marmoteCore: a Markov Modeling Platform

Alain Jean-Marie
Université Côte d'Azur – Inria
Alain.Jean-Marie@inria.fr

ABSTRACT

We present the marmoteCore software project, an open environment for modeling with Markov chains. This platform aims at providing the general scientific user with tools for creating Markov models and accessing the many solution algorithms available for their analysis. We describe its object-oriented architecture, some of its presently available features, and we discuss through examples how existing software can be interfaced with it.

KEYWORDS

Markov modeling, software environment

ACM Reference format:
Alain Jean-Marie. 2017. marmoteCore: a Markov Modeling Platform. In *Proceedings of VALUETOOLS 2017, Venezia, Italia, December 2017,* 6 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Modeling with Markov chains is an activity common to many fields of science and engineering. Interacting particle systems of Physics, genome evolution models of Biology, epidemic models of Medicine, population models of Ecology, queuing systems of Operations Research, Petri nets and stochastic model checking formalisms, all those popular models are based on Markov chains. Monte-Carlo simulation of Markov chains are commonly used for producing samples of distributions of combinatorial objects, physical systems etc.

Despite this practical importance, there exists no software environment providing the general scientific user with, at the same time, a collection of ready-to-use well-known models and general modeling constructions and solution methods, all accessible using an uniform programming interface.

This research was funded by the French National Research Agency under grant ANR-12-MONU-00019, project "MARMOTE" (MARkovian MOdeling Tools and Environments), https://wiki.inria.fr/MARMOTE/Welcome. The author is also affiliated to LIRMM, Universit´e de Montpellier/CNRS.

VALUETOOLS 2017, December 5–7, 2017, Venice, Italy

ACM ISBN 978-1-4503-6346-4/17/12…$15.00
https://doi.org/10.1145/3150928.3150960

marmoteCore is the prototype of such a platform. It consists in a programming library[1] with an object-oriented point of view, that allows programmers to create Markov models and "solve" them by using many available algorithms. These solvers are often *embedded* algorithms or applications written by third parties. Another feature of the library is the availability of ready-made Markov models of several levels of generality, with adapted solution methods.

What is not marmoteCore. The intent of the project is not to program anew all methods that already exist for creating or analyzing Markov chains. The idea is not either to develop an integrated modeling and analysis tool: this sort of functionality is offered by WMS (see Section 5.4) or integrated platforms such as DTK.[2]

The purpose of this document is to demonstrate the potentialities of marmoteCore's architecture. This is not a programming guide, yet several code excerpts will give a flavor of the way programs are written with this library. Indeed, while still being in development, marmoteCore is mature enough so as to be usable in practice.

The paper is organized as follows. Section 2 presents the general purpose and architecture of the software. Section 3 presents the core of the programmer's interface. In Section 4 we describe the principal methods of analysis currently available. In Section 5, we discuss the way marmoteCore interacts with existing scientific software. We conclude with our plans for the future in Section 6.

Acknowledgment. The existence of owes much to the contribution of Issam Rabhi, Emmanuel Hyon and Hlib Mykhailenko. The author wishes to acknowledge their contribution to the architecture and the code of the software.

2 MAIN CHARACTERISTICS

2.1 Background

The idea of providing users with software environments to help them realize their modeling experiments is, of course, not original. The most popular mathematical modeling environments, such as Matlab, Scilab, Mathematica, Maple, R, Sage, all provide packages with functions specialized to families of models (e.g. Matlab's SimuLink for system modeling), or analysis techniques (e.g. statistical packages).

To this day however, there does not exist standard packages specifically devoted to modeling with *discrete space* Markov chains, considered as entities richer than simple matrices. In the particular context of performance modeling, the list of software maintained at [6] is typical of the fact that there is a large variety of solutions techniques adapted

[1] Available at http://marmotecore.gforge.inria.fr.
[2] http://dtk.inria.fr/

to specific models or families of models, but also that there is no uniform presentation or unique access point to these tools, that would help the newcomer to apply one to his/her needs. On the other hand, analysis tools are sometimes presented as "packaged" behind a graphical user interface, as were pioneers like GreatSPN, Tangram-II, and are modern tooks like Java Modeling Tools, CosyVerif, OpenAlea, to name just a few. In some cases, these tools and their advanced algorithms can hardly be used outside of their application context.

We believe there is a need for libraries giving access to efficient numerical algorithms at a lower programming level.

