marmoteCore
user manual

Alain Jean-Marie
Issam Rabhi, Hlib Mykhailenko
Inria

v0.6b, March 14, 2018

0.1 Introduction

`marmoteCore` is a C++ API for the construction and the analysis of Markov Chains. It has veen developed within the MARMOTE project (MARkovian MOdeling Tools and Environments) funded by the Agence Nationale de la Recherche (France), number ANR-12-MONU-0019.

The MARMOTE Software platform was developed with the intent to provide to the general scientist a "modeling environment" which gives access to algorithms developed by specialists. It is intended to be as open as possible, component-oriented, and contributive.

0.2 Markov chains, Markov modeling, Markov modelers

For the purposes of `marmoteCore`, Markov chains (or Markov processes) are a class of stochastic processes $X(t)$, $t \in \mathbb{Z}$ or $\mathbb{R}$

- evolving on a state space $\mathcal{E}$
- described by a transition rule

$$\begin{array}{lcll} i & \longrightarrow & j & \text{with probability } p_{ij}, \text{ for discrete-time Markov chains} \\ i & \longrightarrow & j & \text{with rate } \lambda_{ij}, \text{ for continuous-time Markov chains} \end{array}$$

- from some initial state.

The practical success of Markov chain as models for real-world situations is due in part to the fact that many properties of Markov chains can be obtained

- by analyzing the graph of transitions,
- by solving problems of linear algebra.

Markov modeling consists in

- constructing Markov models
- analyzing them:
  - determine qualitive properties: structure, ergodicity, stability ...
  - compute metrics related with probabilities, frequencies, times, durations ...

At the risk of caricaturing somewhat, the activities of a *Markov modeler* can be classified in two main "profiles":

Theoretician : Aims at developing MC solution methods

- as generic as possible
- yet taking into account the structure of the model

This activity involves:

- invent new formulas/algorithms
- program new methods
- test them on examples/benchmarks
- compare with previous methods (exec. time, accuracy)

Practician : Develops Markov models for specific applications

This activity involves:

- describe/represent model (parameters, structure, ...)
- test model with simulation
- solve model (analytic, numerical), loop until model passes tests
- execute experimental plans
- compare different models (e.g. simplifications)

`marmoteCore` has the ambition to fit the needs of both categories.

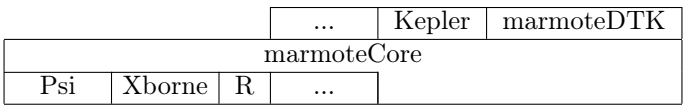
0.3 Architecture

Several requirements have driven the choice of an architecture:

- Modeling abstraction
  - possibility to handle formal models, infinite state spaces, ...
  - possibility to handle hierarchies of models, hierarchies of solution methods.
- Reuse of existing code
  - possibility to communicate with other modeling/solution systems: matlab, R, ...
  - possibility to be used in workflow management systems
  - possibility to implement multiple solution methods for different types of Markov chains

The target architecture has three layers:

- Bottom: solution methods
- Middle: Marmote API, construction of models, handling of data, algorithms, results
- Top: GUI and workflow management



Choice of an object-oriented language: `C++`

Chapter 1

Installation

This section provides instructions for installing the `marmoteCore` library and compiling applications, in particular those provided in the `marmoteApplis` package.

1.1 Installing the core

The latest version of `marmoteCore` is available at `http://marmotecore.gforge.inria.fr/`. It provides libraries for linux systems.[1]

Installation instructions:

1. select a directory for installation, say `MAR_DIR`;
2. become super-user if needed
3. get the tarball from the site above (last version: `marmotecore_1.2.0.tgz`)
4. extract then install with:

```
tar xf marmotecore_1.2.0_plain.tgz
cd marmotecore_1.2.0
make install
```

This will place the header files and the libraries in `MAR_DIR/marmotecore/includes` and `MAR_DIR/marmotecore/lib`, respectively.

1.2 Installing application examples

The latest version of the set of examples is available at `http://marmotecore.gforge.inria.fr/`. It provides application files of various complexities for different application domains.

Installation instructions:

1. select a directory for the applications
2. get the tarball from the site above (last version: `marmoteapplis_1.2.0.tgz`)
3. extract then compile with:

[1] Tested with Fedora 19+.

```
tar xf marmoteapplis_1.2.0.tgz
cd marmoteapplis_1.2.0
make
```

This sequence assumes that the marmoteCore libraries have been installed in /usr/local/marmotecore. If they are in, say, directory MY_MARMOTE, use instead the command:

```
make MARMOTEDIR=MY_MARMOTE
```

Running some applications may require additional setup. See the following sections.

1.2.1 Compiling a marmoteCore application

The Makefiles provided in the marmoteapplis directory automate the compilation process. The easiest way is to adapt them to your needs.

Users not familiar with make may use direct calls to the compiler as follows. This command line assumes that the marmoteCore library has been installed in directory MAR_DIR, and that the application code is in file appli.cpp.

```
g++ -IMAR_DIR/include appli.cpp -o appli -LMAR_DIR/lib -lMarmoteCore -lXborne -lpsi \
    -lboost_thread -lboost_filesystem
```

1.3 Installing supporting environments

Some functionalities of marmoteCore use external programs that must be installed separately.

1.3.1 C++ libraries

Depending on the installation of C++ it may be necessary to install libraries boost_thread and boost_filesystem.

1.3.2 R

Some functionalities of marmoteCore are obtained using the popular R package. In order to take advantage of these, installing the following software is needed.

1. install R, e.g. from the link http://cran.r-project.org/ or through yum/dcf.
2. install the Rcpp: from inside R, issue the command install.packages("Rcpp")
3. then the package RInside with the command install.packages("RInside")
4. then the package markovchain with the command install.packages("markovchain").

1.3.3 Scilab

Scilab is a scientific software comparable to Matlab but not commercial. It is available from

http://www.scilab.org/en

1.3.4 Xborne

Xborne is a suite of applications for creating, manipulating and solving discrete-time Markov chains. It is available upon request from Jean-Michel Fourneau: Jean-Michel.Fourneau@uvsq.fr.

1.3.5 PSI

PSI3 is the environment for Event Modeling of Markov chains and for Perfect Sampling. It is available from:

http://psi.gforge.inria.fr/dokuwiki/

1.4 Running Marmote applications

marmoteCore is designed so as to be able to benefit from the capabilities of external programs.

1.4.1 R

Several methods use the R scientific computation environment and its markovchain package. In order to function, marmoteCore must have been compiled with R enabled, and the necessary R environment must be installed.

1.4.2 PSI

When running applications using PSI1 functionalities, it is necessary to add MAR_DIR/bin in the environment variable PATH.

Chapter 2

Programming with marmoteCore

2.1 Introduction

Practical Markov modeling usually involves the following tasks:

create a Markov model, by specifying the state space and the transitions

analyze the structure of the model, so as to detect qualitative properties and check that the model created is consistent with the model intended

compute metrics associated with the model.

With marmoteCore, these tasks are performed through the creation of objects in a C++ program, and the execution of methods associated with these objects.

2.1.1 The Main Objects

marmoteCore rests on 4 abstract, high-level classes:

- MarkovChain for representing Markov chains
- MarmoteSet for representing state spaces
- TransitionStructure for representing transitions
- Distribution for representing probability distributions.

These main classes and their derived classes will be describe in depth in Chapter 3. In the present chapter, we show through examples how they are used.

2.1.2 Constructing Markov Chains

Creating an instance of MarkovChain objects can be done in one of three ways:

1. read the generator (and the state space) from a file
2. use a predefined class
3. create the generator "by hand".

We describe these three ways below.

2.1.2.1 Getting a Markov Chain from a file

The following code shows three examples of creations of a `MarkovChain` object by reading the model from one or several files:

```
MarkovChain* c1 = new MarkovChain( "Xborne", NULL, 0, "rw1d" );
MarkovChain* c2 = new MarkovChain( "PSI", NULL, 0, "rw1d" );
MarkovChain* c3 = new MarkovChain( "Ers", NULL, 0, "rw1d" );
```

The name of the file is not directly specified. Instead, a model name is supplied (here: `rw1d`) and a file format is specified (here: `Xborne` or `PSI` or `Ers`).

The formats available are described in Appendix D, p. 60. See also 3.1.1.3.

2.1.2.2 Using existing Markov chains

`marmoteCore` provides Markov models of several important families identified in the literature. See Appendix B, p. 57.

Currently implemented (a small part of the Markov Zoo): `TwoStateContinuous`, the two-state continuous-time Markov chain, `Felsenstein81`, a model for BioInformatics, `Homogeneous1DRandomWalk`, `HomogeneousMultiDRandomWalk` and `Homogeneous1DBirthDeath`, models of random walks, in continuous or discrete time.

The following code shows how these models are used.

```
double pro[4] = { 0.1, 0.2, 0.3, 0.4 };
Felsenstein81* c1 = new Felsenstein81( pro, 10.0 );

Homogeneous1DRandomWalk* c3 = new Homogeneous1DRandomWalk( 10, 0.4, 0.3 );
```

The constructor of the `Felsenstein81` object needs an array of probabilities `pro`. The constructor for `Homogeneous1DRandomWalk` just needs a size parameter and two transition probabilities. See respectively §3.1.2.6 and §3.1.2.3.

2.1.2.3 Making a Markov chain

Creating a complex Markov chain is typically done in three steps:

1. create a `MarmoteSet` object, containing the state space representation
2. create a `TransitionStructure` object with the transitions characterizing the model,
3. create the Markov chain from this object.

The first step is optional if the state space is simple enough.

A typical example of creation code following this pattern is:

```
LayeredStateSpace* sp = new LayeredStateSpace( N, E1, E2, M, nu );
SparseMatrix* gen = MakeGenerator( sp, N, E1, E2, M, nu );
MarkovChain* myMC = new MarkovChain( gen );
```

In this example, `SparseMatrix` is a class deriving from `TransitionStructure`, provided by `marmoteCore`. The user has created a class `LayeredStateSpace` which inherits from `MarmoteSet`. Its construction depends on the model parameters `N`, `E1`, `E2`, `M` and `nu`. The user has also programmed a procedure `MakeGenerator` to actually perform the construction.

The code of this method uses the object of type `MarmoteSet` with the following pattern:

```
SparseMatrix* MakeGenerator(LayeredStateSpace* sp, ... ) {

  SparseMatrix* gen = new SparseMatrix( sp->Cardinal() );

  int stateBuffer[5];  // the state space has 5 dimensions
  sp->FirstState(stateBuffer);

  int idx = 0;
  do {
    ...
    // destination state stored in nextBuffer
    nextBuffer[0] = MIN( stateBuffer[0] + 1, someBound );
    ...
    gen->addToEntry( idx, sp->Index(nextBuffer), someRate );
    gen->addToEntry( idx, idx, -someRate );
    ...

    sp->NextState( stateBuffer );
    idx++;
  } while (!sp->IsFirst(stateBuffer));
  ...
  return gen;
}
```

In this procedure, the states of the state space are enumerated in sequence, using the three constructs:

initialization with `sp->FirstState(stateBuffer)`, which sets the state (represented in the array `stateBuffer` to the "first" state of the state space;

increment of the state with `sp->NextState(stateBuffer)`, which moves the state to the next one in the enumeration order;

termination test with `sp->IsFirst(stateBuffer)` which tests whether the enumeration came back to the initial state.

Inside the loop, the current state is modified by the different events to be considered. The result is stored in the variable `nextBuffer`. The rates/probabilities associated with the transition are then added to the generator being constructed with the instruction `gen->addToEntry()`. The index of the destination state is obtained from the state buffer with the fourth construct: `sp->Index(nextBuffer)`.

2.2 Computing on Markov Chains

Many solution methods are available for `MarkovChain` objects and derived classes.

In the following example, we show a comparison of computations for the stationary distribution:

```
// use of specific methods for F81
Felsenstein81* c1 = new Felsenstein81(...);
Distribution* d1 = c1->StationaryDistribution();
Distribution* d2 = c1->SimulateChain(...)->distribution();
// comparison
cout << "Distance L1(d1,d2) = " << d1->distanceL1(d2) << endl;
// use of generic methods for MCs
MarkovChain* c2 = static_cast<MarkovChain*>(c1);
```

```
Distribution* d3 = c2->StationaryDistribution_GaussSeidel();
Distribution* d4 = c2->StationaryDistribution_PowerMethod();
Distribution* d5 = c2->StationaryDistribution_Xborne_LowBound();
Distribution* d6 = c2->StationaryDistributionSample(...);
Distribution* d7 = c2->SimulateChain(...)->distribution();
Distribution* d8 = c2->SimulateChain2(...)->distribution();
```

In the first part of the code, an object of class `Felsenstein81` is created, and two solution methods are used: first `StationaryDistribution()` then `SimulateChain()`. The first one computes exactly the stationary distribution, whereas the second one performs Monte Carlo simulation and returns, in particular, an empirical distribution. The distance between the two distributions is evaluated.

In the second part of the code, the `Felsenstein81` is *cast* to a higer-level `MarkovChain` object, which has more solution methods.

Chapter 3

marmoteCore reference

We describe in this chapter the four main classes of marmoteCore and their derived classes. We present the general interface of these top-level classes. For derived classes, we describe when the general interface has been re-implemented, and when specific methods have been introduced.

As a general rule, attributes of marmoteCore's objects are private and can be accessed (read or write) only through specific "accessor/mutator" methods.

3.1 The MarkovChain object

3.1.1 Common features

The methods common to MarkovChain and derived classes are summarized in the following tables, grouped by functionalities.

3.1.1.1 Definition

This class is accessed with the directive

```
#include "markovChain/markov_chain.h"
```

3.1.1.2 Attributes and accessors

timeType	type_	time type: discrete or continuous
int	state_space_size_	size of the state space
TransitionStructure*	generator_	transition structure of the chain
DiscreteDistribution*	init_distribution_	initial distribution of the process
string	model_name_	name of the model
string	format_	representation format for the model

The "size" of the state space is the number of states it contains.

The generator is supposed to describe the transition structure of the model. It may however be left to null in certain models where the transition structure is implicit.

The initial distribution is used for Monte Carlo simulations. If not set, the Dirac distribution at the state numbered 0 is used by default. It is however advised to set this variable.

