UNIVERSITÀ DEGLI STUDI DI TORINO

TANS Course A.A. 2017/2018 Prof. Massimo Masera

Ising Model Simulation with MC methods

F. Cinus & F. Delussu & N. Sella

Abstract

A computational solution for the Ising model is proposed using Metropolis-Hastings algorithm. Results show critical temperature for 2D and 3D models under periodical conditions. Different sizes and temperatures are studied in agreement with algorithm performances and sensibilities to the parameters. We plotted energy, susceptibility and magnetization vs temperature and we fitted it in order to find critical exponents.

Introduction

The Ising model is a well studied model in Statistical Mechanics which describes ferromagnetism phenomena. It is characterized by a microscopic configuration space based on D-dimensional lattice, that brings to macroscopic statistical quantities. Moreover it can easily generalize the concept of collective effects caused by binary valued points interacting in pairs; for this reason the Ising model became the core of the physics of complex systems. In the last decades computational methods have been applied to search a numerical solution for the 3D Ising model in order to fill the lack of an analytical solution. Alongside Metropolis algorithm became the most popular method of important sampling in MC. The basic idea of a weighted sampling based on the importance of a region determined a great step for the numerical solutions in general. Under these premises we want to outline the purpose this work wants to pursuit: finding numerical solution of the ND-dimensional Ising model with Metropolis algorithm.

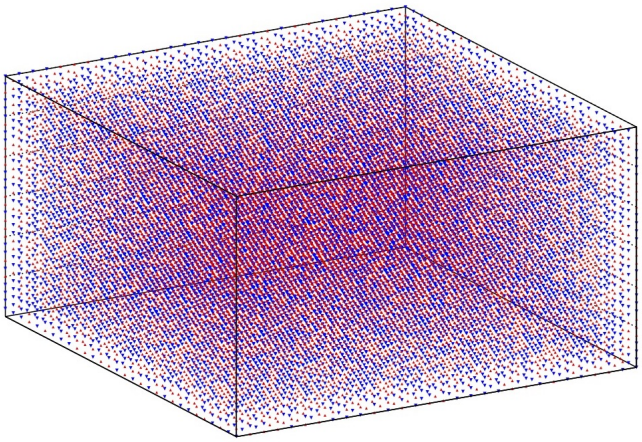

Figure 1: High temperature simulation of 3D Ising model

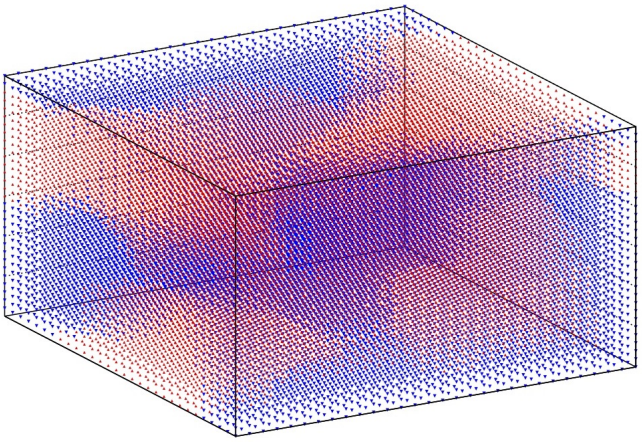

Figure 2: Low temperature simulation of 3D Ising model

1 Theory

1.1 Ising model and importance sampling

The Ising model is a physical-mathematical model characterized by a lattice of local spins in the [-1;+1] domain. The correspondent energy is proportional to the quadratic interaction of near spins (σ_i):

$$H(q,\sigma) = \sum_{<l,m>}{}' -J\sigma_i\sigma_j \tag{1}$$

Where J is the energy interaction term and we sum over the neighbours.
The system has a macroscopic equilibrium state (macrostate) described by statistical quantities: number of spins N, lattice volume V, energy E, magnetization M. According to the Statistical Mechanics theory we consider the thermodynamic limit, i.e. $N, V \to \infty$ and $N/V = cost$ in order to have negligible fluctuations on energy and magnetisation: $(\Delta E)^2/\langle E\rangle = O(1/\sqrt{N})$. From a theoretical point of view we consider an ensemble of lattices, each one with a particular configuration of spins (microstate) that corresponds to the same macrostate; we underlying that all microstates are independent and equiprobable. The configuration space is described through the Hamiltonian formalism indeed, under ergodic hypothesis, the system visits all microstates. Moreover stationary hypothesis implies the existence of an equilibrium state in which the probability distribution of the microstates satisfies the Liouville theorem. These hypotheses bring respectively to the following statements:

1. In the $t \to \infty$ limit: the average (over all the ensemble) of a physical quantity is equal to the temporal average of that quantity.

