Part II Computational Project, 22 March 2014

The Ising Model of a Ferromagnet

Aleksandar Sklyarov, as2202

Abstract
A numerical study of the two-dimensional Ising model of a ferromagnet, using the Metropolis algorithm, is presented. In particular, time and temperature dependences of the energy, magnetization and heat capacity of systems with sizes from 20x20 to 50x50 were investigated. The results were compared against Onsager's theoretical solution and good agreement was established. The value of the critical temperature of transition at zero magnetic field was confirmed to within 2% of its analytical value. Also, the theoretically predicted power-law dependence of the magnetization near the critical temperature was examined and the calculated value of the critical exponent agrees well with the theoretical result. Finally, the behaviour of an Ising system in the presence of an external magnetic field was investigated.

Contents

Introduction

The Ising model is one of the most widely studied models in statistical mechanics. It describes a ferromagnet, which is approximated as a set of spins s_i placed on a fixed lattice. Each spin can take one of two values – spin-up $(+1)$ and spin-down (-1). Analytical studies of the model present tremendous challenges and solutions have been achieved only in one and two dimensions [1]. The intrinsic complex many-body nature of the problem, therefore, makes numerical techniques a preferred choice for theoretical investigations.

This project deals specifically with the two-dimensional Ising model. A system, represented by an $N \times N$ lattice, having $n = N^2$ sites, can be found in 2^n states, and the energy of any particular state is given by:

$$E = E_0 - \mu HM = -J \sum_{<ij>} s_i s_j - \mu H \sum_{i=1}^{N^2} s_i \tag{1}$$

where J is the exchange energy, μ the magnetic moment and H the external magnetic field. The $< ij >$ in the first term indicates that the sum is done over nearest-neighbour spins. The large number of degrees of freedom naturally suggests a Monte Carlo approach. For that reason, the Metropolis algorithm, which is considered to be one of the most widely used Monte Carlo methods, was chosen for this project.

The ultimate goal is to simulate a two-dimensional Ising system and thus estimate the values of various quantities of interest, such as energy, magnetization and heat capacity, and investigate how they depend on time and temperature. Most of the questions related to the Ising model can be answered in the absence of an external magnetic field, $H = 0$, so the main part of the project concentrates on this case. In the last stage, the effects from switching on the field, $H \neq 0$, are investigated.

In the next section, a thorough analysis of the computational aspects of the problem will be presented. A description of how the solution was implemented and an account of its performance will be given in the third section. In the fourth section, I will present and discuss the relevant results from the simulations, after which I will summarise all key points in the Conclusion.

1. Analysis

It is crucial that one thoroughly examines the relevant aspects of computational physics in the problem before an attempt for a solution is made [2, 3, 4, 5, 6, 7, 8]. This helps to recognize in advance not only potential pitfalls but also ways of achieving better efficiency. This section presents a number of key features of the Metropolis algorithm and the Ising model, which were analysed before the actual implementation.

A good place to start the analysis of the problem is from the foundations of the approach that is to be implemented. In a nutshell, the Metropolis algorithm is characterized by having single-spin-flip dynamics and acceptance probability given by:

$$A(\text{flip}) = \begin{cases} e^{-\frac{\Delta E}{k_B T}} & \text{if } \Delta E > 0 \\ 1 & \text{otherwise} \end{cases}$$

The term *single-spin-flip dynamics* means that after each step of the method the difference between the old and the new configuration of the system is at most one spin. This feature allows for the conservation of *ergodicity*, which is the ability of the system to reach any state from any other state if it is left for long enough. Ergodicity is a fundamental property of real physical systems and it is necessary that it is also valid in our simulation.

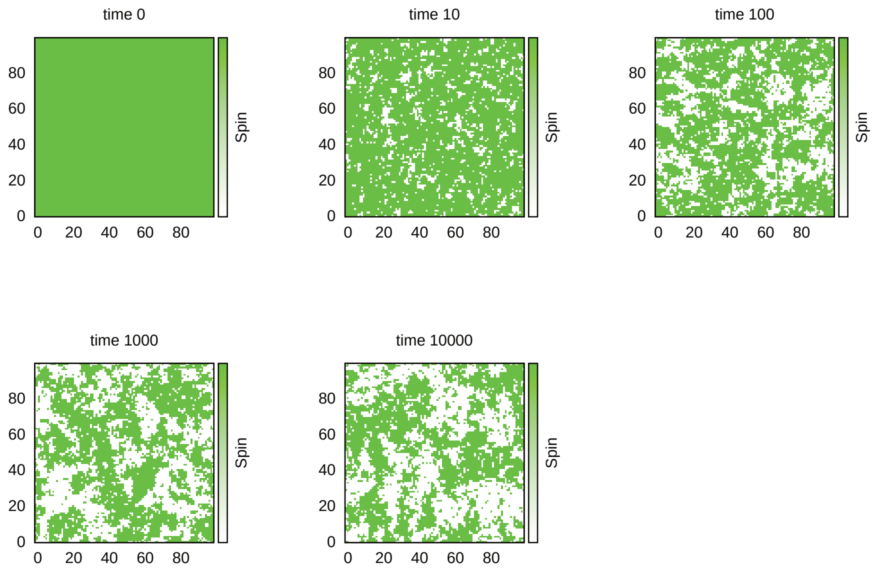

Figure 1. Snapshots of the time evolution of a 100×100 system that has been started with initial conditions corresponding to $T = 0$ with all spins $+1$ at temperature $T = 2.5$. The snapshots have been taken at exponentially increasing time steps, starting from *time 0* up to *time 10000*. By the time of the last snapshot the system has reached equilibrium according to the criteria described in the analysis.