The model name and its "format" are usually deduced when the object is created by reading it from a file. See below.

int	state_space_size()	get the size of the model
TransitionStructure*	generator()	get the generator
void	set_init_distribution(DiscreteDistribution* d)	provide the initial distribution
void	setGenerator(TransitionStructure* tr)	provide the generator
void	setFormat(string format)	set the representation format
void	setModelName(string modelName)	set the model name
string	modelName()	get the model name
string	format()	get the format

3.1.1.3 Constuctors

The class provides three constructors:[1]

```
MarkovChain(int sz, timeType t);
MarkovChain(TransitionStructure* tr);
MarkovChain(string format, string param[], int nbParam, string model_name );
MarkovChain(int multipleOfPeriod, int nStates);
```

The constructor `MarkovChain(int sz, timeType t);` creates Markov chain object over a state space of size `sz` and with type given by `t`. The time type can be `DISCRETE` or `CONTINUOUS`. The construction of the transition structure is left to the user.

In the constructor `MarkovChain(TransitionStructure* tr)`, the transition structure is provided. The size of the state space is deduced from it.

The third version creates a `MarkovChain` object by reading from one or several files. The parameter `format` specifies the format or language in which the model is specified. Formats available are: ERS, PSI1/MARCA and XBORNE. See Appendix D for a description of these formats. Depending on this format, one or several file names or extensions must be provided. Those are listed in the `param` array, the size of which is defined by `nbParam`. The parameter `model_name` serves as a common prefix for file names.

The fourth constructor creates randomly a Markov Chain with a specific periodicity given by $d =$`multipleOfPeriod` and `nStates` states. The number of states is actually always a multiple of d: $d \times \lfloor$`nStates`$/d \rfloor$.

When file is not present, or when its parsing fails, a functional `MarkovChain` object is returned, but with an zero-sized state space and an empty generator.

3.1.1.4 Structural analysis

Methods are provided to perform a structural analysis of the Markov chain (in fact, of its transition structure). They come in two groups:

std::vector<int>	AbsorbingStates()	find the states that are absorbing
std::vector<std::vector<int>>	RecurrentClasses()	computes recurrent classes
std::vector<std::vector<int>>	CommunicatingClasses()	computes communicating classes
bool	IsIrreducible()	checks whether the chain is irreducible
bool	IsAccessible(int from, int to)	checks whether a path exists between two states
int	Period()	computes the periodicity of the Markov Chain
std::vector<MarkovChain*>*	SubChains()	computes the decomposition in irreducible subchains

[1] Plus a `Copy()` method.

`std::vector<int>`	`AbsorbingStatesR()`	find the states that are absorbing
`std::vector<std::vector<int>>`	`RecurrentClassesR()`	computes recurrent classes
`std::vector<std::vector<int>>`	`CommunicatingClassesR()`	computes communicating classes
`bool`	`IsIrreducibleR()`	checks whether the chain is irreducible
`bool`	`IsAccessibleR(int from, int to)`	checks whether a path exists between two states

The first group depends on the BFS exploration methods developed within `marmoteCore`. The second group is an interface to the methods of the `markovchain` package of `R`. See Section 1.4.1.

These methods refer to the notion of *communication* in Markov chains: the existence of a path that goes from one state to another. The method `IsAccessible()` checks this for given pairs of nodes.

The "classes" are equivalence classes for the communication relationship. Those are computed by the method `CommunicatingClasses()`. Among the classes, some are *recurrent*. Those are computed by `RecurrentClasses()`. Both methods return a list of classes (using the `vector` template of `C++`'s Standard Template Library) each class being itself a list of nodes.

Absorbing states are those for which $T_{i,i} = 1$ in discrete time or $T_{i,i} = 0$ in continuous time. They are computed by `AbsorbingStates()` and returned as a list of nodes.

3.1.1.5 Monte Carlo Simulation (forward)

Monte Carlo simulation consists in generating a trajectory of the Markov chain model using (pseudo-)random numbers. The standard methods uses the data from the chain's generator to sample transitions from one state to the next one. Details vary slightly according to the time type of the chain.

The generic interface for Monte Carlo simulation is:

```
SimulationResult*  SimulateChain(double t, bool stats, bool traj, bool incr, bool trace)
SimulationResult*  SimulateChainDT(int t, bool stats, bool traj, bool trace)
SimulationResult*  SimulateChainCT(double t, bool stats, bool traj, bool incr, bool trace)
SimulationResult*  SimulatePSI(int t, bool stats, bool traj, bool trace)
SimulationResult*  SimulateChainR( double t, bool stats, bool traj, bool trace );
```

The methods `SimulateChainDT` and `SimulateChainCT` are specific to discrete-time Markov chains. The method `SimulateChainCT` is specific to continuous-time Markov chains. The method `SimulateChain` is the general entry point: it detects the type of the chain and uses one of `SimulateChainCT` or `SimulateChainDT` to perform the simulation. The method `SimulateChainR` is an interface to the simulation procedure for discrete-time chains in the `R` package `markovchain`.

Input Parameters. Four parameters are common to all methods:

`t` : the maximal time up to which the trajectory should be simulated. For discrete-time chains, this is an integer number and it also represents the number of steps to be simulated. For continuous-time chains, the trajectory is simulated up to this value, which may involve an arbitrary number of transitions.

`stats` : a flag specifying whether statistics must be collected along the way. The standard statistic[2] is to collect empirical state probabilities.

`traj` : a flag specifying whether the trajectory should be stored during the simulation. Storing trajectories may require substantial amounts of memory and may not be useful if statistics are collected and/or the trajectory is printed along the way.

[2]This behavior may be modified in the implementation of these methods in derived classes.

`trace` : a flag specifying whether the trajectory should be printed (to the standard output) along the way. Printing trajectories may slow down the execution of the simulation, but may save memory and offload the burden of statistics to another application.

The fifth parameter is specific to continuous-time simulations:

`incr` : a flag specifying whether the time increments between transitions should be collected, and printed along the trajectory if the `trace` flag is set.

Output. Simulation results are stored in a versatile specific object: `simulationResult`. The attributes of a `simulationResult` object are:

`timeType`	`type_`	type of the Markov chain / trajectory
`int`	`state_space_size_`	size of the state space
`int`	`trajectory_size_`	number of transitions in the trajectory
`bool`	`has_distrib_`	flag corresponding to parameter `stats`
`bool`	`has_trajectory_`	flag corresponding to parameter `traj`
`DiscreteDistribution*`	`distrib_`	the empirical distribution collected when `stats`
`double*`	`dates_`	table of transition times when `traj`
`double*`	`increments_`	table of time increments when `traj` and `incr`
`int*`	`states_`	table of state indices when `traj`

Interface: methods from `simulationResult`

```
// constructors
                        SimulationResult(int size, timeType t, bool stats)
                        SimulationResult(string format, string modelName, bool stats)
// accessors
void                    setTrajectory(bool v)
void                    setTrajectorySize(int l)
void                    setTrajectory(double* d, int *s)
void                    setDistribution(DiscreteDistribution *d)
DiscreteDistribution*   distribution()
int                     trajectorySize()
double*                 dates()
int*                    states()
// I/O
void                    writeTrajectory( FILE* out, string format )
```

The methods `dates()` and `states()` give access to the trajectory, the total size of which is obtained with `trajectorySize()`. The statistics that are collected are returned as a distribution object with `distribution()`.

Initial distribution. The simulation must start from some initial state. This state is obtained by sampling from the `initial_distribution_` attribute of the Markov chain. This value can be set with the method `set_init_distribution()`. If not set, the distribution `DiracDistribution(0)` is used.

Random number generation. There is currently no way to interact with the Random Number Generator that is used in sampling distributions and performing Monte Carlo simulation.

3.1.1.6 Exact sampling from the stationary distribution (backwards)

Exact Sampling consists in drawing directly samples from the stationary distribution of a Markov chain. This applies to discrete-time Markov chains (and continuous-time ones after uniformization) and can be done with the "backwards coupling" technique.

marmoteCore supplies a method implementing this technique for general chains. It uses the implementation from the PSI-1 package, see http://psi.gforge.inria.fr/dokuwiki/.

Usage:

```
SimulationResult* StationaryDistributionSample (int nbSamples);
```

In that case, the attributes of the simulationResult object that is returned, have a signification that differs from the one in "Monte Carlo" methods.

The _states attribute (which is accessed through the states() method contains the list of samples that were obtained. The _dates() attribute (accessed through dates()) contains the backward coupling time that was necessary for each sample. The size of both these arrays is equal to the value of the parameter nbSamples of the StationaryDistributionSample() method.

This method uses three external programs: psi_alias, psi_traj and psi_sample. These should be accessible through the $PATH environment variable.

3.1.1.7 Computation of the stationary distribution

Several methods are provided for computing (usually: numerically approximating) the stationary distribution of Markov chains. There are methods with few controls, supposedly easy to use:

Distribution*	StationaryDistribution(bool progress)
Distribution*	StationaryDistributionCT(bool progress)
Distribution*	StationaryDistributionDT(bool progress)
Distribution*	StationaryDistributionGthLD()
Distribution*	StationaryDistributionSOR()
Distribution*	StationaryDistributionR()

and one entry point with detailed controls for iterative methods:

```
Distribution* StationaryDistributionIterative(
                        string method,
                        int tmax,
                        double precision,
                        string initDistribType,
                        DiscreteDistribution* iDis,
                        bool progress ).
```

Linear Algebra methods. The methods StationaryDistributionGthLD() and StationaryDistributionR() use algorithms for solving the linear system

$$\pi T = 0 \quad \text{(continuous time), or} \qquad \pi T = \pi \quad \text{(discrete time)}$$

together with the constraint that π is a probability vector. They are exact, up to numerical errors. They use an amount of memory proportional to the size of a full matrix and may not be suited for large problems.

The method StationaryDistributionR() uses the R environment. See Section 1.4.1.

Iterative methods. The remaining methods are iterative in the sense that they build a sequence of vectors $\pi_0, \pi_1, \ldots, \pi_n, \ldots$ that ideally converges to the solution π. Usually, these methods have several control parameters. These parameters are fixed to some default values in the convenient methods StationaryDistribution(), StationaryDistributionCT(), StationaryDistributionDT() and StationaryDistributionSOR(). The first one is actually a common entry point that selects one of the two following ones depending on the type of the Markov chain.

For cases where a finer control is needed, the method StationaryDistributionIterative() is provided with all parameters. It currently supports two algorithms (specified in the method variable): "Power" and

"Embed". The first algorithm is the standard Power method on probability transition matrices. It is applied to the uniformized chain in the case of continuous-time. The second algorith is specific of continuous-time. It applies the power method to the discrete-time Markov chain embedded at jump times, then corrects the distribution obtained so as to provide the stationary distribution.

Alternately, a direct call to the corresponding methods can be done with the interface:

```
Distribution* StationaryDistributionCTEmbedding(int tMax, double precision,
                                                DiscreteDistribution *iDis, bool progress );
Distribution* StationaryDistributionPower(int tMax, double precision,
                                          DiscreteDistribution *iDis, bool progress );
```

Other parameters common to these methods are:

`progress` : flag specifying if the iteration numbers must be issued along the way.

`tmax` : the maximum number of iterations to be performed. When this number is reached, a message is issued warning the user that the computation may be imprecise. Note that iterative methods usually do not provide a guarantee of precision anyway.

`precision` : a precision parameter used for stopping iterations. Typically, iterations stop when two consecutive results have a "distance" less than this parameter. Note that this does not imply in general that the result produced is within a distance of the exact value.

`initDistribType` : a specification of the initial distribution to be used in the iterations. Recognized types are: `"Zero"` for the Dirac distribution concentrated on the state with index 0; `"Max"` for the Dirac distribution concentrated on the state with largest index; `"Uniform"` for the uniform distribution over the whole state space, and `"Custom"` in which case the `iDis` parameter is used.

`iDis` : the initial distribution to be used in the iterations.

3.1.1.8 Transient distributions

The transient distribution is the distribution of the state of the Markov chain after a given time, starting from some initial state. Methods performing this calculation are:

```
Distribution*  TransientDistributionR( int fromState, double t )
Distribution*  TransientDistributionDT( int fromState, int t )
```

The method `TransientDistributionR` performs the calculation for continuous-time chains. It uses the `R` environment, see Section 1.4.1. The method `TransientDistributionDT` performs the calculation for discrete-time chains. It uses the power method.

3.1.1.9 Hitting times

Hitting times are random variables defined from the Markov chain. They measure the time it takes for the chain to reach a certain set of states, starting from some initial state.

The methods computing these distributions (or sampling from them) are:

```
Distribution*  HittingTimeDistribution(int iState, bool *hitSetIndicator)
int*           SimulateHittingTime(int iState, bool *hittingSet, int nbSamples, int tMax)
double*        AverageHittingTime(bool *hitSetIndicator)
double*        AverageHittingTimeDT(bool *hitSetIndicator)
double*        AverageHittingTimeDTIterative(bool *hitSetIndicator)
```

The parameters common to these methods are:

`iState` : the index of the initial state.

`hitSetIndicator` : an array of boolean marking the states that are in the hitting set with `true`.

The method `AverageHittingTime()` uses the Gauss-Jordan method to solve the linear system that provides average hitting times. The method `AverageHittingTimeIterative()` solves (approximately) the same system with the power method. Both return arrays of the size of the state space, contaning the average hitting time from every state in the state space.

The method `SimulateHittingTime` obtains sample of the distribution with Monte Carlo simulation. It uses the parameters `nbSamples` to specify how many samples should be collected, and `tMax` to limit the duration of simulations. When the limit is reached, it is returned as the sample. This method is specific to discrete-time chains.

3.1.1.10 Output

Markov chain objects can be saved to a file using a number of formats. The method `Write(string format, string modelName)` does this. Supported formats are: Ers, R, SCILAB. See Appendix D for their description.

3.1.2 Implementations

The following Markov chain models implemented. The specific implementation `AbstractMarkovChain` is described in Section 3.1.3.