2. The probability distribution of the microstates depends only on the Hamiltonian:

$$\rho(q,\sigma) \propto \exp\{-\beta H(q,\sigma)\} \tag{2}$$

   Where q is the spin position on the lattice and β is the product of the Boltzmann constant and the temperature T of the system.

In a finite case the Ising model system is characterized by magnetization and energy fluctuations that brings us to consider it under canonical formalism. We define the partition function as follow: $Z = \sum_q exp[-\beta H(q)]$. The model simulation can be done through Monte Carlo method, i.e. we generate a pseudo-random chain of numbers in order to extract a microstate sample (from all the configuration space) distributed by the following equilibrium distribution of probability:

$$P_{eq}(q) = \exp\{-\beta H(q)\}/Z \tag{3}$$

In practice we cannot compute the partition function, moreover an algorithm that search over all the phase space has not a great performance.

The Boltzmann's factor as probability of choosing a configuration gives a great improvement for the simulation. In this way we are not sampling all the configuration space but a selected and uniform distributed microstates collection. This approach is called *importance sampling* and it allows us to calculate physical quantities as the average number over the extracted configuration:

$$\langle E \rangle_N = \frac{1}{N} \sum_{i=1}^{N} E(q_i)$$

This intuition can be proved under ergodic hypothesis: in fact we can consider an evolving Ising system in which there is a spin flip at each time step. The temporal collection of microstates is a Markov chain with a transition probability (W) that leads the system to the unique equilibrium probability in the $t \to \infty$ limit. Indeed the ratio of the transition probability is:

$$\frac{W(q \to q')}{W(q' \to q)} = \frac{P_{eq}(q')}{P_{eq}(q)} = \exp\{\beta[E(q) - E(q')]\} \tag{4}$$

Where q and q' are respectively the configuration before the spin-flip and after. This implies the equivalence between importance sampling and random extraction. The arbitrary choose on $W(q \to q')$ determines the particular algorithm; for the Metropolis-Hastings algorithm:

$$W(q \to q') = min\left\{1, \frac{P_r(q')P_{eq}(q')}{P_r(q)P_{eq}(q)}\right\} \tag{5}$$

Where $P_r(q)$ is the probability to be in the q configuration; we consider symmetric probability: $P_r(q) = P_r(q')$. In this way the configuration q' is more probable than q if $P_{eq}(q') > P_{eq}(q)$ and the step transition could occurs with the following probabilities and conditions:

$$\begin{cases} W(q \to q') = 1 & \Delta E \leq 0 \\ W(q \to q') = \exp\{-\beta \Delta E\} & \Delta E > 0 \end{cases}$$

Phase transition in Ising model

Magnetization in Ising model shows phase transition at a certain temperature called T_c. This means that the logarithm of the partition function has critical point of the first order in this point and two different statistical descriptions concurrently exist at T_c. Indeed the magnetization curve has two different fits before this point and after, that corresponds to inner configuration of spins: random that corresponds to $\langle M \rangle = 0$ and ordered with $\langle M \rangle \to +1/-1$.

2 Model

2.1 Lattice Implementation

A D-dimensional Lattice with N spins along each edge is described by

$$\mathbf{L} = (L_0, L_1, .., L_{N^D-1})$$

Let i denote the index of array $\mathbf{L}$; $i = \{0, 1, ..., N^D - 1\}$
$L_i = \{0, 1\}$ is a boolean entry resprenting down and up spin by convention
Let *i-spin* denote the spin represented by the L_i boolean entry.
The cartesian coordinates $\mathbf{a} = (a_0, a_1, .., a_{D-1})$ of the i-spin can be computed from index i if a convention is set on the Lattice arrangement, which in our Model is the following:

- Origin O of the D-dimensional space is a spin-vertex of the cubic Lattice, each edge starting from O is aligned along one of the D positive directions.
- Unitary distance between each couple of adjacent spins.
- Spins are indexed starting from increasing the first dimension's coordinate, than the second and so forth.