2.2 The general purpose of `marmoteCore`

The development of the `marmoteCore` software pursues several objectives at the same time. On the one hand, it aims at providing to the general scientific user a "modeling environment" for Markovian systems. This environment should provide at the same time an access to basic models of the literature (an thus serve as a repository of models, together with their "classical" results), and an access to state-of-the-art solution algorithms. Through a programming interface, this *Markov user* should be able to use `marmoteCore`'s library as "routine" in a scientific project.

In the practice of Markov modeling for the performance evaluation of concrete situations, it may happen that the modeler defines a family of parametric models and develops a specific experimental setup centered around this specific family. In that case, modeling with the aid of a graphical interface, or developing a set of ad-hoc scripts is convenient. In other situations, and especially in other fields of science, "solving" a Markov chain is just a subroutine in a much larger process involving statistical estimation, learning, decisions, etc. In those situations, providing solution methods through an API is a requirement of the end user.

At a second level, the environment should also be useful to *Markov developers* who are working on advanced solution methods, be they generic or adapted to specific classes of Markov models. `marmoteCore` will then be used as a library of algorithms as well as benchmark cases, and will serve as an experimental platform for comparing the performance of methods. It would then be seen as some middleware, providing a communication between some generic scientific software or some experiment management environment (e.g. a WMS) and solution algorithms/programs.

In coping with these simultaneous objectives, the main challenge is to produce a framework as *open* and flexible as possible, in order to welcome software contributed by external parties, while at the same time allowing the general user to use their favorite modeling language/software.

2.3 Architectural choices

2.3.1 Object-oriented organization. The objectives of the project make the choice of object-oriented programming almost obvious. Object-oriented languages give the possibility of designing high-level abstractions for mathematical concepts, representing objects with common properties. Yet, through the mechanism of inheritance, the user has the flexibility of controlling the implementation of specific instances of the model. The organization of `marmoteCore` relies a lot on hierarchies of models, and polymorphism. We illustrate the power of this idea in Section 3.

We have chosen the `C++` language, partly to ease up the integration of legacy code available in `C` or `C++`. The first phase of the development, presented here, consisted in the development of a solid code base demonstrating the potentialities of the architecture. This choice is consistent with the longer-term objectives of the project: in a second phase, we will make the library available as packages, plugins or libraries to other languages/ platforms. The most popular ones (R, Python, Sage, ...) indeed all provide a way to incorporate functions written in `C++`.

2.3.2 Separation between the core and packages through wrapping. The idea that `marmoteCore` should provide a core of objects and functionalities, and give access to more functionalities through wrapping of packages, comes from the following factors. On the one hand, there exists multiple incompatible data formats for models (typically, the matrix representation of Markov generator) and results of analyses (distributions, trajectories, etc.). On the other hand, there exists a wealth of software programmed in a variety of languages, and it is out of question to re-program all these applications.

A solution is to provide a small number of high-level abstractions that are common to any Markov model, and group them into a "core" which can be seen as an exchange hub. The core should be able to handle Markov models in various formats and call solution methods from different sources. This is done using three principal paradigms: (1) native programming within the core, using `marmoteCore` objects; (2) reusing methods already programmed, typically `C` or `C++`. Methods of the `XBORNE` package are available this way, see Section 5.3; (3) wrapping of *external calls* to separate applications. This usually consists in a) generating a file suitable for some external application; b) execute this application through a "system call", directing the output to some other file; c) parsing of results into `marmoteCore`. We explain in Section 5 how these techniques are used concretely.

3 THE `MARMOTECORE` INTERFACE

The interface of `marmoteCore` is based on only four principal abstractions: `MarkovChain`, `TransitionStructure`, `MarmoteSet` and `Distribution`. These high-level classes are intended to be specialized for specific purposes. We present now their principal features: attributes and functionality, and give examples of this specialization.

3.1 Class `MarkovChain`

Obviously, the class `MarkovChain` is the central one for a Markov modeling software. Technically however, as it appears from Figure 1, it is just a container for the descriptive elements: a state space, a transition structure and an initial distribution. Those are described in the following sections.

```
timeType type_;
MarmoteSet* stateSpace_;
TransitionStructure* generator_;
DiscreteDistribution* initDistribution_;
```

Figure 1: Attributes of the `MarkovChain` class

The class has a large number of methods, which are reviewed in Section 4.

3.1.1 Implementations: a hierarchy of Markov Chains. The principal novelty and full potential of the software resides in its organization in a hierarchy of concepts. Markov Chain and Transition Structure objects in particular can be organized from the most specific to the most generic.