Name	description	inherits from
`TwoStateContinuous`	generic continuous-time chain with two states	`MarkovChain`
`Homogeneous1DRandomWalk`	discrete-time random walk on subsets of $\mathbb{N}$	`MarkovChain`
`Homogeneous1DBirthDeath`	continuous-time birth-death on subsets of $\mathbb{N}$	`MarkovChain`
`HomogeneousMultiDRandomWalk`	discrete-time random walk on subsets of $\mathbb{N}^d$	`MarkovChain`
`PoissonProcess`	the usual Poisson process on $\mathbb{N}$	`Homogeneous1DBirthDeath`
`Felsenstein81`	continuous-time model for genome evolution	`MarkovChain`

3.1.2.1 `TwoStateContinuous`

This class implements the two-state continuous-time Markov chain. This is a continuous-time Markov chain model, characterized by:

- the rate of jumps from state 0 to state 1, α,
- the rate of jumps from state 1 to state 0, β.

It is currently a class derived from `MarkovChain`: it probably should be derived from `Homogeneous1DBirthDeath`.

Definition. This class is accessed with the directive

```
#include "markovChain/two_state_continuous.h"
```

Constructors. This class has a single constructor:

```
TwoStateContinuous( double alpha, double beta );
```

Re-implemented methods. The following methods have been re-implemented within `TwoStateContinuous`:

```
std::vector<int> AbsorbingStates();
std::vector< std::vector<int> > RecurrentClasses();
std::vector< std::vector<int> > CommunicatingClasses();
bool IsIrreducible();
bool IsAccessible(int stateFrom, int stateTo);
SimulationResult* SimulateChain(double tMax, bool stat, bool traj, bool incr, bool trace);
```

Specific methods. The following methods are specific to the class:

```
BernoulliDistribution* StationaryDistribution();
BernoulliDistribution* TransientDistribution( double t );
```

The parameter of the second methods is `t`, the time at which the distributions should be evaluated.
The method `TransientDistribution()` computes the transient distribution $\pi(t)$ with the exact formulas:

$$\begin{aligned} p_0(t) &= \frac{1}{\alpha+\beta}\left(\beta + e^{-(\alpha+\beta)t}(\alpha p_0(0) - \beta p_1(0))\right) \\ p_1(t) &= \frac{1}{\alpha+\beta}\left(\alpha - e^{-(\alpha+\beta)t}(\alpha p_0(0) - \beta p_1(0))\right) \ . \end{aligned}$$

The method `StationaryDistribution()` returns the stationary distribution: it is a Bernoulli distribution given by:

$$\pi = \left(\frac{\beta}{\alpha+\beta}, \frac{\alpha}{\alpha+\beta}\right)$$

and is defined only when $(\alpha, \beta) \neq (0, 0)$.

3.1.2.2 `Homogeneous1DBirthDeath`

This class implements the 1-dimensional birth and death process with homogeneous transition rates. This is a continuous-time Markov chain model, characterized by:

- the number of states (or "size") N, possibly `INFINITE_STATE_SPACE_SIZE`
- the rate of jumps to the right, λ,
- the rate of jumps to the left, μ.

Definition. This class is accessed with the directive

```
#include "markovChain/homogeneous_1d_birth_death.h"
```

Constructors. Two constructors are available.

```
Homogeneous1DBirthDeath( double lambda, double mu );
Homogeneous1DBirthDeath( int n, double lambda, double mu );
```

The first form defines a birth-death process with $\mathbb{N}$ as state space. The second one defines a birth-death process with $[0..n-1]$ as state space.

Re-implemented methods. The following methods have been re-implemented within `Homogeneous1DBirthDeath`:

```
SimulationResult* SimulateChain(double tMax, bool stat, bool traj, bool incr, bool trace);
```

Simulation is possible for infinite-state birth-death processes. However, overflow of the state is not handled currently.

Specific methods. The following methods are specific to the class:

```
DiscreteDistribution* TransientDistribution(double t,int nMax);
DiscreteDistribution* ApproxTransientDistribution(double t,int nMax);
GeometricDistribution* StationaryDistribution();
DiscreteDistribution* StationaryDistribution(int nMax);
void MakeMarkovChain();
```

The parameters of these methods are: `t`, the time at which distributions should be evaluated, and `nMax`, the index of the largest state. Note that the "size" parameter specified at the creation of the object is ignored by these methods.

The method `TransientDistribution()` computes the transient distribution $\pi(t)$ with exact formulas (currently incomplete).

The method `ApproxTransientDistribution()` computes an approximation to the transient distribution for a `Homogeneous1DBirthDeath` chain, computed as an interpolation between the initial distribution $\pi(0)$ and the stationary distribution π:

$$\pi(t) \;=\; (1 - e^{-(\lambda+\mu)t})\pi + e^{-(\lambda+\mu)t}\pi_0.$$

The method `StationaryDistribution()` returns the stationary distribution for the birth-death process on $\mathbb{N}$. It is a geometric distribution. When $\lambda \geq \mu$, the geometric distribution with parameter 1 is returned, to represent the defective distribution with a "Dirac mass at $+\infty$".

The method `StationaryDistribution(int n)` returns the stationary distribution for the birth-death process on $[0..n]$ (and not $[0..n-1]$). It is a truncated geometric distribution:

$$\pi_k \;=\; \frac{(1-\lambda/\mu)(\lambda/\mu)^k}{1-(\lambda/\mu)^{n+1}}, \quad \text{if } \lambda \neq \mu, \qquad \pi_k \;=\; \frac{1}{n+1} \quad \text{if } \lambda = \mu, \quad k = 0..n.$$

When $\lambda = \mu$, this is a discrete uniform distribution, but the class `UniformDiscreteDistribution` is not used.

The method `MakeMarkovChain()` creates a generator for the Markov chain object, of the type `SparseMatrix`. This does not apply to chains on $\mathbb{N}$.

3.1.2.3 `Homogeneous1DRandomWalk`

This class implements the 1-dimensional random walk with homogeneous transition probabilities. This is a discrete-time Markov chain model, characterized by:

- the number of states (or "size") N, possibly `INFINITE_STATE_SPACE_SIZE`
- the probability to jump to the right, p
- the probability to jump to the left, q.

The model is valid if $p + q \leq 1$. The probability to stay at the same position is $1 - p - q$.

Definition. This class is accessed with the directive

```
#include "markovChain/homogeneous_1d_random_walk.h"
```

Constructors. Two constructors are available.

```
Homogeneous1DRandomWalk( double p, double q );
Homogeneous1DRandomWalk( int n, double p, double q );
```

The first form defines a random walk process with $\mathbb{N}$ as state space. The second one defines a random walk process with $[0..n-1]$ as state space.

Re-implemented methods. None.

Specific methods. The following methods are specific to the class:

```
SimulationResult* SimulateChain(long int tMax, bool stat, bool traj, bool trace);
DiscreteDistribution* ApproxTransientDistribution(int t, int nMax);
Distribution* StationaryDistribution();
void MakeMarkovChain();
void Write(string format, string prefix);
```

The method `SimulateChain` performs Monte Carlo simulation of the chain. Simulation is possible for infinite-state birth-death processes. However, overflow of the state is not handled currently.[3]

The parameters of the methods devoted to transient and stationary distributions are: `t`, the time at which distributions should be evaluated, and `nMax`, the index of the largest state.

The method `ApproxTransientDistribution()` computes an approximation to the transient distribution for a `Homogeneous1DRandomWalk` chain, computed as an interpolation between the initial distribution $\pi(0)$ and the stationary distribution π:

$$\pi(t) \;=\; (1-\omega^t)\pi + \omega^t\pi_0, \quad \text{with } \omega = 1-p-q+2\sqrt{pq}\cos(\pi/n).$$

Note that method uses its own parameter `n` and ignores the "size" parameter specified at the creation of the object.

The method `StationaryDistribution()` returns the stationary distribution for the birth-death process. When the process is on $\mathbb{N}$, this stationary distribution is a geometric distribution. When $p \geq q$, the geometric distribution with parameter 1 is returned, to represent the defective distribution with a "Dirac mass at $+\infty$".

The method `MakeMarkovChain()` creates a generator for the Markov chain object, of the type `SparseMatrix`. This does not apply to chains on $\mathbb{N}$.

The method `Write()` writes a representation of the model in one or several files, using a specified format and a specified name for the model (parameter `prefix`). The prefix is used to form the file name, by appending extensions specific to the format. Supported formats are (see Appendix D):

- Xborne: both Rii and Cuu files are written
- Psi, Ers, `R`.

3.1.2.4 `HomogeneousMultiDRandomWalk`

This class implements the multidimensional random walk with homogeneous transition probabilities. This is a discrete-time Markov chain model, characterized by:

- the number of dimensions d,
- the number of states in each dimension, possibly `INFINITE_STATE_SPACE_SIZE`
- the probabilities to jump to the right in each dimension, $(p_1, \ldots, p_d)$,
- the probability to jump to the left in each dimension, $(q_1, \ldots, q_d)$.

The model is valid if $\sum_{k=1}^{d}(p_i+q_i) \leq 1$. The probability to stay at the same position is $r = 1-\sum_{k=1}^{d}(p_i+q_i)$.

Objects of this class are equipped with a generator object, from the `multiDimHomTransition` class.

Definition. This class is accessed with the directive

```
#include "markovChain/homogeneous_multi_d_random_walk.h"
```

[3] Observe that the method does *not* override `MarkovChain::SimulateChain` because its signature is actually that of `MarkovChain::SimulateChainDT`. This mismatch will be corrected in a lated version.

Constructors. Two constructors are available.

```
HomogeneousMultiDRandomWalk( int nbDims, double* p, double* q );
HomogeneousMultiDRandomWalk( int nbDims, int* sz, double* p, double* q );
```

In both, `nbDims` is the number of dimensions d and `p`, `q` are arrays of size d containing the probabilities p_i and q_i. The first form defines a random walk with $\mathbb{N}^d$ as state space. The second one defines a random walk with $\times_{i=1}^{d}[0..n_i - 1]$ as state space, where the n_i are the values in the array `sz`.

Re-implemented methods. The following method is reimplemented.

```
int* SimulateHittingTime(int iState, bool *hittingSet, int nbSamples, int tMax);
```

The method works only for finite chains, although this condition is not currently enforced.

Specific methods. The following methods are specific to the class:

```
void MakeMarkovChain();
DiscreteDistribution* StationaryDistribution();
void Write(string format);
```

The method `MakeMarkovChain()` creates a generator for the Markov chain object, of the type `SparseMatrix`. It applies only to finite chains of dimension 1 or 2.

The method `StationaryDistribution()` returns the stationary distribution for the birth-death process. It applies only to chains on finite state spaces. The distribution is a product of truncated geometric distribution, see `Homogeneous1DRandomWalk`.

The method `Write()` writes a representation of the model in one or several files, using a specified format. The attribute `model_name_` is used to form the file name, by appending extensions specific to the format. The only supported format is XBORNE (see Appendix D). This method will probably be made obsolete.

3.1.2.5 `PoissonProcess`

This class implements the Poisson counting process. This is a continuous-time Markov chain model, characterized by a unique parameter λ: its rate or intensity. It is a "pure birth" process, and is derived from `Homogeneous1DBirthDeath`: the parameter μ of this model is set to 0.

Definition. This class is accessed with the directive

```
#include "markovChain/poisson_process.h"
```

Constructors. This class has a single constructor:

```
PoissonProcess( double lambda );
```

Re-implemented methods. The following methods have been re-implemented within `PoissonProcess`:

```
Distribution* TransientDistribution(double t);
GeometricDistribution* StationaryDistribution();
SimulationResult* SimulateChain(double tMax, bool stat, bool traj, bool incr, bool trace);
```

The method `TransientDistribution()` returns a Poisson distribution. There is no stationary distributions for Poisson processes: the method `StationaryDistribution()` returns a `GeometricDistribution(1.0)` object, that is, a Dirac mass at infinity. See Section 3.4.2.5.

Specific methods. None.

3.1.2.6 `Felsenstein81`

The Felsenstein 81 model is a continuous-time Markov chain with a state space of size 4 an generator defined by:

$$q_{i,j} \;=\; \mu p_i \;, \quad i \neq j \qquad q_{i,i} \;=\; \mu(1-p_i) \;,$$

where $\pi = (p_1, p_2, p_3, p_4)$ is a probability distribution, and $\mu > 0$. The distribution π turns out to be the stationary distribution.

Definition. This class is accessed with the directive

```
#include "markovChain/biology/felsenstein81.h"
```

Constructors. Two constructors are available:

```
Felsenstein81( double p[4] , double mu);
Felsenstein81( DiscreteDistribution* d, double mu);
```

In both versions, the parameter μ is passed as variable `mu`. The distribution π is passed as an array of four elements in the first version, and as a `DiscreteDistribution` object in the second. It must have 4 values, although this condition is not currently enforced.

Re-implemented methods. The following method has been re-implemented within `Felsenstein81`:

```
Distribution* HittingTimeDistribution(int iState, bool* hittingSet);
double* AverageHittingTime(bool* hittingSet);
SimulationResult* SimulateChain(double tMax,
                                bool stats, bool traj, bool incr, bool trace );
```

The method `HittingTimeDistribution()` uses the exact formula:

$$P(\tau_{\mathcal{H}} \leq t) \;=\; 1 - e^{-\mu t \pi(\mathcal{H})},$$

where $\pi(\mathcal{H})$ is the mass of the hitting set $\mathcal{H}$ with the measure π, and returns an `ExponentialDistribution` object. The method `AverageHittingTime` uses the same formula.

Specific methods. The following methods are specific to the class.

```
void MakeMarkovChain();
DiscreteDistribution* TransientDistribution(int fromState, double t);
DiscreteDistribution* TransientDistribution(double t);
DiscreteDistribution* StationaryDistribution();
```

The method `MakeMarkovChain()` creates a generator for the Markov chain object, of the type `SparseMatrix` (although this matrix is full in general).

The methods `TransientDistribution()` compute the transient distribution $\pi(t)$ with exact formulas. The first form assumes that the chain starts in the state specified as argument `fromState`. The second one assumes that the chain starts with its initial distribution as specified by attribute `init_distribution_`.

The method `StationaryDistribution()` returns the exact stationary distribution π.

3.1.3 Abstract Markov chains

The parameter `IsAbstract` refers to the possibility that a `MarkovChain` object be kept "abstract" (or virtual) in the sense that its contents is left in files but not read in memory.

