Introduction to Probability Theory
Problem set #2

Student: Sergi Blanco Cuaresma

November 23, 2010

Abstract

Solutions to the problem set number two of the subject Statistics, Monte Carlo Methods and Data Processing - Master in Astrophysics, Particle Physics and Cosmology (Universitat de Barcelona).

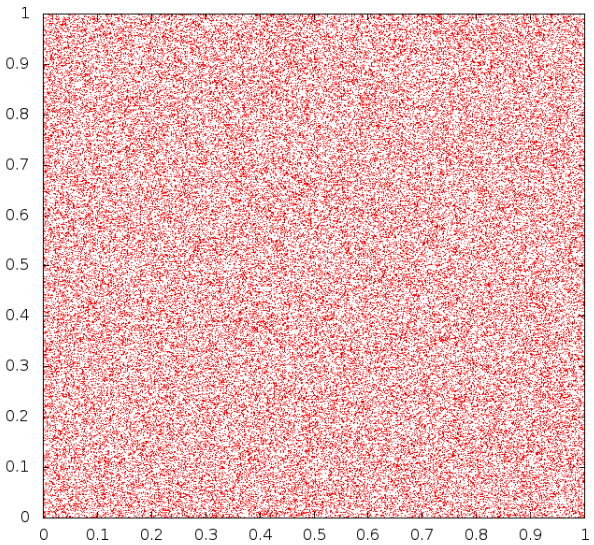
Contents

1 Random number generator

The chosen pseudo random number generator (RNG) algorithm is Mersenne twister (MT19937). Although it is not suitable for certain applications such as cryptography, it has all the properties to be a good RNG for Monte Carlo simulations and statistics (i.e. It has a very long period of $2^{19937}-1$). This algorithm is used in popular applications such as R, Maple and Matlab.

Instead of doing a new development of this RNG algorithm, an already existing open source implementation has been used (C Language). Concretely a new variant of Mersenne Twister (MT) named SIMD-oriented Fast Mersenne Twister (SFMT)[2]. It is better to use a widely proven and tested RNG for the simulations that will be develop in this problem set. Additionally, SFMT is specially designed to have a better performance with recent parallelism of modern CPUs, such as multi-stage pipelining and SIMD (e.g. 128-bit integer) instructions, which will affect positively to the simulations performance.

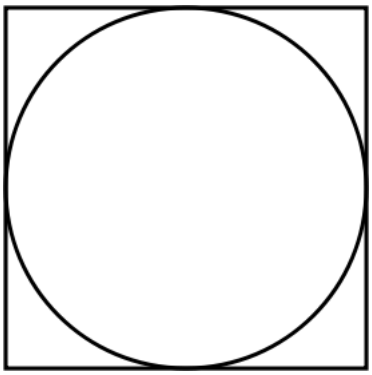
The SFMT implementation allows to select the period from a wide range of options, but it has been kept the original $2^{19937}-1$ for this problem set (specified in the compilation command line). The source code of SFMT has been integrated in the Monte Carlo simulator built for this problem set, and additionally a small test has been implemented (100.000 points generation) to visually (but not formally) validate its uniformity:



2 Approximation to π using the Metropolis algorithm

2.1 Design

In order to make an approximation to π using the Monte Carlo simulation (the Metropolis algorithm has not been used directly for this problem, no intuitive application has been found), we will consider a circle inside an square where random points will be drawn. It is known that the distance between two point is obtained with $d^2 = (x_1 - x_0)^2 + (y_1 - y_0)^2$, and if one of the points is the origin $(x_0 = 0, y_0 = 0)$ then the formula is simplified to $d^2 = (x_1)^2 + (y_1)^2$.



For each random point, it is necessary to check if it is within a circle of radius $r = 1$ and center $(0, 0)$, therefore $1 > x^2 + y^2$ must be true. If we consider that the circle is perfectly fitted into a square of length $l = 2$, and we generate a bunch of random points, the probability of a point to be inside the circle is:

$$P\left(1 > x^2 + y^2\right) = \frac{Circle\,Area}{Square\,Area} = \frac{\pi r^2}{l^2} = \frac{\pi}{4} \tag{1}$$

If we generate N random points and M of those are located inside the unit circle, the probability of that a random point lies inside the unit circle is given as:

$$P\left(1 > x^2 + y^2\right) = \frac{M}{N} \tag{2}$$

Therefore, if N becomes extremely large, π can be approximated making equal both probabilities:

$$\frac{\pi}{4} = \frac{M}{N} \Rightarrow \pi = \frac{4 \cdot M}{N} \tag{3}$$

The implemented simulation generates 10^9 random points with a uniform distribution between zero and one and counts how many of them are within the circle. Finally it computes the π approximation using the calculation presented in this section.

2.2 Compilation

The simulator can be compiled with GNU CC compiler, available for Unix/Linux systems and Windows. From the command line on the same directory where the source code is located, the following instruction will generate the "runMonteCarloPi" file:

```
$ gcc -O3 -msse2 -fno-strict-aliasing -DHAVE_SSE2=1 -lm -DDSFMT_MEXP=19937 -o runMonteCarloPi dSFMT.c mainMonteCarloPi.c
```

2.3 Execution

The execution[1] of the simulation gives the following result:

```
$ time ./runMonteCarloPi
Pi aproximation = (785392169/1000000000)*4 = 3.141569

real    0m9.661s
user    0m9.580s
sys     0m0.010s
```

2.4 Source code

```c
#include <stdio.h>
#include <stdlib.h>
#include "dSFMT.h"

int main(int argc, char* argv[]) {
    double pi; // Result
    int m = 0; // Number of points inside the circle
    const int N = 10000; // Number of random points
    double x, y; // Point coordinates

    dsfmt_t dsfmt;
    int i;

    int seed = 12345;

    // Initialize the random number generator (RNG)
    dsfmt_init_gen_rand(&dsfmt, seed);

    for (i = 0; i < N; i++) {
        // Generate a random X coordinate between [0,1)
        x = dsfmt_genrand_close_open(&dsfmt);
        x = (x*2) - 1; // Increment range to (-1, 1)
        // Generate a random X coordinate between [0,1)
        y = dsfmt_genrand_close_open(&dsfmt)*2;
        y = y - 1; // Increment range to (-1, 1)

        // If the point is inside the radius (circle)
        if (x * x + y * y < 1.0) {
```

