`Marmote`
user manual

Alain Jean-Marie

v0.2.0, May 16, 2024

0.1 Introduction

`Marmote` is a C++ API for the construction and the analysis of Markov Chains. It is an evolution of the (legacy) `marmoteCore` API, itself 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.

(legacy) `marmoteCore` was developed by Alain Jean-Marie, Issam Rabhi and Hlib Mykhailenko.

`Marmote` is currently developed by Alain Jean-Marie, Emmanuel Hyon and Patrick Brown.

0.2 Markov chains, Markov modeling, Markov modelers

For the purposes of `Marmote`, 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

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

- 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)

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

0.3 Architecture

The general idea of the software is sketched in the following diagram.

The purpose of `Marmote` is to provide the *generic* API devoted to Markov Chains, in the sub-package `MarmoteMarkovChain`. The code for Markov chains itself is developed using objects provided by the `MarmoteCore` sub-package, or possibly external libraries or applications (e.g. imported from Xborne or Psi), or methods of some programming language (e.g. R, Python, Scilab/Matlab, etc.).

At higher levels, thematic libraries using Markov objects can be developed. `MarmoteMDP` is a library devoted to Markov Decision Processes. It is described in a separate document. As possible examples, MarmoteQueue and MarmoteGame are fictitious libraries which could exist one day, devoted to queueing theory and game theory models.

Ultimately, the libraries can be accessed either directly in applications, or linked to Scientific Workflow Management systems such as Kepler.

`Marmote` is developped in `C++` in order to ease the resuse of legacy `C`-style code. It offers naturally an API in `C++`. An API in `Python` is currently under development.

Chapter 1

Installation

This section provides instructions for installing the `MarmoteCore` library and compiling applications. These instructions, together with many commented examples, are available at `https://marmote.gitlabpages.inria.fr/marmote/`.

The recommended installation and compilation procedure is via `conda` and `cmake`. Conventional installation via "tarballs" is possible for some architectures, but not supported.

1.1 Installing Anaconda/miniconda

Before installing Marmote you will need to install `Anaconda` or `miniconda` following instructions at `https://conda.io/miniconda.html`.

The command-line instructions are executed in a terminal window (linux and macOS) or in a "anaconda prompt" or a "conda powershell" (MS Windows).

1.2 Preparing a Marmote workspace

Throughout these instructions, the directory `MAR_DIR` is used as the root directory. First create this directory and move into it:

```
$ mkdir MAR_DIR
$ cd MAR_DIR
```

1.3 Preparing the `conda` environment

The next step is to create the conda environment, containing the required conda packages, and activate it. This environment will be called `marmote-use`. First download the conda configuration file `marmote-use.yaml` corresponding to your architecture, from `https://marmote.gitlabpages.inria.fr/marmote/instructions.html`. Then execute:

```
$ conda env update -f marmote-use.yaml
$ conda activate marmote-use
```

Alternative: create the conda environment specifying the channels and the minimal packages on the command line.

```
$ conda create -n marmote-use
$ conda install -c marmote -c conda-forge marmote boost cmake -n marmote-use
$ conda activate marmote-use
```

1.4 Compiling application examples

Compiling an application using `Marmote` consists in creating a `cmake`. This simply requires placing the `C++` source files and a `CMakeLists.txt` configuration file in some directory. [1]

This process is explaned here using the illustrating examples available at `https://marmote.gitlabpages.inria.fr/marmote/examples.html`.

1.4.1 Compilation of a single file

First download the source code and the configuration file.

Next, execute:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

This produces the executable `example1`. To run the application:

```
$ ./example1 3 0.2 0.2 0.4
```

1.4.2 Compilation of two project files at the same time

First download the source files for example #1 and example #MDP10, together with the configuration file. Then execute:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

This produces two executables `example1` and `exampleMDP10`. To run the applications:

```
$ ./example1 3 0.2 0.2 0.6
$ ./exampleMDP10
```

1.4.3 Compilation all examples

First download the source files as a tar archive or a zip archive, together with the configuration file. Unpack the source files:

```
$ tar xf all_examples.tar
```

or

```
$ unzip all_examples.zip
```

Then execute:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

This produces all executables `example1` to `example7` and `exampleMDP10` to `example40`. To run the applications:

[1] Optionally, a separate directory can be created for each project.

```
$ ./example1 3 0.2 0.2 0.6
$ ./example7
$ ./exampleMDP10
$ ./exampleMDP20
```

Chapter 2

Programming with `Marmote`

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 `Marmote`, 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

`Marmote` rests on 4 abstract, high-level classes:

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

The classes `MarmoteSet`, `Distribution` and `TransitionStructure` and their different instances are provided by the module `MarmoteCore`. The class `MarkovChain` and its different instances is provided in the module `MarmoteMarkovChain`.

These main classes and their derived classes will be described 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 C, p. 66. See also 3.2.1.3.

2.1.2.2 Using existing Markov chains

`MarmoteCore` provides Markov models of several important families identified in the literature. See Appendix B, p. 64.

Currently implemented (a small part of the Markov Zoo): `TwoStateContinuous` and `TwoStateDiscrete`, the two-state Markov chains in continuous and discrete time, `Homogeneous1DRandomWalk`, `HomogeneousMultidRandomWalk`, `Homogeneous1DBirthDeath` and `HomogeneousMultidBirthDeath`, models of random walks, in continuous or discrete time, `PoissonProcess` and `MMPP`, models of arrival processes in continuous time, and `Felsenstein81`, a model for BioInformatics.

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 §3.2.2.9 and §3.2.2.4.

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` object is *cast* to a higer-level `MarkovChain` object, which has more solution methods.

Chapter 3

`Marmote` reference

Contents

We describe in this chapter the four main classes of `marmote` 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 `marmote`'s objects are private and can be accessed (read or write) only through specific "accessor/mutator" methods.

3.1 Basic types and constants

`Marmote` uses a few numeric and symbolic types. While using standard type like `int` or `long int` will work most of the time, it is advised to conform to this type convention.

timeType : describing whether the time model of the Markov Chain is discrete or contiuous; may be either of `DISCRETE`, `CONTINUOUS`, `UNDEFINED`, `UNKNOWN`.

stateType : the integer type used for state indices in state spaces. It may be negative since practical modeling sometime uses negative indices. It is currently implemented as `long long int` but it is discouraged to rely on this assumption.

cardinalType : the integer type used for the cardinal of state spaces. It may *not* be negative. It is currently implemented as `unsigned long long int` but it is discouraged to rely on this assumption.

cacheType : related to Monte-Carlo simulation of Markov Chains (see Section 3.2.1.6). May be either of `CACHE_FULL`, `CACHE_BASIC` and `CACHE_NONE`.

simLenType : related to Monte-Carlo simulation of Markov Chains (see Section 3.2.1.6). It is currently implemented as `unsigned long int` but it is discouraged to rely on this assumption.

inoutFormat : formats of representation of objects in files for input/output. Some formats apply specifically to matrices or vectors or sets, others apply to several object types.