N.B. This rules imply that, given a specific spin, each coordinate a_j takes values in range $\{0, 1, \ldots, N-1\}$.

index i can be expressed as a power series of N with coefficients equal to the a_j coordinates :

$$i = \sum_{j=1}^{D-1} a_j N^j \tag{6}$$

The a_j coordinates can then be computed from i as :

$$a_j = [i\%N^{j+1}]/N^j \tag{7}$$

2.2 Energy Computation

The Lattice's energy is computed according to equation (1). The Lattice class provides a method wich returns the energy of the system.
The method's implemented algorithm takes the lattice array and performs two for loops, one over the system's dimension $d = (0, 1, .., D-1)$ and the other over the lattice array's index $i = (0, 1, .., N^D - 1)$.
The key idea is that, given a fixed i-spin, along each dimension d two neighbours $i_{d_\pm}$-spin are found on the increasing and decreasing d-coordinate respectively.
So each single spin has $2D$ interacting neighbours in total, by summing up their energy interaction terms the contribute of the single spin to the total energy is obtained. So the energy can be rewritten as :

$$H = \frac{1}{2} \sum_{i=0}^{N^D-1} \sum_{d=0}^{D-1} \sum_{\pm} -J\sigma_i\sigma_{i_{d_\pm}} \tag{8}$$

Performing $\sum_i$, double countings of couples occur and a factor $1/2$ is required.
For each fixed index i ,the algorithm takes into account only the interaction term with the i_{d_+}-spin so that the number of operation is halved. So the algorithm computes :

$$H = \sum_{i=0}^{N^D-1} \sum_{d=0}^{D-1} -J(L_i \ \oplus \ L_{i_{d_+}}) \tag{9}$$

$\sigma_i\sigma_{i_{d_+}}$ has been replaced by $L_i \ \oplus \ L_{i_{d_+}}$, the former can take values ± 1 wether the two spins are aligned or not. Since spins are represented by boolean entries of array $\mathbf{L}$, the XOR bitwise operator $\oplus$ is applied on couple $(L_i, L_{i_{d_+}})$ so that it returns the same value of $\sigma_i\sigma_{i_{d_+}}$.
From the i index coordinates $\mathbf{a} = (a_0, a_1, .., a_{D-1})$ the i_{d_+} index coordinates
$\mathbf{a^{d_+}} = (a_0^{d_+}, a_1^{d_+}, .., a_{D-1}^{d_+})$ can be computed.
The vector $\mathbf{a^{d_+}}$ differs from $\mathbf{a}$ only on the d deg entry $a_d^{d_+}$ which is the only one to be increased, this is computed as $(a_d + 1)\%N$ since we are applying periodic boundary conditions on the Lattice. From this formulas we can express the i_{d_+} index as the sum of two terms:

$$\begin{aligned} i_{d_+} &= \left\{ \sum_{j=0}^{d-1} a_j N^j + [(a_d + 1)\%N]N^d \right\} + \left\{ \sum_{j=d+1}^{D-1} a_j N^j \right\} \\ &= \left\{ (i + N^d)\%N^{d+1} \right\} + \left\{ (i/N^{d+1})N^{d+1} \right\} \end{aligned}$$

n.b. : if $d = D - 1$ the second term equals 0, so it's not computed by the algorithm

2.3 Metropolis Algorithm

At each step Metropolis Algorithm proposes a Lattice configuration q' which differs from the initial configuration q only by one spin.
The number of possible q' configurations equal to the total number of spins N^D , q' is chosen with uniform probability.
Defining the ratio $a = P_{eq}(q')/P_{eq}(q)$

1. If $a \geq 1$ than q' is accepted
2. If $a < 1$:

   q' is accepted with probability a

   if q' is rejected (with probability $1 - a$) configuration q is kept

That means q' is always accepted when its probability is greater than q , otherwise it is accepted with probability $P_{eq}(q')/P_{eq}$.

According to formula (3) $a = \exp\{-\beta\Delta E\}$
It' easy to observe that when $a \geq 1$ than $\Delta E \leq 0$. The latter inequality is taken as the condition of point 1.

The Algorithm's single step is implemented as following :

1. Random integer i is extracted with uniform probability from $\{0, 1, ..., N^D - 1\}$
2. The energy variation ΔE resulting from flipping the $i - spin$ is computed
3. If $\Delta E \leq 0$ than the i-spin is flipped
4. If $\Delta E > 0$:

   A random double number u is extracted with uniform probability from $[0, 1]$

   if $u < \exp\{-\beta\Delta E\}$

   i-spin is flipped

   if $u > \exp\{-\beta\Delta E\}$

   i-spin is not flipped

3 Appendix

Classes

Lattice

The Lattice class permits to construct a Lattice, compute its termodynamic quantities and change its state with the Metropolis Algorithm.
Computation details of functions such as **energy**() and **cooling**() are thoroughly explained in Model section.

