Laboratoire ID

PERFECT SIMULATOR

Ψ 1.0 - User Manual

The **PSI** team

Corine Marchand, Florent Morata, Jean Marc Vincent .

December 15, 2002

Projet IMAG - DECORE

Abstract

P.S.I. is a software package which permit to simulate stationary distribution or directly cost function of large Markov chains.
The simulation kernel is based on the *Coupling From The Past* algorithm of Propp & Wilson and uses the *Aliasing* technique to simulate transitions of the model.

Key Features

* Permits to obtain a coding of Markov chains.
* Accepts discrete and continuous time models.
* Knowledge of stopping criteria simulation.
* Provides perfect sampling.
* Possibility to directly simulate functional of stationary distribution.
* Includes several numerical tools[1] (statistics and GnuPlot graphics).

Application Area

* Computer performance modelling.
* Performance evaluation.
* Markov chains simulation.

Key words: Markov chains, CFTP, Walker's algorithm, Aliasing.

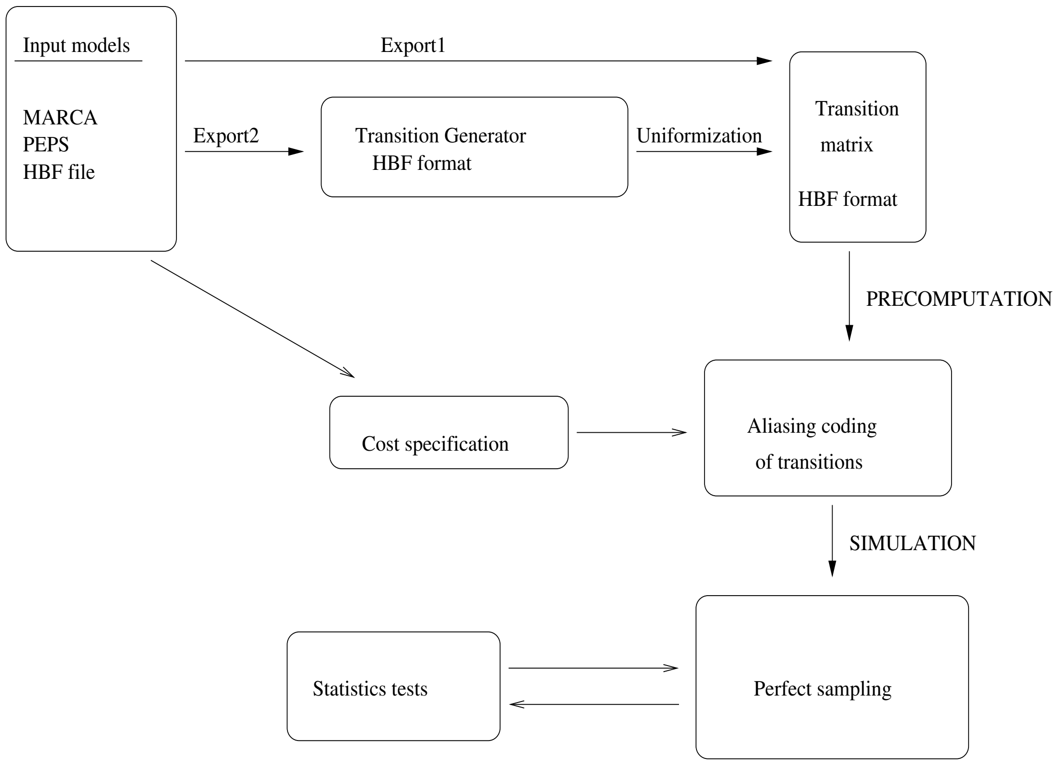
[1]Only with interface textual version.

Contents

Chapter 1

Introduction

1.1 PSI presentation



PSI is able to :

* Read different input formats : HBF (classical and PEPS[1] use), MARCA[2] .
* Uniformizate continuous time model.
* Code any Markov chain transitions in terms of **Aliasing** data.
* Comparate **backward** simulation and **MCMC** simulation[3] .
* Output results directly exploiting with GnuPlot or any Unix tools.

1.2 HBF format

Harwell Boeing Form [Saa91] is a format which facilites the storage of sparse matrix. Instead of store all the values, the idea is to only store the non-zero values.
So, to have a efficient data storage, we must be able to provide three vectors, admitting two delimiters : the number of non-zero values (*'nz_number'*) and the number of rows and columns of the matrix (*'dimension'*)

- *Val* : array of reals of size *'nz_number'*, containing the non-zero values.
- *Col* : array of integers of size *'nz_number'*, containing the column indexes.
- *Row* : array of integers of size *'dimension'* +1, containing the beginning index of each row.

Example 1 (HBF format)

$$\mathbf{M} = \begin{pmatrix} 10 & -4 & 3 & 6 \\ 5 & 2 & 1 & 0 \\ 0 & 3 & 0 & -2 \\ 0 & -4 & 0 & 0 \end{pmatrix}$$

$\mathbf{M}$ *matrix can be stored by :*

Val	10	3	-4	6	1	2	5	3	-2	-4
Col	0	2	1	3	2	1	0	1	3	1

Row	0	4	7	9

Note that the order of the non-zero elements of a row is not important. Moreover, ***PSI*** *accepts HBF format with 'column' storage (PEPS models by example) and indifferently states numerotation from* 0 *to* $dimension - 1$ *and from* 1 *to* $dimension$*.*

[1] *Performance Evaluation of Parallel Systems.*
[2] *Markov Chains Analyzer.*
[3] available with textual interface version.

1.3 Uniformization

PSI is able to simulate continuous time model, but to realize this operation, we need to transform the *generator* matrix $\mathbf{Q}$ into an equivalent *transition* matrix $\mathbf{P}$.
That is $\mathbf{P} = I + \frac{1}{\Lambda_{max}}\mathbf{Q}$, where Λ_{max} is the maximum of the absolute values of diagonal elements of $\mathbf{Q}$. This transformation [Ste94] permit to obtain a discrete time Markov chain verifying :

$$(\pi\mathbf{Q} = 0 \Leftrightarrow \pi\mathbf{P} = \pi)$$

1.4 Aliasing construction

To use **Walker**'s algorithm, as described in [Wal74] and [Sch83], to simulate transitions of Markov chains, we must build **aliasing** arrays.
Consider a construction example, issued from [Mor02].

Example 2 (Aliasing precomputation) *Let X a random variable defined on the space states $[\![0..5]\!]$ by the probability distribution $p = (p_i)_{0 \le i \le 5}$:*

$$p = (0.1, 0.3, 0.2, 0.1, 0.2, 0.1)$$

At the start of computation, we have respectively, "low" and "high" set :

$$L = \{0, 3, 5\}$$

and

$$H = \{1, 2, 4\}$$

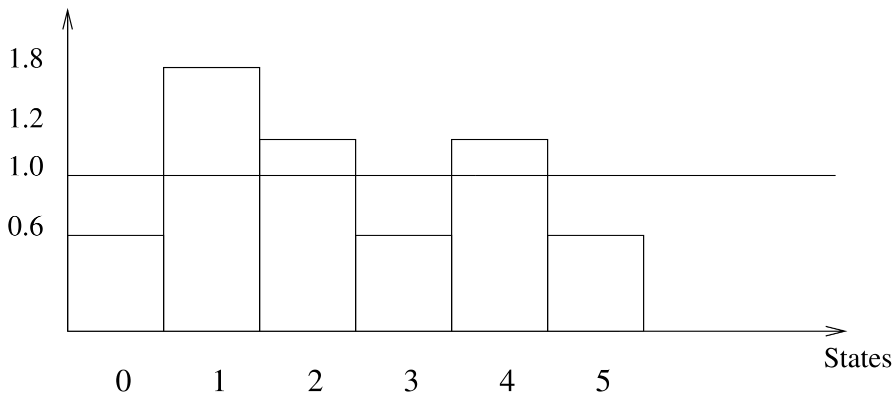


Figure 1.1: Aliasing construction : initial step.