A target hierarchy for Markov chains in the `marmoteCore` environment is displayed in Figure 2 (for continuous-time chains: a similar hierarchy exists for discrete-time ones). It illustrates the project to populate the environment with

- standard generic models from the theory, such as
  - Markovian counting processes: Poisson processes, Interrupted Poisson processes (IPP), Markov-Modulated Poisson processes (MMPP), Markov Additive Processes, Galton-Watson models, Urn models;
  - Standard Markov chains: two-state chains, random walks (birth-death in continuous time) in one and many dimensions, with constant or state-dependent rates, etc.;
- classes of models with recognized structure such as "Quasi-birth death" or "Poisson Systems" [1], etc.;
- models with specific scientific applications, including:
  - nucleotide replacement models of Biology: Jukes-Cantor, Kimura, Felsenstein, Tamura-Nei, etc. [5];
  - interacting particle models of Statistical Physics, like: Asymmetric Exclusion Processes (ASEP), Contact Processes, Ising systems, etc. [9];
  - models from Stochastic Operations Research, including Markov-modulated queues, Jackson and BCMP networks, or G-Nets.

This proposal is by no means exhaustive, and can be adapted to host more "high level" modeling paradigms (Stochastic/Timed Petri Nets, Stochastic Automata Networks, Stochastic Process Algebras) as well as "low level" models with few parameters and strong structure stemming from various fields: population models, epidemics, etc.

Figure 3 illustrates the use of polymorphism, that enables to use, for some model in the hierarchy, methods that have been attached to models higher in the hierarchy. In this toy example, several ways of computing or approximating the stationary distribution are used. This example also illustrates that the same functionality can (and should) be offered with several implementations.

3.2 Class `MarmoteSet`

The `MarmoteSet` class implements discrete sets of states, with elementary objects such as integer intervals, and elementary constructions such as Cartesian products and unions.

3.2.1 Functionalities. The principal methods of `MarmoteSet` objects are represented in Figure 4.

Methods `index()` and `decodeState()` provide the two-way maps from possibly complex state spaces, represented as vectors of numbers, to a linear numbering of states. They are necessary since numerical methods usually involve the manipulation of arrays with a standard numbering of entries. In some sense, they are also sufficient to implement functionalities using state spaces.

Methods `firstState()` and `nextState()` respectively set some state vector to the conventional "initial" state of the set, and transform this state vector into the following state in the conventional order. Together with `isZero()` which tests equality of a state with the initial state, they are enough to enumerate all states of some set, as shown in Figure 7.

3.2.2 Implementation. Currently implemented sets include `MarmoteInterval`, the standard integer interval, `MarmoteBox`, its multidimensional extension, `BinarySequences`, the representation of $\{0,1\}^N$ suitable for many models in computer science and statistical physics, and more complex geometries such as `Simplex` and `BinarySimplex`.

3.3 Class `TransitionStructure`

The `TransitionStructure` class is an abstraction for the labeled state-to-state transitions that are common to discrete-time and continuous-time Markov chains. The mathematical representation of this concept is the matrix, with the usual convention that origin states are rows and destination states are columns.

3.3.1 Functionalities. The principal methods/functionalities for this object are listed in Figure 5. There are basic methods for constructing the structure: setting and accessing the entries. A somewhat advanced access to the structure is through method `getTransDistrib()` which returns a probability distribution over the destination states. It is particularly useful for Monte-Carlo simulation.

Methods `uniformize()` and `embed()` provide the standard ways for passing from continuous to discrete time.

Methods `evaluateMeasure()` and `evaluateValue()` represent respectively the action of the operator on row vectors (usually, probability measures) and column vectors (usually, rewards or weights of some sort, attached to states). These correspond to left- and right- vector/matrix multiplications in the matrix algebraic representation.

3.3.2 Implementations. This class has currently several implementations. The one that is dedicated to numerical computation, uses a sparse matrix structure and consistently is named `SparseMatrix`.