PRIVATE DATA MEMBERS

const uint	**N**	number of spins along one edge
const uint	**dim**	dimension
const uint	**num_spin**	total number of spins : N^{dim}
bool *	**lattice**	boolean array of size **num_spin**
static double	**T**	temperature

PUBLIC MEMBER FUNCTIONS

CONSTRUCTORS

Lattice()

Default Constructor
Creates Lattice object with N=1 , dim=1
The single entry of **lattice** is set to 0 or 1 with 0.5 probability

Lattice(const uint& _N , const uint& _dim)

Standard Constructor
Creates Lattice object with N = _N, dim = _dim
Sets each **lattice** entry to 0 or 1 with 0.5 probability

Lattice(const Lattice& obj)

Copy Constructor
Creates Lattice object with obj's data members

DESTRUCTOR

~ **Lattice**

Frees up memory allocated by **lattice**

PHYSICAL AND NUMERICAL FUNCTIONS

bool **flipSpin**(const uint& n)

If n < **num_spin**
Sets **lattice**[n] to !**lattice**[n] and returns true
Else it doesn't change **lattice** and returns false

int **dE**(const uint& n) const

Returns energy variation resulting from applying **flipSpin**(n)
n.b. : it's a const method, it doesn't apply **flipSpin**(n)

int **energy**() const;

Returns total Energy

float **magnetization**() const

Returns Magnetization per site

void **cooling**(const uint& iter)

Applies one step of Metropolis Algorithm

void **cooling**(const uint& iter)

Applies **iter** steps of Metropolis Algorithm

double * **coolingPar**()

Applies **cooling**() and returns a four-dimensional array **arr**
arr are respectively variations of :
Temperature , Energy , Magnetization , Energy per site

OVERLOADED OPERATORS

Lattice& **operator=**(const Lattice& obj)

Assignment operator

friend std::ostream& **operator**<<(std::ostream& out, const Lattice& lat)

Taking lat as a reference to a Lattice object
Prints **lattice** member of lat by typing the command cout << lat ;

bool **operator==**(const Lattice& obj)

Taking lat1 and lat2 as references to Lattice objects
lat1 == lat2 returns true if lat1 and lat2 have same data members

GETTERS AND SETTERS

uint **getN**() const ; uint **getDim**() const; uint **getNumSpin**() const

returns respectively **N** , **dim** , **num_spin**

bool **getSpin**(const uint & n) const

returns **lattice**[n] entry

double **getT**() ; static void **setT**(const double& _T)

respectively :
Returns **T** ; Sets **T** = _T

OTHERS

void **printLatticeCSV**(const TString& name) const

prints **lattice** in a csv file saved in the working directory

void **printLatticeROOT**(const TString& name , const TString& ln = "lat") const

saves Lattice object in a root file

DrawLattice

DrawLattice class permits to draw a 2D or 3D Lattice inside a TCanvas
The Lattice object to be drawn can be imported in two ways with two Constructors
The first constructor takes a Lattice object as argument
The second constructor takes the name of a root file and the name of a Lattice saved with the **printLatticeROOT** method
The Lattice is displayed in a TMultiGraph which has two TGraphs for up and down spins, with respectively kRed and kBlue color.
The computation of the spin coordinates is explained in the Model Section.

PRIVATE DATA MEMBERS

Lattice	**lattice**	Lattice object
uint	**N**, **dim**, **num_spin**	**N**, **dim**, **num_spin** of **lattice**
TString	**fname**	name of root file
TString	**lname**	name of **Lattice** saved in root file fname
TString	**cname** , **ctitle**	Tcanvas name and title
TString	**gname** , **gtitle**	TMultiGraph name and title

cname, **ctitle**, **gname**, **gtitle** are assigned the same way in each of the three Constructors as : cname("cv"), ctitle("default canvas"), gname("gr"), gtitle("Ising")

PUBLIC MEMBER FUNCTIONS

CONSTRUCTORS

DrawLattice()

Default Constructor
Creates lattice with Lattice Default Constructor

DrawLattice(const Lattice& lat)

First Standard Constructor
Sets **lattice** = lat

DrawLattice(const TString& _fname, const TString& _lname)

Second Standard Constructor
Searches a Tfile with name _fname
Sets **lattice** with **readFile**()
void **readFile**()
Gets Lattice object with name _lname from the TFile and sets it to **lattice**

DRAW FUNCTION

void **draw**()

performs a switch statement on **dim**
if **dim** = 2 applies **draw2D**()
if **dim** = 3 applies **draw3D**()
else returns without applying a draw Function

GETTERS

uint **getN**() ; uint **getDim**() ; uint **getNumSpin**()

returns respectively **N**, **dim**, **num_spin**

PRIVATE MEMBER FUNCTIONS

DRAW FUNCTIONS

void **draw2D**()

draws a bidimensional TMultiGraph

void **draw3D**()

draw a threedimensional TMultiGraph

SETTERS

void **setN**() ; void **setDim**() ; void **setNumSpin**()