`FORMAT_NONE` : Unspecified format

`FORMAT_MARMOTE` : Marmote format, also known as "ERS" format, described in Appendix C.3; applies to all objects

`FORMAT_MATLAB_FULL`,`FORMAT_MATLAB_SPARSE` : Matlab formats for matrices; apply to transition structures

`FORMAT_SCILAB_FULL`,`FORMAT_SCILAB_SPARSE` : Scilab formats for matrice, described in Appendix C.5; apply to transition structures

`FORMAT_MATRIXMARKET_SPARSE`, `FORMAT_MATRIXMARKET_FULL` : MatrixMarket formats for matrices; apply to transition structures

`FORMAT_MAPLE` : Maple format with full matrices; applies to transition structures

`FORMAT_R` : Matrix format for the `R` environment, described in Appendix C.4; applies to transition structures

`FORMAT_NUMPY` : Python format with full matrices; applies to transition structures

`FORMAT_MARCA` : Format for the `MARCA` software, described in Appendix C.2

`FORMAT_MARMOTE_FULL` : Marmote format with full matrices; applies to transition structures

`FORMAT_PSI3` : Psi3 yaml format; applies to transition structures

`FORMAT_XBORNE_CII`, `FORMAT_XBORNE_CUU`, `FORMAT_XBORNE_RII` : Xborne formats for matrices, described in Appendix C.1; apply to transition structures

`FORMAT_XBORNE_SIZE` : Xborne format for the matrix size specification ".sz"; applies to transition structures

`FORMAT_FLAT` : Flat format for sets

`FORMAT_STRUCTURED` : Structured format for sets

`FORMAT_XBORNE_SET` : Xborne format for set files ".cd", described in Appendix C.1

The default format is usually `FORMAT_MARMOTE`.

3.2 The `MarkovChain` object

3.2.1 Common features

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

3.2.1.1 Definition

This class is accessed with the directive

```
#include <marmoteMarkovChain/marmoteMarkovChain.h>
```

3.2.1.2 Attributes and accessors

`timeType`	`type_`	time type: discrete or continuous
`stateType`	`state_space_size_`	size of the state space
`MarmoteSet*`	`state_space_`	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. It coincides with the size of the `MarmoteSet` object which represents the state space.

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.

`stateType`	`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

Note that there is currently no accessor for the variable `state_space_`.

3.2.1.3 Constuctors

The class provides three constructors:[1]

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

The constructor `MarkovChain(stateType sz, timeType t);` creates a Markov chain object over a state space of size `sz` and with type given by `t`. The space type (`stateType`) is an integer, see Section 3.1. The time type can be `DISCRETE` or `CONTINUOUS`. The state space is implicitly a `MarmoteInterval` object.

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 C 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.

When a problem occurs during the construction (*e.g.* a 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.

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

3.2.1.4 Pseudo-constructor

A fourth way to create Markov chain objects is to use the class (static) method:

```
RandomMarkovChain(int multipleOfPeriod, stateType nStates);
```

This 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`$/d \rfloor$.

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

`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`. This feature is not available in the current version.

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.

3.2.1.6 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(long t, bool stats, bool traj, bool trace)
SimulationResult*  SimulateChainCT(double t, bool stats, bool traj, bool incr, bool trace)
SimulationResult*  SimulateChainCT_AllOpt(double t, bool stats, bool traj, bool incr,
                                          bool trace, bool fullState, cacheType ctype )
SimulationResult*  SimulateChainDT_AllOpt(long t, bool stats, bool traj,
                                          bool trace, bool fullState, cacheType ctype )
SimulationResult*  SimulatePSI(long t, bool stats, bool traj, bool trace)
SimulationResult*  SimulateChainR( double t, bool stats, bool traj, bool trace );
```

The methods `SimulateChainDT` and `SimulateChainDT_AllOpt` are specific to discrete-time Markov chains. Similarly, method `SimulateChainCT` and `SimulateChainCT_AllOpt` are 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.

`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.

Two more parameters are specific to methods `SimulateChainDT_AllOpt` and `SimulateChainCT_AllOpt`, which allow a finer control on the simulation and its output:

`fullState` : a flag specifying whether the complete representation of the state should be printed when `trace` is active; when set, the method `PrintState()` of the state space (see Section 3.4) is called after the index of the state is printed;

`cacheType` : controls the way transition distributions are stored during the simulation. The possibilities are `CACHE_FULL`, `CACHE_BASIC` and `CACHE_NONE`. Under `CACHE_FULL` (which is the default for simulations), all transitions are stored. This saves times when states are visited several times, but is impractical for large state spaces, and impossible for infinite state spaces. Under `CACHE_BASIC`, the two last transitions are kept in cache. Under `CACHE_NONE`, transition distributions are computed every time.

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

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
MarmoteSet*	state_space_	state space of the samples
cardinalType	state_space_size_	size of the state space
simLenType	trajectory_size_	number of transitions in the trajectory
bool	has_distrib_	flag corresponding to parameters *_cumTime_
bool	has_trajectory_	flag corresponding to parameters *_dates_
bool	has_increments_;	flag corresponding to parameter increments_
bool	trace_;	indicator of whether trajectory is traced
std::vector<double>	CT_dates_;	table of times in the trajectory
std::vector<simLenType>	DT_dates_;	table of times in the trajectory
std::vector<double>	increments_;	table of time increments in the trajectory
std::vector<cardinalType>	state_idx_;	table of states in the trajectory
cardinalType	max_state_;	maximal state reached, in case of infinite state space
std::vector<simLenType>	DT_cumTime_;	table of cumulated times in states for discrete time
std::vector<double>	CT_cumTime_;	table of cumulated times in states for continuous time
cardinalType	last_state_;	state of last record
double	last_time_;	time of last record, in continuous-time
std::ostream*	out_;	stream to which tracing is sent
bool	fullstate_;	indicates whether the indices should be expanded into states

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()
vector<double>*       CT_dates()
vector<simLenType>*   DT_dates()
vector<cardinalType>* states()
void                  recordCTSample( double date, cardinalType state )
void                  recordDTSample( simLenType date, cardinalType state )
// 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.

3.2.1.7 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.

3.2.1.8 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 `StationaryDistributionGthLD()` performs a call to the corresponding application from `Xborne`. This feature is not available in the current version.

The method `StationaryDistributionR()` uses the `R` environment. This feature is not available in the current version.

Standard 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 algorithm 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.

Red Light Green Light (RLGL) Iterative methods. The entry point to RLGL algorithms[3] is the method:

```
Distribution* StationaryDistributionRLGL(int tMax, double epsilon,
                                         DiscreteDistribution *iDis, bool progress,
                                         double alpha, string criterion)
```

Specific parameters to the `StationaryDistributionRLGL` are:

`alpha` : is a parameter in $[0, 1]$, used for the PageRank variant: $\alpha P + (1 - \alpha)\pi_0$; it has the default value 1;

`criterion` : indicates how nodes whose cash should be routed are chosen: available options are

- **`"thresh_1"`** : absolute cash greater than average (the default value),
- **`"thresh_2"`** : cash greater than sqare root of second moment,
- **`"RR"`** : round robin,
- **`"PI"`** : all cash is routed, equivalent to power iteration.

[3]The algorithms are described in `https://arxiv.org/abs/2008.02710`.

3.2.1.9 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. This feature is not available in the current version. The method `TransientDistributionDT` performs the calculation for discrete-time chains. It uses the power method.

3.2.1.10 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. Methods are available for directly computing their distribution, computing their average, or obtaining samples of them via Monte Carlo simulation.

Computing the hitting time distribution. There are no top-level methods for this in the current version. Derived classes implement such a method when it is known from the theory.

Computing the average hitting time. The methods computing the average of the hitting time distributions are:

```
double*   AverageHittingTime(bool *hitSetIndicator)
double*   AverageHittingTimeDT(bool *hitSetIndicator)
double*   AverageHittingTimeDTIterative(bool *hitSetIndicator)
double**  AverageHittingTimeDT_Conditional(bool *hitSetIndicator)
double**  AverageHittingTimeDT_ConditionalIterative(bool *hitSetIndicator)
```

The parameter common to these methods is:

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

The method `AverageHittingTime()` is available for both discrete-time and continuous-time Markov chains. It calls `AverageHittingTimeDT()` for discrete-time chains. For continuous-time chains, it performs uniformization, then calls `AverageHittingTimeDT()` on the uniforized chain.

The method `AverageHittingTimeDT()` 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 of the hitting set, from every state in the state space.

Likewise, the method `AverageHittingTimeDT_Conditional()` computes the average hitting times *conditioned* on the state hit. It uses uses the Gauss-Jordan method. The iterative version `AverageHittingTimeDT_ConditionalIterative()` uses an approximate fixed point method to solve each of the linear systems involved. Both return a two-dimensional square array of the size of the state space, where entry (i, j) represents the conditional hitting time of state j starting from state i. If j is not in the hitting set, this number is 0.

These methods apply only to discrete-time chains. They should not be used with large state spaces.

Simulating hitting times. Methods for obtaining samples of the hitting times are:

```
SimulationResult* SimulateHittingTime(DiscreteDistribution *iDis, bool *hitSetIndicator,
                                      unsigned long int nbSamples, double tMax)
```

```
SimulationResult* SimulateHittingTime(cardinalType iState, bool *hitSetIndicator,
                                      unsigned long int nbSamples, double tMax )
SimulationResult* SimulateHittingTimeCT(DiscreteDistribution *iDis, bool *hitSetIndicator,
                                        unsigned long int nbSamples, double tMax )
SimulationResult* SimulateHittingTimeDT(DiscreteDistribution *iDis, bool *hitSetIndicator,
                                        unsigned long int nbSamples, double tMax )
```

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. For discrete-time chains, it is interpreted as maximal the number of steps to perform, and for continuous-time chains, the maximal value of time. When this limit is reached, it is returned as the sample. Two variants of the method accept as parameter: either some initial distribution `iDis`, or some initial state `iState`.

Depending on the type of Markov chain, the specific methods `SimulateHittingTimeCT()` or `Simulate HittingTimeDT()` are called. They accept only an initial distribution parameter `iDis`. If the hitting time from a specific initial state is needed, then convert this state to a `DiracDistribution()` (see Section 3.5.2.2) or use the generic `SimulateHittingTime()`.

All simulation methods return a `SimulationResult` object. See its description in Section 3.2.1.6. The samples are recovered as a `vector<double>` for continuous-time Markov chains, and a `vector<simLenType>` for discrete-time Makov chains.

3.2.1.11 Output

Markov chain objects can be saved to a file using a number of formats. The methods for doing this are:

```
void  Write(std::ostream* out, string modelName, inoutFormat format )
void  Store(std::string modelName, inoutFormat format )
```

The method `Write()` places the output in the stream `out`. The method `Store()` places the output into a file. The file name is obtained by concatenating the `modelName` with an extension depending on the output format specified by parameter `format`. The formats currently supported are described in Section 3.1.

There is also a serialization method

```
std::string  toString( inoutFormat format )
```

which converts the result of `Write()` into a string.

3.2.2 Implementations

The following Markov chain models implemented. When a model can be considered as a sub-case of another model, the corresponding class inherits of that of the super-model. A hierarchy of some known models of the literature is proposed in Appendix B, in continuous time (Appendix B.1) and discrete time (Appendix B.2). Since not all the models are implemented yet, the inheritance relation between the classes described below is restricted to implemented ones.

Currently implemented models:

Name	description	inherits from
`TwoStateDiscrete`	generic discrete-time chain with two states	`MarkovChain`
`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`
`HomogeneousMultiDBirthDeath`	continuous-time random walk on subsets of $\mathbb{N}^d$	`MarkovChain`
`MMPP`	the Markov-modulated Poisson process on $\mathbb{N}$	`MarkovChain`
`PoissonProcess`	the usual Poisson process on $\mathbb{N}$	`Homogeneous1DBirthDeath`
`Felsenstein81`	continuous-time model for genome evolution	`MarkovChain`

3.2.2.1 `TwoStateDiscrete`

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

- the probability of jumps from state 0 to state 0, a,
- the probability of jumps from state 1 to state 1, b.

It is a currently class derived from `MarkovChain`: it probably should be derived from `Homogeneous1DRandomWalk`.

Definition. This class is accessed with the directive

```
#include "marmoteMarkovChain/marmoteTwoStateDiscrete.h"
```

Constructors. This class has a single constructor:

```
TwoStateContinuous( double a, double b );
```

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

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

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

```
TwoStateDiscrete* Copy();
BernoulliDistribution* StationaryDistribution();
BernoulliDistribution* TransientDistribution( double t );
std::vector<Distribution*> HittingTimeDistribution( bool* hitSetIndicator );
double* AverageHittingTime( bool* hitSetIndicator );
```

The parameter of the second methods is `t`, the time (measured in the number of time steps) at which the distributions should be evaluated.

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

$$\begin{aligned} p_0(t) &= \frac{1}{a+b-2}\left(b-1+(a+b-1)^t((a-1)p_0(0)+(1-b)p_1(0))\right) \\ p_1(t) &= \frac{1}{a+b-2}\left(a-1-(a+b-1)^t((a-1)p_0(0)+(1-b)p_1(0))\right) . \end{aligned}$$

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

$$\pi \;=\; \left(\frac{b-1}{a+b-2}, \frac{a-1}{a+b-2}\right)$$

and is defined only when $(a, b) \neq (1, 1)$.

Hitting time distributions are either Dirac distributions at 0, or geometric distributions on $\mathbb{N}^*$. The situation where the hitting set is empty is handled: the (defective) distribution `GeometricDistribution(1.0)` is returned by `HittingTimeDistribution()`, and the constant `INFINITE_DURATION` is returned by `AverageHittingTime()` (see Section 3.5.1).

3.2.2.2 `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 "marmoteMarkovChain/marmoteTwoStateContinuous.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<cardinalType> AbsorbingStates();
std::vector< std::vector<cardinalType> > RecurrentClasses();
std::vector< std::vector<cardinalType> > CommunicatingClasses();
stateType Period();
bool IsIrreducible();
bool IsAccessible(stateType stateFrom, stateType stateTo);
SimulationResult* SimulateChain(double tMax, bool stat, bool traj, bool incr, bool trace);
```

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

```
TwoStateDiscrete* Copy();
TwoStateDiscrete* Uniformize();
TwoStateDiscrete* Embed();
BernoulliDistribution* StationaryDistribution();
BernoulliDistribution* TransientDistribution( double t );
std::vector<Distribution*> HittingTimeDistribution( bool* hitSetIndicator );
double* AverageHittingTime( bool* hitSetIndicator );
```

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 $(\alpha, \beta) \neq (0, 0)$.

Hitting time distributions are either Dirac distributions at 0, or exponential distributions. The situation where the hitting set is empty is handled: the (defective) distribution `ExponentialDistribution(INFINITE_DURATION)` is returned by `HittingTimeDistribution()`, and the constant `INFINITE_DURATION` is returned by `Average HittingTime()` (see Section 3.5.1).

3.2.2.3 `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 "marmoteMarkovChain/marmoteHomogeneous1dBirthDeath.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:

```
Homogeneous1DBirthDeath* Copy();
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 $\mathbb{N}$.

3.2.2.4 `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 "marmoteMarkovChain/marmoteHomogeneous1dRandomWalk.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. The following methods have been re-implemented within `Homogeneous1DBirthDeath`:

```
SimulationResult* SimulateChain(double tMax, bool stat, bool traj, bool incr, bool trace);
void Write( std::ostream* out, string modelName, inoutFormat format );
void Store( string modelName, inoutFormat format );
```

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

```
Homogeneous1DRandomWalk* Copy();
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.[4]

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 $\mathbb{N}$.