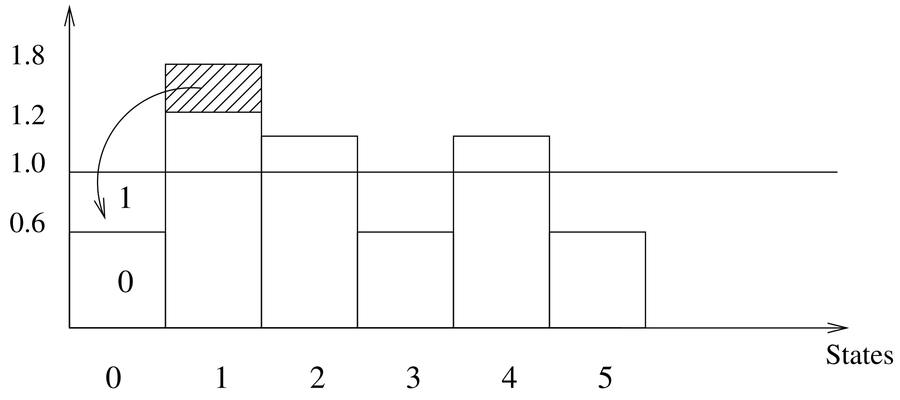


Figure 1.2: Aliasing construction : first step.

$$R[1] = R[1] + R[0] - 1 = 1.8 + 0.6 - 1 = 1.4$$
$$A[0] = 1$$

$L = \{0, 3, 5\}$ *et* $H = \{1, 2, 4\}$.

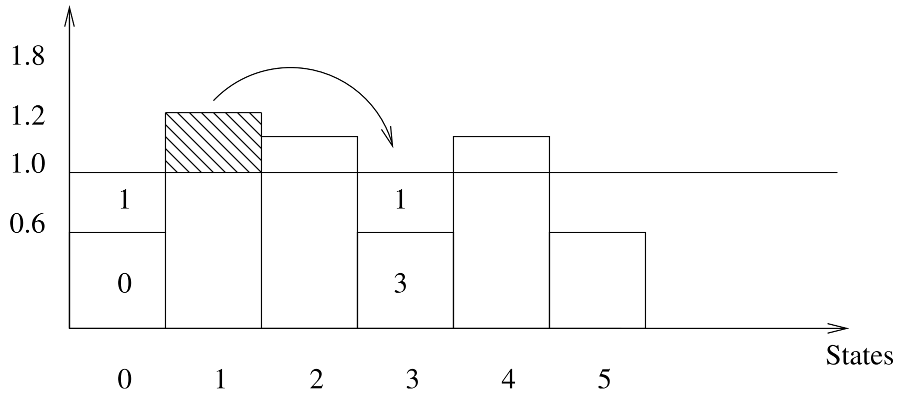


Figure 1.3: Aliasing construction : second step.

$$R[1] = R[1] + R[3] - 1 = 1.0$$
$$A[3] = 1 \to L = \{0, 1, 3, 5\},\ H = \{2, 4\}.$$

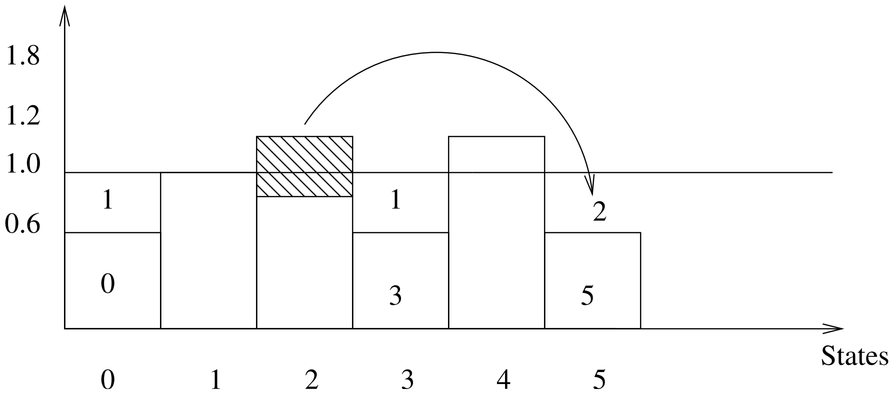


Figure 1.4: Aliasing construction : third step.

$$R[2] = R[2] + R[5] - 1 = 0.8$$
$$A[5] = 2$$

$L = \{0, 1, 2, 3, 5\}$ *et* $H = \{4\}$.

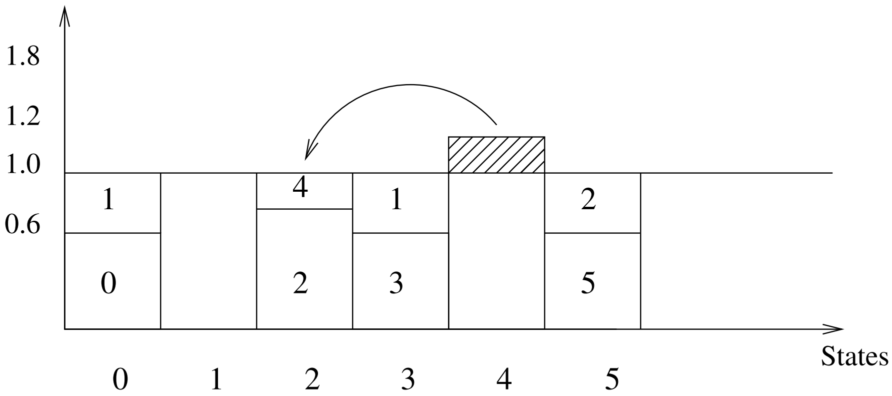


Figure 1.5: Aliasing construction : final step.

$$R[4] = R[4] + R[2] - 1 = 1.0$$
$$A[2] = 4$$

$L = \{0, 1, 2, 3, 4, 5\}$ *et* $H = \emptyset$

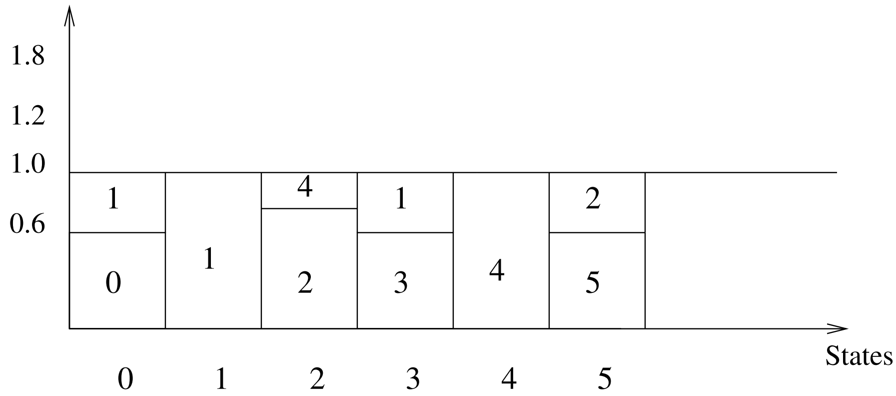


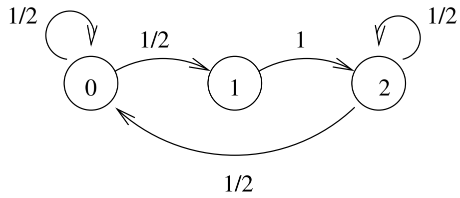
Figure 1.6: Aliasing construction : end of construction.