Sets **N**, **dim**, **num_spin** respectively to **lattice.N**, **lattice.dim**, **lattice.num_spin**
n.b. applied in First and Second Standard Constructor after **lattice** is set

void **setN**(_N) ; void **setDim**(_dim) ; void **setNumSpin**(_num_spin)

Sets **N**, **dim**, **num_spin** respectively to _N, _dim, _num_spin

```
/*-------------------------------------------------------------------*
 *                                                                   *
 * This macro explains how to construct handle and draw a Lattice *
 *                                                                   *
 *-------------------------------------------------------------------*/

#include "Lattice.h"
#include "DrawLattice.h"

{

  //LATTICE CLASS

  Lattice::setT(0.5); //Set static member Temperature to 0.5
                      //Default is T = 0.

  int N = 10;
  int dim = 2;
  Lattice lat(N,dim); //Construct a bi-dimensional Lattice
                      //with 10 spins along each edge

  cout << "lat visualization :" << endl;
  cout << lat << endl; //Print Lattice spins on the console
                       //The lattice spins will be disposed randomly

  cout << "energy E of lat : " << lat.energy() << endl;
  cout << "magnetization M of lat : " << lat.magnetization() << endl;

  lat.cooling(); //Perform a Metropolis step

  double * data = new double[4];
  data = lat.coolingPar(); //perform a Metropolis step collecting data

  cout << "T =  " << data[0] << " : temperature" << endl;
  cout << "dE = " << data[1] << " : energy variation" << endl;
  cout << "dM = " << data[2] << " : magnetization variation" << endl;
  cout << "dS = " << data[3] << " : energy per site variation" << endl;
  delete[] data;

  /*
   *
   * According to Metropolis algortihm if dE <0
   * there's a probability [1 - exp(-dE/T)]  (K=1)
   * that the step will not change lat state with a spinFlip
   * in that case dE=0 ; dM=0 ; dS=0;
   *
   */
```

```
54
55    const unsigned iter = 10000;
56    lat.cooling(iter); // Perform 10000 Metropolis steps
57
58    cout << lat << endl; //Print Lattice spins on the console
59                         //The lattice will now be ordered
60
61    cout << "energy E of lat : " << lat.energy() << endl;
62    // Energy will be around -200
63    cout << "magnetization M of lat : " << lat.magnetization() << endl;
64    // Magnetization per site will be about 1 or -1
65
66    /*
67     *
68     * If T is set to a value greater than the critic temperature
69     * (for 2D Ising Model Tc = 2.27)
70     * and the same Macro is executed,
71     * lat will not thermalize.
72     *
73     */
74
75    //DRAWLATTICE
76
77
78    lat = Lattice(10,2);
79
80    DrawLattice drawLat1(lat); // Create a DrawLattice object
81                               // lat is passed in order to draw it
82
83
84    cout << "lat with dim = " << lat.getDim() << endl;
85    cout << "and edge = " << lat.getN() << "will be drawn" << endl;
86
87    drawLat1.draw(); // Draws the three-dimensional Lattice lat
88
89
90    Lattice lat3D(20,3); // Construct a three-dimensional Lattice
91                         // with 20 spins along each edge
92
93    lat = lat3D;   // Use assignment operator to reassign lat
94
95
96    cout << "lat with dim = " << lat.getDim() << endl;
97    cout << "and edge = " << lat.getN() << "will be drawn" << endl;
98
99
100   DrawLattice drawLat2(lat);
101   drawLat2.draw();
102
103 }
```

Listing 1: Caption

SimulationLattice

Simulation Lattice permits to run a simulation and collect data during runtime. The simulation is performed on a set of Lattice objects over a range of temperatures
All Lattices have same edge length and same dimension For each temperature :

each Lattice is submitted to **I0** cooling iterations in order to reach thermal equilibrium.
In this step data are not collected

each Lattice is submitted to **iter** cooling iterations,
during each iteration measures of temperature, energy, magnetization and susceptibility are collected.

PRIVATE DATA MEMBERS

Lattice *	**lattice_vector**	vector storing a Lattice for each entry
const uint	**N**	edge length of each Lattice
const uint	**dim**	dimension of each Lattice
const uint	**dim_vector**	dimension of **lattice_vector**
const TString	**file**	root file name in which data are collected
uint	**I0**	number of iterations not collecting data
uint	**iter**	number of iterations collecting data
double	**tempmin**	minimum range's temperature
double	**tempmax**	maximum range's temperature
uint	**tempstep**	number of temperatures in the range

n.b. : **tempstep** should be a multiple of 4 for implementation reasons

PUBLIC MEMBER FUNCTIONS

CONSTRUCTORS

SimulationLattice()