Other implementations are *not* based on the comprehensive storage, for instance `MultiDimHomTransition` that represents the multi-dimensional birth/death transitions with constant rates or probabilities. In the code excerpt in Figure 6 (for the 1-D case), no bounds on state numbers appear: it is possible to represent this way processes on infinite state spaces. Some analysis methods, e.g. simulation, can work

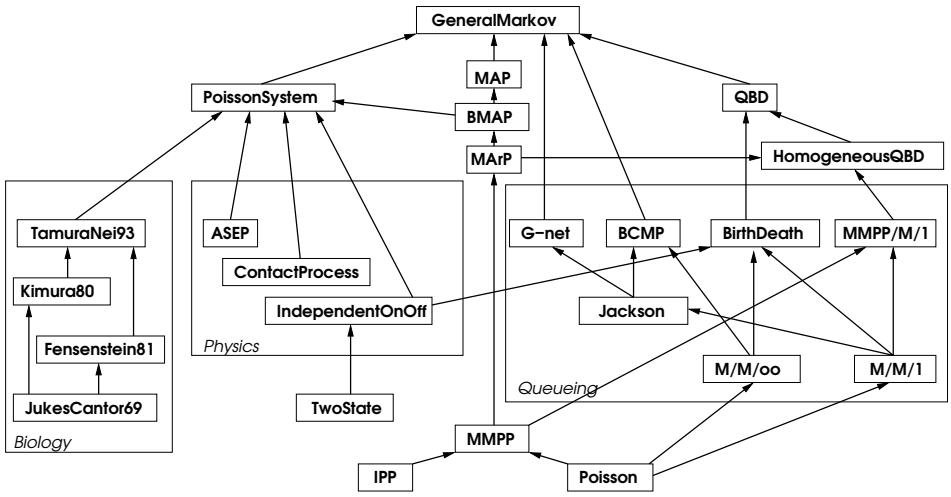

Figure 2: Hierarchy of continuous-time Markov chains

```
// methods specific to the MC under study
Felsenstein81* c1 = new Felsenstein81(...);
Distribution* d[8];
d[1] = c1->stationaryDistribution();
d[2] = c1->simulateChain(...)-->getDistribution();
// generic methods for MCs, called on same object
MarkovChain* c2 = static_cast<MarkovChain*>(c1);
d[3] = c2->stationaryDistributionSOR();
d[4] = c2->stationaryDistributionGthLD();
d[5] = c2->replicateSamples_Psi3(...);
d[6] = c2->simulateChain(...)-->getDistribution();
```

Figure 3: Illustration of polymorphism for the call to solution methods

```
// state-index conversions
void decodeState(int index, int* buffer);
int index(int* buffer);
// state space exploration
void firstState(int* buffer);
void nextState(int* buffer);
bool isZero(int* buffer);
```

Figure 4: Minimal methods of the `MarmoteSet` class

with these objects. The "block transition" class, suitable for modeling QBDs, currently being implemented, has also this feature.

The class `EventMixture` is adapted to "event modeling", where elementary events, occuring at constant rates/probabilities provoke elementary transitions (births, deaths, movements, etc.). These are the basic elements for building many complex models, in the view of Poisson Systems. They can also be used to construct Markov decision models or Markov games.

```
timeType getType()
// entry manipulation
double getEntry(int,int);
double setEntry(int,int);
double addToEntry(int,int);
// exploitation of the transition structure
DiscreteDistribution* getTransDistrib(int);
// transformation of transition structures
TransitionStructure* uniformize();
TransitionStructure* embed();
// transition structure as an operator
void evaluateMeasure(double*,double*);
void evaluateValue(double*,double*);
```

Figure 5: Methods of a `TransitionStructure`

```
double BirthDeath::getEntry( int i, int j ) {
   if ( i == j ) return r_;
   else if ( i == j + 1 ) return p_;
   else if ( i == j - 1 ) return q_;
   else return 0.0;
};
```

Figure 6: Implementation of an unbounded birth-death transition structure

3.4 Class `Distribution`

The `Distribution` class implements classical attributes for probability distributions, including the calculation of moments, Laplace transforms, and sampling of pseudo-random values. The specific objects implemented in `marmoteCore` include the ubiquitous Dirac, Bernoulli, Geometric, Poisson and Exponential distributions. The class `DiscreteDistribution` is adequate for generic distributions on state spaces.

4 ANALYSIS METHODS

We survey in this section some analysis and "solution" methods presently available in marmoteCore. Those are originally present in existing software packages: the presentation also serves for illustrating the way existing Markov software can be integrated into the platform.

4.1 Structural Analysis

Structural analysis methods are related to the communication structure in the transition graph. They include absorbingStates(), recurrent/communicatingClasses(), which compute state space decompositions into corresponding classes. Their results are used by isirreducible() and isaccessible(i,j), as well as period(). The method subChains() extracts the set of independent MarkovChain objects corresponding to recurrent classes of some Markov chain.