[1] Execution has been made using "time" command, which prints the CPU time that the program has used.

```
            m++;
        }
    }

    pi = ((double)m / N) * 4;
    printf("Pi aproximation = (%i/%i)*4 = %f\n", m, N, pi);
    return 0;
}
```

3 Ising model

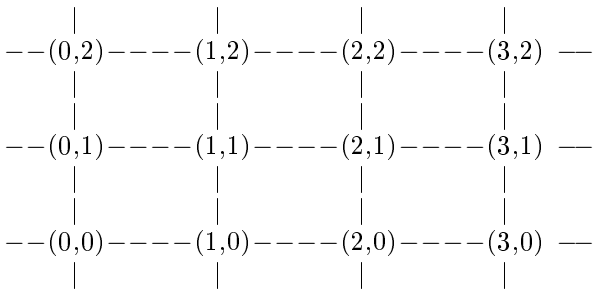
In statistical mechanics the probability theory is applied to the study of the thermodynamic behavior of systems composed of a large number of particles. Among its techniques, there is the Ising model which is a mathematical model of ferromagnetism. This is the basic mechanism by which certain materials (such as iron) form permanent magnets or are attracted to magnets.

The model consists of discrete variables called spins which can be found in one of two states ($\sigma_i = \{-1, +1\}$) and are arranged in a lattice ($N_1 x N_2$). The goal is to find phase changes in the Ising model, as a simplified form of phase changes in real substances.

The hamiltonian of this system is:

$$H = -\sum_{\langle i,j \rangle} J_{ij}\sigma_i\sigma_j - \sum_i h_i\sigma_i \tag{4}$$

Where $J_{ij}$is the magnetic coupling between spin i and j, $h_i$is the magnetic field at site i. Spins are arranged in a lattice with periodic boundary conditions:



The partition function of the system is:

$$Z(T) = \sum_{\sigma} e^{-\frac{H(\sigma)}{T}} \tag{5}$$

And the average energy:

$$\langle H \rangle = \sum_{\sigma} H(\sigma) \cdot P(\sigma) = \sum_{\sigma} H(\sigma)\, e^{-\frac{H(\sigma)}{T}} \tag{6}$$

For a 20 × 20 spin lattice interacting via the Ising model, 2400 states are presented and it is difficult to examine them all. In that sense, with Monte Carlo simulations and the Metropolis algorithm a set of states can be picked based on their probability distribution (weighting them equally).

3.1 Simulator design

In order to perform an Monte Carlo simulation (applying the metropolis algorithm) for each of the cases presented in the problem set (including the last one with random coupling constants and magnetic fields, which was not a required task), a complete parametrizable program has been develop:

1. Mersenne Twister (MT) is the used random number generator (SFMT implementation).
2. The lattice with the spins and the magnetic coupling / magnetic field are represented by three different matrices of $N_1 x N_2$.
3. A group of N samples are generated for a given temperature T, each of them starts with an initial state and evolve (transitions) driven by the metropolis algorithm and the probability that the system occupies a given state σ ($\pi_T(\sigma) \equiv \frac{e^{-\frac{H(\sigma)}{T}}}{Z(T)}$):
4. A spin is randomly chosen with a uniform distribution, and a change in its states is proposed (σ_{new}).
   (a) The energy difference of the new state relative to the present state ($\sigma_{current}$) is calculated.
   (b) A random number $0 \leq n \leq 1$ is generated (uniform distribution), and the new state is only accepted if $\frac{\pi_T(\sigma_{new})}{\pi_T(\sigma_{current})} = \frac{e^{-\frac{H(\sigma_{new})}{T}}}{e^{-\frac{H(\sigma_{current})}{T}}} = e^{-\frac{Increment\, H(\sigma)}{T}} > n$.
5. The probability of a given state σ is the number of times that this states appears over the total number of states generated (samples):

$$\lim_{N\to\infty} \frac{N(\sigma)}{N} \equiv \pi(\sigma) \tag{7}$$

6. As an input, a configuration file named "config.txt" has been created in the directory "input" in order to allow customizable executions. The file contains groups of simulations, which begin by the keyword "OUTPUT_NAME", and establishes a set of execution parameters for each of them (Monte Carlo loops, samples, lattice size, temperature ranges and magnetic constants). An illustrative example:

```
OUTPUT_NAME=GenuineCase
MONTECARLO_LOOPS=10000
SAMPLE_NUM=1000
# Lattice size
NX=12
NY=12
TEMP_MIN=0.01
TEMP_MAX=5
# Number of measurement in the interval above
TEMP_MEASUREMENTS=40
# Comment the next line if a random magnetic coupling is needed
CONSTANT_MAGNETIC_COUPLING=1
# Comment the next line if a random magnetic field is needed
CONSTANT_MAGNETIC_FIELD=0

OUTPUT_NAME=CompleteRandomnessCase
...
```

7. After the execution of the group of simulations defined in the configuration file, results are presented in different formats:

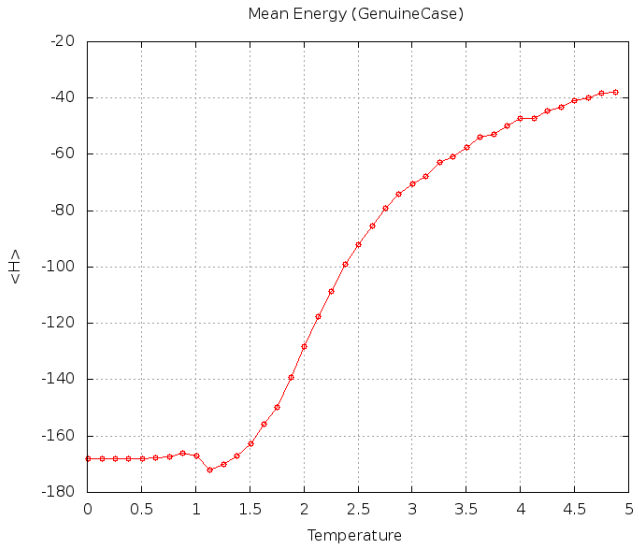
   (a) A table for each simulation is printed to the screen and saved in a file located in the "output" directory. For example:

```
* Next output: CompleteRandomnessCase.txt
-------------------------------------------------------------------------------
Level    Temperature      MeanEnergyEst    SpecificHeat     Magnetization
-------------------------------------------------------------------------------
0        0.010000         -2.464674        0.818280         0.069625
1        0.059900         -2.605526        0.497470         0.024083
2        0.109800         -2.626684        0.141379         0.017833
3        0.159700         -3.327892        0.233652         0.147319
4        0.209600         -3.331454        0.052610         0.148681
5        0.259500         -3.190520        0.696360         0.107208
...
```