In addition to the attributes already listed in Section 3.1, including the `format_` and `model_name_` specifiers, these objects have specific attributes related to abstractness:

`int`	`abstract_nbr_`	the number of parameters describing the chain
`string*`	`abstract_param_`	an array of names describing the chain

The size of array `abstract_param_` is `abstract_nbr_`.

Definition. This class is accessed with the directive

```
#include "markovChain/abstract_markov_chain.h"
```

Constructors. Two constructors are available:

```
AbstractMarkovChain (int sz, timeType t)
AbstractMarkovChain (string format, string param[], int nbParam, string model_name)
```

The first form creates an `AbstractMarkovChain` object with minimal information. The second one creates an object corresponding to a model with name `model_name` written in format/language `format`, the files describing it being listed in array `param` (with size given by `nbParam`. The following formats are compatible with abstract chains: PSI1/MARCA, ERS, XBORNE, XBORNEPres. They are described in Appendix D. However, only abstract chains of the XBORNE type have currently a functional interface.

Re-implemented methods. The following methods of `MarkovChain` are re-implemented:

```
Write (string format, string model_name)
Distribution *  StationaryDistributionGthLD ()
Distribution *  StationaryDistributionSOR ()
```

Method `Write()` is currently re-implemented to issue an error message. Read/Write operations must be performed with a concrete chain. This situation may change if external packages provide applications to convert their models in other formats.

The two other methods refer to solution methods provided by XBORNE. Their re-implementation consists in using the applications of XBORNE instead of the embedded code.

Specific methods. The following methods are specific to the class.

```
set_abstract_nbr(int abstract_nbr)
set_abstract(string abstract[])
abstract_nbr()
toString()
NCDProperty(double epsilon)
BandIMSUB(std::string model_name)
Vincent()
RowVincent()
Absorbing()
ProdFundSW(std::string model_name)
RowSum(std::string model_name)
```

Methods `set_abstract_nbr()` and `set_abstract()` are mutators for the corresponding attributes. `abstract_nbr()` is the accessor for `abstract_nbr_`. There is no accessor for `abstract_param_` but `toString()` is a utility to provide a description of the object.

The methods `NCDProperty`, `BandIMSUB`, `Vincent()`, `RowVincent()`, `Absorbing()`, `ProdFundSW()` and `RowSum()` are entry points to the applications of XBORNE with the same names. See the documentation of XBORNE for details.

3.2 The `TransitionStructure` object

Transition structures are abstractions for matrices, or weighted graphs. Transition structures commonly encountered in Markov modeling are *probability transition matrices*, for discrete-time Markov chains, and *infinitesimal generators* (or *rate matrices*) for continuous-time Markov chains. Matrices or transition functions can occur in various contexts. `marmoteCore` provides a unified presentation with the `TransitionStructure` object and several implementations.

A transition structure maps an "origin" space to a "destination" space and associates a value to these "transitions". In the description below, this will be represented by an "operator" T, mapping some set $\mathcal{O}$ to some set $\mathcal{D}$. The values will be denoted as $T_{i,j}$ for $i \in \mathcal{O}$ and $j \in \mathcal{D}$. The origins i are associated with *rows* and the destinations j with *columns*.

By convention, when the value associated to some possible transition is $T_{i,j} = 0$, then the transition does not actually exist. Accordingly, it is possible to speak of the number of transitions from some particular origin i, since it does not necessarily coincide with the cardinal of the set $\mathcal{D}$. In the vocabulary of (directed) graphs, this is known as the *outdegree* of node i.

3.2.0.1 Definition

This class is accessed with the directive

```
#include "transitionStructure/transition_structure.h"
```

3.2.1 Common features

All `TransitionStructure` objects have the following attributes:

`timeType`	`type_`	the time type of the structure: discrete or continuous
`long int`	`orig_size_`	size of the origin state space
`long int`	`dest_size_`	size of the destination state space

The `type_` attribute has an influence on the type of entries. It has two possible values: `DISCRETE` and `CONTINUOUS`. When it is `DISCRETE`, entries must be probabilities, that is, comprised between 0 and 1. When it is `CONTINUOUS`, entries are arbitrary real numbers.

The "size" attributes are, *a priori*, nonnegative integer numbers. It is however possible to define transition structures over sets which are not finite but denumerable. In that case, the value of the size attribute is `INFINITE_STATE_SPACE_SIZE`.

These attributes are accessed with the following methods:

`// accessing the attributes`		
`timeType`	`getType()`	returns `CONTINUOUS` or `DISCRETE`
`int`	`origSize()`	returns `orig_size_`
`int`	`destSize()`	returns `dest_size_`

Other Methods common to all `TransitionStructure` objects are:

`// accessing the entries`		
`bool`	`setEntry(int,int)`	set the value of entry $T_{i,j}$
`bool`	`addToEntry(int,int,double)`	add a value to entry $T_{i,j}$
`double`	`getEntry(int,int)`	obtain the entry $T_{i,j}$
`int`	`getNbElts(int)`	obtain the number of non-zero entries (outdegree) for some origin i
`int`	`getCol(int,int)`	obtain the destination of the k transition from some origin i
`double`	`getEntryByCol(int,int)`	obtain the k-th non-zero entry in some row i
`DiscreteDistribution*`	`TransDistrib(int)`	see Section 3.2.1.1
`bool`	`ReadEntry(FILE*);`	
`double`	`RowSum(int)`	evaluation of the sum $\sum_j T_{i,j}$ for some i
`// transformations`		
`TransitionStructure*`	`Uniformize()`	see Section 3.2.1.3
`TransitionStructure*`	`Embed()`	see Section 3.2.1.3
`// actions of a transition`		
`void`	`EvaluateMeasure(double*,double*)`	evaluate the action on a mesure: πT
`void`	`EvaluateMeasure(DiscreteDistribution*, DiscreteDistribution*)`	evaluate the action on a mesure: πT version with `Distribution` objects
`void`	`EvaluateValue(double*,double*)`	evaluate the action on a value: Tv

Note that the `addToEntry()` method modifies an entry by adding some value `val` to it, or creates an entry if none was found. The methods `setEntry()` and `addToEntry()` return a boolean, `true` if the operation was successful, `false` otherwise, typically when parameters `i,j` are out of range.

3.2.1.1 Probabilistic Transitions

In the context of Markov chains, transitions are of random nature. The entries in a row of a transition structure encode the random law of transitions from the corresponding origin state i to the destination space $\mathcal{D}$. The `TransDistrib()` method extracts this law as a discrete distribution on $\mathcal{D}$.

Unless the convention is explicitly different in the implementation of a derived class, the distribution is obtained as follows:

- When the time type is discrete, the values $T_{i,j}$ are directly intepreted as transition probabilities.
- When the time type is continuous, the probabilities returned are values $T_{i,j}$ are

$$p_{i,j} \;=\; \frac{T_{i,j}}{\sum_{k\in\mathcal{D}} T_{i,k}}.$$

3.2.1.2 Actions as an operator

Transition structures can "operate" on measures and values. In linear algebra terminology, these operations correspond to left- and right- vector/matrix multiplications.

Measures are defined on the destination space. The action of operator T on measure π, denoted as πT, results in another measure with weights:

$$(\pi T)_j \;=\; \sum_{i\in\mathcal{D}} \pi_i T_{i,j}, \qquad \text{for all } j \in \mathcal{D}.$$

Values are defined on the origin space. The action of operator T on value v, denoted as Tv, results in another value:

$$(Tv)_i \;=\; \sum_{j\in\mathcal{D}} T_{i,j} v_j, \qquad \text{for all } j \in \mathcal{D}.$$

These operations are realized by methods `evaluateMeasure()` (two forms, depending on the representation of the measure) and `evaluateValue()`.

3.2.1.3 Uniformization and Embedding

The `uniformize()` and `embed()` methods transform a continuous-time structure into a discrete-time one. As such, they do not operate on discrete-time transition structures: they just return a copy of the original object in that case.

Uniformization consists in considering the evolution of a continuous-time Markov chain at the pace of a Poisson process with a constant rate ν, called the uniformization factor. All events of the Markov chain occurs at instants of this Poisson process. However, some events may be self-transitions that do not change the state. The result is a discrete-time structure. Algebraically,

$$T^{\nu} \;=\; \frac{1}{\nu} T.$$

The operation is possible for a range of values of ν. By default, the value chosen is the minimum possible: $\nu = \max_i |T_{ii}|$.

Embedding consists in returning the discrete-time chain with transition probabilities obtained with the `TransDistrib()` method (see Section 3.2.1.1).

3.2.1.4 I/O

The `Write()` method outputs the structure on some file decriptor, using a given format (see Appendix D). Supported formats are: XBORNE (Rii variant: by increasing row and increasing columns), MARCA, Matrix-Market sparse and full, Ers, Maple, R, SCILAB, Full, and Matlab.

3.2.2 Implementations

Transition structures implemented:

- `TransitionStructure/SparseMatrix`
- `TransitionStructure/MultiDimHomTransition` (generalized birth-death)
- `TransitionStructure/EventMixture`

Projected:

- `TransitionStructure/Matrix`
- `TransitionStructure/QBD`

3.2.2.1 `SparseMatrix`

The class `SparseMatrix` implement sparse matrix storage, by rows.

Definition. This class is accessed with the directive

```
#include "transitionStructure/sparse_matrix.h"
```

Constructors. Two constructors are available:

```
SparseMatrix(int size);
SparseMatrix(int rowSize, int colSize);
```

In the first one, the `size` parameter applies to the origin and the destination space (square transition structures). In the second one, they are speficied separately.

Re-implemented methods. The following methods have been re-implemented in `SparseMatrix`:

```
bool setEntry(int row, int col, double val);
bool addToEntry(int row, int col, double val);
double getEntry(int,int);
int getNbElts(int row);
int getCol(int row, int numCol );
double getEntryByCol(int row, int numCol);
double RowSum(int row);
void EvaluateMeasure(double* m, double* res);
DiscreteDistribution* EvaluateMeasure(DiscreteDistribution* d);
void EvaluateValue(double* v, double* res);
SparseMatrix* Copy();
SparseMatrix* Uniformize();
SparseMatrix* Embed();
void Write(FILE* out, std::string format);
```

The `Write()` method outputs the structure on some file decriptor, using a given format (see Appendix D). Supported formats are "Ers", "Full", "MatrixMarket-sparse", "MatrixMarket-full", "Maple", "MARCA", "R", "scilab" and "XBORNE".

Specific methods. Additional methods with respect to the top class are:

```
double EvaluateValueState(double* v, int stateIndex);
void Normalize();
SparseMatrix* Revert();
void Diagnose(FILE* out);
SparseMatrix* getReverted();
std::pair<std::vector<SCC>*, SparseMatrix*> getStronglyConnectedComponents(double ignore);
```

The method `EvaluateValueState()` has the same function as `EvaluateValue()` but returns the value for a single state passed as parameter `stateIndex`.

The `Normalize()` method reorganizes the internal storage of transitions so that: a) no duplicate columns appear in rows; b) column numbers appear in increasing order. The resulting structure makes some algorithms more efficient.

The `Diagnose()` method produces diagnostics and counts on the structure.

The `Revert()` and `getReverted()` methods compute the transposed transition structure, in which origin and destination states are exchanged, and the directions of transitions are reverted while keeping their weight or label.

The `getStronglyConnectedComponents()` method computes a decomposition into strongly connected components of the graph of the matrix. The result is returned as a pair $(\vec{C}, M)$. Here, $\vec{C}$ is the list of strongly connected components, each being coded in a structure with description:

```
struct SCC {
  int id;  /**< index of the SCC */
  int period; /**< period of this SCC */
  std::set<int> states; /**< list of states indices in the SCC */
};
```

The second part of the result, M, is the transition matrix between these strongly connected components. The parameter `ignore` is a threshold: entries with values less or equal to it are ignored in the computation. The default value is 0. The method applies to square transition structures, although this is not currently enforced.

3.2.2.2 `EventMixture`

The `EventMixture` class represents discrete-time transition structures that are probabilistic mixtures of more elementary transition. The idea is that transitions are governed by a finite set of "events", each occuring with a fixed probability over time, and causing a fixed transition.

Definition. This class is accessed with the directive

```
#include "transitionStructure/event_mixture.h"
```

Constructors. Two constructors are available:

```
EventMixture(int size,int nbEvents,double* probas,std::string* names,int** transitions );
EventMixture(SparseMatrix* spMat);
```

The first constructor uses as arguments:

`size` the size of the state space

`nbEvents` the number of different events

`probas` the respective probabilities of these events

`names` the respective names given to these events

`transitions` the array of elementary transitions corresponding to these events. Each of these array describes a mapping from one state to another one.

The second constructor extracts an event structure from a `SparseMatrix` structure, in a greedy way (experimental).

Re-implemented methods.

```
bool setEntry(int i, int j, double val);
double getEntry(int i, int j);
int getNbElts(int i);
int getCol(int i, int k);
double getEntryByCol(int i, int k);
DiscreteDistribution* TransDistrib(int i);
double RowSum(int i);
EventMixture* Copy();
EventMixture* Uniformize();
EventMixture* Embed();
void EvaluateMeasure(double* d, double* res) ;
DiscreteDistribution* EvaluateMeasure(DiscreteDistribution* d);
void EvaluateValue(double* v, double* res);
void Write(FILE* out, std::string format);
```

The `Write()` method supports formats XBORNE, MARCA, Ers, Maple and PSI3 (experimental).

Specific methods.

```
int nb_events();
double event_proba(int e);
double EvaluateValueState(double* v, int stateIndex);
```

The two first ones are accessors to the number of events and the respective probabilies of these events.

The method `EvaluateValueState()` has the same function as `EvaluateValue()` but returns the value for a single state passed as parameter `stateIndex`.

3.2.2.3 `MultiDimHomTransition`

The classe `MultiDimHomTransition` represents multidimensional, homogeneous random walk transition structures.

- the number of dimensions d,
- the number of states in each dimension, assumed finite,
- the probabilities to jump to the right in each dimension, $(p_1, \ldots, p_d)$,
- the probability to jump to the left in each dimension, $(q_1, \ldots, q_d)$.

The model is valid if $\sum_{k=1}^{d}(p_i+q_i) \leq 1$. The probability to stay at the same position is $r = 1-\sum_{k=1}^{d}(p_i+q_i)$. The boundaries are absorbing: when a transition goes out of bounds, it is assumed to stay on the boundary.