One implementation of these methods uses algorithms of [8] (see also [13]) suitably modified to be parallelized on multi-core machines when available. Another implementation uses the R markovchain package [12]. See also Section 5.1. This illustrates again the possibility of having the same functionality implemented with different algorithms.

4.2 Monte-Carlo simulation

The standard forward Monte-Carlo simulation is available with the simulateChain() method. The method includes several controls for the printing of the trajectory and the collection of empirical occupancy measures. Perfect sampling of Markov chains using the method of Propp and Wilson [10] is also available. This method is imported from the PSI package [11], see also Section 5.2.

4.3 Steady-state distributions

One of the aims of the marmoteCore project is to give access to several methods for solving the same problem, using a relatively common interface. This is the case for the computation of (approximate) steady-state distributions for Markov chains, the basic problem of the field.

In the current version, five methods are available for the top-level class MarkovChain. One is the standard power method. Two provide an interface to the linear-algebraic methods GTH and SOR (see Stewart [13]): these are borrowed from the XBORNE package [4], see Section 5.3. The last one is an interface to the functionality offered by the R package markovchain already mentioned [12].

In addition, the hierarchical organization of Markov models allows to derive specific methods for models with well-known structure. This is the case for classes TwoState, Homogeneous1DBirthDeath (aka $M/M/1/K$), Homogeneous1DRandomWalk, Felsenstein81, representing (a partial list of) models for which the steady-state distribution can be found in closed form. Abstraction allows to handle some infinite-state models: for the infinite-state random walk and birth-death, an object of type GeometricDistribution is returned by stationaryDistribution().

```
SparseMatrix* makeGenerator( adHocStateSpace* sp, ... )
{
   SparseMatrix* gen=new SparseMatrix(sp->cardinal());

   int stateBuffer[5]; // this state space has 5 dims
   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->isZero(stateBuffer));
}
```

Figure 7: Generic way of constructing an infinitesimal generator

4.4 Transient distributions

Computing transient distributions for general Markov models is a highly technical task and marmoteCore applies its "reuse" philosophy to delegate it to specialized tools. Currently, the transientDistribution() method of the top-level MarkovChain uses the package described in [2].

Similarly to stationary distributions, closed form solutions that exist can be implemented at lower levels of the hierarchy. This is the case for Homogeneous1DRandomWalk, Homogeneous1DBirthDeath, Poisson, TwoState, Felsenstein81.

4.5 Hitting times

A method is provided to obtain (approximate) *average* hitting times of an arbitrary subset of the state space, starting from every initial state. Hitting time *distributions* can be sampled using Monte-Carlo simulations. The exact value for this distribution is returned for some elementary models, *e.g.* currently: TwoState, Poisson, Felsenstein81.

5 INTEGRATION WITH EXISTING SCIENTIFIC SOFTWARE

In this section, we explain how marmoteCore can (and does) interface with existing software systems. Indeed, it is essential to the project that existing methods of analysis (solution algorithms, data assimilation and model construction, e.g. [7], statistical analysis, ...) be accessible easily through the programming interface. We explain through several examples that this idea works well. The functionalities of marmoteCore are limited only by the time needed to realize the interfaces, and the accessibility of the software to be integrated.

5.1 R

The case of R is particularly interesting because there exist two ways of interacting with it. One first way is through wrapping (see Section 2.3), as for any external program. A second, more intimate way, is to create an instance of R's execution engine inside the program, using the `Rinside` library. Instructions are passed to this engine, and results recovered as `C++` objects. We have used this technique to wrap some functionalities of the `markovchain` package [12] and the transient probabilities solver [2]. The method `sample` of the class `PoissonDistribution` also uses R: a validation that the non-trivial statistical methods need not be reimplemented.

5.2 PSI

PSI, the **P**erfect **SI**mulator, is a unix tool that provides a simulation kernel for continuous- or discrete-time Markov chains, based on backwards coupling [11]. For `marmoteCore`, this represents the example of wrapping the call of external applications. Markov matrices are written in the MARCA format [13] suitable for executing the `psi_sample` application that generates exact samples of the stationary distribution.

5.3 XBORNE

XBORNE is a set of independent programs, written in `C`, for building and transforming stochastic matrices, and computing bounds or exact results for their steady-state and transient distributions [4]. For `marmoteCore`, this is an example of direct reuse of `C` code. Courtesy of the package's maintainers, applications from XBORNE have been slightly modified so as to provide their functionality through function calls.[3] Communication of the Markov model is done through files, although in principle the translation of data structures could be performed. The methods SOR and GTH for computing stationary distributions [13] are included this way in `marmoteCore`.