   (b) A GNU Plot[2] script is crated in the "output" directory, which can be used to generate graphs files in PNG format (in the same directory) for each simulation. For example:

```
$ cd output/
$ gnuplot generateGraphs.plot
```

[2] http://www.gnuplot.info/



3.2 Compilation

The simulator can be compiled with GNU CC compiler, available for Unix/Linux systems and Windows. From the command line on the same directory where the source code is located, the following instruction will generate the "runMonteCarloIsingModel" file:

```
1 $ gcc -O3 -msse2 -fno-strict-aliasing -DHAVE_SSE2=1 -lm -
    DDSFMT_MEXP=19937 -o run-MonteCarloIsingModel dSFMT.c
    main-MonteCarloIsingModel.c
```

3.3 Execution and results

As explained above, the program reads a configuration file located at the "input" directory, where simulations parameters should be established. By default, there are three simulations defined corresponding to the three cases exposed in this problem set. Once compiled, the simulations can be started by executing the following sentence from the command line:

```
1 $ ./run-MonteCarloIsingModel
```

3.3.1 Simple case

Partition function The partition function encodes how the probabilities are partitioned among the different microstates (based on their individual energies), which implies that it counts the (weighted) number of states a system can

occupy. Since spins are not interacting $J_{ij} = 0$ and the magnetic field is constant $h_i = 1 \forall i$, therefore $H = -\sum_i \sigma_i$.
On the other hand, it is known the following relationship:

$$Z_N(T) = (Z_1(T))^N \tag{8}$$

Where $N = N_x \cdot N_y$ is the number of spins in the lattice. Therefore:

$$Z_N(T) = \left(\sum_\sigma e^{-\frac{H(\sigma)}{T}} \right)^N = \left(\sum_\sigma e^{\frac{\sum_i \sigma_i}{T}} \right)^N \tag{9}$$

And because we only have two possible states $\sigma = \{-1, +1\}$:

$$Z_N(T) = \left(\sum_\sigma e^{\frac{\sum_i \sigma_i}{T}} \right)^N = \left(e^{\frac{-1}{T}} + e^{\frac{1}{T}} \right)^N \tag{10}$$

Which is known to be:

$$Z_N(T) = \left(e^{\frac{-1}{T}} + e^{\frac{1}{T}} \right)^N = \left(2\cos\left(\frac{1}{T}\right) \right)^N \tag{11}$$

Average energy $\langle H \rangle$

$$\langle H \rangle = \frac{1}{Z(T)} \sum_\sigma H(\sigma) e^{-\frac{H(\sigma)}{T}} \tag{12}$$

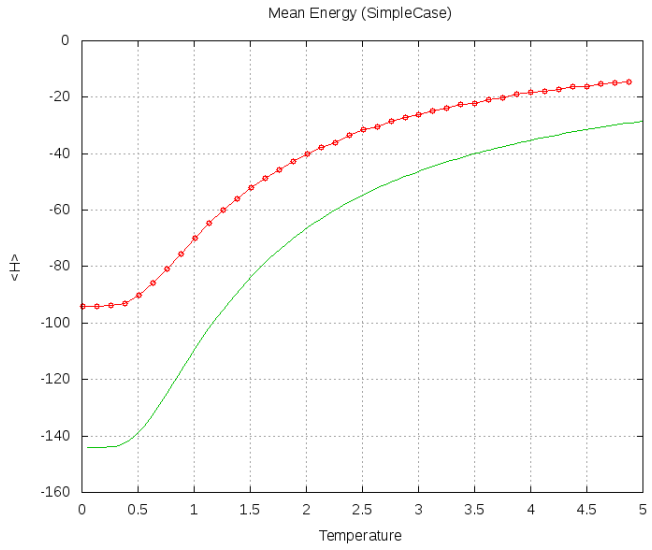
Considering $\beta = \frac{1}{T}$, the average energy is also related to the partition function in the following way:

$$\langle H \rangle = -\frac{\partial}{\partial \beta} \ln(Z(T)) = -\frac{1}{Z(T)} \frac{\partial}{\partial \beta} Z(T) = \frac{2^N \cdot N \cdot \cosh^{N-1}(\beta)}{2^N \cos^2(\beta)} = -N \cdot \tanh\left(\frac{1}{T}\right) \tag{13}$$

Mean value estimation $\tilde{E}$ and average energy $\langle H \rangle$ For a simulation with the following parameters:

```
OUTPUT_NAME=SimpleCase
MONTECARLO_LOOPS=10000
SAMPLE_NUM=1000
# Lattice size
NX=12
NY=12
TEMP_MIN=0.01
TEMP_MAX=5
# Number of measurement in the interval above
TEMP_MEASUREMENTS=40
# Comment the next line if a random magnetic coupling is needed
CONSTANT_MAGNETIC_COUPLING=0
# Comment the next line if a random magnetic field is needed
CONSTANT_MAGNETIC_FIELD=1
```

This is the results for the mean value estimator[3] (in red) and the average energy (in green) based on the analytical formulation (previous section) :



Average square distances between the estimator $\tilde{E}$ and$\langle H \rangle$ Average energy:

$$\langle H \rangle = \sum_{\sigma} H(\sigma) e^{-\frac{H(\sigma)}{T}} \tag{14}$$