Definition. This class is accessed with the directive

```
#include "transitionStructure/multi_dim_hom_transition.h"
```

Constructors. The class has a single constructor:

```
MultiDimHomTransition(int nbDims, int* dimSize, double* p, double* q);
```

The parameter `nbDims` is the number of dimensions d, and `p`, `q` are arrays of size d containing the probabilities p_i and q_i. process with $\times_{i=1}^{d}[0..n_i - 1]$ as state space, where the n_i are the values in the array `dimSize`.

Re-implemented methods.

```
bool setEntry(int i, int j, double val);
double getEntry(int i, int j);
int getNbElts(int i);
int getCol(int i, int k);
double getEntryByCol(int i, int k);
DiscreteDistribution* TransDistrib(int i);
double RowSum(int i);
MultiDimHomTransition* Copy();
MultiDimHomTransition* Uniformize();
MultiDimHomTransition* Embed();
void EvaluateMeasure(double* d,double* res);
void EvaluateValue(double* v, double* res);
void Write(FILE* out, string format);
DiscreteDistribution* EvaluateMeasure(DiscreteDistribution* d);
```

The `Write()` method supports formats XBORNE, MARCA, Ers, Maple.

Specific methods.

```
int dim_size(int d);
double p(int d);
double q(int d);
DiscreteDistribution* JumpDistribution();
```

The three first ones are accessors to the number of dimensions d and the probability vectors p and q, corresponding to the constuctor of the class.

The method `JumpDistribution` returns a discrete distribution representing the generic jumps. The distribution has $2d+1$ values: $\{0, \pm 1, ..., \pm d\}$. Assuming a numbering of dimensions from 1 to d, the coding is:

- 0 codes the self jump
- $+i$ codes a jump upwards in dimension i
- $-i$ codes a jump downwards in dimension i.

3.3 The `MarmoteSet` object

In `marmoteCore`, sets are represented by unions of discrete (hyper-)rectangles. The simplest set is an integer interval. Other sets are constructed from cartesian products of such intervals, and unions of them.

3.3.0.1 Definition

This class is accessed with the directive

```
#include "Set/marmote_set.h"
```

3.3.1 Common features

All `MarmoteSet` objects have the following common attributes:

`enum opType`	`UNION, PRODUCT, SIMPLE`	specify the type of construction
`bool`	`is_simple_, is_union_, is_product_`	flags the type of construction
`int`	`nb_dimensions_`	number of dimensions for products
`int`	`nb_Zones_`	number of subsets for unions
`long int`	`cardinal_`	total cardinal
`MarmoteSet**`	`zone_`	array of subsets for unions
`MarmoteSet**`	`dimension_`	array of dimensions for products
`int*`	`state_buffer_`	
`int*`	`dim_offset_`	
`int*`	`idx_offset_`	
`int`	`tot_nb_dims_`	
`int*`	`first_state_`	value of the state indexed 0

Methods common to all `MarmoteSet` objects are summarized in the following table.

```
// constructors
          MarmoteSet()
          MarmoteSet( MarmoteSet **list, int nb, opType t )
                                                    construction from ...
// accessors
long int  Cardinal()                                number of elements in the set
bool      IsFinite()
bool      IsSimple()
bool      IsUnion()
bool      IsProduct()
int       tot_nb_dims()                             number of dimensions for products
// state-index conversions
void      DecodeState(int index, int* buffer);
int       Index(int* buffer)
// state space exploration
void      FirstState(int* buffer)
void      NextState(int* buffer)
bool      IsFirst(int* buffer)
// utilities
void      enumerate()
void      PrintState(FILE* out, int index);
```

3.3.1.1 Constructors

The simple `MarmoteSet()` initializes a set with the minimal features corresponding to an empty set. The user is responsible for setting up the attributes of the set. This is normally used only in derived classes.

The constuctor `MarmoteSet( MarmoteSet **list, int nb, opType t )` builds a set from more elementary ones, using the construction of type `t` given as argument. The type may be one of `UNION` or `PRODUCT`. The number of subsets is `nb` and they are provided in the array `list`.

3.3.1.2 State representation and indexing

For computational purposes, all states are represented by a vector (array) of integers. The number of elements in this array is the "total dimension" of the set, stored in the attribute `tot_nb_dims_` and can be retrieved with the accessor `tot_nb_dims()`. However, most objects like transition structures and distributions require that the elements of the set be represented as consecutive integers. Any `MarmoteSet` class must then implement a one-to-one correspondence between its states and some numbers called the *indices* of the states.[4]

The two methods performing this conversion are `Index()` to pass from a state to an index, and `DecodeState()` to do the reverse. These methods are used very often and should be as efficient as possible.

3.3.1.3 Walking through sets

An elementary operation on sets is to consider all states sequentially. For this purpose, every `MarmoteSet` object identifies a particular state called the "first" or "initial" state, and stored in the attribute `first_state_`. This is usually the state with index 0 but need not be so.

In addition, `MarmoteSet` provides three functions:

initialization with `FirstState(stateBuffer)`, which sets the state (represented in the array `stateBuffer` to the first state of the state space;

increment of the state with `NextState(stateBuffer)`, which moves the state to the next one in the enumeration order;

[4]When no confusion can occur, states and their indices are assimilated in this manual.

termination test with `IsFirst(stateBuffer)` which tests whether the enumeration came back to the initial state.

The `Enumerate()` utility walks through the state space using just these three constructs, printing each state in the process.

3.3.1.4 I/O

The method `PrintState()` writes a representation of the state to the stream passed as parameter.

There are currently no provision for reading or writing state spaces as a whole.

3.3.2 Implementations

Five sets are currently implemented:

Name	description	inherits from
`MarmoteInterval`	a simple 1-dimensional discrete interval, possibly infinite	`MarmoteSet`
`MarmoteBox`	cartesian products of intervals	`MarmoteSet`
`BinarySequence`	sequences of bits	`MarmoteSet`
`BinarySimplex`	sequences of bits with given count of ones	`MarmoteSet`
`Simplex`	sequences of integers with given total sum	`MarmoteSet`

3.3.2.1 `MarmoteInterval`

This class implements sets of the form $\{a, a+1, \ldots, b\}$ where a and b are integers. The cardinal of the set is $b - a + 1$, provided that $a \leq b$. Sets of this class are always finite.

Definition. This class is accessed with the directive

```
#include "Set/marmote_interval.h"
```

Constructors. The class has a single constructor:

```
MarmoteInterval( int min, int max );
```

By convention, if `max` < `min`, then the interval is empty. Otherwise, both `min` and `max` are inside the interval.

Re-implemented methods.

```
bool IsFinite();
bool IsFirst(int* buffer);
void FirstState(int* buffer);
void NextState(int *buffer);
void DecodeState(int index, int* buffer);
int Index(int* buffer);
void PrintState(FILE* out, int *buffer);
void enumerate();
```

The first state is set as a. The index of state s is $s - a$.

The `PrintState()` method writes the state value with a leading white space and a minimal formating width equal to 4 characters.

Specific methods. The class does not provide specific methods. In particular, the values of a and b cannot be directly accessed after the creation of the object.

3.3.2.2 `MarmoteBox`

The `MarmoteBox` class represents "rectangular" sets. They are cartesian products of one-dimensional intervals: $\times_{i=1}^{d}[a_b..b_i]$. They are not implemented using the `MarmoteInterval` objects however. These sets *may be infinite.*

Definition. This class is accessed with the directive

```
#include "Set/marmote_box.h"
```

Constructors. The class provides two constructors:

```
MarmoteBox(int nbDims, int* dimSize);
MarmoteBox(int nbDims, int *lower, int* upper);
```

In both, the parameter `nbDims` specifies the dimension d. It must be larger than 1 although this condition is not currently enforced.

In the first variant, the array `dimSize`, which must have d elements, contains the sizes of the different dimensions. These numbers may be `INFINITE_STATE_SPACE_SIZE`, in which case the corresponding dimension will be considered as $\mathbb{N}$, or a finite value n, in which case the dimension will be considered as the interval $\{0, \ldots, n-1\}$.

In the second variant, the values a_i and b_i are provided in the arrays `lower` and `upper`. These must be *nonnegative* values, or `INFINITE_STATE_SPACE_SIZE`. By convention, if $a_i > b_i$, the interval of the corresponding dimension is assumed to be $\{a_i\}$.

Re-implemented methods.

```
bool IsFinite();
bool IsFirst(int* buffer);
void FirstState(int* buffer);
void NextState(int *buffer);
void DecodeState(int index, int* buf);
int Index(int* buf);
void PrintState(FILE* out, int *buffer);
```

The first state is set as $(a_1, \ldots, a_d)$. States are ordered lexicographically so that the index of state $(s_1, \ldots, s_d)$ is given by the formula:

$$\texttt{index}(s_1, \ldots, s_d) \;=\; \sum_{i=1}^{d} s_i \prod_{j=i+1}^{d} (b_j - a_j + 1) \;.$$

The `PrintState()` method writes the state value as a sequence of numbers between parentheses and separated by commas, each number having a leading white space and a minimal formating width equal to 3 characters. Example: `(  3, 150,  20)`.

Specific methods. The class does not provide specific methods. In particular, the values of used for creating the object cannot be directly accessed after the creation.

3.3.2.3 `BinarySequence`

The `BinarySequence` class represents sequences (or words) of bits. They can be seen as cartesian powers of the set $\{0, 1\}$ and therefore as rectangular sets. They are not implemented using the `MarmoteBox` objects however. These sets are always finite.

Definition. This class is accessed with the directive

```
#include "Set/binary_sequence.h"
```

Constructors. The class has a single constructor:

```
BinarySequence(int n);
```

The parameter `n` is the length of the sequence. The set has then 2^n states.

Re-implemented methods. The following methods are re-implemented in the class:

```
bool IsFinite();
bool IsFirst(int* buffer);
void FirstState(int* buffer);
void NextState(int *buffer);
void DecodeState(int index, int* buf);
int Index(int* buf);
void PrintState(FILE* out, int *buffer);
```

The first state is the sequence $(0, \ldots, 0)$. States are ordered lexicographically so that the index of state $(s_1, \ldots, s_d)$ is given by the formula:

$$\texttt{index}(s_1, \ldots, s_d) \;=\; \sum_{i=1}^{d} s_i \; 2^{d-i} \; .$$

The `PrintState()` method writes the state value as a sequence of bits (0 or 1) between parentheses and separated by white spaces. Example: `( 1 1 1 0 0 0 1 0 1 )`.

Specific methods. The class does not provide specific methods. In particular, the value `n` used for creating the object cannot be directly accessed after the creation.

3.3.2.4 `BinarySimplex`

The `BinarySimplex` class represents sequences (or words) of bits in which the number of ones is constant. Formally,

$$\mathcal{S}_{n,p} := \{\sigma \in \{0,1\}^n, |\sigma|_1 = p\}.$$

These sets are always finite, with cardinal $\binom{n}{p}$.

Definition. This class is accessed with the directive

```
#include "Set/binary_simplex.h"
```

Constructors. The class provides a single constructor:

```
BinarySimplex(int n, int p);
```

The parameter `n` specifies the length of the sequence, and `p` specifies the number of ones. It is required that $0 \leq p \leq n$, although this condition is not currently enforced.

Re-implemented methods.

```
bool IsFinite();
bool IsFirst(int* buffer);
void FirstState(int* buffer);
void NextState(int *buffer);
void DecodeState(int index, int* buf);
int Index(int* buf);
void PrintState(FILE* out, int *buffer);
```

The first state is the sequence $(1, \ldots, 1, 0, \ldots, 0)$ where the p ones are leading the sequence. States are ordered lexicographically *with letter order 1 < 0*. The index of state $(s_1, \ldots, s_n)$ is given by the recursive formula:

$$\texttt{index}(n, p; s_1, \ldots, s_n) = \begin{cases} \texttt{index}(n-1, p-1; s_2, \ldots, s_n) & \text{if } s_1 = 1 \\ \binom{n-1}{p-1} + \texttt{index}(n-1, p; s_2, \ldots, s_n) & \text{if } s_1 = 0 \end{cases}$$

with the boundary condition $\texttt{index}(n, 0; 0, \ldots, 0) = 0$.

The `PrintState()` method writes the state value as a sequence of bits (0 or 1) between parentheses and separated by white spaces. Example: `( 1 1 1 0 0 0 1 0 1 )`.

Specific methods. The class does not provide specific methods. In particular, the values `n` and `p` used for creating the object cannot be directly accessed after the creation.

3.3.2.5 `Simplex`

The `Simplex` class represents sequences of nonnegative integer numbers with a given total sum. Formally,

$$\mathcal{S}_{n,p} := \{\sigma \in \mathbb{N}^n, \sum_{i=1}^{n} \sigma_n = p\}.$$

These sets are always finite, with cardinal: $\binom{n+p-1}{n-1}$.

Definition. This class is accessed with the directive

```
#include "Set/simplex.h"
```

Constructors. The class provides a single constructor:

```
Simplex(int n, int p);
```

The parameter `n` specifies the length of the sequence, and `p` specifies the total sum.

Re-implemented methods.

```
bool IsFinite();
void DecodeState(int index, int* buf);
int Index(int* buf);
void PrintState(FILE* out, int *buffer);
```

The first state is the sequence $(0, 0, \ldots, 0, p)$. States are ordered lexicographically. That the index of state $(s_1, \ldots, s_n)$ is given by the recursive formula:

$$\texttt{index}(n, p; s_1, \ldots, s_n) \;=\; \begin{cases} 0 & \text{if } n = 1 \\ \\ L(s_1, n, p) + \texttt{index}(n-1, p-s_1; s_2, \ldots, s_n) & \text{if } n \geq 2 \end{cases}$$

where

$$L(j, n, p) \;=\; \sum_{i=0}^{j-1} \binom{p-i+k-2}{k-2} .$$

The `PrintState()` method writes the state value as a sequence of numbers between parentheses and separated by white spaces. Example: `( 1 4 1 0 0 0 7 0 1 )`.

Specific methods. The class does not provide specific methods. In particular, the values `n` and `p` used for creating the object cannot be directly accessed after the creation.