Default Constructor

SimulationLattice(const uint& _N, const uint& _dim, const uint& _dim_vector)

3-Parameters Constructor
The other data members can be assigned with the setter methods

SimulationLattice(const uint& _N, const uint& _dim, const uint& _dim_vector, const TString& _file, const uint& _i0, const uint& _iter, const double& _tempmin, const double& _tempmax, const uint& _tempstep)

Full Parameter Constructor

SimulationLattice(const Lattice& _lat , const uint& _dim_vector, const TString& _file, const uint& _i0, const uint& _iter, const double& _tempmin, const double& _tempmax, const uint& _tempstep)

Constructor based on a Lattice
Sets **N** and **dim** as those of lat

SimulationLattice(const SimulationLattice& obj)

Copy Constructor

SimulationLattice& **operator=**(const SimulationLattice& obj)

Assignment Operator

DESTRUCTOR

$\sim$ **SimulationLattice**

Frees up memory allocated by **lattice_vector**

RUN FUNCTION

void **run**() const

Performs the simulation

GETTERS AND SETTERS

Getters methods are defined for all the Private Data Members except for **lattice_vector**, but its entries can be obtained with :

Lattice **getLattice**(const uint& i)

returns **lattice_vector**[i]

Setters methods are defined for all the Private Data Members except for **lattice_vector N**, **dim** and **dim_vector**

AnalysisLattice

AnalysisLattice performs an analysis on raw data collected on a simulation input-file
The analysis creates an output-file which stores :

mean and standard deviation of physical quantities for each Lattice and temperature of the simulation.

mean and standard deviation performed over the Lattices for each temperature of the simulation

Once the output file is created, the physical functions can be plotted into a graph
The curves of Magnetization and Susceptibility vs Temperature can be fitted in order to estimate the Critical Temperature and Exponenents of the phase transition.

PRIVATE DATA MEMBERS

const TString	**file_in**	input root file name
const TString	**file_out**	output root file name
static double	**TempCritic**	critic temperature

PUBLIC MEMBER FUNCTIONS

CONSTRUCTOR

AnalysisLattice(const TString& file_input, const TString& file_output)

Parametric Constructor
Sets input and output file names

RUN FUNCTION

void **run**()

Performs the analysis of the previous simulation.
Recreates output file

DRAW FUNCTIONS

TGraphErrors * **drawLattice**(cuint& lattice_number, cuint& x_axis, cuint& y_axis)

returns a TGraphErrors of x and y, integers representing physical qunatities
See the following macro for define statements of x and y
Usually x is TEMPERATURE while y could be ENERGY, MAGNETIZATION or SUSCEPTIBILITY
The graph represent a single Simulation's Lattice identified by lattice_number

TGraphErrors * **drawLatticeMean**(cuint& x_axis, cuint& y_axis)

Returns graph of x and y of physical quantities averaged over all Simulation's Lattices

TGraphErrors * **draw**(cuint& x_axis, cuint& y_axis)

draw a TMultiGraph containing the graphs for all Lattices obtained with the **drawLattice()** method

FIT FUNCTIONS

static double **analiticX**(double * x, double * par)

function used for fitting SUSCEPTIBILITY
has parameters : Critic Temperature , Critical Exponent and a proportional constant

static double **analiticM**(double * x, double * par)

function used for fitting MAGNETIZATION
has parameters : Critic Temperature , Critical Exponent and a proportional constant

void **findTcritic**()

sets TempCritic to the temperature for which suscpetibility is maximum
in the TGraphErrors returned by **drawLatticeMean**(TEMPERATURE,SUSCEPTIBILIT

void **fitLattice**(bool mean, cuint& x_axis, cuint& y_axis, double fit_temp_min, double fit_temp_max, int lat_number);

Performs fit on MAGNETIZATION or SUSCEPTIBILITY vs TEMPERATURE graph
respectively with fit funcion analiticM or analiticX
if mean==true
fits TGraphErrors returned by drawLatticeMean(x_axis,y_axis)
if mean==false
fits TGraphErrors returned by drawLattice(lat_number,x_axis,y _axis)
The fit is performed in [temp_min,temp_max] range