And the estimator:

$$\tilde{E} = \frac{1}{N} \sum_{n=1}^{N} H(\sigma_n) \tag{15}$$

Where $\sigma_n \forall n$ has been chosen with the probability distribution $e^{-\frac{H(\sigma)}{T}}$.
The average square distances:

$$\varsigma^2 = \int_D (f(x) - I)^2 dx \tag{16}$$

$$\varsigma^2 = (\tilde{E} - \langle H \rangle)^2 = \left(\frac{1}{N} \sum_{n=1}^{N} H(\sigma_n) - \left(-N \cdot \tanh\left(\frac{1}{T}\right) \right) \right)^2 = \left(\sum_{n=1}^{N} H(\sigma_n) + \tanh\left(\frac{1}{T}\right) \right)^2 \tag{17}$$

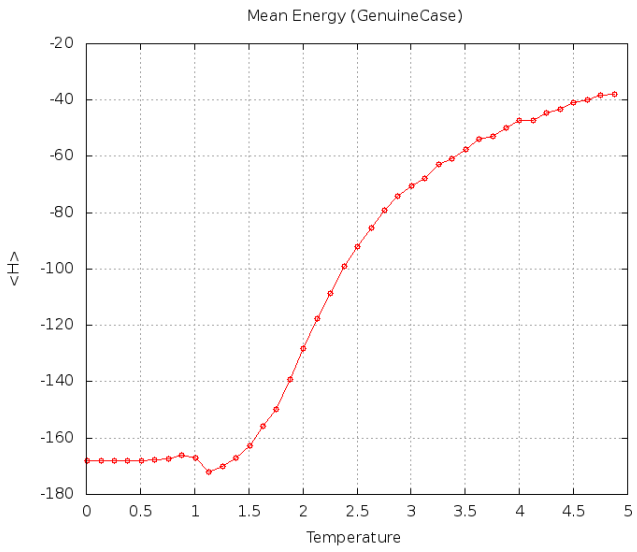
[3]Data table can be found in the output directory of the source code

3.3.2 Genuine case

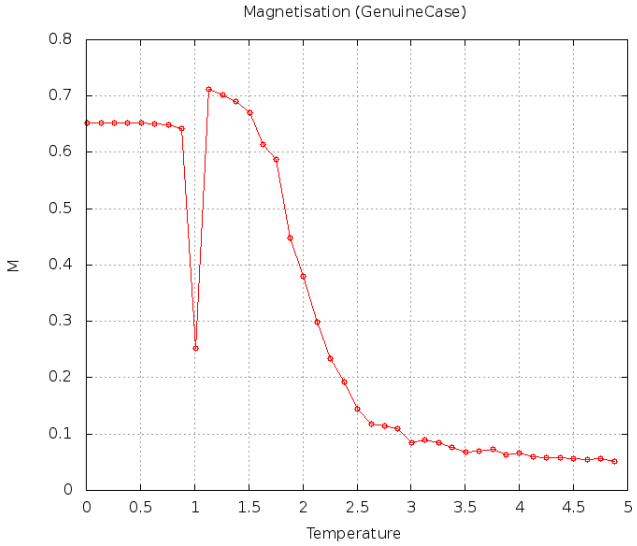
Results For a simulation with the following parameters:

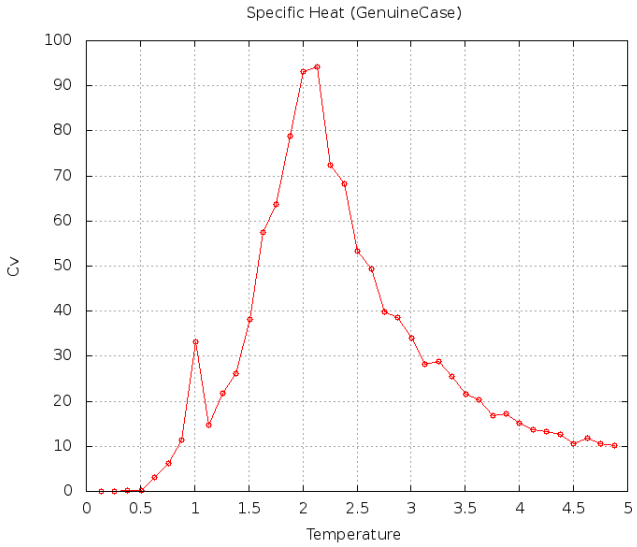
```
OUTPUT_NAME=GenuineCase
MONTECARLO_LOOPS=10000
SAMPLE_NUM=1000
# Lattice size
NX=12
NY=12
TEMP_MIN=0.01
TEMP_MAX=5
# Number of measurement in the interval above
TEMP_MEASUREMENTS=40
# Comment the next line if a random magnetic coupling is needed
CONSTANT_MAGNETIC_COUPLING=1
# Comment the next line if a random magnetic field is needed
CONSTANT_MAGNETIC_FIELD=0
```

These are the results[4]:



[4]Data table can be found in the output directory of the source code





Critical temperature On a 2D square lattice with external magnetic field $H = 0$, the critical temperature is observed to be on $T \simeq \frac{1}{0.44} = 2,2727$.

3.3.3 Complete randomness case

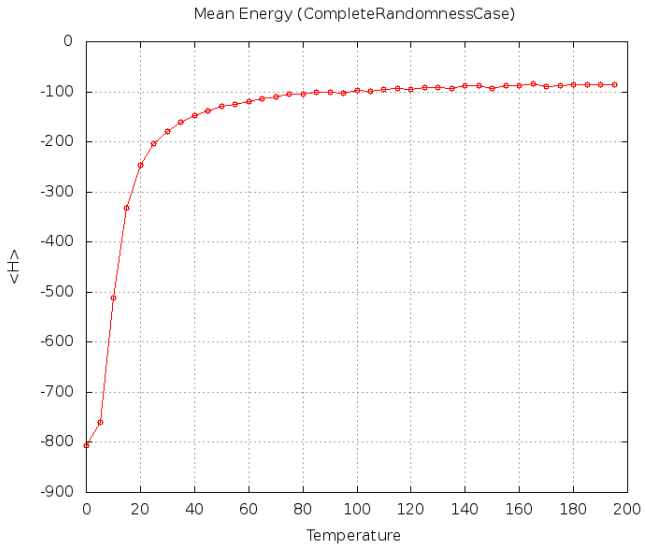
Although the problem set did not demand for an explicit computation with all coupling constants and magnetic fields established by random variables, it

has been implemented into the simulation application. If it is indicated in the configuration file, at the moment of the spin initialization the magnetic coupling and magnetic fields are also established by a pair of random variables distributed uniformly between $[0, 5)$.