5.4 Workflow Management Systems

`marmoteCore` will ultimately offer a large variety of analysis tools. The effective use of those features within a complex experiment will probably be easier by interfacing `marmoteCore` with a Workflow Management System (WMS).

We have experimented this possibility with a well-known WMS: Kepler. The Kepler Project[4] is an open source workflow application, designed to help scientists, for designing, executing, reusing, evolving, archiving, and sharing scientific workflows. Kepler is a java-based application: it was necessary to wrap our `C++` code into it. `FC2K` (Fortran/C/C++ to Kepler) is a tool which offer this functionality by generating a Kepler actor that executes a given `C/C++` function or Fortran routine via JNI (java native interface).

[3] Technically: using `extern "C"` declarations.
[4] https://kepler-project.org/.

6 CONCLUSION AND FUTURE WORK

We have presented the general orientations of the `marmoteCore` software project. These features are implemented in the current release of the core, and are fully operational: `marmoteCore` has been used to realize numerical and simulation experiments in [3, 14]. The current development plans include the development of QBD-like formalisms, interfacing available QBD software. The extension of modeling capabilities towards Markov Reward /Accumulation processes (MMPP, MAP, MArP, aka Markov Arrival Processes, of Figure 2), controlled Markov chains and Markov games is also foreseen. Finally, we will start the second phase of the project with the development of a Python interface.

REFERENCES

[1] P. Brémaud. *Markov chains: Gibbs fields, Monte Carlo simulation, and queues*. Vol. 31. Springer Science & Business Media, 1999.
[2] L. Cerdà-Alabern. Closed form transient solution of continuous time markov chains through uniformization. *In proc. 7th International Conference on Performance Evaluation Methodologies and Tools*, Valuetools, Torino, 2013.
[3] I. Dimitriou, S. Alouf and A. Jean-Marie. A Markovian queueing system for modeling a smart green base station. In: *Proc. EPEW 2015, 12th Europ. Works. on Perf. Engineering*, Madrid, 2015.
[4] J.-M. Fourneau, Y. Ait El Mahjoub, F. Quessette and D. Vekris. XBorne 2016: A Brief Introduction, In: *Proc. 31st Intl. Symp. Computer and Information Sciences, ISCIS 2016*, Kraków, pp. 134–141, 2016.
[5] N. Galtier, O. Gascuel and A. Jean-Marie, Introduction to Markov Models in Molecular Evolution. In: *Statistical Methods in Molecular Evolution*, R. Nielsen (ed.), Springer, 2005.
[6] M. Hlynka. List of queueing theory software. Technical report, Univ. Windsor, http://web2.uwindsor.ca/math/hlynka/qsoft.html, dated March 10, 2017.
[7] A.S. Horváth and M.S. Telek. PhFit: A General Phase-Type Fitting Tool, *Computer Performance Evaluation: Modelling Techniques and Tools*, LNCS #2324, p. 82, 2002.
[8] J. P. Jarvis and D. R. Shier. Graph-theoretic analysis of finite Markov chains. *In Applied Mathematical Modeling: A Multidisciplinary Approach*, CRC Press, 1996.
[9] T.M. Liggett. *Stochastic interacting systems: contact, voter and exclusion processes*. Wiley, 1999.
[10] J.-G. Propp and D.B. Wilson. Exact Sampling with Coupled Markov Chains and Applications to Statistical Mechanics, *Random Structures and Algorithms*, 9(1-2), pp. 223–252, 1996.
[11] PSI, Perfect Simulator. Software project http://psi.gforge.inria.fr/dokuwiki/doku.php, accessed 27/07/2017.
[12] G. A. Spedicato, T. S. Kang, S. B. Yalamanchi and D. Yadav. The markovchain Package: A Package for Easily Handling Discrete Markov Chains in R, https://cran.r-project.org/web/packages/markovchain/vignettes/an_introduction_to_markovchain_package.pdf
[13] W.J. Stewart. *Introduction to the numerical Solution of Markov Chains*. Princeton University Press, New Jersey, 1995.
[14] A. Vallet, L. Chusseau, F. Phillippe and A. Jean-Marie. Semiconductor laser Markov models in the micro-canonical, canonical and grand-canonical ensembles. *SigmaPhi 2017, Intl. Conf. Statistical Physics*, Corfu, 2017.