3.4 The `Distribution` object

3.4.0.1 Definition

This class is accessed with the directive

```
#include "Distribution/distribution.h"
```

3.4.1 Common features

The methods common to `Distribution` objects are summarized in the following table.

`double`	`Mean()`	mathematical expectation
`double`	`Rate()`	inverse of the mean
`double`	`Moment(int n)`	n-th moment
`double`	`Variance();`	variance
`double`	`Laplace(double s)`	Laplace transform evaluated at real s
`double`	`DLaplace(double s)`	derivative of the Laplace transform
`double`	`Cdf(double x)`	cumulative distribution function
`double`	`Ccdf(double x)`	complementary cumulative distribution function
`bool`	`HasMoment(int n)`	check that moments exist
`Distribution*`	`Rescale(double factor)`	scaling the distribution by a real factor
`Distribution*`	`Copy()`	
`double`	`Sample()`	generate a pseudo-random sample from the distribution
`void`	`IidSample(int n, double* s)`	generate several samples
`double`	`DistanceL1(Distribution*)`	compute the L1 distance for distributions
`bool`	`HasProperty(std::string)`	tests whether the distribution has some property

The `Cdf()` and `Ccdf()` methods return respectively, for x supplied as argument:

$$F_X(x) := P(X \leq x) \qquad P(X > x) = 1 - F_X(x).$$

The `DistanceL1()` method actually applies only to discrete distributions. It uses the re-implementation of `DistanceL1()` in derived classes to perform the computation. It should be eventually superseded by the Total Variation distance.

3.4.2 Implementations

The following distributions are implemented:

Name	description	inherits from
`DiscreteDistribution`	finite discrete distribution	`Distribution`
`DiracDistribution`	Dirac mass	`DiscreteDistribution`
`BernoulliDistribution`	Bernoulli distribution	`DiscreteDistribution`
`UniformDiscreteDistribution`	uniform distribution over integer intervals	`DiscreteDistribution`
`GeometricDistribution`	geometric distribution over $\mathbb{N}$	`Distribution`
`PoissonDistribution`	Poisson distribution	`Distribution`
`GammaDistribution`	Gamma distribution	`Distribution`
`ErlangDistribution`	Erlang distribution	`GammaDistribution`
`ExponentialDistribution`	negative exponential distribution	`ErlangDistribution`
`UniformDistribution`	uniform continuous distribution	`Distribution`

3.4.2.1 `DiscreteDistribution`

This is the general finite discrete distribution. It consists in a list of n *values* $v_1, \ldots, v_n$ and a list of n probabilities $p_1, \ldots, p_n$.

Definition. This class is accessed with the directive

```
#include "Distribution/discrete_distribution.h"
```

Constructors. The class provides two constructors.

```
DiscreteDistribution( int sz, double* vals, double* probas );
DiscreteDistribution( int sz, char *name );
```

The first one creates the object from existing tables of values and probabilities. These arrays must be (at least) of size `sz`. They are *copied* in the object that is created. The second constructor reads the distribution from the file which name is provided as argument `name`. The file should consist in `sz` real numbers, one per line. They should be positive and add up to 1.0, although this is not currently enforced. If anything goes wrong with the file (not accessible, too short, ...) missing probabilities are assumed to be 0. The values are implicitly assume to be 0, 1, ..., `sz-1`.

It is *not* assumed that the v_i are distinct, nor ordered.

Both constructors calculate and store the expectation of the distribution.

Re-implemented methods. The following methods are re-implemented in the class.

```
double    Mean();
double    Moment( int order );
double    Cdf( double x );
bool      HasMoment( int order );
DiscreteDistribution *Rescale( double factor );
DiscreteDistribution *Copy();
double    Sample();
```

These distributions have moments of any order: the method `HasMoment()` always returns `true`. Rescaling by a factor f produces a distribution on values $f \times v_1, \ldots, f \times v_n$ and the same probabilities. The `Sample()` produces a pseudo-random sample of the distribution. It uses a simple linear algorithm and may be inefficient for large values of n.

Specific methods. The following methods are specific to the class:

```
int       nb_vals();
double*   values();
double*   probas();
double    getProbaByIndex(int i);
double    getProba(double value);
double    getValue(int i);
bool      setProba(int i, double v);
double    distanceL1( DiscreteDistribution* d );
double    distanceL2( DiscreteDistribution* d );
double    distanceLinfinity( DiscreteDistribution* d );
void      Write( FILE *out, int mode );
```

Methods `nb_vals()`, `values()` and `probas()` are accessors to n and the tables of values and probabilities, respectively.

Method `getProbaByIndex()` returns p_i where i is specified by `i`. The user has no control on the order of the entries. This method is normally used to browse through all probabilities. Method `getValue()` returns v_i where i is given by parameter `i`.

Method `getProba()` returns the probability of the value `v`. Since values v_i are not necessarily distinct, this is computed as $\sum\{p_j | v_j = v\}$. This method may be inefficient for large values of n.

Method `setProba()` allows to change the value p_i where i is spefified as arguement. The resulting object is not necessarily a distribution anymore. Its mean is incorrect. To be used with caution (or not at all).

The methods `DistanceL1()`, `DistanceL2()` and `DistanceLinfinity()` compute respectively:

$$\sum_{i=1}^{n} |p_i - p'_i|, \qquad \sqrt{\sum_{i=1}^{n} |p_i - p'_i|^2}, \qquad \max\left\{|p_i - p'_i|, 1 \leq i \leq n\right\}.$$

They all *assume implicitly that the set of values is the same* for the distributions that are compared.

The `Write()` method prints a representation of the distribution on file descriptor `out` with format specified as `mode`. Available formats are `DEFAULT_PRINT_MODE` and `MAPLE_PRINT_MODE`. They produce respective results as:

```
discrete [ v1 v2 ... vn ] [ p1 p2 ... pn ]
Vector( [ p1, p2, ..., pn ] );
```

3.4.2.2 `DiracDistribution`

This is the Dirac distribution, concentrated at some value v. It is a special case of finite discrete distribution and the class inherits from `DiscreteDistribution`.

Definition. This class is accessed with the directive

```
#include "Distribution/dirac_distribution.h"
```

Constructors. There is only one constructor to this class:

```
DiracDistribution( double val )
```

Re-implemented methods.

```
double Mean();
double Rate();
double Moment( int order );
double Variance();
double Laplace( double s );
double DLaplace( double s );
double Cdf( double x );
bool HasMoment( int order );
DiracDistribution *Rescale( double factor );
DiracDistribution *Copy();
double Sample();
void IidSample( int n, double* s );
double    getProba(double value);
void Write( FILE *out, int mode );
```

These distributions have moments of any order: the method `HasMoment()` always returns `true`. Rescaling by a factor f produces a Dirac distribution on value $f \times v$.

Methods `getProba()` and `Write()` reimplement the methods from `DiscreteDistribution`. The last one writes "`Dirac distribution at` v" whatever the format specified.

Specific methods. The following method is specific to the class:

```
double  value(int);
```

It is an accessor for v.

3.4.2.3 `BernoulliDistribution`

This is the Bernoulli distribution with parameter p. It is the distribution $\{0, 1\}$ with probabilities p and $1 - p$. It is a special case of finite discrete distribution and the class inherits from `DiscreteDistribution`.

Definition. This class is accessed with the directive

```
#include "Distribution/bernoulli_distribution.h"
```

Constructors. The class has a single constructor:

```
BernoulliDistribution( double val );
```

Its parameter is the probability p.

Re-implemented methods.

```
double Mean();
double Rate();
double Moment( int order );
double Cdf( double x );
bool HasMoment( int order );
BernoulliDistribution *Rescale( double factor );
BernoulliDistribution *Copy();
double Sample();
void Write( FILE *out, int mode );
```

These distributions have moments of any order: the method `HasMoment()` always returns `true`. Rescaling is not possible and returns a copy of the original distribution.

Method `Write()` reimplement the method from `DiscreteDistribution`. It writes "`Bernoulli distribution with proba` p" whatever the mode specified.

Specific methods. The following methods are specific to the class:

```
double getParameter();
double proba();
```

Both methods `getParameter()` and `proba()` are accessors to the parameter p.

3.4.2.4 `UniformDiscreteDistribution`

This is the uniform distribution over some integer interval $a..b = \{a, a+1, \ldots, b-1, b\}$. It is a special case of finite discrete distribution and the class inherits from `DiscreteDistribution`.

Definition. This class is accessed with the directive

```
#include "Distribution/uniform_discrete_distribution.h"
```

Constructors. The class has a single constructor:

```
UniformDiscreteDistribution( int valInf, int valSup );
```

It defines the values of a and b by parameters `valInf` and `valSup` respectively. It is necessary that $a \leq b$ although this is not currently enforced.

Re-implemented methods.

```
double Mean();
double Rate();
double Moment( int order );
double Variance();
double Laplace( double s );
double DLaplace( double s );
double Cdf( double x );
double Cdf( double x );
bool HasMoment( int order );
UniformDiscreteDistribution *Rescale( double factor );
UniformDiscreteDistribution *Copy();
double Sample();
void IidSample( int n, double* s );
double getProba(double value);
void Write( FILE *out, int mode );
```

These distributions have moments of any order: the method `HasMoment()` always returns `true`. Rescaling is not possible and returns a copy of the original distribution.

Method `Write()` reimplement the method from `DiscreteDistribution`. It writes "`uniform distribution on` $[a..b]$" whatever the mode specified.

Specific methods. The following methods are specific to the class:

```
int valInf();
int valSup();
```

They give access to the parameters a and b respectively.

3.4.2.5 `GeometricDistribution`

This is the geometric distribution on $\mathbb{N}$. It has one parameter, a probability p, interpreted as the probability that X is *not* 0. In other words, the distribution is:

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

The value $p = 1$ is accepted, in which case the distribution is interpreted as a Dirac mass at infinity. Its mean and higher moments are then infinite.

Definition. This class is accessed with the directive

```
#include "Distribution/geometric_distribution.h"
```

Constructors. The class has a single constructor,

```
GeometricDistribution( double p );
```

which sets the parameter p.

Re-implemented methods.

```
double Mean();
double Rate();
double Moment( int order );
double Variance( int order );
double Laplace( double s );
double DLaplace( double s );
double Cdf( double x );
bool HasMoment( int order );
GeometricDistribution *Rescale( double factor );
GeometricDistribution *Copy();
double Sample();
void  Write( FILE *out, int mode );
```

These distributions have moments of any order, except when $p = 1$. Rescaling is not possible and returns a copy of the original distribution. Method `Write()` writes "`G` p" whatever the format specified.

Specific methods. The following methods are specific to the class:

```
double  p();
double  getRatio();
double  getProba(double k);
```

Both methods `p()` and `getRatio()` give access to the parameter p. Methods `getProba()` returns the probability $P(X = k)$.

3.4.2.6 `PoissonDistribution`

This is the Poisson distribution with some real parameter λ. It is given by:

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

Definition. This class is accessed with the directive

```
#include "Distribution/poisson_distribution.h"
```

Constructors. The class has a single constructor,

```
PoissonDistribution( double lambda );
```

which sets the parameter λ. The value of λ should be positive, although this is not currently enforced.

Re-implemented methods.

```
double   Mean();
double   Rate();
double   Moment( int order );
double   Variance();
double   Laplace( double s );
double   DLaplace( double s );
double   Cdf( double x );
double   Ccdf( double x );
bool    HasMoment( int order );
PoissonDistribution *Rescale( double factor );
PoissonDistribution *Copy();
double   Sample();
void    IidSample( int n, double* s );
void     Write( FILE *out, int mode );
```

These distributions have moments of any order: the method `HasMoment()` always returns `true`. Rescaling a Poisson distribution by a factor f returns a Poisson distribution with parameter $\lambda \times f$.

Sampling from this distribution is possible only when `R` is enabled. See Section 1.3.2.

Specific methods. The following methods are specific to the class:

```
double  lambda();
double  getProba(double k );
```

The first one is an accessor for parameter λ. The second one returns $P(X = k)$.

3.4.2.7 `GammaDistribution`

This is the Gamma distribution with shape parameter k and scale parameter θ (or rate parameter $\lambda = \theta^{-1}$). It is given by its density:

$$\mathrm{d}P(X \le x) \;=\; \lambda\, \frac{(\lambda x)^{k-1}}{\Gamma(k)} e^{-\lambda x} \mathrm{d}x, \quad x \ge 0.$$

The mean of the distribution is $k\theta$.

Definition. This class is accessed with the directive

```
#include "Distribution/gamma_distribution.h"
```

Constructors. This class has a single constructor

```
GammaDistribution(double shape, double scale)
```

The shape parameter must be strictly positive. The scale parameter must be positive. It can be equal to 0, in which case the distribution is equivalent to a Dirac distribution at 0. The rate parameter is then infinite. If illegal parameters are supplied to the constructor, the default is returned, namely, the Exponential distribution with parameter one, corresponding to $k = \lambda = \theta = 1.0$.

Re-implemented methods. These methods are reimplemented from `Distribution`.

```
double Mean();
double Rate();
double Moment( int order );
double Variance();
double Laplace( double s );
double DLaplace( double s );
double Cdf( double x );
bool HasMoment( int order );
GammaDistribution *Rescale( double factor );
GammaDistribution *Copy();
double Sample();
void Write( FILE *out, int mode );
std::string   toString();
```

These distributions have moments of any order: the method `HasMoment()` always returns `true`. Rescaling results in a new Gamma distribution.

The methods `Cdf()` and `Sample()` are currently not implemented: the constant value `0.0` is returned, with a warning.

Method `Write()` reimplement the method from `Distribution`. It writes "`Gamma shape` k `rate` λ" when the mode is `PNED_PRINT_MODE`. Otherwise, it writes the result of `toString()`, which is: `Gamma(`k`,`θ`)`.

Specific methods. None.

3.4.2.8 `ErlangDistribution`

This is the Erlang distribution with k phases of mean θ. It is actually the Gamma distribution with shape parameter k and scale parameter θ (or rate parameter $\lambda = \theta^{-1}$. Accordingly, its density is

$$\mathrm{d}P(X \leq x) \;=\; \lambda\, \frac{(\lambda x)^{k-1}}{(k-1)!} e^{-\lambda x} \mathrm{d}x, \quad x \geq 0,$$

its cdf is:

$$P(X \leq x) \;=\; 1 - \sum_{j=0}^{k-1} \frac{(\lambda x)^{j}}{j!} e^{-\lambda x}, \quad x \geq 0,$$

and its mean is $k\theta$.