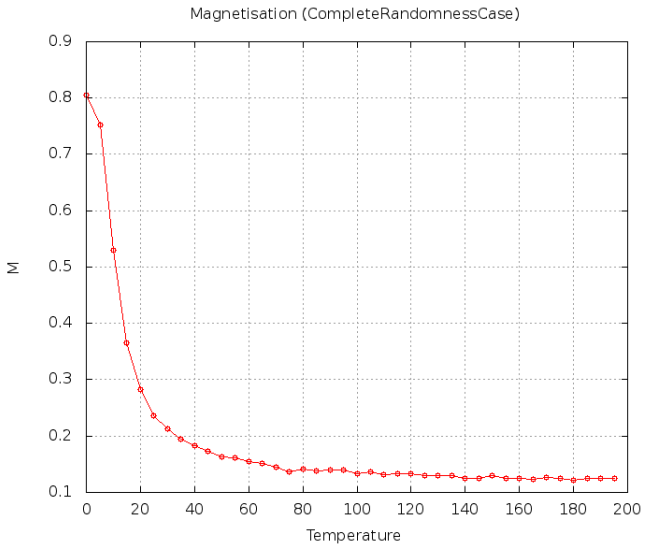
For a simulation with the following parameters:

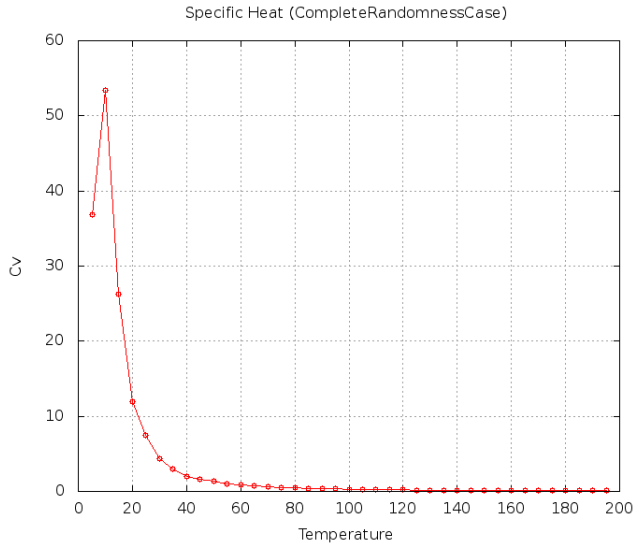
```
OUTPUT_NAME=CompleteRandomnessCase
MONTECARLO_LOOPS=10000
SAMPLE_NUM=1000
# Lattice size
NX=12
NY=12
TEMP_MIN=0.01
TEMP_MAX=200
# Number of measurement in the interval above
TEMP_MEASUREMENTS=40
# Comment the next line if a random magnetic coupling is needed
#CONSTANT_MAGNETIC_COUPLING=0
# Comment the next line if a random magnetic field is needed
#CONSTANT_MAGNETIC_FIELD=0
```

These are the results[5]:



[5]Data table can be found in the output directory of the source code





3.4 Source code

The program contains the following relevant functions:

- main: First function to be executed from which the simulations are orchestrated.
- initialize: Create data structures and initialize spin state (s), magnetic coupling (J) and magnetic field (h)

- update_state: Modify current state with a given probability that depends on the amount of energy change and temperature
- register_sample_measurement: Save into matrices the measurements needed for a given sample
- register_measurement: Sum up measurements for a group of sample and save the results into arrays
- Input/output file operations:
  - read_config: Read configuration file with one or more simulation sets
  - save_measurements: Print results and save them into an output file

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "dSFMT.h"

// Constants
#define CONFIG_FILE "config.txt"
#define GNUPLOT_FILE "generateGraphs.plot"
#define STRING_LIMIT 256
#define TRUE 1
#define FALSE 0


typedef struct {
    char output_name[STRING_LIMIT];
    int montecarlo_loops;
    int sample_num;

    // Lattice size, adjust
    int nx;
    int ny;

    double temp_min;
    double temp_max;
    int temp_measurements; // Number of measurement in the interval above
    int constant_magnetic_coupling; // TRUE or FALSE
    double magnetic_coupling;
    int constant_magnetic_field; // TRUE or FALSE
    double magnetic_field;
} configuration;

/////////////////////////////////////////////////////////////////////////
/// BEGIN: Global Variables
configuration config;

// Spin, magnetic coupling and magnetic field at site x,y
int **s;
double **J;
double **h;

double **vEnergySample;
double **vEnergySquareSample;
double **vMagnetisationSample;

double *vMagnetisation;
double *vTemperature;
double *vMeanEnergyEstimator;
double *vEnergyEstimator; // Needed to calculate Specific Heat
```

```
50  double *vSpecificHeat;
51  /// END: Global Variables
52  //////////////////////////////////////////////////////////////////////////////
53
54
55  //////////////////////////////////////////////////////////////////////////////
56  /// BEGIN: Matrix manipulation functions
57  int xUp(int i) {
58      return ((i+1) % config.nx);
59  }
60
61  int xDown(int i) {
62      return ((i-1+config.nx) % config.nx);
63  }
64  int yUp(int i) {
65      return ((i+1) % config.ny);
66  }
67  int yDown(int i) {
68      return ((i-1+config.ny) % config.ny);
69  }
70
71
72  int** create_imatrix(int size_x, int size_y) {
73      int i;
74      int** matrix;
75      matrix = calloc(size_x, sizeof(int*));
76      for(i = 0;i<size_x;i++) {
77          matrix[i] = calloc(size_y, sizeof(int));
78      }
79      return matrix;
80  }
81
82  double** create_dmatrix(int size_x, int size_y) {
83      int i;
84      double** matrix;
85      matrix = calloc(size_x, sizeof(double*));
86      for(i = 0;i<size_x;i++) {
87          matrix[i] = calloc(size_y, sizeof(double));
88      }
89      return matrix;
90  }
91
92  void destroy_matrix(void** matrix) {
93      int i;
94      for( i=0; matrix[i] != NULL; i++ ) {
95          free( matrix[i] );
96      }
97      free( matrix );
98  }
99  /// END: Matrix manipulation functions
100 //////////////////////////////////////////////////////////////////////////////
101
102
103
104 //////////////////////////////////////////////////////////////////////////////
105 /// BEGIN: Monte carlo functions
106
107 // Create data structures and initialize spin state (s), magnetic coupling (J
        ) and magnetic field (h)
108 void initialize(dsfmt_t *dsfmt){
109     int x, y;
110
111     // Structures creation
112     s = create_imatrix(config.nx, config.ny);
113     J = create_dmatrix(config.nx, config.ny);
114     h = create_dmatrix(config.nx, config.ny);
115
```