It follows this (non unique) **Aliasing** construction :

States	0	1	2	3	4	5
Thresholds	0.6	1.0	0.8	0.6	1.0	0.6
Alias	1	/	4	1	/	2

1.5 Perfect Simulation

The **perfect simulation** [Wil96], derived from the algorithm of *Coupling from the past*, designed by Propp & Wilson (1996), gives a method to directly generate a sample according to the stationary distribution of Markov chains.
The scheme of this algorithm is to consider all possible initial values and observe their trajectories.
But to obtain an exact sample, we must run the simulation from a distant point in the past up until the present. The stopping criteria of the simulation is therefore the coalesce time of different trajectories at time 0.



$$\mathbf{P} = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 1 \\ \frac{1}{2} & 0 & \frac{1}{2} \end{pmatrix}$$

In use, for example, the inverse of pdf, apply to each transition probability $p(x, .)$, we obtain a (non-unique) representation of the chain:

For $x = 0, 2$,

$$\Phi(x,u) = \begin{cases} 0 & \text{si } u \leq \frac{1}{2}. \\ min_x(x+1,2) & \text{sinon.} \end{cases}$$

and

$$\Phi(1,u) = 2.$$

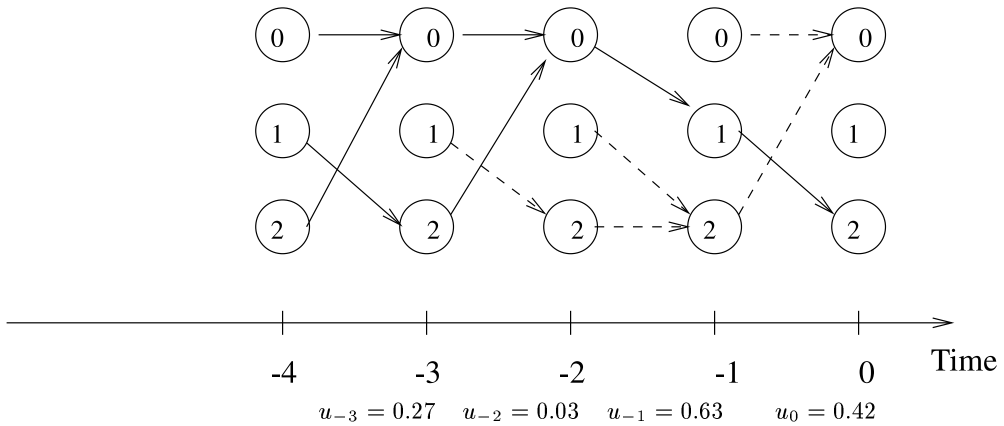


Figure 1.7: Coupling after 4 iterations.

Chapter 2

Versions

Release 1 of **PSI** version 1.0 is available in two versions :

- Unix-like version.
- Textual interface version.

One is a traditional use of Unix shell, and the other permit to dispose of an user interface.

2.1 Unix-like

2.1.1 Precomputation

'i' option of *psi_alias* executable specify the full input path of the model :

```
#  psi_alias -i ~/PSI/n3p1.marca

 File : '/users/huron/fmorata/PSI/n3p1.simu' created ...
```

A particular output path can be specified with the **'o'** option :

```
 # psi_alias -i ~/PSI/n3p1.marca -o ~/n3p1

 File : '/users/huron/fmorata/n3p1.simu' created ...
```

'h' option permit to recall help syntax :

```
#  psi_alias -h

 USAGE : ./psi_alias [-io] argument [-hv]

  -i : input model (full path)

  -o : output file (automatically generate with 'simu' extension)

       By default, output file has the same name than input file.

  -h : help

  -v : version
```

'v' option permit to know version number and internal informations of **PSI** :

```
#  psi_alias -v

 P.S.I. version 1.0 - Aliasing coding
 Authors : Florent.Morata@imag.fr
           Corine.Marchand@imag.fr
           Jean-Marc.Vincent@imag.fr
 Web : http://www-id.imag.fr
```

2.1.2 Simulation

psi_sample executable has the same syntax than *psi_alias* and permit to operate the **Perfect Simulation** with or without cost function :

```
#psi_sample -i ~/PSI/n3p1.simu -c ~/PSI/Cost_file/n3p1.cost -d 10000

 File : '/users/huron/fmorata/PSI/n3p1.sample' created ...
```

```
#  psi_sample -h

 USAGE : ./psi_sample [-idoc] argument [-hv]

  -i : input 'simu' file (full path)

  -d : sample size

  -o : output file (automatically generate with 'sample' extension)

       By default, output file has the same name than input file.

  -c : input cost file (full path)

       If -c option is not specified, cost function is the space states.

  -h : help.

  -v : version
```

```
# psi_sample -v

 P.S.I. version 1.0 - Exact sampling
 Authors : Florent.Morata@imag.fr
           Corine.Marchand@imag.fr
           Jean-Marc.Vincent@imag.fr
 Web : http://www-id.imag.fr
```

2.2 Textual Interface

When **PSI** software interface textual package is installed, by typing **psi**, the following menu is displayed.

```
      +-----------------------------------------------+
      |                                               |
      |             PSI - Perfect Simulation          |
      |                                               |
      |                    ID - IMAG                  |
      |                                               |
      +-----------------------------------------------+

 (1) Precomputation     (3) Graphical analyze      (5) About this version
 (2) Simulation         (4) Tools                  (0) Exit
```

Options (1) and (2) correspond to *psi_alias* and *psi_sample* executables of **Unix** version. However options (3) and (4) is available only with user interface version and

permit to use diferent tools, like **GnuPlot** graphics and statistics results of iteration number.
In any menu, by typing 0, you quit the current level.

2.2.1 Precomputation

Choosing *Precomputation* option leads the following :

```
#### PRECOMPUTATION ####

- Input model (full path) : /users/huron/fmorata/PSI/n3p1.hbf_r

        :-) File '/users/huron/fmorata/PSI/n3p1.simu' created ...

    :-) Aliasing construction time : 2220 micro-seconds
    :-) Reading file time : 7128 micro-seconds
    :-) Writing file time : 6592 micro-seconds
```

2.2.2 Simulation

Choosing otion (2) *Simulation* from the main menu leads :

```
#### PSI simulation ####

(1) Backward    (2) Monte Carlo  (0) Back
(3) Forward     (4) Trajectory
```

Backward

```
#### EXACT SIMULATION ####

- Input 'simu' file (full path) : /users/huron/fmorata/PSI/n3p1.simu

- Sample's size : 10000

- Output file (automatically generate with 'sample' extension) :
  /users/huron/fmorata/PSI/n3p1


  - Do you want functional simulation ? (y/n) : y

   - Input cost file (full path) :/users/huron/fmorata/PSI/Cost_file/n3p1.co

      :-) File '/users/huron/fmorata/PSI/n3p1.sample' created ...

   :-) Simulation of one realization : 7 micro-seconds
   :-) Reading aliasing file time : 0 micro-seconds
```