Methods `Write()` and `Store()` (see Section 3.2.1.11) produce a representation of the model. Supported formats for objects `Homogeneous1DRandomWalk` are:

- `FORMAT_MARMOTE`: the standard sparse `Marmote` format;
- `Xborne` format `FORMAT_XBORNE_SIZE` for state space representations, `FORMAT_XBORNE_RII` and `FORMAT_XBORNE_CUU` for row-based and column-based outputs;
- matrix formats `FORMAT_MARCA` and `FORMAT_MATRIXMARKET_SPARSE`;
- Psi3 `yaml` configuration file `FORMAT_PSI3`;
- `R` full-matrix format `FORMAT_R`.

3.2.2.5 `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 `HomogeneousMultidTransition` class (see Section 3.3.2.2).

[4] 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.

Definition. This class is accessed with the directive

```
#include "marmoteMarkovChain/marmoteHomogeneousMultidRandomWalk.h"
```

Constructors. Two constructors are available.

```
HomogeneousMultiDRandomWalk( unsigned int nbDims, double* p, double* q );
HomogeneousMultiDRandomWalk( unsigned int nbDims, stateType* 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 methods are reimplemented.

```
DiscreteDistribution* StationaryDistribution();
int* SimulateHittingTime(cardinalType iState, bool *hitSetIndicator,
                         unsigned long int nbSamples, simLenType tMax);
Store(string modelName, inoutFormat format);
```

The `SimulateHittingTime` 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();
```

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 distributions, see `Homogeneous1DRandomWalk`.

The method `Store()` (see Section 3.2.1.11) handles specifically the format `FORMAT_XBORNE_RII`. Other formats are directly handled by the main class `MarkovChain`.

3.2.2.6 `HomogeneousMultiDBirthDeath`

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

- the number of dimensions d,
- the number of states in each dimension, possibly `INFINITE_STATE_SPACE_SIZE`
- the rates of jumps to the right in each dimension, $(\lambda_1, \ldots, \lambda_d)$,
- the rates of jumps to the left in each dimension, $(\mu_1, \ldots, \mu_d)$.

Objects of this class are equipped with a generator object, from the `HomogeneousMultidTransition` class (see Section 3.3.2.2).

Definition. This class is accessed with the directive

```
#include "marmoteMarkovChain/marmoteHomogeneousMultidBirthDeath.h"
```

Constructors. Two constructors are available.

```
HomogeneousMultiDBirthDeath(unsigned int nbDims, double* lambda, double* mu);
HomogeneousMultiDBirthDeath(unsigned int nbDims, stateType* sz, double* lambda, double* mu);
```

In both, `nbDims` is the number of dimensions d and `lambda`, `mu` are arrays of size d containing the rates λ_i and μ_i. The first form defines a birth-death process with $\mathbb{N}^d$ as state space. The second one defines a birth-death process 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 methods are reimplemented.

```
DiscreteDistribution* StationaryDistribution();
int* SimulateHittingTime(cardinalType iState, bool *hitSetIndicator,
                         unsigned long int nbSamples, simLenType tMax);
Store(string modelName, inoutFormat format);
```

The `SimulateHittingTime` 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();
```

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 distributions, see `Homogeneous1DRandomWalk`.

The method `Store()` (see Section 3.2.1.11) handles specifically the format `FORMAT_XBORNE_RII`. Other formats are directly handled by the main class `MarkovChain`.

3.2.2.7 `MMPP`

This class implements the generic Markov-modulated Poisson process. This is a continuous-time Markov chain model, characterized by:

- a continuous-time Markov chain, called the "environment", with generator Q;
- a vector of arrival rates r.

It is a "pure birth" process which behaves as follows. When the environment is in state i, arrivals occur according to a Poisson process of rate r_i. State transitions of the environment occur independently of the arrival processes.

Definition. This class is accessed with the directive

```
#include "marmoteMarkovChain/marmotePoissonProcess.h"
```

Constructors. This class has a single constructor:

```
MMPP( TransitionStructure* env, double* rates );
```

The parameter `env` is the generator of the environment, the parameter `rates` contains the rates r_i.

Re-implemented methods. None.

Specific methods. The following methods are specific to `MMPP`:

```
SimulationResult* SimulateChain(double tMax,
                          DiscreteDistribution* numberDis, DiscreteDistribution* phaseDis,
                          bool traj, bool trace=false, bool withIncrements=false );
MMPP* Copy();
```

The parameters and behavior of `SimulateChain()` are similar to `MarkovChain::SimulateChain()` (see Section 3.2.1.6) but it admits *two* parameters for specifying the initial conditions. Parameter `numberDis` is for the initial value of the state of the arrival process: a random number in $\mathbb{N}$. Parameter `phaseDis` is the distribution of the state (also known as "phase") of the environment at time $t = 0$.

3.2.2.8 `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 "marmoteMarkovChain/marmotePoissonProcess.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 3.5.2.6.

Specific methods. None.

3.2.2.9 `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 "marmoteMarkovChain/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.

3.3 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.

3.3.1 Common features

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

3.3.1.1 Definition

This class is accessed with the directive

```
#include "marmoteCore/marmoteTransitionStructure.h"
```

3.3.1.2 Attributes and accessors

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 3.3.1.3
`bool`	`ReadEntry(FILE*);`	
`double`	`RowSum(int)`	evaluation of the sum $\sum_j T_{i,j}$ for some i
`// transformations`		
`TransitionStructure*`	`Uniformize()`	see Section 3.3.1.5
`TransitionStructure*`	`Embed()`	see Section 3.3.1.5
`// 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.

3.3.1.3 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

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

3.3.1.4 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()`.

3.3.1.5 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,

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

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 3.3.1.3).

3.3.1.6 I/O

The `Write()` method outputs the structure on some file decriptor, using a given format (see Appendix C). 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.

3.3.2 Implementations

Transition structures implemented:

- `TransitionStructure/SparseMatrix`
- `TransitionStructure/HomogeneousMultidTransition` (generalized birth-death)

Projected:

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

3.3.2.1 `SparseMatrix`

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

Definition. This class is accessed with the directive

```
#include "marmoteCore/marmoteSparseMatrix.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 C). 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()` method computes 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 result is stored internally. The `getReverted()` method returns this transposed matrix; it computes it before if not already present.

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.