```
116     vEnergySample = create_dmatrix(config.temp_measurements, config.
            sample_num);
117     vEnergySquareSample = create_dmatrix(config.temp_measurements, config.
            sample_num);
118     vMagnetisationSample = create_dmatrix(config.temp_measurements, config.
            sample_num);
119
120     vMagnetisation = malloc(config.temp_measurements*sizeof(double));
121     vTemperature = malloc(config.temp_measurements*sizeof(double));
122     vMeanEnergyEstimator = malloc(config.temp_measurements*sizeof(double));
123     vEnergyEstimator = malloc(config.temp_measurements*sizeof(double));
124     vSpecificHeat = malloc(config.temp_measurements*sizeof(double));
125
126
127     for (x=0; x<config.nx; x++) {
128         for (y=0; y<config.ny; y++) {
129             // Generate a random number distributed uniformly between [0,1)
                    for selecting spin status
130             // Initial state: highly disordered (hot)
131             if (dsfmt_genrand_close_open(dsfmt) < 0.5) {
132                 s[x][y] = 1;
133             } else {
134                 s[x][y] = -1;
135             }
136
137             if (config.constant_magnetic_coupling == TRUE) {
138                 // Constant magnetic coupling for every bond
139                 J[x][y] = config.magnetic_coupling;
140             } else {
141                 // Generate a random X coordinate between [0,5)
142                 J[x][y] = dsfmt_genrand_close_open(dsfmt)*5;
143             }
144             if (config.constant_magnetic_field  == TRUE) {
145                 // Constant magnetic field for every field
146                 h[x][y] = config.magnetic_field;
147             } else {
148                 // Generate a random X coordinate between [0,5)
149                 h[x][y] = dsfmt_genrand_close_open(dsfmt)*5;
150             }
151         }
152     }
153
154 }
155
156 // Modify current state with a given probability that depends on the amount
        of energy change and temperature
157 void update_state(dsfmt_t *dsfmt, double temp) {
158     int x,y;
159     double p_plus,p_minus;
160     int current_spin, new_spin;
161     double H, new_H, H_change;
162     double random_var;
163
164     x = (int)(dsfmt_genrand_close_open(dsfmt) * 10) % config.nx;
165     y = (int)(dsfmt_genrand_close_open(dsfmt) * 10) % config.ny;
166
167     current_spin = s[x][y];
168     new_spin = - s[x][y];
169
170     // Current energy at chosen point
171     double a = J[xUp(x)][y] * current_spin * s[xUp(x)][y];
172     double b = J[xDown(x)][y] * current_spin * s[xDown(x)][y];
173     double c = J[x][yUp(y)] * current_spin * s[x][yUp(y)];
174     double d = J[x][yDown(y)] * current_spin * s[x][yDown(y)];
175     // NOTE: Double counting is not produced because we are measuring a
            single point,
176     //       not the whole matrix
177     if (config.nx == 1 && config.ny == 1) {
```

```
178             H = - h[x][y]*current_spin;
179         } else if (config.nx == 1){
180             H = - (c + d) - h[x][y]*current_spin;
181         } else if (config.ny == 1){
182             H = - (a + b) - h[x][y]*current_spin;
183         } else {
184             H = - (a + b + c + d) - h[x][y]*current_spin;
185         }
186 
187         // New energy at chosen point
188         a = J[xUp(x)][y] * new_spin * s[xUp(x)][y];
189         b = J[xDown(x)][y] * new_spin * s[xDown(x)][y];
190         c = J[x][yUp(y)] * new_spin * s[x][yUp(y)];
191         d = J[x][yDown(y)] * new_spin * s[x][yDown(y)];
192         // NOTE: Double counting is not produced because we are measuring a
                single point,
193         //       not the whole matrix
194         if (config.nx == 1 && config.ny == 1) {
195             new_H = - h[x][y]*new_spin;
196         } else if (config.nx == 1){
197             new_H = - (c + d) - h[x][y]*new_spin;
198         } else if (config.ny == 1){
199             new_H = - (a + b) - h[x][y]*new_spin;
200         } else {
201             new_H = - (a + b + c + d) - h[x][y]*new_spin;
202         }
203 
204         // Contribution to total energy if change is made
205         H_change = new_H - H;
206 
207         /*if (H_change < 0) {
208             // New state is less energetic
209             s[x][y] = new_spin;
210         } else {*/
211             random_var = dsfmt_genrand_close_open(dsfmt); // random number 0-1
212             if (random_var < exp(-H_change/temp)) {
213                 s[x][y] = new_spin;
214             }
215         //}
216     }
217     /// END: Monte carlo functions
218     ////////////////////////////////////////////////////////////////////////////
219 
220 
221     ////////////////////////////////////////////////////////////////////////////
222     /// BEGIN: File functions
223 
224 
225     void create_gnuplotfile(){
226         FILE *file;
227 
228         chdir("output");
229         file = fopen(GNUPLOT_FILE,"w");
230         chdir("..");
231         fprintf(file, "#!/usr/bin/gnuplot\n");
232         fprintf(file, "set␣grid\n");
233         fprintf(file, "set␣xlabel␣'Temperature'\n");
234         fprintf(file, "unset␣key\n");
235         fprintf(file, "set␣rmargin␣10\n");
236         fprintf(file, "set␣lmargin␣10\n");
237         fprintf(file, "set␣terminal␣png␣size␣800,␣600\n");
238         fprintf(file, "\n");
239 
240         fclose(file);
241     }
242 
243     void append_to_gnuplotfile(char *output_name, int xmin, int xmax){
244         FILE *file;
```