Definition. This class is accessed with the directive

```
#include "Distribution/erlang_distribution.h"
```

Constructors. This class has a single constructor

```
ErlangDistribution(int phases, double scale)
```

The phases parameter is a strictly positive integer numbers. The scale parameter must be positive. It can be equal to 0, in which case the distribution is equivalent to a Dirac distribution at 0. The rate parameter is then infinite. If illegal parameters are supplied to the constructor, the default is returned, namely, the Exponential distribution with parameter one, corresponding to $k = 1$, $\lambda = \theta = 1.0$.

Re-implemented methods. These methods are reimplemented from `GammaDistribution`.

```
double Mean();
double Rate();
double Moment( int order );
double Variance();
double Laplace( double s );
double DLaplace( double s );
double Cdf( double x );
bool HasMoment( int order );
ErlangDistribution *Rescale( double factor );
ErlangDistribution *Copy( double factor );
double Sample();
void Write( FILE *out, int mode );
std::string   toString();
```

These distributions have moments of any order: the method `HasMoment()` always returns `true`. Rescaling results in a new Erlang distribution.

Method `Write()` reimplement the method from `GammaDistribution`. It writes "`Erlang shape` k `rate` λ" when the mode is `PNED_PRINT_MODE`. Otherwise, it writes the result of `toString()`, which is: `Erlang(`k`,`θ`)`.

Specific methods. None.

3.4.2.9 `ExponentialDistribution`

This is the exponential distribution with parameter λ. It is given by:

$$P(X \leq x) \;=\; 1 - e^{-\lambda x}, \quad x \geq 0.$$

This parameter λ is the *rate*. The *mean* of the distribution is $1/\lambda$.

Definition. This class is accessed with the directive

```
#include "Distribution/exponential_distribution.h"
```

Constructors. This class has a single constructor

```
ExponentialDistribution(double m)
```

The value `m` is the *mean* of the random variable. It may be equal to 0, in which case the rate parameter λ is infinite. The mean should be positive. If an illegal parameter is provided, the default Exponential(1.0) is returned.

Re-implemented methods.

```
double Mean();
double Rate();
double Moment( int order );
double Variance();
double Laplace( double s );
double DLaplace( double s );
double Cdf( double x );
bool HasMoment( int order );
ExponentialDistribution *Rescale( double factor );
ExponentialDistribution *Copy();
double Sample();
void Write( FILE *out, int mode );
std::string   toString();
```

These distributions have moments of any order: the method `HasMoment()` always returns `true`. Rescaling results in a new exponential distribution.

Method `Write()` reimplement the method from `Discrete`. It writes "`Exponential distribution with mean` $1/\lambda$" whatever the mode specified. The `toString()` method returns "`Exponential(`$1/\lambda$`)`".

Specific methods. None.

3.4.2.10 `UniformDistribution`

This is the uniform distribution over some real interval $[a, b]$. It is given by

$$P(X \leq x) \;=\; \max\left\{0, \min\left\{1, \frac{x-a}{b-a}\right\}\right\}.$$

Definition. This class is accessed with the directive

```
#include "Distribution/uniform_distribution.h"
```

Constructors. The class has a single constructor:

```
UniformDistribution( double inf, double sup )
```

where `inf` and `sup` denote a and b. The values should be such that $a \leq b$ although this is not currently enforced.

Re-implemented methods.

```
double Mean();
double Rate();
double Moment( int order );
double Laplace( double s );
double DLaplace( double s );
double Cdf( double x );
bool HasMoment( int order );
UniformDistribution *Rescale( double factor );
UniformDistribution *Copy();
double Sample();
double IidSample();
void Write( FILE *out, int mode );
```

Methods `Rescale()` and `Copy()` return `UniformDistribution` objects.

Specific methods. The following methods are specific to the class:

```
double valInf()
double valSup()
```

These are the accessors to the values of a and b, respectively.

Chapter 4

Developer's guide

This chapter is for people who would like to develop new models and methods in the environment of `marmoteCore`.

4.1 Contributing to `marmoteCore`

4.1.1 Joining the forge

`marmoteCore` is free software. You can develop your own version of it privately. However you are welcome to contribute to the common software base, either in the core or through examples of applications.

In order to do that, you need an account on Inria's forge `http://gforge.inria.fr/`, section "creating an account". In case of trouble, contact an administrator.

Once the account active, get the `marmoteCore`project from the forge:

```
git clone git+ssh://USER@scm.gforge.inria.fr//gitroot/marmotecore/marmotecore.git
```

replacing `USER` by your user name.

4.1.2 Coding conventions

`marmoteCore` tentatively follows the conventions of Google's coding style.[1] With respect to names:

- naming of files (`C++` header and code files): all lowercase with underscores;
- naming of classes: "CamelCase" convention, starting with upper case, no underscores;
- naming of functions: starting with lower case, no underscores
- naming of variables: "SnakeCase" convention, all lower case with underscores; "core" variables with a trailing underscore.

With respect to coding:

- accessors (read) without "get"
- mutators (write) with "set" (in lower case, an exception accepted to the naming convention)
- all variables private.

In addition, the following adaptations of the convention:

[1] `https://google.github.io/styleguide/cppguide.html`

- pseudo-accessors returning a simple quantity not exactly a class variable, with "get".
- misc utilities like `toString()`...

Example:

```
class MyClassA
{
  private:
    int price_;
};
int myFunction(int var1, int var2);
// Accessor (read access)
int price() { return price_; }
// Mutator (write access)
void set_price(int price){ price_=price; }
```

4.2 Compiling the core

A `Makefile` allows the (re)compilation of the core by issuing a simple command `make`.

The compilation parameter `WITH_R` controls whether `R`-related code is taken into account or not in the compilation.

4.3 Developing `MarkovChain` objects

4.3.1 The Markov Chain object

Principles:

- "concrete" chains
  - read/create from files in several formats Ers, Marca...
    $\Longrightarrow$ being implemented: HBF, Xborne transition spec, PSI spec
  - create from a `TransitionStructure` (e.g. matrix)
  - write to file in Ers format
    $\Longrightarrow$ being implemented: Marca, HBF, R, scilab/matlab, Maple, ...
- "abstract" chains specified by
  - format
  - name of model
  - optional liste of extensions and/or file names

  "lazy" evaluation

4.3.2 Virtual methods to be reimplemented

There are syntactically no pure virtual methods in the class `MarkovChain`. However, derived classes will likely not benefit from the top-level implementations because the representation of their generator will be different.

The implementation of the top-level class uses a `SparseMatrix` object as generator. Derived classes, representing models with some structure, will rely on a different `TransitionStructure` object, or simply on some "abstract" representation with few parameters.[2]

[2]Not to be confused with `AbstractMarkovChain` objects.

4.4 Developing `TransitionStructure` objects

The class `TransitionStructure` is intended mostly as a template. The following methods are pure virtual and must be reimplemented:

```
virtual bool setEntry(int i, int j, double val) = 0;
virtual double getEntry(int i, int j) = 0;
virtual int getNbElts(int i) = 0;
virtual int getCol(int i, int k) = 0;
virtual double getEntryByCol(int i, int k) = 0;
virtual DiscreteDistribution* TransDistrib(int i) = 0;
virtual double RowSum(int i) = 0;
virtual TransitionStructure* Copy() = 0;
virtual TransitionStructure* Uniformize() = 0;
virtual TransitionStructure* Embed() = 0;
virtual void EvaluateValue(double* v, double* res) = 0;
```

4.5 Developing `Distribution` objects

4.5.1 General distributions

The class `Distribution` is intended mostly as a template. The following methods are pure virtual and must be reimplemented:

```
virtual double Mean() = 0;
virtual double Rate() = 0;
virtual double Moment( int order ) = 0;
virtual double Laplace( double s ) = 0;
virtual double DLaplace( double s ) = 0;
virtual double Cdf( double x ) = 0;
virtual bool HasMoment( int order ) = 0;
virtual Distribution* Rescale( double factor ) = 0;
virtual Distribution* Copy() = 0;
virtual double Sample() = 0;
virtual std::string toString() = 0;
virtual void Write( FILE *out, int mode ) = 0;
```

However, some methods have a default implementation such as `Ccdf()`, `Variance()` or `IidSample()`.

4.5.2 Discrete distributions

Many distributions derive from `DiscreteDistribution`. This class has a protected constructor

```
DiscreteDistribution( int sz )
```

which creates an object with just the number n of values/probabilities instantiated. The responsibility of the programmer is to: either create the tables `values_` and `probas_` from other data, or supply data structures and methods that override the use of these.

Generic methods of `DiscreteDistributions` expect that the variable `mean_` is computed at creation time. This value should be re-computed if the distribution changes in any way after creation.

4.6 Developing MarmoteSet objects

The MarmoteSet class has, syntactically, no pure virtual methods. However, when developing a new "atomic" object, it is necessary to implement the basic methods Index(), DecodeState() and NextState().

4.7 Interfacing with R

The interface with R uses the Rcpp library. The documentation is at http://cran.r-project.org/web/packages/Rcpp/index.html

The library defines types RInside and SEXP.

The code of marmoteCore uses the directive #WITH_R to isolate pieces of code that use specifically this library.

For instance, the excerpt from MarkovChain.cpp.

```
#ifdef WITH_R
#include <RInside.h>
using namespace Rcpp;
#endif

#ifdef WITH_R
RInside* MarkovChain::_Rmotor = (RInside*)NULL;
#else
typedef void* RInside;
typedef void* SEXP;
#endif
```

When a method that uses R is called, the first thing it does is check whether the variable _RMotor is set. If not, it creates one with a call to Rinside(). If R is not enabled, this call raises an exception. The user can catch it and continue if the operation is not essential to the application.

Methods that need R to work are, for instance, MarkovChain::StationaryDistributionR() or poissonDistribution::s

Contents

Appendix A

Examples

The examples of use of `marmoteCore` are organized in two sets: Basic and Advanced.

A.1 Basic examples

The basic examples show how to construct simple Markov chains and call basic solution functions. We briefly comment below the principal functionalities and programming specificities.

A.1.1 Example 1

This example creates a 3-state, discrete-time Markov chain, then performs a (Monte-Carlo) simulation of it. Usage:

```
example1 <n> <p1> <p2> <p3>
```

Here, `n` is the number of steps for the simulation, and `p1`, `p2`, `p3` are the respective initial probabilities of the three states.

Tasks performed:

- create a `DiscreteDistribution` object to hold the initial distribution of the process
- create a `SparseMatrix` object to hold the transition matrix of the chain, entry by entry with the `addToEntry()` function;
- create a `MarkovChain` object and link the previous elements to it;
- output the Markov chain object to the screen;
- create a simulation of a trajectory and store it in a `SimulationResult` object;
- write the trajectory to the screen;
- clean up.

A.1.2 Example 2

This example creates the same 3-state discrete-time Markov chain as in Example 1, then computes the transient distribution of the chain after a given number of steps. Usage:

```
example2 <n> <p1> <p2> <p3>
```

Here, `n` is the number of steps for the transient distribution, and `p1`, `p2`, `p3` are the respective initial probabilities of the three states.

Tasks performed:

- create a `DiscreteDistribution` object to hold the initial distribution of the process

- create a `SparseMatrix` object to hold the transition matrix of the chain, entry by entry with the `addToEntry()` function;
- create a `MarkovChain` object and link the previous elements to it;
- output the Markov chain object to the screen;
- calculate the transient distributions after `n` steps and store it in a `Distribution` object;
- write the distribution to the screen;
- clean up.

A.1.3 Example 3

This example creates three slightly different 8-state, discrete-time Markov chains, then computes the transient distribution of the chain after a given number of steps. Usage:

```
example3 <n> <p1> <p2> <p3> <p4> <p5> <p6> <p7> <p8>
```

Here, `n` is the number of steps for the simulation, and `p1`, `p2`, etc. are the respective initial probabilities of the eight states.

Tasks performed:

- create a `DiscreteDistribution` object to hold the initial distribution of the process
- create three `SparseMatrix` objects to hold the transition matrix of the chain, entry by entry with the `addToEntry()` function;
- create three `MarkovChain` objects and link the previous elements to it;
- calculate the transient distributions after `n` steps for each Markov chain and store it in a `Distribution` object;
- write the distributions to the screen;
- clean up.

A.1.4 Example 4

This example creates two Markov chains by reading them from files, then outputs them to other files: either as whole Markov chains, either only their generators. Usage:

```
example4
```

Tasks performed:

- create two `MarkovChain` objects by reading their description in files `aexample.mcl` and `rw1d.mcl`;
- output them again to files `example4_chain1.mcl` and `example4_chain2.mcl`;
- write the generator of these chains to files `example4_chain1.gen` and `example4_chain2.gen`;
- clean up.

A specificity is the test that the generator is present (not `NULL`) before attempting to write it to the file. Indeed, if the file is not found, or not in the proper format, the chain is created but with an empty generator (see Section 3.1.1.3).

A.1.5 Example 5

This example creates the same 3x3 Markov Chain as in Example 1, then tries two different methods for computing the stationary distribution. The invariance of this distribution is tested. It is also compared with a transient distribution. Usage:

```
example5
```

Tasks performed:

- create a random `DiscreteDistribution` object to hold the initial distribution of the process
- create a `SparseMatrix` object to hold the transition matrix of the chain, entry by entry with the `addToEntry()` function;
- create a `MarkovChain` object and link the previous elements to it;
- compute the stationary distribution using methods `StationaryDistribution()` and `StationaryDistributionPower()` (see Section 3.1.1.7);
- compute the one-step transient distribution π_1, starting from one of these stationary distributions π_∞ taken as initial distribution π_0, and calculate the L1 distance $||\pi_0 - \pi_1||_1$;
- compute the 100-step transient distribution π_{100}, starting from the random initial distribution, and calculate the L1 distance $||\pi_\infty - \pi_{100}||_1$;
- clean up.

This example features the use of a `UniformDistribution` object to generate random values.