3.3.2.2 `HomogeneousMultidTransition`

The classe `HomogeneousMultidTransition` 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 "marmoteCore/marmoteHomogenousMultidTransition.h"
```

Constructors. The class has a single constructor:

```
HomogeneousMultidTransition(timeType type, unsigned int nbDims, stateType* dim_size,
                            double* left_trans, double* right_trans);
```

The parameter `type` specifies whether the transition structure is in continuous or discrete time. The parameter `nbDims` is the number of dimensions d and the parameter `dimSize` specifies the extension in each dimension. The process has $\times_{i=1}^{d}[0..n_i - 1]$ as state space, where the n_i are the values in the array `dimSize`. The arrays `left_trans` and `right_trans` will be interpreted as jumping probabilities or rates, depending on the time type. See Section 3.2.2.5 or Section 3.2.2.6 for intepretations.

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);
HomogeneousMultidTransition* Copy();
HomogeneousMultidTransition* Uniformize();
HomogeneousMultidTransition* 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.

3.4 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.

3.4.1 Common features

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

States are represented by arrays of integers. A special type is used for states:

MarmoteState : the type representing a state, that is, a member of some `MarmoteSet` object. It is currently implemented as an array of `cardinalType` but it is discouraged to rely on this assumption.

3.4.1.1 Definition

This class is accessed with the directive

```
#include "marmoteCore/marmoteSet.h"
```

3.4.1.2 Attributes and accessors

All `MarmoteSet` objects have the following common attributes:

`enum opType`	`UNION, PRODUCT, SIMPLE`	specify the type of construction
`int`	`nb_dimensions_`	number of dimensions for products
`int`	`nb_zones_`	number of subsets for unions
`stateType`	`cardinal_`	total cardinal
`MarmoteSet**`	`zone_`	array of subsets for unions
`MarmoteSet**`	`dimension_`	array of dimensions for products
`MarmoteState`	`state_buffer_`	

`// 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

3.4.1.3 Constructors

```
// constructors
  MarmoteSet()
  MarmoteSet( MarmoteSet **list, unsigned int nb, opType t )
```

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`.

3.4.1.4 State representation and indexing

```
// state-index conversions
void  DecodeState(int index, MarmoteState);
int   Index(MarmoteState)
```

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.[5]

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.

3.4.1.5 Walking through sets

Walking through sets is performed using the following methods.

```
// state space exploration
void  FirstState(MarmoteState)
void  NextState(MarmoteState)
bool  IsFirst(MarmoteState)
```

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;

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

3.4.1.6 I/O

I/O methods related to states and set are the following:

```
// I/O utilities
void  enumerate()
void  PrintState(FILE* out, int index);
```

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

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.

3.4.2 Implementations

Six sets or classes of sets are currently implemented:

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

Name	description	inherits from
`EmptySet`	the empty set	`MarmoteSet`
`MarmoteInterval`	a simple 1-dimensional discrete interval, possibly infinite	`MarmoteSet`
`Integers`	the set of all integers	`MarmoteInterval`
`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`

3.4.2.1 `EmptySet`

This class implements the empty set. It is intended primarily as an exercise, but can also be used as a default value or a place-holder in some cases.

Definition. This class is accessed with the directive

```
#include "marmoteCore/marmoteEmptySet.h"
```

Constructors. The class has a single constructor:

```
EmptySet();
```

Re-implemented methods.

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

Method `IsFinite()` returns `true`. Method `Belongs()` returns `false`, since there are no states in the set. Likewise for `IsFirst()`. The other methods issue error messages and do nothing. Method `Index()` returns 0 by convention.

Specific methods. The class does not provide specific methods, except for `Copy()`.

3.4.2.2 `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 "marmoteCore/marmoteInterval.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(MarmoteState);
void FirstState(MarmoteState);
void NextState(MarmoteState);
void DecodeState(int index, MarmoteState);
int Index(MarmoteState);
void PrintState(FILE* out, MarmoteState);
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.

3.4.2.3 `MarmoteIntegers`

This class implements the set of all natural integers $\mathbb{N} = \{0, 1, 2, \ldots\}$. It is a particular case of `MarmoteInterval`, from which it inherits.

Definition. This class is accessed with the directive

```
#include "marmoteCore/marmoteIntegers.h"
```

Constructors. The class has a single constructor:

```
MarmoteIntegers();
```

Re-implemented methods. The following methods are reimplemented from `MarmoteInterval`:

```
bool IsFinite();
bool Belongs(MarmoteState);
```

Method `IsFinite()` returns false.

Specific methods. The class does not provide specific methods, except for `Copy()`.

3.4.2.4 `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 "marmoteCore/marmoteBox.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(MarmoteState);
void FirstState(MarmoteState);
void NextState(MarmoteState);
void DecodeState(int index, int* buf);
int Index(int* buf);
void PrintState(FILE* out, MarmoteState);
```

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.

3.4.2.5 `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 "marmoteCore/marmoteBinarySequence.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(MarmoteState);
void FirstState(MarmoteState);
void NextState(MarmoteState);
void DecodeState(int index, int* buf);
int Index(int* buf);
void PrintState(FILE* out, MarmoteState);
```

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.

3.4.2.6 `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 "marmoteCore/marmoteBinarySimplex.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(MarmoteState);
void FirstState(MarmoteState);
void NextState(MarmoteState);
void DecodeState(int index, int* buf);
int Index(int* buf);
void PrintState(FILE* out, MarmoteState);
```

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.

3.4.2.7 `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 "marmoteCore/marmoteSimplex.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, MarmoteState);
```

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.

3.5 The `Distribution` object

3.5.1 Common features

3.5.1.1 Definition

This class is accessed with the directive

```
#include "marmoteCore/marmoteDistribution.h"
```

3.5.1.2 Constants and types

Some constants are specifically defined for `Distribution` objects.

`distType` : type of distances between distributions; existing values are `DISTANCE_L1`, `DISTANCE_L2`, `DISTANCE_LINFINITY`, `DISTANCE_TV` (for the total variation distance);

`INFINITE_DURATION` : representation of infinity, in the case where some moments of the distribution can be infinite;

`INFINITE_RATE` : representation of infinity, in the case where the "rate" is a meaningful concept for the distribution, and its mathematical value is infinity (inverse of 0).

3.5.1.3 Attributes and accessors

`Distribution` objects have one common attribute:

`double`	`mean_`	the mathematical expectation of the distribution

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`	`Distance(distType, Distribution*, Distribution*)`	compute some 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:

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

In addition to the generic `Distance()` method, some shorthand forms exist for the four distances of the `distType` type: `DistanceL1(Distribution*,Distribution*)`, `DistanceL2(Distribution*,Distribution*)`, `DistanceLInfinity(Distribution*,Distribution*)`, `DistanceTV(Distribution*,Distribution*)`. Not all distances are defined for all distributions, and even when defined, not all are implemented.

3.5.2 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`
`ShiftedGeometricDistribution`	geometric distribution over $\{a, a+1, \dots\}$	`DiscreteDistribution`
`GeometricDistribution`	geometric distribution over $\mathbb{N}$	`ShiftedGeometricDistribution`
`PoissonDistribution`	Poisson distribution	`DiscreteDistribution`
`PhaseTypeDiscreteDistribution`	discrete phase-type distribution	`DiscreteDistribution`
`GammaDistribution`	Gamma distribution	`Distribution`
`ErlangDistribution`	Erlang distribution	`GammaDistribution`
`ExponentialDistribution`	negative exponential distribution	`ErlangDistribution`
`GaussianDistribution`	univariate gaussian distribution	`Distribution`
`UniformDistribution`	uniform continuous distribution	`Distribution`
`PhaseTypeDistribution`	continuous phase-type distribution	`Distribution`

3.5.2.1 `DiscreteDistribution`

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

Definition. This class is accessed with the directive

```
#include "marmoteCore/marmoteDiscreteDistribution.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`. A *tolerance* of 10^{-8} is applied to the parameter `value`.

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 ] );
```

3.5.2.2 `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 "marmoteCore/marmoteDiracDistribution.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.

3.5.2.3 `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 "marmoteCore/marmoteBernoulliDistribution.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.

3.5.2.4 `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 "marmoteCore/marmoteUniformDiscreteDistribution.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 Laplace( double s );
double DLaplace( double s );
double Cdf( double x );
double Ccdf( double x );
bool HasMoment( int order );
PhaseTypeDistribution *Rescale( double factor );
PhaseTypeDistribution *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.

3.5.2.5 `ShiftedGeometricDistribution`

This is the geometric distribution on the set $\{a, a+1, \ldots\}$ where $a \in \mathbb{N}$. It has two parameters: an integer offset a which is the smallest value the random variable can take, and a probability p, interpreted as the probability that X is *strictly* larger than a. In other words, the distribution is:

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

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 "marmoteCore/marmoteShiftedGeometricDistribution.h"
```

Constructors. The class has a single constructor,

```
ShiftedGeometricDistribution( long int offset, double p );
```

which sets the parameters a (the "offset") and p.