```
245
246         chdir("output");
247         file = fopen(GNUPLOT_FILE,"a");
248         chdir("..");
249         fprintf(file, "set␣xrange␣[%i:%i]\n", xmin, xmax);
250         fprintf(file, "set␣xtics␣%f\n", (xmax - xmin) / 10.0);
251         fprintf(file, "set␣title␣'Mean␣Energy␣(%s)'\n", output_name);
252         fprintf(file, "set␣ylabel␣'<H>'\n");
253         fprintf(file, "set␣output␣\"%s-MeanEnergy.png\"\n", output_name);
254         fprintf(file, "plot␣\"%s.txt\"␣every␣::1␣using␣2:3␣with␣linespoints␣pt␣6␣
              title␣'Mean␣energy'\n", output_name);
255         fprintf(file, "\n");
256         fprintf(file, "set␣title␣'Specific␣Heat␣(%s)'\n", output_name);
257         fprintf(file, "set␣ylabel␣'Cv'\n");
258         fprintf(file, "set␣output␣\"%s-SpecificHeat.png\"\n", output_name);
259         fprintf(file, "plot␣\"%s.txt\"␣every␣::2␣using␣2:4␣with␣linespoints␣pt␣6␣
              title␣'Specific␣Heat'\n", output_name);
260         fprintf(file, "\n");
261         fprintf(file, "set␣title␣'Magnetisation␣(%s)'\n", output_name);
262         fprintf(file, "set␣ylabel␣'M'\n");
263         fprintf(file, "set␣output␣\"%s-Magnetisation.png\"\n", output_name);
264         fprintf(file, "plot␣\"%s.txt\"␣every␣::1␣using␣2:5␣with␣linespoints␣pt␣6␣
              title␣'Magnetisation'\n", output_name);
265         fprintf(file, "\n\n\n");
266         fclose(file);
267     }
268
269     // Read config file with one or more config sets and extract one
270     configuration *read_config(FILE *file) {
271         configuration config = {"default", 1000, 10, 2 , 2, 0.1, 5, 10, FALSE, 0,
                FALSE, 0};
272         configuration *pconfig = &config;
273         int level;
274         char line[STRING_LIMIT], *param, *value;
275         fpos_t pos;
276
277         config.constant_magnetic_coupling = FALSE;
278         config.constant_magnetic_field = FALSE;
279
280         while (!feof(file)) {
281             fgetpos(file, &pos); // Position before reading
282
283             if (fgets(line, sizeof(line), file) == NULL) {
284                 continue;
285             }
286             // Ignore comments and empty lines
287             if (line[0] != '#' && strlen(line) > 1) {
288                 param = strtok(line, "=");
289                 value = strtok(NULL, "\n"); // Remove \n from line
290
291                 if (strcmp(param, "OUTPUT_NAME") == 0) {
292
293                     if ((strcmp(config.output_name, "default") == 0)) { // First
                          time: config begins
294                         strcpy(config.output_name, value);
295                     } else {    // New config: end reading
296                         fsetpos(file, &pos); // Go back to last position
297                         break;
298                     }
299
300                 } else if (strcmp(param, "MONTECARLO_LOOPS") == 0) {
301                     config.montecarlo_loops = atoi(value);
302                 } else if (strcmp(param, "SAMPLE_NUM") == 0) {
303                     config.sample_num = atoi(value);
304                 } else if (strcmp(param, "NX") == 0) {
305                     config.nx = atoi(value);
306                 } else if (strcmp(param, "NY") == 0) {
307                     config.ny = atoi(value);
```

```
308             } else if (strcmp(param, "TEMP_MIN") == 0) {
309                 config.temp_min = atof(value);
310             } else if (strcmp(param, "TEMP_MAX") == 0) {
311                 config.temp_max = atof(value);
312             } else if (strcmp(param, "TEMP_MEASUREMENTS") == 0) {
313                 config.temp_measurements = atoi(value);
314             } else if (strcmp(param, "CONSTANT_MAGNETIC_COUPLING") == 0) {
315                 config.constant_magnetic_coupling = TRUE;
316                 config.magnetic_coupling = atof(value);
317             } else if (strcmp(param, "CONSTANT_MAGNETIC_FIELD") == 0) {
318                 config.constant_magnetic_field = TRUE;
319                 config.magnetic_field = atof(value);
320             }
321         }
322     }
323 
324     if ((strcmp(config.output_name, "default") == 0)) {
325         // No more configurations found in file
326         return NULL;
327     } else {
328         return pconfig;
329     }
330 }
331 
332 // Save into matrices the measurements needed for a given sample
333 void register_sample_measurement(int temperature_level, int sample, double
        temperature) {
334     // Calculate mean energy and magnetization, and print out
335     int x,y;
336     double magnetisation,H, Ha, Hb;
337 
338     // Sum over the neighbour sites
339     H = magnetisation = Ha = Hb = 0.0;
340 
341     for (x=0; x<config.nx; x++) {
342         for (y=0; y<config.ny; y++) {
343 
344             // Calculate magnetic coupling
345             double a = J[xUp(x)][y] * s[x][y] * s[xUp(x)][y];
346             double b = J[xDown(x)][y] * s[x][y] * s[xDown(x)][y];
347             double c = J[x][yUp(y)] * s[x][y] * s[x][yUp(y)];
348             double d = J[x][yDown(y)] * s[x][y] * s[x][yDown(y)];
349             // NOTE: Double counting is produced because we pass through the
350             //       whole matrix and every link will be sum twice
351             if (config.nx == 1 && config.ny == 1) {
352                 Ha += 0;
353             } else if (config.nx == 1){
354                 Ha += (c + d)/2; // Correct double counting dividing by 2
355             } else if (config.ny == 1){
356                 Ha += (a + b)/2; // Correct double counting dividing by 2
357             } else {
358                 Ha += (a + b + c + d)/2; // Correct double counting dividing
                        by 2
359             }
360 
361             Hb += h[x][y]*s[x][y];
362             magnetisation += s[x][y];
363         }
364     }
365 
366     H = - Ha - Hb;  // Energy
367     //H = H/(config.nx*config.ny);  // Mean energy
368     magnetisation = magnetisation/(config.nx*config.ny);
369 
370     vEnergySample[temperature_level][sample] = H;
371     vEnergySquareSample[temperature_level][sample] = H*H;
372     vMagnetisationSample[temperature_level][sample] = magnetisation;
373 }
```