Monte Carlo

```
 #### MCMC SIMULATION ####

 - Input 'simu' file (full path) : /users/huron/fmorata/PSI/n3p1.simu

 - Sample's size : 10000

 - Output file (automatically generate with 'sample' extension) :
   /users/huron/fmorata/PSI/n3p1_mc


   - Do you want functional simulation ? (y/n) : n

    - Initial state number (0 .. 3) : 0

    - Steps number of simulation : 150

       :-) File '/users/huron/fmorata/PSI/n3p1_mc.sample' created ...

    :-) Simulation of one realization : 79 micro-seconds
    :-) Reading aliasing file time : 0 micro-seconds
```

Forward

```
 #### FORWARD SIMULATION ####

 - Input 'simu' file (full path) : /users/huron/fmorata/PSI/n3p1.simu

 - Sample's size : 10000

 - Output file (automatically generate with 'sample' extension) :
   /users/huron/fmorata/PSI/n3p1_for


   - Do you want functional simulation ? (y/n) : n


       :-) File '/users/huron/fmorata/PSI/n3p1_for.sample' created ...

    :-) Simulation of one realization : 12 micro-seconds
    :-) Reading aliasing file time : 0 micro-seconds
```

Trajectory

```
   #### TRAJECTORY SIMULATION ####

 - Input 'simu' file (full path) : /users/huron/fmorata/PSI/n3p1.simu

 - Output file (automatically generate with 'traj' extension) :
```

```
   /users/huron/fmorata/PSI/Models/n3p1


   - Do you want functional simulation ? (y/n) : n


   - Initial state ( 0 .. 3) : 2

   - Length of trajectory : 100


       :-) Final state : 2

       :-) See '/users/huron/fmorata/PSI/n3p1.traj' file for complete
 trajectory.
```

2.2.3 Graphical analyze

```
#### GRAPHICAL ANALYZE ####

 - Input 'sample' file (full path) : /users/huron/fmorata/PSI/n3p1.sample

  Press enter to continue ...



  - Exportation : (1) eps / (2) fig / (3) tex / (4) jpeg / (5) ps
                  / (6) pstex


  Press enter to continue ...

  - Exportation : (1) eps / (2) fig / (3) tex / (4) jpeg / (5) ps
                  / (6) pstex
```

2.2.4 Tools

```
    #### PSI tools ####

   (1) HBF facilities      (3) Aliasing coding
   (2) Results             (0) Back
```

HBF facilities

```
 #### HBF facilities ####

 (1) See model  (3) Uniformization
 (2) Conversion (0) Back
```

See model

```
#### MODEL INFORMATIONS ####

-----------------------------------------------------------

 Model '/users/huron/fmorata/PSI/n3p1.marca' in memory

 ----------------------------------------------------------

 - Space states : 4

 - Transitions number : 9
```

Conversion

```
#### HBF CONVERSION ####

 :-)  Full path of HBF 'column' name model : /users/huron/fmorata/PSI/n3p1.hbf_c

 File '/users/huron/fmorata/PSI/Models/n3p1.hbf_c' created ...
```

Uniformization

```
 #### UNIFORMIZATION ####

 - Continuous model (full path) : /users/huron/fmorata/PSI/n12p4.hbf_r

 File '/users/huron/fmorata/PSI/n12p4.unif' created ...
```

Results

```
#### Results ####

 (1) Statistics  (0) Back
 (2) Distribution
```

Statistics

```
 #### STATISTICS ####


  - Input 'sample' file (full path) : /users/huron/fmorata/PSI/n3p1.sample
---------------------------------------------------------------------------

  :-) sample size : 10000
  :-) coalesce time mean estimated : 5.6403
  :-) standard deviation estimated : 4.01636
  :-) relative error : 0.0787206
  :-) confidence interval : [5.56158;5.71902]

---------------------------------------------------------------------------
```

Distribution

```
 #### SIMULATION RESULTS ####

   - Input 'sample' file (full path) :
     /users/huron/fmorata/PSI/n3p1.sample

--------------------------------------
(0.1842 , 0.275 , 0.2674 , 0.2734 , )
--------------------------------------
```

Aliasing coding

```
#### ALIASING CODING ####

  - Dicrete model (full path)  : /users/huron/fmorata/PSI/bisto.hbf_r

  - Choice of D (>= 2) : 15

  - Do you want 'row' or 'column' arrays ? (r/c) r

  :-) File '/users/huron/fmorata/PSI/bisto.simu' created ...
```

2.2.5 About this version

```
  ----------------------------------------------

   P.S.I. - Perfect Simulation - version 1.0

 ----------------------------------------------

 PSI is a simulation software for markovian models in discrete
  and continuous time :

 Simulation kernel is based on the CFTP algorithm of Propp & Wilson and
 transitions generation is based on the Aliasing method of Walker.
  PSI permit to generate exact sampling of stationary distribution
  or directly functional of stationary distribution.

  PSI is a free sofware under GPL License and that's why you can
  redistribute it and/or modify it under the terms of the GNU
 General Public License as published by the Free Software Foundation.
              ( see : http://www.fsf.org )
```

Chapter 3

Installation

PSI is a free software under the **GPL**[1] General Public License. Any changes in the source code is under the full responsability of the user.
PSI has been writing in C++ language, and successfully testing with g++ GNU compilator in versions 2.96 and 3.2.

3.1 Getting the source files

The source files and documentation for **PSI** are available at the URL :
http://www-id.imag.fr/Logiciels/psi

3.2 Compiling

Uncompress the package[2] in the directory of your choice ; for example *'/usr/src'* :

```
# cd usr/src
# tar -zxvf /where/is/the/package.tar.gz
```

Basic Compilation

```
# ./configure   // configuration for your system
# make          // compile the package
# make install   // in root mode

# make uninstall // for uninstall psi
```

Optimized Compilation It is advised to use *optimization mode* to obtain better performances :

```
# ./configure --enable-optim
```

[1]See http://www.fsf.org, or the COPYING file of the distribution, for more details.
[2]**psi-1.0.tar.gz** or **psi_unix-1.0.tar.gz**

and especially for *i686* processors :

```
# ./configure --enable-optim686
```

3.3 RPM packages

For *RedHat*, *Mandrake* and *Suse* Linux platform, you can avoid the previous installation procedure in using a **RPM** package :

```
# rpm -i --test package // check the package without installing
# rpm -i package.rpm // for installing the package
# rpm -e package // uninstall package.rpm
```

Chapter 4

PSI Extensions

4.1 Input files

There are three possibilities to import a markovian model in **PSI** :

* **MARCA** model : *'.marca'* extension file.
* **HBF** model : *'.hbf_r'* extension file.
* **PEPS** model : *'.hbf_c'* extension file.

Note that for **MARCA** models, numeroration is from 1 to $dimension$, for **PEPS** models, from 0 to $dimension - 1$, and for **HBF** models, both numerotation can be accepted.
On the other hand, internal and output **PSI** numerotation is from 0 to $dimension$.

Model type	*Extension*	*Header values*	*Storage*	*Numerotation*
MARCA	'.marca'	columns number, rows number, *'nz_number'*	row	1 .. *'dimension'*
PEPS	'.hbf_c'	*'dimension'*, *'nz_number'*	column	0 .. *'dimension'*-1
HBF	'.hbf_r'	*'dimension'*, *'nz_number'*	row	both

Note that for markovian models, the rows number equals the columns number and its exact value is *'dimension'*.

4.2 Simu file

PSI precomputation provides an output file with *'simu'* extension. This file contains **Aliasing** data, storing in column , corresponding to the transitions of equivalent input model Markov.
Each row of file corresponds to a *threshold*, a *standard value*, and an *aliased value*.

