105
- statesToString(), 51
- stdDeviation(), 269, 274, 280
- steadyProbabilities(), 20
- SteadyStateSolver, 250
- SteadyStateSolver(MarkovProcess), 250
- StochasticShortestPath, 170
- StochasticShortestPath(States), 170
- StochasticShortestPathSolver, 221
- StochasticShortestPathSolver(StochasticShortestPath), 221
- StructureException, 121
- StructureException(String), 121
- subState, 23, 26
- sum(ContPhaseVar), 259, 274
- sum(ContPhaseVar, ContPhaseVar), 259, 274
- sum(DiscPhaseVar), 263, 280
- sum(DiscPhaseVar, DiscPhaseVar), 264, 280
- sum(double[]), 386
- sum(SuperErlang), 340
- sum(SuperErlang, SuperErlang), 340
- sumAbs(double[]), 386
- sumGeom(double), 259, 264, 275, 280
- sumMatPower(Matrix, int, Vector, Vector), 321
- sumPH(DiscPhaseVar), 260, 264, 275, 280
- sumPH(DiscPhaseVar, ContPhaseVar), 260, 275
- sumPH(DiscPhaseVar, DiscPhaseVar), 264, 280
- SuperErlang, 336
- SuperErlang(), 337
- SuperErlang(double, int, double), 337
- SuperErlang(Term), 337
- survival(double), 269, 275, 280
- survival(int, double), 269, 275, 281
- SUSPENDED, 53
- Term, 342
- Term(double, int, double), 343
- term(int), 340

theStates, 32
 times(double), 260, 275, 305, 341
 times(Matrix), 305
 timesOne(), 305
 toEventArray(), 81
 toMarkovMatrix(Matrix), 305
 toStateArray(), 107
 toString(), 51, 64, 72, 77, 79, 81, 84, 99, 107,
 109, 112, 115, 220, 249, 260, 264, 275,
 281, 305, 327, 341, 347
 toStringE(), 341, 347
 toStringP(), 341, 347
 toStringRTF(), 305, 341, 347
 toTxt(), 305
 TransientSolver, 252
 TransientSolver(MarkovProcess), 252
 Transition, 108
 Transition(S, double), 108
Transitions, 69
 TransitionsSet, 110
 TransitionsSet(), 110

 uminus(), 305
 useErrorBounds(boolean), 227
 useGaussSeidel(boolean), 227
 usesErrorBounds(), 228
 usesGaussSeidel(), 228
 useUniformization, 302
 Utils, 348
 Utils(), 348

 v, 337
 ValueFunction, 113
 valueFunction, 217
 ValueFunction(), 113
 ValueFunction(String), 113
 ValueFunction(ValueFunction), 113
 ValueFunction(ValueFunction, String), 114
 ValueIterationSolver, 224
 ValueIterationSolver(CTMDP, double), 225
 ValueIterationSolver(DTMDP, double), 225
 valueOf(String), 54, 245, 246
 values(), 54, 245, 246
 var, 354, 356, 391
 varAt(int), 327
 variance(), 269, 275, 281
 variance(double[]), 321
 vars, 323
 vLine(), 51

 waitingQ(double), 260, 275

 WRITING, 53

 Zeros(int, int), 305