```
374
375 // Sum up measurements for a group of sample and save the results into arrays
376 void register_measurement(int temperature_level, double temperature) {
377     // Calculate mean energy and magnetization, and print out
378     int sample;
379     double H, H2, magnetization;
380
381     /* sum over the neighbour sites - typewriter fashion */
382     H = H2 = magnetization = 0.0;
383     for (sample=0; sample<config.sample_num; sample++) {
384         H += vEnergySample[temperature_level][sample];
385         H2 += vEnergySquareSample[temperature_level][sample];
386         magnetization += vMagnetisationSample[temperature_level][sample];
387     }
388
389     vTemperature[temperature_level] = temperature;
390
391     vMagnetisation[temperature_level] = fabs(magnetization/(config.sample_num
          ));
392     vMeanEnergyEstimator[temperature_level] = H/(config.sample_num);
393     vSpecificHeat[temperature_level] = ( (H2/(config.sample_num)) - pow(H/(
          config.sample_num),2)) / pow(temperature,2);
394 }
395
396 // Print results and save them into an output file
397 void save_measurements(){
398     int level;
399     FILE *file;
400
401     append_to_gnuplotfile(config.output_name, config.temp_min, config.
          temp_max);
402
403     chdir("output");
404     file = fopen(strcat(config.output_name, ".txt"),"w");
405     chdir("..");
406     printf("* Next output: %s\n", config.output_name);
407     printf("
          ------------------------------------------------------------------------\
          n");
408     printf("Level\tTemperature\tMeanEnergyEst\tSpecificHeat\tMagnetization\n"
          );
409     printf("
          ------------------------------------------------------------------------\
          n");
410
411     fprintf(file, "Level\tTemperature\tMeanEnergyEst\tSpecificHeat\
          tMagnetization\n");
412     for (level=0; level<config.temp_measurements; level++) {
413         fprintf(file, "%i\t%f\t%f\t%f\t%f\n", level, vTemperature[level],
              vMeanEnergyEstimator[level], vSpecificHeat[level],
              vMagnetisation[level]);
414         printf("%i\t%f\t%f\t%f\t%f\n", level, vTemperature[level],
              vMeanEnergyEstimator[level], vSpecificHeat[level],
              vMagnetisation[level]);
415     }
416     printf("
          ------------------------------------------------------------------------\
          n");
417     fclose(file);
418 }
419 /// END: File functions
420 ////////////////////////////////////////////////////////////////////////////
421
422
423
424 ////////////////////////////////////////////////////////////////////////////
425 /// BEGIN: MAIN
426 int main(int argc, char* argv[]) {
```

```
427     FILE *file;
428     dsfmt_t dsfmt; // Random number generator
429     int i, sample, temperature_level;
430     double temperature;
431     int seed = 12345;
432 
433     create_gnuplotfile();
434 
435     // For each config set in the config file
436     chdir("input");
437     file = fopen(CONFIG_FILE,"r");
438     chdir("..");
439     configuration *pconfig = read_config(file);
440     while (pconfig != NULL) {
441         memcpy(&config, pconfig, sizeof(config)); // Make read config global
442 
443         // Initialize the random number generator (RNG)
444         dsfmt_init_gen_rand(&dsfmt, seed);
445 
446         temperature = config.temp_min;
447         if (temperature == 0) temperature += 0.00001; // Avoid zero division
448         temperature_level = 0;
449 
450         // Create data structures and initialize spin state (s), magnetic
                coupling (J) and magnetic field (h)
451         initialize(&dsfmt);
452 
453         // For each temperature
454         while (temperature < config.temp_max && temperature_level < config.
                temp_measurements) {
455 
456             // Generate samples
457             for (sample=0; sample < config.sample_num; sample++) {
458                 // Monte carlo simulation for a given sample
459                 for (i=0; i < config.montecarlo_loops; i++) {
460                     update_state(&dsfmt, temperature);
461                 }
462                 register_sample_measurement(temperature_level, sample,
                        temperature);
463             }
464 
465             // Calculate mean values for the group of samples at a given
                    temperature
466             register_measurement(temperature_level, temperature);
467 
468             temperature_level++;
469             temperature += ((double)config.temp_max - config.temp_min) /
                    config.temp_measurements;
470             if (temperature == 0) temperature += 0.00001; // Avoid zero
                    division
471         }
472 
473         // Print and save outputs
474         save_measurements();
475 
476         // Next config set
477         pconfig = read_config(file);
478     }
479 
480     fclose(file);
481     return 0;
482 }
483 /// END: MAIN
484 ////////////////////////////////////////////////////////////////////////////
```

References

[1] Iblisdir, S. (2010). *An Introduction to Probability Theory.* Universitat de Barcelona.

[2] Saito, M. Matsumoto, M. *SIMD-oriented Fast Mersenne Twister (SFMT).* Retrieved November 19, 2010, from the World Wide Web: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html

[3] Wikipedia (2010). *Mersenne twister.* Retrieved November 19, 2010, from the World Wide Web: http://en.wikipedia.org/wiki/Mersenne_twister

[4] Wikipedia (2010). *Partition function.* Retrieved November 19, 2010, from the World Wide Web: http://en.wikipedia.org/wiki/Partition_function_(statistical_mechanics)

[5] Khan, W.U. Kar, P. Approximation of pi using the Monte Carlo Method. Retrieved November 19, 2010, from the World Wide Web: http://www.scribd.com/doc/6987974/Approximation-of-Pi-Using-the-Monte-Carlo-Method

[6] Lisa Larrimore. Monte Carlo Simulation of the 2D Ising Model.

[7] Peter Meyer (2009). Lattice Geometries. Retrieved November 19, 2010, from the World Wide Web: http://www.hermetic.ch/compsci/lattgeom.htm