Re-implemented methods.

```
double  getProbaByIndex(cardinalType idx);
double  getProba(double k);
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()` is available only for default format `FORMAT_MARMOTE` and writes "`ShiftedGeometric on [`a`..+oo) with proba` p`: P(k) =` $(1-p)$ `x (`p`)^(k-`a`) k=`a`..+oo`".

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

```
long int  offset();
double  p();
double  getRatio();
```

The method `offset()` gives access to the parameter a. Observe that its return type is `double`. Both methods `p()` and `getRatio()` give access to the parameter p. Methods `getProba()` returns the probability $P(X = k)$.

3.5.2.6 `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.

Since it is a particular case of `ShiftedGeometricDistribution` with offset $a = 0$, the class inherits from it.

Definition. This class is accessed with the directive

```
#include "marmoteCore/marmoteGeometricDistribution.h"
```

Constructors. The class has a single constructor,

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

which sets the parameter p.

Re-implemented methods.

```
double  getProbaByIndex(cardinalType idx);
double  getProba(double k);
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()` accepts the standard format `FORMAT_MARMOTE` but also the Maple format `FORMAT_MAPLE`. In the first case, it writes: "`Geometric on N with proba` p`: P(k) =` $(1 - p)$ `x (`p`)^k, k=0..+oo`". In the second case, it writes: "`Statistics[RandomVariable](GeometricDistribution(` $1 - p$ `) )`".

Specific methods. None.

3.5.2.7 `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 "marmoteCore/marmotePoissonDistribution.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   getProba( double k );
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. This feature is not available in the current version.

Method `Write()` accepts the standard format `FORMAT_MARMOTE` but also the Maple format `FORMAT_MAPLE`. In the first case, it writes: "`Poisson with rate` λ". In the second case, it writes: "`Statistics[RandomVariable](PoissonDistribution(` λ `))`".

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

```
double  lambda();
```

It is an accessor for parameter λ.

3.5.2.8 `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:

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

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

Definition. This class is accessed with the directive

```
#include "marmoteCore/marmoteGammaDistribution.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()` reimplements the method from `Distribution`. It accepts the standard format `FORMAT_MARMOTE` but also the Maple format `FORMAT_MAPLE`. In the first case, it writes: "`Gamma shape` k `rate` λ". In the second case, it writes: "`Statistics[RandomVariable]( GammaDistribution(` θ`,` k `) )`".

Specific methods. None.

3.5.2.9 `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 $k\theta$. The class `ErlangDistribution` inherits from `GammaDistribution`.

Definition. This class is accessed with the directive

```
#include "marmoteCore/marmoteErlangDistribution.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()` reimplements the method from `GammaDistribution`. It accepts the standard format `FORMAT_MARMOTE` but also the Maple format `FORMAT_MAPLE`. In the first case, it writes: "`Erlang phases` k `rate` λ". In the second case, it writes: "`Statistics[RandomVariable]( ErlangDistribution(` θ`,` k `) )`".

Specific methods. None.

3.5.2.10 `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 "marmoteCore/marmoteExponentialDistribution.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()` reimplements the method from `ErlangDistribution`. It accepts the standard format `FORMAT_MARMOTE` but also the Maple format `FORMAT_MAPLE`. In the first case, it writes: "`Exponential distribution with mean` $1/\lambda$". In the second case, it writes: "`Statistics[RandomVariable]( Exponential Distribution(` $1/\lambda$ `) )`". The `toString()` method returns "`Exponential(m=`$1/\lambda$`)`".

Specific methods. None.

3.5.2.11 `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 "marmoteCore/marmoteUniformDistribution.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.

3.5.2.12 `GaussianDistribution`

This is the scalar (univariate) Gaussian distribution. It is specified by two parameters: its mean m and variance v. The variance is the square of the standard deviation σ. The distribution is given by its density:

$$\frac{\mathrm{d}P}{\mathrm{d}x}(X \leq x) \;=\; \frac{1}{\sqrt{2\pi v}} \;\exp\left(-\frac{(x-m)^2}{2v}\right).$$

Definition. This class is accessed with the directive

```
#include "marmoteCore/marmoteGaussianDistribution.h"
```

Constructors. This class has a single constructor:

```
GaussianDistribution( double mean, double variance )
```

which sets parameters m and v.

Re-implemented methods.

```
double Mean();
double Rate();
double Moment( int order );
double Laplace( double s );
double DLaplace( double s );
double Cdf( double x );
double Ccdf( double x );
bool HasMoment( int order );
UniformDistribution *Rescale( double factor );
UniformDistribution *Copy();
double Sample();
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 Gaussian distribution with mean $f \times m$ and variance $f^2 \times v$.

Method `Write()` reimplements the method from `Distribution`. It accepts the standard format `FORMAT_MARMOTE` but also the Maple format `FORMAT_MAPLE`. In the first case, it writes: "`Gaussian mean` m `variance` v". In the second case, it writes: "`Statistics[RandomVariable]( GaussianDistribution(` m, $v^{1/2}$ `) )`".

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

```
double Variance()
```

It gives access to parameter v.

3.5.2.13 `PhaseTypeDistribution`

This class represents the phase-type distribution with a discrete (and finite) phase space. It is characterized by a continuous-time transition structure T and an initial distribution β on the phase space. The transition structure is a priori "sub-stochastic" so that jumps are possible to some state outside the phase state, called the terminal state. The distribution is that of the time it takes for the continuous-time Markov chain with generator T to hit the terminal state (i.e. exit the phase space), when it starts from distribution β. It may be that the distribution is defective: when the terminal state is never reached with some positive probability.

Definition. This class is accessed with the directive

```
#include "marmoteCore/marmotePhaseTypeDistribution.h"
```

Constructors. The class has a single constructor:

```
PhaseTypeDistribution(TransitionStructure* T, DiscreteDistribution* beta );
```

It defines the values of T and β.

Re-implemented methods. The following methods are re-implemented from `Distribution`:

```
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 );
UniformDiscreteDistribution *Rescale( double factor );
UniformDiscreteDistribution *Copy();
double Sample();
void    IidSample( int n, double* s );
void    Write( FILE *out, int mode );
```

The mean is given by the formula $m = \beta T^{-1}\mathbf{1}$. In the defective case, this evaluates to `INFINITE_DURATION`. The `Moment()` method is restricted to order 2; the second moment is $m_2 = 2\beta T^{-2}\mathbf{1}$. The distribution has moments of all orders if $m < \infty$, so that the argument of `HasMoment()` is ignored. The computation of cdf and its Laplace transform, `Cdf()`, `Ccdf()`, `Laplace()` and `DLaplace()`, is currently not implemented. Rescaling by a factor f produces a phase-type distribution with same distribution β and phase transitions fT.

Sampling of the distribution with `Sample()` and `IidSample()` by simulating hitting times of the associated Markov chain (see Section 3.2.1.10).

The `Write()` method accepts only the standard format `FORMAT_MARMOTE` and produces as output:

```
Continuous-time Phase-type with rate matrix: [ ... ] and proba vector ...
```

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