4.3 Cost file

If user wants to directly simulate a functional of stationary distribution, this one must provide a file containing the different cost values for each states. Numerotation can indifferently begin from 0 or from 1, but output simulation results correspond **always** to a numerotation beginning from 0.

For example, this *'cost'* file corresponds to a model with 10 states (numeroted from 0) and 3 cost classes.

```
0   0
1   1
2   1
3   2
4   1
5   2
6   2
7   3
8   1
9   2
```

4.4 Results files

PSI outputs perfect sampling results in file with *'sample'* extension.
The header value is the *sample size*, and each row corresponds to the *cost value* (state number by default) and the *iteration number* to obtain it.

```
####################################################
#                 PSI 2002 (backward simulation)   #
#                                                  #
#               sample’s size                      #
#                                                  #
#         Cost + iterations                        #
###################################################!

6

2 5
3 3
2 4
3 9
1 6
0 12
```

4.5 Recapitulation

Here is the list of all **PSI** extensions :

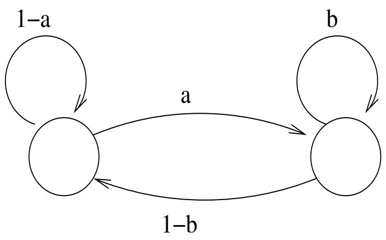
Extension	*Description*	*Type*	*Interface version*	*Unix version*
'cost'	Cost function values	input	yes	yes
'gnu'	GnuPlot script	output	yes	no
'hbf_c'	HBF 'column' model	input	yes	yes
'hbf_r'	HBF 'row' model	input	yes	yes
'marca'	MARCA model	input	yes	yes
'sample'	Perfect sampling	input/output	yes/yes	no/yes
'simu'	Aliasing data	input/output	yes/yes	yes/yes
'stat_cl'	Estimated probability	output	yes	no
'stat_time'	Estimated coalesce time	output	yes	no
'traj'	Markov chain trajectory	output	yes	no
'unif'	Uniformized continuous process	output	yes	no

Chapter 5

Examples

5.1 Discrete time

Consider the classical Markov chain with two states :



$$\mathbf{P} = \begin{pmatrix} \text{1-a} & \text{a} \\ \text{b} & \text{1-b} \end{pmatrix}$$

where $0 < a, b < 1$, and $a + b > 0$.

This stochastic matrix admit as stationary distribution :

$$\pi = (\frac{b}{a+b}, \frac{a}{a+b})$$

Take for example : $a = \frac{1}{2}$ and $b = \frac{1}{3}$, we obtain the transition matrix :

$$\mathbf{P} = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{3} & \frac{2}{3} \end{pmatrix}$$

Look three importation possibilities in **PSI** :

- **'2states.hbf_r'**.
- **'2states.hbf_c'**.
- **'2states.marca'**.

```
2 4
0.5
0.5
0.33333333
0.66666666
0 1 0 1
0 2 4
```

```
2 4
0.5
0.33333333
0.5
0.66666666
1 2 1 2
1 3 5
```

```
2 2 4
1 1 0.5
1 2 0.5
2 1 0.33333333
2 2 0.66666666
```

Note that it can be important to specify several decimal values to avoid numerical problems in Aliasing construction.

After *precomputation*, we obtain a **'2states.simu'** file (modulo row permutations) :

```
2 2

1 0 -1
0.666667 0 1
1 1 -1
1 1 -1
```

5.2 Continuous time

Consider a model of resource sharing - **Mutex1** [Fer92].
In this model, N distinguishable processes share a resource. Each process alternates between *sleeping* state and *using* state. The number of processes concurrently use is bounded by P, where $1 \leq P \leq N$.
We shall let λ_i be the rate at which proces i awakes from the sleeping state wishing to access the resource, and μ_i, the rate at which this same process releases the resource after possession of this one.

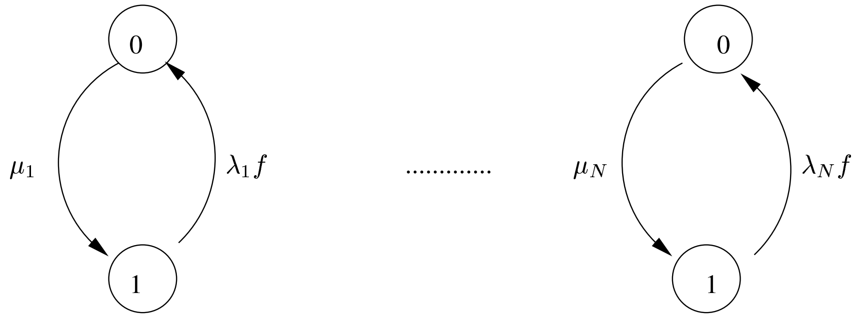


Figure 5.1: Resource Sharing Model - Mutex1

Take for example, a **Mutex1** model, with $N = 3$ processes, $P = 1$ resource, and rates values :

- $\lambda = 6$.
- $\mu = 9$.

We obtain as *generator* matrix :

$$\mathbf{Q} = \begin{pmatrix} -18 & 6 & 6 & 6 \\ 4 & -4 & 0 & 0 \\ 4 & 0 & -4 & 0 \\ 4 & 0 & 0 & -4 \end{pmatrix}$$

Note that we are :

- $\Lambda_{max} = 18$.
- dimension $= 4$.
- nz_number $= 10$.
- $\pi = (\frac{2}{11}, \frac{3}{11}, \frac{3}{11}, \frac{3}{11})$.

Consider this**'n3p1.marca'** importation file :

```
4 4 10

1 1 -18
1 2 6
1 3 6
1 4 6
2 2 -4
2 1 4
3 3 -4
3 1 4
4 4 -4
4 1 4
```

Then, **uniformization** step, gives the equivalent discrete[1] time Markov chain :

```
################################################
#                PSI 2002                      #
#                                              #
#      HBF format of the discreted process     #
#                                              #
################################################


 Non-null values of the transition matrix :

0.333333 , 0.333333 , 0.333333 , 0.777778 ,
0.222222 , 0.777778 , 0.222222 , 0.777778 ,
0.222222 ,

 Column indexes :

1 , 2 , 3 , 1 , 0 , 2 , 0 , 3 , 0 ,

 Beginning row indexes :

0 , 3 , 5 , 7 , 9 ,
```

[1]**PSI** generates automatically uniformized data and takes by default the maximum out-degree to realize *'simu'* file. This file can be generated by using *'Uniformization'* modulate.

And it follows the **Aliasing** data file :

```
4 3

1 1 -1
1 1 -1
1 2 -1
1 3 -1
1 2 -1
0.666667 0 1
0.666667 0 2
0.666667 0 3
1 3 -1
0 -1 1
0 -1 2
0 -1 3
```

Note that it is ESSENTIAL for *simulation*, to provide a *'simu'* file, where **Aliasing** data are stored by column !!!

■ You're now ready for ***Perfect Simulation***. ■

Chapter 6

Source Code

6.1 PSI Files List

Here is a list of all files with brief descriptions:

alea.cpp	Random functions for PSI
alea.h	Header file of the random library
alias.cpp	Methods of Alias class
alias.h	Header of Alias class
alias.inl	Inline methods of Alias class
analyze.cpp	Methods of Analyze class
analyze.h	Header of Analyze class
analyze.inl	Inline methods of Analyze class
cost.cpp	Methods of Cost class
cost.h	Header of Cost file
cost.inl	Inline methods of Cost file
fic_gnu.cpp	Gnuplot manipulation in C++ code
fic_gnu.h	Header of GnuPlot library
hbf.cpp	Methods of HBF class
hbf.h	Header of HBF class
hbf.inl	Inline methods of HBF class
interface.cpp	PSI User Interface
interface.h	Header file of the interface library
percent.cpp	Methods of Percent class
percent.h	Header of Percent class
percent.inl	Inline methods of Percent class
psi.cpp	Main of PSI with user interface
psi_alias.cpp	PSI Precomputation Main
psi_sample.cpp	PSI Simulation Main

For more details, see the *HTML* sources full guides[1] . Note that *Analyze* an *Percent* classes are especially use for textual interface version of **PSI**.