BIN FUNCTIONS

std::vector<double> **bin**(const std::vector<double>& vec)

returns std::vector with half size of vec according to binning procedure

std::vector<double> **binN**(cuint& num_bin, const std::vector<double>& vec)

returns std::vector from applying **bin** on vec num_times

TGraph * evalBinning(cuint& nb)

returns a graph representig energy standard deviation versus bin level nb

GETTERS AND SETTERS

Getters and Setters are implemented for all the private Data Members

```
/*--------------------------------------------------------------------*
 *                                                                    *
 * This macro explains how to perform a complete simulation           *
 * using th libraries.                                                *
 *                                                                    *
 *--------------------------------------------------------------------*/
#include "SimulationLattice.h" // Lattice.h included
#include "AnalysisLattice.h"   //
#include "TString.h"           // Other useful root inclusions
#include "TStopwatch.h"        //

{
  const unsigned int lattice_size(10);
  const unsigned int lattice_dimension(2);
  // this way a 10x10 lattice is created
  const unsigned int number_of_lattices(5);
  // simulation performed simultaneously on 5 lattices
  TString simulation_file("simulation_file.root");
  TString analysis_file("analysis_file.root");
  unsigned int iter_pre_simulation(1000000);
  unsigned int iter_for_simulation(500000);
  double min_temperature(0.5);
  double max_temperature(3.5);
  double steps_of_temperature(30);

  TStopwatch timer;

  SimulationLattice s(lattice_size, lattice_dimension,
                      number_of_lattices, simulation_file,
                      iter_pre_simulation, iter_for_simulation,
                      min_temperature, max_temperature,
                      steps_of_temperature);
  /*
   * Alternative way to call it:
   *
   * SimulationLattice s(lattice_size, lattice_dimension,
       number_of_lattices);
   * s.setFile(simulation_file);
   * s.setI0(iter_pre_simulation);
   * ...
   * s.run();
   */
  timer.Start();   // Simulation started with parameters set
  s.run();         // simulation_file always recreated
  timer.Stop();    //
  timer.Print();   //
```

```
53
54
55
56
57
58    AnalysisLattice a(simulation_file, analysis_file);
59    /*
60     * input and output file must be provided even
61     * if the output file already exists.
62     * The output file is recreated only by run()
63     * but is called in reading mode by the other
64     * data members
65     */
66    timer.Start();   // Analysis started: parameters got
67    a.run();         // from input file.
68    timer.Stop();    // Output file Recreated.
69    timer.Print();   //
70
71    /*
72     * Once the analysis is performed:
73     * a.draw(TEMPERATURE, MAGNETIZATION);
74     *
75     * defined name   defined name   defined value
76     *
77     * ENERGY                        1
78     * TEMPERATURE    TEMP           2
79     * MAGNETIZATION  MAG            3
80     * SITE_ENERGY    SENERGY        4
81     * SUSCEPTIBILITY SUSC           5
82     */
83
84 }
```

Listing 2: Caption

4 Code

This section's aim is to introduce the reader to the C++ implementation of the model. The code is based on TObject class of Root and its scheme is the following:

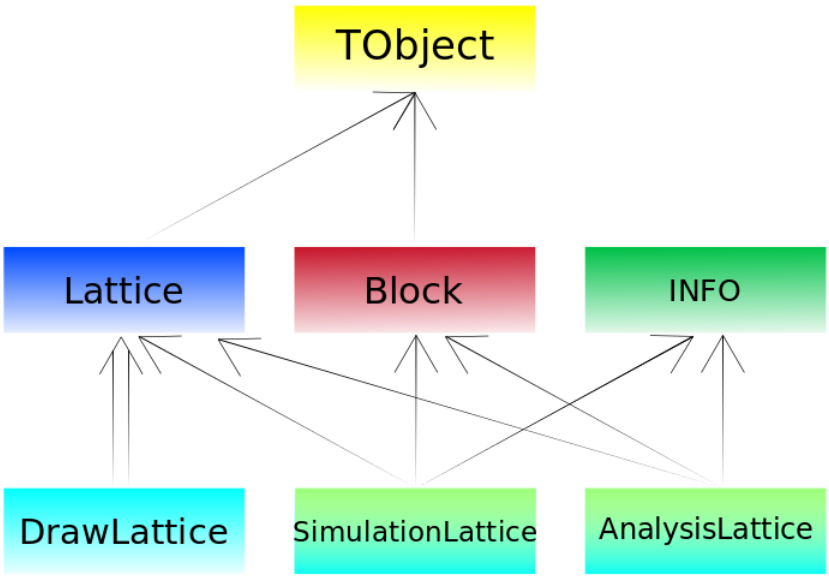

Figure 3: Class scheme

5 Results

This section wants to summarize the simulation results. Under this purpose we underline two main effects:

- Finite size and finite volume of the lattice imply magnetization and energy fluctuations; periodic conditions reduce this phenomena. Moreover finite size leads to no residual magnetization: this effect is out of the temperature's range studied.
- Importance sampling algorithms can produce correlated configurations corresponding to different steps of the simulation. This phenomena is in contrast with theoretical hypothesis of microstates independence and can be correctly treated through binning technique.