```
TransitionStructure* trans() { return trans_; }
DiscreteDistribution* iDis() { return iDis_; }
```

They are the accessors to the parameters T and β of the distribution.

3.5.2.14 `PhaseTypeDiscreteDistribution`

This class represents the discrete-time phase-type distribution with a discrete (and finite) phase space. It is characterized by a probability transition structure P and an initial distribution β on the phase space. The transition structure is a priori sub-stochastic so that jumps are possible to some state outside the phase state, called the terminal state. The distribution is that of the time (number of steps) it takes for the continuous-time Markov chain with generator T to hit the terminal state (i.e. exit the phase space), when it starts from distribution β. It may be that the distribution is defective: when the terminal state is never reached with some positive probability.

Definition. This class is accessed with the directive

```
#include "marmoteCore/marmotePhaseTypeDiscreteDistribution.h"
```

Constructors. The class has a single constructor:

```
PhaseTypeDiscreteDistribution(TransitionStructure* T, DiscreteDistribution* beta );
```

Re-implemented methods. The following methods are re-implemented from `Distribution`:

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

The mean is given by the formula $m = \beta(I-P)^{-1}\mathbf{1}$. In the defective case, this evaluates to `INFINITE_DURATION`. The `Moment()` method is currently restricted to order 1. The distribution has moments of all orders if $m < \infty$, so that the argument of `HasMoment()` is ignored. The computation of the Laplace transform, `Laplace()`

and `DLaplace()`, is currently not implemented. Rescaling is not possible for these distributions: a copy is returned by `Rescale()` with a warning message.

Sampling of the distribution with `Sample()` and `IidSample()` by simulating hitting times of the associated Markov chain (see Section 3.2.1.10).

The `Write()` method accepts only the standard format `FORMAT_MARMOTE` and produces as output:

```
Discrete-time Phase-type with rate matrix: [ ... ] and proba vector ...
```

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

```
TransitionStructure* trans() { return trans_; }
DiscreteDistribution* iDis() { return iDis_; }
```

They are the accessors to the parameters P and β of the distribution.

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. The matrix is:

$$P = \begin{pmatrix} 0.25 & 0.5 & 0.25 \\ 0.4 & 0.2 & 0.4 \\ 0.4 & 0.3 & 0.3 \end{pmatrix}.$$

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. The matrices are:

$$P_1 = \begin{pmatrix} 0.2 & 0.8 & & & & & & \\ 0.25 & 0.25 & 0.25 & 0.25 & & & & \\ 0.6 & & 0.4 & & & & & \\ & & 0.3 & 0.2 & 0.25 & 0.25 & & \\ & & & & 0.1 & 0.3 & 0.3 & 0.3 \\ & & & & & 1.0 & & \\ & & & & 0.5 & & 0.5 & \\ & & & & 0.4 & 0.2 & 0.2 & 0.5 \end{pmatrix} \quad P_2 = \begin{pmatrix} 0.2 & 0.8 & & & & & & \\ 0.25 & 0.25 & 0.25 & 0.25 & & & & \\ 0.6 & & 0.4 & & & & & \\ & & 0.3 & 0.2 & 0.25 & 0.25 & & \\ & & & & 0.1 & 0.3 & 0.3 & 0.3 \\ & & & & & 0.5 & & 0.5 \\ & & & & 0.5 & & 0.5 & \\ & & & & 0.4 & 0.2 & 0.2 & 0.5 \end{pmatrix}$$

$$P_3 = \begin{pmatrix} 0.2 & 0.8 & & & & & & \\ 0.25 & 0.25 & 0.25 & 0.25 & & & & \\ 0.6 & & 0.4 & & & & & \\ & & 0.3 & 0.2 & 0.25 & 0.25 & & \\ & & & & 0.1 & 0.3 & 0.3 & 0.3 \\ & & & 0.5 & & 0.5 & & \\ & & & & 0.5 & & 0.5 & \\ & & & & 0.4 & 0.2 & 0.2 & 0.5 \end{pmatrix}$$

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. The matrices are:

$$P_1 = \begin{pmatrix} 0.2 & 0.8 & & & & & & \\ 0.25 & 0.25 & 0.25 & 0.25 & & & & \\ 0.6 & & 0.4 & & & & & \\ & & 0.3 & 0.2 & 0.25 & 0.25 & & \\ & & & & 0.1 & 0.3 & 0.3 & 0.3 \\ & & & & & 1 & & \\ & & & & 0.5 & & 0.5 & \\ & & & & 0.4 & 0.2 & 0.2 & 0.2 \end{pmatrix} \quad P_2 = \begin{pmatrix} 0.6 & 0.4 & & & & & & \\ 0.3 & 0.3 & 0.4 & & & & & \\ & 0.3 & 0.3 & 0.4 & & & & \\ & & 0.3 & 0.3 & 0.4 & & & \\ & & & 0.3 & 0.3 & 0.4 & & \\ & & & & 0.3 & 0.3 & 0.4 & \\ & & & & & 0.3 & 0.3 & 0.4 \\ & & & & & & 0.3 & 0.7 \end{pmatrix}$$

Usage:

```
example4
```

Tasks performed:

- create two `MarkovChain` objects by reading their description in files `IO_example_in1.mcl` and `IO_example_in2.mcl`;
- output them again to files `IO_example_out1.mcl` and `IO_example_out2.mcl`;
- write the generator of these chains to files `IO_example_mat1.mmt` and `IO_example_mat2.mmt`;
- 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 3.2.1.3).

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 3.2.1.8);
- 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) `MarkovChain`s:
  - 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 three first of them are matrices P_1, P_2 and P_3 of Example 3. The probabilities of transitions are not represented since they are not relevant to this analysis.

A.2 Advanced examples

A.2.1 Using complex `MarmoteSet` objects

TBD

A.2.2 Using the hierarchy of models

TBD

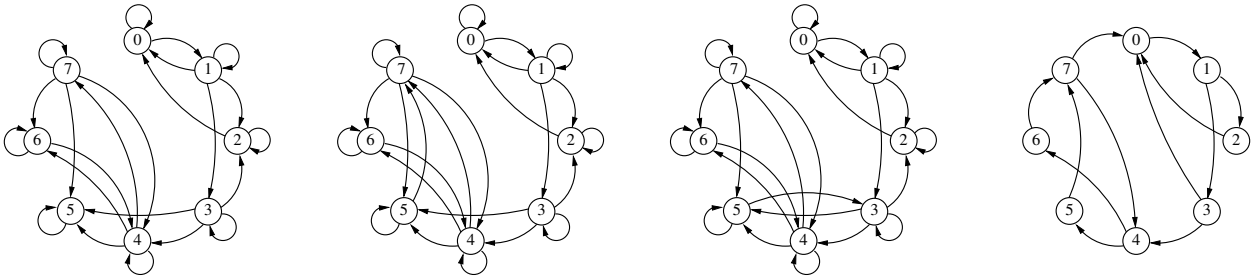


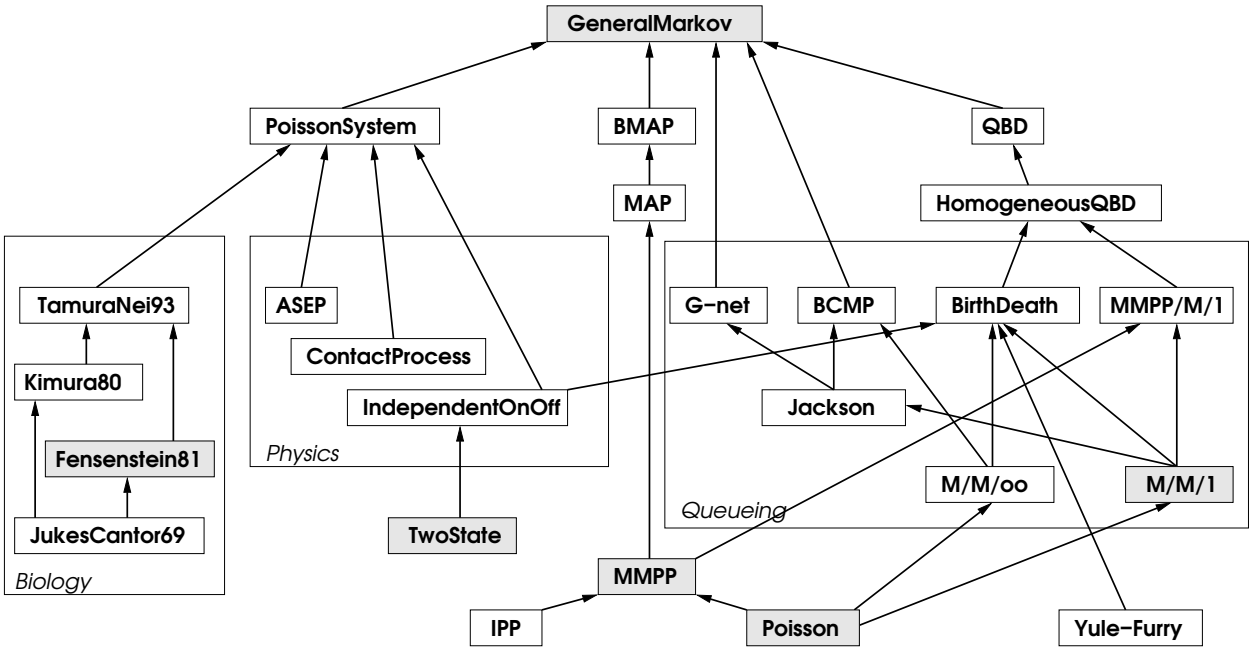
Figure A.1: Diagrams of the four Markov chains analyzed in Example 7

Appendix B

The Markov Zoo

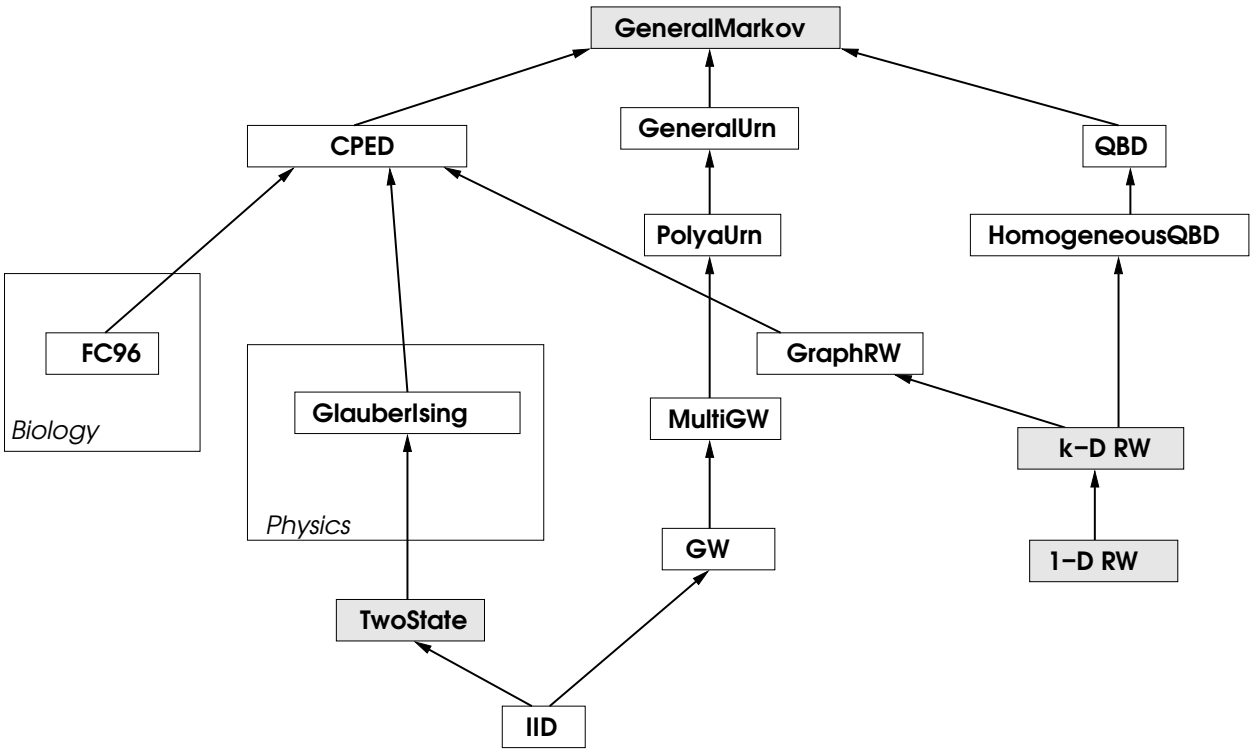
A hierarchy of continuous-time models is proposed in the diagram below. Models featured in grey are those currently impemented. Abbreviations used are: ASEP for Asymmetric Exclusion Process, MAP for Markov Arrival Process (and not Markov *Additive* Process), BMAP for Batch Markov Arrival Process, IPP for Interrupted Poisson Process, MMPP for Markov-Modulated Poisson Process, QBD for Quasi-Birth-Death. Other names are classical in thir respective fields.

B.1 The Continuous-Time Markov Zoo



B.2 The Discrete-Time Markov Zoo

A hierarchy of discrete-time models is proposed in the diagram below. Models featured in grey are those currently impemented. Abbreviations used are: GW for Galton-Watson, QBD for Quasi-Birth-Death, FC96 for Felsenstein-Churchill 96, CPED for Constant-Probability Event-Driven process. Other names are classical in thir respective fields.


GeneralMarkov
CPED
GeneralUrn
QBD
PolyaUrn
HomogeneousQBD
FC96
Biology
GraphRW
GlauberIsing
MultiGW
k−D RW
Physics
GW
1−D RW
TwoState
IID

Appendix C

File formats

C.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
```

C.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
```

C.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
```

C.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)`

C.5 Scilab

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

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

C.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.

C.7 Matrix Market

The `MatrixMarket` output format is available only for transition structures. This format has been specified by NIST, see: `https://math.nist.gov/MatrixMarket/formats.html`. There are two variants: the "coordinate" format for sparse matrices, and the "array" format for full matrices. In general, a line beginning with "`%`" is a comment.

- Sparse matrices: the file begins with the header
  `%%MatrixMarket matrix coordinate real general` .
  The first (non-comment) line contains three numbers: row size, column size and number of non-zero entries. The entries then follow in the format `row col value`.

  This format is similar to the `MARCA` format (Appendix C.2) except that the blank line after the size specification is not mandatory. On the other hand, the format is "flexible" in the sense that extra blank lines may be inserted.

- Full matrices: the file begins with the header
  `%%MatrixMarket matrix array real general` .
  The first (non-comment) line contains two numbers: row size and column size. The entries then follow, one by row.

C.8 Harwell-Boeing (HB)

The Harwell-Boeing format (HB or HBF) is a compact coding of matrices. Its implementation in `marmote` is pending.