[1]http://www-id.imag.fr/Logiciels/psi/doc_src.html.

6.2 Alias Class Reference

Aliasing data.

`#include <alias.h>`

Public Methods

- int **size** () const

  Give the size of aliasing array.
- double **threshold** (const int i) const

  Give the threshold corresponding to the ith aliasing array index.
- int **col** (const int i) const

  Give the column index corresponding to the ith aliasing array index.
- int **alias** (const int i) const

  Give the alias index corresponding to the ith aliasing array index.
- void **init** (const int d)

  Initialization of aliasing tables.
- void **mk_walker** (const int d)

  Construction of Walker array.
- void **transpose** (const int d)

  Transposition of Walker array.
- int **read** (const string path)

  Reading of aliasing data for simulation step.
- void **write_construct** (const string file_name, const int d)

  Writing of definitive Walker array (stored by column).
- int **walk** (const double rand, const int j, const int n, const int d, const int i)

  Walker's algorithm for discrete distributions.

Private Types

- typedef **AliasStruct stock**

Private Attributes

- vector< **stock** > **S**

6.2.1 Detailed Description

Aliasing data.

Walker's algorithm is used to generate internal transitions in the simulation kernel.

6.2.2 Member Typedef Documentation

typedef struct AliasStruct Alias::stock `[private]`

Type stock to manipulate **AliasStruct** in PSI processes.

6.2.3 Member Function Documentation

int Alias::alias (const int *i*) const `[inline]`

Give the alias index corresponding to the ith aliasing array index.

int Alias::col (const int *i*) const `[inline]`

Give the column index corresponding to the ith aliasing array index.

void Alias::init (const int *d*)

Initialization of aliasing tables.

Parameters: ***d*** an integer corresponding to the matrix width.

void Alias::mk_walker (const int *d*)

Construction of Walker array.

This array is built by rows and provides a representetion of the Markov chain to simulate.

Parameters: ***d*** an integer corresponding to the matrix width.

int Alias::read (const string *path*)

Reading of aliasing data for simulation step.

Parameters: ***file*** a string corresponding to the name of the input file '.simu'.

int Alias::size () const `[inline]`

Give the size of aliasing array.

double Alias::threshold (const int *i*) const `[inline]`

Give the threshold corresponding to the ith aliasing array index.

void Alias::transpose (const int *d*)

Transposition of Walker array.

Parameters: ***d*** an integer corresponding to the matrix width. We obtain by this way a contiguous representation for the simulation step.

int Alias::walk (const double *rand*, const int *j*, const int *n*, const int *d*, const int *i*)

Walker's algorithm for discrete distributions.

Parameters: ***rand*** a double value according to uniform law on [0;1].

j an integer corresponding to a random choice of column index.

n an integer corresponding to the space state dimension.

d an integer corresponding to the matrix width.

i an integer between 0 and N-1 (where N:= matrix dimension) being the current cursor.

void Alias::write_construct (const string *file_name*, const int *d*)

Writing of definitive Walker array (stored by column).

Parameters: ***namefile*** a string corresponding to the name of the output file '.simu'.

d an integer corresponding to the matrix width.

6.2.4 Member Data Documentation

vector<stock> Alias::S `[private]`

Storage of aliasing arrays.

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

- **alias.h**
- **alias.cpp**
- **alias.inl**

6.3 Alias::AliasStruct Struct Reference

Data structure to avoid a contiguous representation of aliasing data.

Public Attributes

- **Threshold t**
- **Index col**
- **Index alias**

6.3.1 Detailed Description

Data structure to avoid a contiguous representation of aliasing data.

6.3.2 Member Data Documentation

Index Alias::AliasStruct::alias

Alias index : second value.

Index Alias::AliasStruct::col

Column index : standard value.

Threshold Alias::AliasStruct::t

Reject threshold.

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

- **alias.h**

6.4 Cost Class Reference

Simulation data for CFTP algorithm.

```
#include <cost.h>
```

Public Methods

- void **init** ()

  Initialization of the trajectories.

- void **init_state** (const int nb)

  Creation of Cout vector without functional.

- int **value** () const

  Give the first element of Class vector.

- int **value** (const int i) const

  Give the ith value of Class vector.

- int **value_next** () const

  Give the first element of Class_next vector.

- int **value_next** (const int i) const

  Give the ith element of Class_next vector.

- void **swap** ()

  Update of the trajectories after each CFTP iteration.

- bool **test** ()

  Test the global backward trajectories coupling.

- bool **test_next** ()

  Test the global forward trajectories coupling.

- void **free_cl** ()

  Memory liberation of simulation vector.

- void **change** (const int i, const int j)

  Modification of the ith Class_next element with the value of the jth element of Class vector.

- void **mk_cost** (const string path)

  Reading of a cost file.

- int **nb_cl** ()

  Give the class (according to the study functional) number.

Private Attributes

- vector< int > **Class**
- vector< int > **Class_next**
- vector< int > **Cout**

6.4.1 Detailed Description

Simulation data for CFTP algorithm.

6.4.2 Member Function Documentation

void Cost::change (const int *i*, const int *j*)

Modification of the ith Class_next element with the value of the jth element of Class vector.

This public method is used to generate next state of trajectories by the Walker aliasing algorithm.

Parameters: ***i*** an integer
j an integer

void Cost::free_cl ()

Memory liberation of simulation vector.

void Cost::init ()

Initialization of the trajectories.

void Cost::init_state (const int *nb*)

Creation of Cout vector without functional.

Parameters: ***nb*** an integer corresponding to the space state dimension.

void Cost::mk_cost (const string *path*)

Reading of a cost file.

Parameters: ***file*** a string corresponding to the cost file name.

int Cost::nb_cl ()

Give the class (according to the study functional) number.

void Cost::swap ()

Update of the trajectories after each CFTP iteration.

bool Cost::test ()

Test the global backward trajectories coupling.

Returns: true if there is coupling, false otherwise.

bool Cost::test_next ()

Test the global forward trajectories coupling.

Returns: true if there is coupling, false otherwise.

int Cost::value (const int *i*) const `[inline]`

Give the ith value of Class vector.
It be useful to MCMC simulation to estimate a functional of stationary distribution.

int Cost::value () const `[inline]`

Give the first element of Class vector.
Its public method is used to obtain the generate value of each realization.

int Cost::value_next (const int *i*) const `[inline]`

Give the ith element of Class_next vector.

int Cost::value_next () const `[inline]`

Give the first element of Class_next vector.
Its public method is used to obtain the generate value of each realization in Forward simulation.

6.4.3 Member Data Documentation

vector<int> Cost::Class `[private]`

Storage of trajectory values at time -n

vector<int> Cost::Class_next `[private]`

Storage of trajectory values at time -n+1

vector<int> Cost::Cout `[private]`

Storage of space states cost function
The documentation for this class was generated from the following files:

- **cost.h**
- **cost.cpp**
- **cost.inl**

6.5 HBF Class Reference

Storage of matrix in Harwell Boeing Format.