A.1.6 Example 6

This example handles `Set` objects, more precisely one `MarmoteInterval` object, and `MarmoteBox` objects of several dimensions. Usage:

```
example6
```

Tasks performed:

- create two `MarmoteBox` objects with one and two dimensions and one `MarmoteInterval` object
- enumerate them with different methods:
  - use the member function `Enumerate()`
  - use the walkthrough facilities `FirstState()/NextState()/IsFirst()`
  - use the `Cardinal()/DecodeState()` facilities
- redo the test with upcast pointers of type MarmoteSet
- clean up.

A.1.7 Example 7

This example demonstrates the structural analysis possibilities for `MarkovChain` objects. Usage:

```
example7
```

Tasks performed:

- create four discrete-time `SparseMatrix` objects
- create four `MarkovChain` objects and set their generators to the previously created `SparseMatrix` objects
- perform a structural analysis of the (graph of the) `MarkovChains`:
  - run the `Diagnose()` utility on the generator
  - find absorbing states with `AbsorbingStates()` and list them
  - find communicating classes with `CommunicatingClasses()` and list them
  - find recurrent classes with `RecurrentClasses()` and list them
  - compute the period with `Period()` and print it
- clean up.

The four transition structures analyzed are displayed in Figure A.1. The probabilities of transitions are not represented since they are not relevant to this analysis.

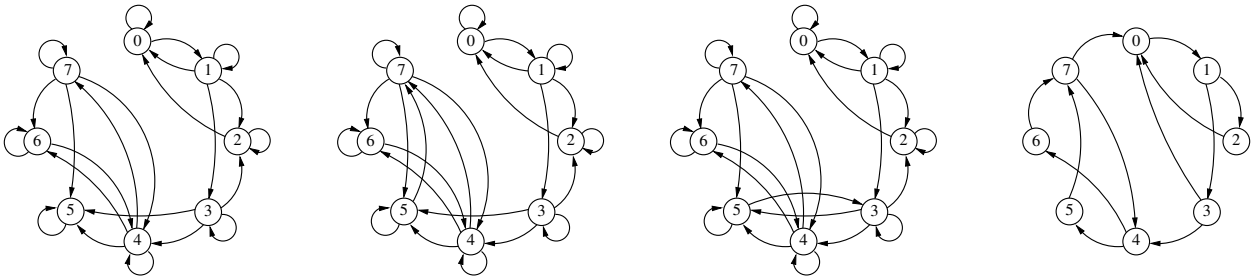


Figure A.1: Diagrams of the four Markov chains analyzed in Example 7

A.2 Advanced examples

A.2.1 Using complex `MarmoteSet` objects

TBD

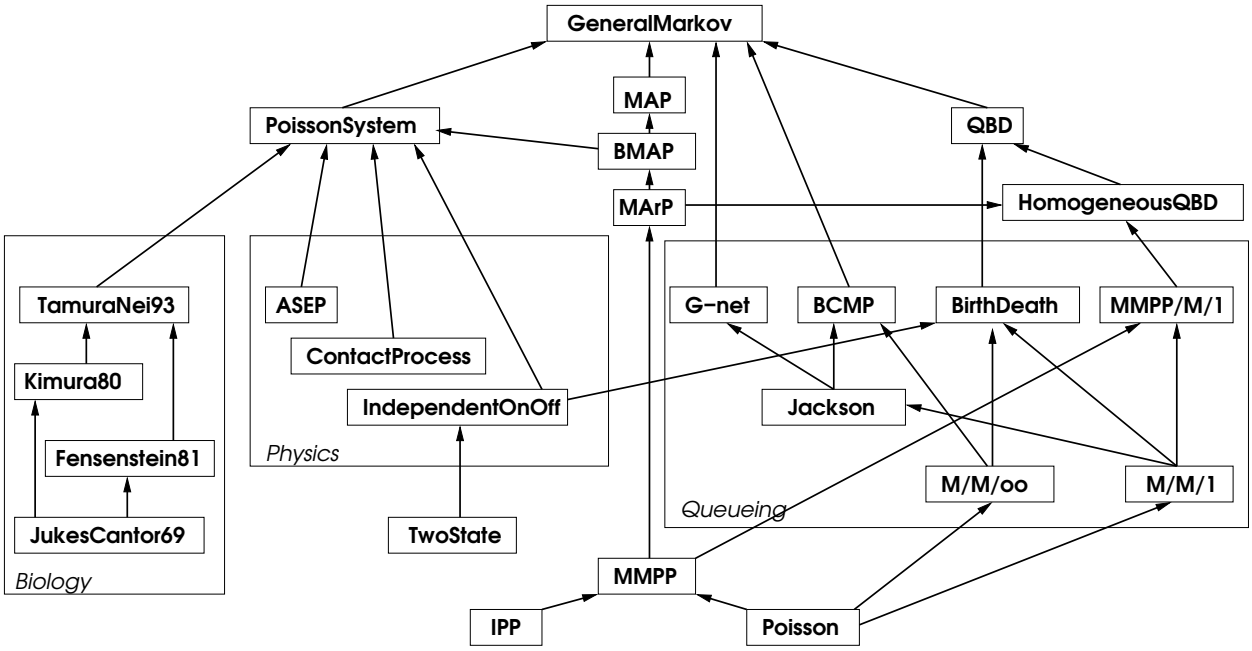
A.2.2 Using the hierarchy of models

TBD

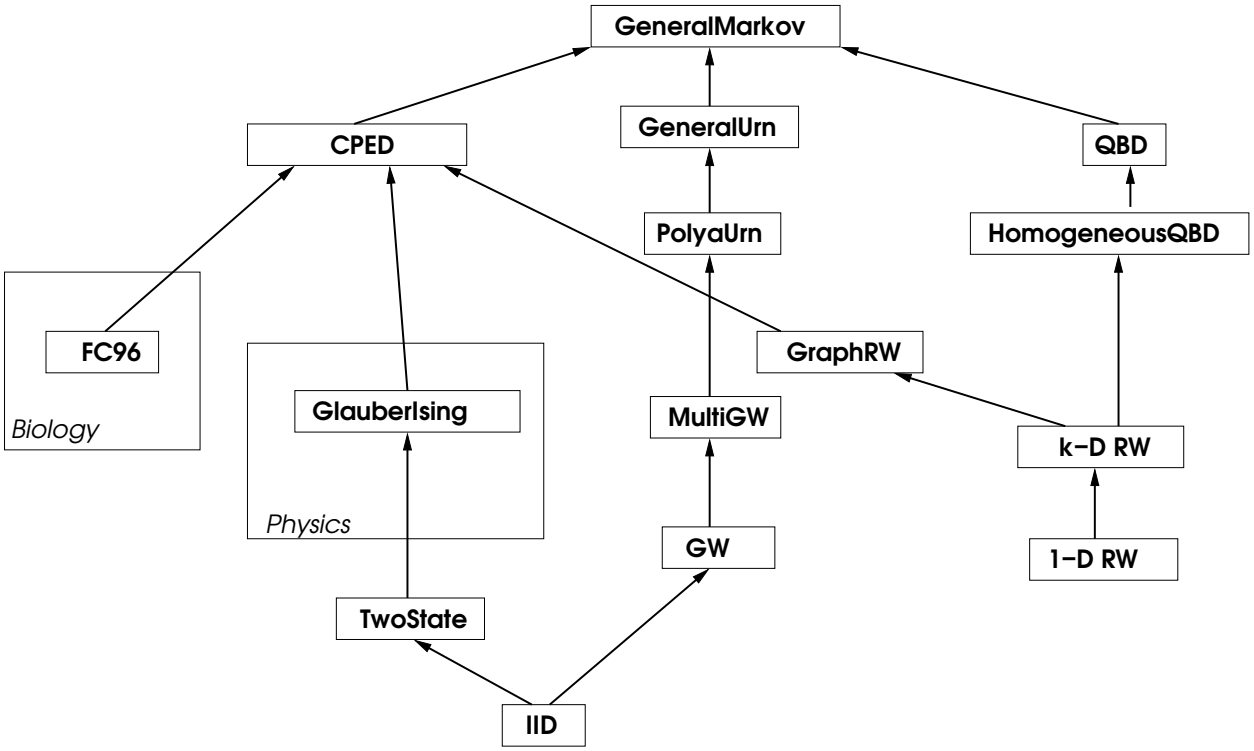
Appendix B

The Markov Zoo

B.1 The Continuous-Time Markov Zoo



B.2 The Discrete-Time Markov Zoo



Appendix C

Interacting with R

C.1 Using `Rcpp`

1. run `R`:

   ```
   R
   ```

2. load the `Rcpp` package:

   ```
   library(Rcpp)
   ```

3. create an empty `Rcpp` package:

   ```
   Rcpp.package.skeleton("testpackage", example\_code=FALSE, attributes=TRUE)
   ```

4. créer un fichier cpp dans /testpackage/src

5. include the following code:

   ```
   #include<Rcpp.h>

   using namespace Rcpp;

   //[[Rcpp::export]]
   ```

6. retourner à R et générer les wrapper:

   ```
   compileAttributes("testpackage")
   ```

7. install the package

   ```
   R CMD INSTALL -- build testpackage
   ```

8. load the new package

   ```
   library(testpackage)
   ```

9. finally, call the functions of the package.

C.2 Using `Rinside`

1. exemple "plot".

Appendix D

File formats

D.1 Xborne

The Xborne suite uses a multi-file format for representing matrices and the state spaces on which they are defined.

Size. A file with extension `.sz` normally contains three integer numbers on three lines. The first line is the number of non-zero entries, then the number of states reachable from the initial state, then the dimension of the state space, considered as a subset of $\mathbb{N}^d$.

State space. Xborne represents state spaces as a subset of $\mathbb{N}^d$. The value of d is in the "`.sz` file. The files with extensions `.cd` do the mapping between this multidimensional representation and the numbering of states. States are numbered starting with 0. Each row of the file begins with the index number of some state, and continues with the list of d coordinates, separated with white spaces or tabulations. States need not be present in increasing index order in the file.

The following example `testB3.cd` from the distribution of Xborne, represents the state $\{0,1,2\}^3$ with 27 elements.

```
          0          0          0          0
          1          1          0          0
          2          2          0          0
          3          0          1          0
          4          1          1          0
          5          2          1          0
          6          0          2          0
...
         21          0          1          2
         22          1          1          2
         23          2          1          2
         24          0          2          2
         25          1          2          2
         26          2          2          2
```

Matrices. Matrices of Xborne are stored in different formats. `marmoteCore` uses primarily the "increasing row/increasing column" format, abbreviated as "Rii".

In the Rii format, each line starts with the origin state index. It is followed by the number of entries in that row of the matrix. Then the entries themselves follow as pairs probability/destination state. All fields are separated by white spaces.

The following is the complete specification of the homogeneous 1-d random walk with 10 states and left/right probabilities 0.3 and 0.4.

```
0          2 6.000000e-01          0 4.000000e-01          1
1          3 3.000000e-01          0 3.000000e-01          1 4.000000e-01          2
2          3 3.000000e-01          1 3.000000e-01          2 4.000000e-01          3
3          3 3.000000e-01          2 3.000000e-01          3 4.000000e-01          4
4          3 3.000000e-01          3 3.000000e-01          4 4.000000e-01          5
5          3 3.000000e-01          4 3.000000e-01          5 4.000000e-01          6
6          3 3.000000e-01          5 3.000000e-01          6 4.000000e-01          7
7          3 3.000000e-01          6 3.000000e-01          7 4.000000e-01          8
8          3 3.000000e-01          7 3.000000e-01          8 4.000000e-01          9
9          2 3.000000e-01          8 7.000000e-01          9
```

D.2 MARCA

The MARCA format is defined in William Stewart's MARCA suite. It is one of the formats supported by the PSI1 suite of programs.

The first line comprises three integer numbers, respectively the row dimension, the column dimension and the number of non-zero entries of the matrix.

The second line is empty. The following ones list the entries as triples $(i, j, T_{i,j})$. States are numbered starting from 1.

The following example is the complete specification of a homogeneous 1-d random walk with 5 states and left/right probabilities 0.3 and 0.4.

```
5     5     13

      1          1 6.000000e-01
      1          2 4.000000e-01
      2          1 3.000000e-01
      2          2 3.000000e-01
      2          3 4.000000e-01
      3          2 3.000000e-01
      3          3 3.000000e-01
      3          4 4.000000e-01
      4          3 3.000000e-01
      4          4 3.000000e-01
      4          5 4.000000e-01
      5          4 3.000000e-01
      5          5 7.000000e-01
```

D.3 Ers

A simple text format where transitions are listed as triples $(i, j, T_{i,j})$. The first line indicates whether the model is discrete or continuous. The keyword `sparse` is implicit. The second line describes the size of the state space. The following ones are the entries. The lists ends with the keyword `stop`. A last line specifies the initial state.

States are numbered starting with 0.

The following example specifies a discrete-time Markov chain with 16 states and 0 initial state.

```
discrete sparse
```

```
16
0 0 0.9091
0 4 0.0909
1 1 0.9091
...
stop
0
```

The following example specifies a continuous-time Markov chain with 101 states and 100 as initial state. Observe that diagonal entries are *not* listed: they are deduced from the off-diagonal entries.

```
continuous sparse
101
0 1 1.000000
1 0 1.000000
1 2 0.500000
2 1 1.000000
2 3 0.333333
3 2 1.000000
3 4 0.250000
...
99 98 1.000000
100 99 1.000000
stop
100
```

D.4 R

The `R` output format is compatible with the `markovchain` package. It is a full matrix format. The Markov chain is written as:

- a matrix, in the form: `generator<-matrix(c(<entry>,<entry>,...),nrow=<size>,byrow=TRUE)`
- a state space, in the form: `statenames<-c("<name>","<name>",...)`
- the chain itself, in the form: `mc<-new("markovchain",states=statenames,transitionmatrix=generator)`

D.5 Scilab

The `Scilab` output format is available only for transition structures. The matrix is written as:

```
gen = [<entry> <entry> <entry> ....; <entry> <entry> <entry> ....; ...];
```

D.6 Maple

The `Maple` output format is available only for transition structures. It uses the sparse matrix format of Maple: entries are listed as a list of `(i,j)=value`. The user is responsible for wrapping this list into the linalg/matrix or LinearAlgebra/Matrix formats.

Index