Calibrating the model

Simulations are done with 10^8 steps for thermalization (10^9 for the 3D lattice) and more than 10^7 steps with useful data. We varied the lattice size from 40x40 to 100x100 number of spins. Simulation time is really sensible to the number of temperature steps. We choose 100 steps of temperature from 0.5K to 3.5K centred around theoretical critical temperature (2.27K for 2D, 4K for 3D). (See Figure 12 in Appendix)

Figure 4: Energy-Temperature graph for 2D Lattices with I=10,000,000 and 100 steps of T

Figure 5: Magnetization-Temperature graph for 2D Lattices with I=10,000,000 and 100 steps of T

Figure 6: Susceptibility-Temperature graph for 2D Lattices with I=10,000,000 and 100 steps of T

Figure 7: 10,000,000 steps of cooling for the 2D Lattice with L=100

Figure 8: Energy-Temperature graph for 3D Lattices with I=40,000,000 and 25 steps of T

Figure 9: Magnetization-Temperature graph for 3D Lattices with I=40,000,000 and 25 steps of T

Figure 10: Susceptibility-Temperature graph for 3D Lattices with I=40,000,000 and 25 steps of T

The binning method can take into account the correlation between different configurations. Measures corresponding to different steps of the simulation can have a correct

Figure 11: 1,000,000,000 steps of cooling for the 3D Lattice with L=40

error valuation through this technique: we divide a set of N ordered configurations in $N/2^n$ groups (each one with n elements), we calculate the mean for each bin and we iterate this process until the desired level of binning; then we calculate the global mean and its error. In this way we correlate the measure at a particular level of binning (higher level $\rightarrow$ correlation length $<<$ size of the bins $\rightarrow$ lower correlation $\rightarrow$ higher error). We chose the eighth level of binning for which we found a plateau in the error-level graph.

6 Conclusions

Metropolis-Hastings algorithm has been implemented in Root (C++) in order to show phase transitions in the 2D and 3D Ising model. Simulation parameters are the following:

- number of spins for side: 40-60-80-100
- thermalization steps: 10^8-10^9
- data collection steps: $> 10^7$
- temperature measures: $25 - 100$

We recall the theoretical critical temperature of the 2D Ising model with infinite size: 2.27K. The 3D Ising model does not have an analytical solution so we present a result in harmony with the results accepted by scientific community.

Model	Critical T	Critical Exp
2D	Numb. Users	Max Delivery Cost
3D	700	1.75 €

Performance

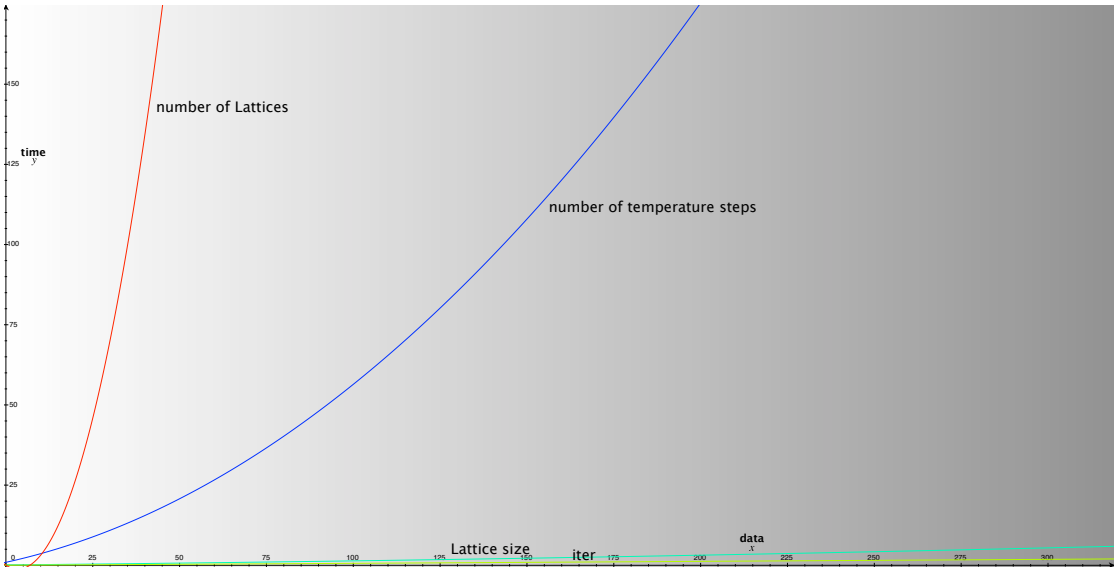

Figure 12: Time scaling for different data member initialization