`#include <hbf.h>`

Public Methods

- **HBF** ()

  Empty constructor.

- void **create** (const int d, const int n)

  Full constructor with input parameters.

- ~**HBF** ()

  Destructor.

- int **size** () const

  Give the non-zero values number of matrix.

- int **order** () const

  Give the dimension of matrix.

- **psi_type type** () const

  Give the current model type.

- double **maxi** () const

  Give the largest element (in modulo) of continuous time matrix.

- bool **is_empty** () const

  Test if a current HBF structure exists.

- double **val** (const int i) const

  Give the ith non-zero value of matrix.

- int **col** (const int i) const

  Give the ith column index.

- int **row** (const int i) const

  Give the ith row beginning index of matrix.

- int **valbyrow** (const int i) const

  Give the non-null values number of the ith row.

- int **zero_or_one** ()

  Test the numerotation of input model.

- void **update** ()

  Transformation of HBF numerotation.

- void **read_hbf_r** (const **psi_name** path)

Reading of a 'row' HBF model.

- void **read_marca** (const **psi_name** path)

  Reading of a MARCA model.

- void **read_hbf_c** (const **psi_name** path)

  Reading of a PEPS model.

- void **uniformization** ()

  Uniformization of a continuous time model.

- void **write_unif** ()

  Writing of a unif file in HBF (classical) format.

- void **col2row** ()

  Conversion of a column HBF model to a row HBF model.

- void **convert** ()

  Writing of a **col2row***() conversion.*

- void **scale** (const double d)

  Multiplication of matrix by a double input value.

- void **give_a_name** (const **psi_name** file)

  Baptise HBF storage.

- **psi_name name** () const

  Give the current model's name.

- int **out_degree** () const

  Give the maximum out_degree of matrix.

- void **transform** (const int d)

  Width transformation of matrix.

Static Public Attributes

- HBF **hbf** = HBF()

Private Attributes

- **psi_name model_name**
- **psi_type model_type**
- int **dimension**
- int **nz_number**
- vector< double > **Val**
- vector< int > **Col**

- vector< int > **Row**
- vector< int > **ValbyRow**
- double **biggest**

6.5.1 Detailed Description

Storage of matrix in Harwell Boeing Format.

There exists three possibilities to import a matrix :

- hbf_r : row hbf format.(classical data structure)
- hbf_c : column hbf format. (like PEPS format)
- marca : like import file from MARCA software.

The state numerotation can start from zero or from one.

6.5.2 Constructor & Destructor Documentation

HBF::HBF ()

Empty constructor.

HBF::~HBF ()

Destructor.

6.5.3 Member Function Documentation

int HBF::col (const int *i*) const `[inline]`

Give the ith column index.

void HBF::col2row ()

Conversion of a column HBF model to a row HBF model.

void HBF::convert ()

Writing of a **col2row**() conversion.

void HBF::create (const int *d*, const int *n*)

Full constructor with input parameters.

Parameters:
- ***d*** an integer corresponding to the dimension.
- ***n*** an integer corresponding to the non-null values number.

void HBF::give_a_name (const psi_name *file*) `[inline]`

Baptise HBF storage.

Parameters:
- ***file*** an input psi_name.

bool HBF::is_empty () const `[inline]`

Test if a current HBF structure exists.

Returns: true if an HBF structure is in memory, false otherwise.

double HBF::maxi () const `[inline]`

Give the largest element (in modulo) of continuous time matrix.

psi_name HBF::name () const `[inline]`

Give the current model's name.

int HBF::order () const `[inline]`

Give the dimension of matrix.

int HBF::out_degree () const

Give the maximum out_degree of matrix.

void HBF::read_hbf_c (const psi_name *path*)

Reading of a PEPS model.

Parameters: ***path*** an input psi_name path.

void HBF::read_hbf_r (const psi_name *path*)

Reading of a 'row' HBF model.

Parameters: ***path*** an input psi_name path.

void HBF::read_marca (const psi_name *path*)

Reading of a MARCA model.

Parameters: ***path*** an input psi_name path.

int HBF::row (const int *i*) const `[inline]`

Give the ith row beginning index of matrix.

void HBF::scale (const double *d*)

Multiplication of matrix by a double input value.

int HBF::size () const `[inline]`

Give the non-zero values number of matrix.

void HBF::transform (const int *d*)

Width transformation of matrix.

Parameters: ***d*** an integer corresponding to the new matrix width.

psi_type HBF::type () const `[inline]`

Give the current model type.
PSI accepts discrete and continuous time model.

void HBF::uniformization ()

Uniformization of a continuous time model.
$\pi Q = 0 \Leftrightarrow \pi P = \pi$.

void HBF::update ()

Transformation of HBF numerotation.
PSI numerotation begin from zero and so needs a modification for models started from one.

double HBF::val (const int *i*) const `[inline]`

Give the ith non-zero value of matrix.

int HBF::valbyrow (const int *i*) const `[inline]`

Give the non-null values number of the ith row.

void HBF::write_unif (const psi_name *file*))

Writing of a unif file in HBF (classical) format.

Parameters: ***file*** a string corresponding to the output file.

int HBF::zero_or_one () `[inline]`

Test the numerotation of input model.
This method permit to the PSI user to construct his model with numerotation choice.

Returns: 0 if the input model start from zero and 1 if the input model start from one.

6.5.4 Member Data Documentation

double HBF::biggest `[private]`

Largest element (in modulo) of the generator

vector<int> HBF::Col `[private]`

Storage of column indexes

int HBF::dimension `[private]`

States number of the model

HBF HBF::hbf = HBF() `[static]`

psi_name HBF::model_name `[private]`

Model's name

psi_type HBF::model_type `[private]`

Time type model

int HBF::nz_number `[private]`

Number of non zero elements of generator matrix

vector<int> HBF::Row `[private]`

Storage of row beginning

vector<double> HBF::Val `[private]`

Storage of non zero values

vector<int> HBF::ValbyRow `[private]`

Storage of the non-zero values number by row

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

- **hbf.h**
- **hbf.cpp**
- **hbf.inl**

6.6 Analyze Class Reference

Statistics treatment of output PSI file '.sample'.

```
#include <analyze.h>
```

Public Methods

- void **open_sample** (const string path)

    reading of a sample file.

- int **elt_cl** (const int i) const

    Give the ith generate value.

- int **elt_time** (const int i) const

    Give the ith coalesce time.

- int **nb_cl** () const

    Give the class number of the generation.

- int **size_sample** () const

    Give the sample size.

- int **MaxTau** () const

    Give the largest coalesce time value.

- void **create_cl** ()

    Creation of histo_cl vector.

- void **create_time** ()

    Creation of histo_time vector.

- int **elt_histo_time** (const int i) const

    Give the apparition number of the class value i.

- string **norm_cl** (const string file)

    Normalization of histo_cl vector.

- string **norm_time** (const string file)

    Normalization of histo_time vector.

- double **pi** (const int i) const

    Give the ith estimated probability value.

- template<class T> T **sum** (vector< T > V, const int choice)

    Sum of T vector element.

- double **tau** (vector< int > V)

    Give the mean estimator.

- double **sigma** (vector< int > V)

Give the standard deviation estimator.

- double **error** (vector< int > V)

  Give the relative error.

- **ldouble int_conf** (vector< int > V, const int choice)

  Construction of confidence interval.

Private Attributes

- vector< int > **cl**
- vector< int > **t**
- int **size**
- vector< int > **histo_cl**
- vector< int > **histo_t**

6.6.1 Detailed Description

Statistics treatment of output PSI file '.sample'.

6.6.2 Member Function Documentation

void Analyze::create_cl ()

Creation of histo_cl vector.

void Analyze::create_time ()

Creation of histo_time vector.

int Analyze::elt_cl (const int *i*) const

Give the ith generate value.

int Analyze::elt_histo_time (const int *i*) const

Give the apparition number of the class value i.

int Analyze::elt_time (const int *i*) const `[inline]`

Give the ith coalesce time.

double Analyze::error (vector< int > *V*)

Give the relative error.

Parameters: ***V*** an integer vector of sample values.

ldouble Analyze::int_conf (vector$<$ int $>$ *V*, const int *choice*)

Construction of confidence interval.

The choice variable permit to choose lower or upper born of the confidence interval.

Parameters: ***V*** an integer vector of sample values.

choice an integer to specify the born.

int Analyze::MaxTau () const

Give the largest coalesce time value.

int Analyze::nb_cl () const

Give the class number of the generation.

If user simulate without functional, the class number is exactly the dimension of the space state.

string Analyze::norm_cl (const string *file*)

Normalization of histo_cl vector.

It results a file '.stat_cl'.

Parameters: ***file*** a string corresponding to the sample name.

string Analyze::norm_time (const string *file*)

Normalization of histo_time vector.

It results a file '.stat_time'.

Parameters: ***file*** a string corresponding to the sample name.

void Analyze::open_sample (const string *path*)

reading of a sample file.

Parameters: ***path*** a string corresponding to the input full path name.

double Analyze::pi (const int *i*) const

Give the ith estimated probability value.

double Analyze::sigma (vector$<$ int $>$ *V*)

Give the standard deviation estimator.

Parameters: ***V*** an integer vector of sample values.

int Analyze::size_sample () const `[inline]`

Give the sample size.

template<class T> T Analyze::sum (vector< T > *V*, const int *choice*)

Sum of T vector element.
We can calculate $\sum_{i=1}^{size} elt$, or $\sum_{i=1}^{size} elt^2$.

Parameters:
- ***V*** a T vector, where T is a generic type.
- ***choice*** an integer to specify the kind of sum to execute.

double Analyze::tau (vector< int > *V*)

Give the mean estimator.

Parameters:
- ***V*** an integer vector of sample values.

6.6.3 Member Data Documentation

vector<int> Analyze::cl `[private]`

vector<int> Analyze::histo_cl `[private]`

Count of different generate values apparition.

vector<int> Analyze::histo_t `[private]`

Count of different iteration number values apparition.

int Analyze::size `[private]`

Sample size.

vector<int> Analyze::t `[private]`

Storage of iteration number values : these values correspond to the coalesce time of an iteration.
The documentation for this class was generated from the following files:

- **analyze.h**
- **analyze.cpp**
- **analyze.inl**

6.7 Percent Class Reference

Gestion of integer percents to see the simulation progress.

```
#include <percent.h>
```

Public Methods

- **Percent** ()

  Constructor.

- ~**Percent** ()

  Destructor.

- void **create** (const int total, const int step)

  Calculate the new percent value at each iteration.

- bool **is_new** ()

  Return true if new value if different of old value, false otherwise.

- void **progress** ()

  Write the percent progression.

- void **update** ()

  Update of percent values.

- **percent give_new** ()

  Give the new percent value.

- **percent give_old** ()

  Give the old percent value.

Private Attributes

- **percent old**
- **percent nw**

6.7.1 Detailed Description

Gestion of integer percents to see the simulation progress.

6.7.2 Constructor & Destructor Documentation

Percent::Percent ()

Constructor.

Percent::~Percent ()

Destructor.

6.7.3 Member Function Documentation

void Percent::create (const int *total*, const int *step*) `[inline]`

Calculate the new percent value at each iteration.

Parameters:
- ***total*** an integer corresponding to the total number of data.
- ***step*** an integer corresponding to the current value of iteration.

percent Percent::give_new () `[inline]`

Give the new percent value.

percent Percent::give_old () `[inline]`

Give the old percent value.

bool Percent::is_new () `[inline]`

Return true if new value if different of old value, false otherwise.

void Percent::progress ()

Write the percent progression.

void Percent::update () `[inline]`

Update of percent values.

6.7.4 Member Data Documentation

percent Percent::nw `[private]`

New percent value ent

percent Percent::old `[private]`

Old percent value

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

- **percent.h**
- **percent.cpp**
- **percent.inl**

Bibliography

[Bré99] P. Brémaud. *Markov Chains Gibbs Fields, Monte Carlo Simulation and Queues*. Springer Verlag, 1999.

[Dur96] R. Durett. *Probability Theory and Examples*. Duxbury Press, 1996.

[Fer92] P. Fernandes. *Méthodes numériques pour la solution de systèmes markoviens à grand espace d'états*. PhD thesis, INPG, 1992.

[Fil98] J.A. Fill. An interruptible algorithm for perfect sampling via markov chains. *Annals of Applied Probability*, 8:131–162, 1998.

[Hag01] Olle Haggstrom. Finite markov chains and algorithmic applications. Lecture note, Course at Chalmers University of Technology, january 2001.

[Law95] G.F. Lawler. *Introduction to Stochastic Processes*. Chapman & Hall / CRC, 1995.

[Mar01] Corine Marchand. Simulation de grands modèles markoviens par technique de couplage. *Mémoire de D.E.A. I.S.C., U.J.F. Grenoble*, 2001.

[Mor02] Florent Morata. Simulation parfaite de grands modèles markoviens, implantation de la méthode d'aliasing. *Rapport de stage D.E.S.S. S.I.T.N., U.C.B.L. Lyon, (http://www.firenzo.fr.fm/Monsite/Rapport.pdf)*, 2002.

[Ros91] S.M. Ross. *A Course in Simulation*. Macmillan Publishing Company, New-York, 1991.

[Ros99] Duncan J. Murdoch & Jeffrey S. Rosenthal. Efficient use of exact samples. 1999.

[Saa91] Y. Saad. Numerical methods for large eigenvalue problems. *Manchester University Press Series in Algorithms*, 1991.

[Sch83] P. Bratley & B.L. Fox & L.E. Schrage. *A Guide to Simulation*. Springer Verlag, NY Inc., 1983.

[Ste94] W.J. Stewart. *Introduction to the Numerical analysis of Markov Chains*. Princeton, 1994.

[Ste00] O. Stenflo. Ergodic theorems for markov chains represented by iterated function systems. *Bull Polish Acad Sci Math*, 2000.

[Thö01] E. Thönnes. A primer on perfect simulation, 2001. citeseer.nj.nec.com/366900.html.

[Twe98] S. Foss & R. Tweedie. Perfect simulation and backward coupling. *Communication in Statistics - Stochastic Models*, 14:187–204, 1998.

[Vin02] Corine Marchand & Jean Marc Vincent. On the exact simulation of functionals of stationary markov chains. Soumis, 2002.

[Wal74] A.J. Walker. An efficient method for generating discrete random variables with general distributions. *ACM Trans. Math. Software*, pages 253–256, 1974.

[Wil96] J.G. Propp & D.B. Wilson. Exact sampling with coupled markov chains and applications to statistical mechanics. *Random structures and Algorithms*, 9:223–252, 1996.

[Wil97] James G. Propp & David B. Wilson. Coupling from the past : a user's guide. *Mathematics Subject Classification*, Primary 11K45, 1997.

[Win95] L.Lovàsz & P. Winkler. Exact mixing in a unknown markov chain. *The Electric Journal of Combinatorics*, # R15, 1995.