The *acceptance probability* determines how likely it is for a particular spin to be flipped. If the energy of the new state is lower than that of the present state, the spin is always flipped, otherwise it is flipped with probability $e^{-\frac{\Delta E}{k_B T}}$. A random spin is chosen at each step of the method. For an $N \times N$ lattice, a single *time step* consists of N^2 steps of the algorithm, so that on average all spins would have had the chance to be flipped once.

Periodic boundary conditions are to be applied. That is, the spins on one edge of the lattice become neighbours of the corresponding spins on the other edge. This ensures that all spins have the same number of neighbours and that there are no special ones with differing properties from the rest.

Once the key aspects of the Metropolis algorithm have been analysed, one can move to the specifics of the simulation. For a start, it needs to be decided what the *initial conditions* of the simulated system should be. From all possible configurations there are two that stand out – $T = 0$ and $T \to \infty$. At $T = 0$ the Ising model is in its ground state with all spins aligned in the same direction. At $T \to \infty$ the thermal energy is infinitely larger than the exchange energy, which effectively results in the spins being randomly oriented throughout the lattice.

Another important element of the simulation is the *calculation of* ΔE. One can choose to do this directly by calculating the two energy states separately, using the expression for E from Equation 1, and subtracting them, $\Delta E = E_{\text{new}} - E_{\text{old}}$. However, this is not a very efficient approach, since even in zero magnetic field the number of operations required is proportional to the number of bonds in the system, which is $O(n)$. A better approach could take advantage of the fact that:

$$E_{\text{new}} - E_{\text{old}} = \Delta E_0 - \mu H \Delta M \tag{2}$$

where,

$$\begin{aligned} \Delta E_0 &= -J \sum_{<ij>} s_i^{\text{new}} s_j^{\text{new}} + J \sum_{<ij>} s_i^{\text{old}} s_j^{\text{old}} \\ &= -J \sum_{i \text{ n.n. } k} s_i^{\text{old}} \left(s_k^{\text{new}} - s_k^{\text{old}} \right) \\ &= 2J \sum_{i \text{ n.n. } k} s_i^{\text{old}} s_k^{\text{old}} \\ &= 2J s_k^{\text{old}} \sum_{i \text{ n.n. } k} s_i^{\text{old}} \end{aligned} \tag{3}$$

and

$$\begin{aligned} \Delta M &= \sum_{i=1}^{N^2} \left(s_i^{\text{new}} - s_i^{\text{old}} \right) \\ &= s_k^{\text{new}} - s_k^{\text{old}} \\ &= -2 s_k^{\text{old}} \end{aligned} \tag{4}$$

where the sum $\sum_{i \text{ n.n. } k}$ is over the nearest neighbours of the flipped spin k. Therefore,

$$E_{\text{new}} - E_{\text{old}} = 2 s_k^{\text{old}} \left(J \sum_{i \text{ n.n. } k} s_i^{\text{old}} + \mu H \right) \tag{5}$$

This expression now requires only the sum of four terms, which is $O(1)$, so it is much more efficient. Furthermore, it is not needed to change the state of the system to calculate it.

The last expression for ΔE suggests yet another improvement that one could introduce into their implementation. It can be seen that ΔE can only take one of five possible values. Therefore, instead of *calculating* $e^{-\frac{\Delta E}{k_B T}}$, which is a slow operation, every time when $\Delta E > 0$, its five possible values can be calculated only once in the beginning of the simulation and then reused when needed. This trick should increase the speed of the algorithm steps significantly. It should be noted that in the actual implementation ΔE_0 and ΔM from Equation 2 have been considered separately, which has lead to ΔE having ten possible values. Nevertheless, the concept is unchanged.

Another critical part of the simulation is to decide when a state of *equilibrium* has been reached. When the system is started from some initial configuration, it has to be left for a suitably long period of time until it equilibrates. This period is called the equilibration time – τ_{eq}. At the very least, one can expect roughly n time steps to suffice for equilibrium, since every spin needs to be allowed to flip at least once. There exist many sophisticated approaches of estimating τ_{eq}, but taking $\tau_{\text{eq}} = n$ is considered to be satisfactory.

A potential pitfall of defining equilibrium is the confusion with a state of *metastable equilibrium*. This is generally a challenge for the sophisticated methods mentioned above and is easily overcome by repeating the simulation with different initial conditions.

Once the system has reached equilibrium (Figure 1), *measurements* of the quantities of interest need to be done. In particular, these are the energy, magnetization and heat capacity.

As explained a few paragraphs ago, calculating the energy directly is not efficient. Instead, one can calculate the initial energy once and then use ΔE, which is calculated anyway, to update the value of the total energy at every flip,

$$E_{\text{new}} = E_{\text{old}} + \Delta E.$$

A similar approach can be adopted for the calculation of the magnetization, following from Equation 4,

$$M_{\text{new}} = M_{\text{old}} + \Delta M.$$

The measurement of the mean energy and magnetization, however, requires averaging the calculated values over some time during the run. To do this, one needs to know the correlation time, τ_{corr}, of the simulation. The correlation time is a measure of how long it takes for the system to get from one state to a significantly different one, in which the number of spins that are the same as in the initial state is no more than what it would be expected to be found by chance. There are some advanced methods of estimating τ_{corr} from which it is known that usually the equilibration time is significantly larger than the correlation time, $\tau_{\text{eq}} > \tau_{\text{corr}}$. Therefore, taking $\tau_{\text{corr}} = \tau_{\text{eq}}$ would be a safe assumption, which avoids the need for implementing any involved techniques.

The last part of the preliminary analysis deals with *errors*. Since the measured quantities are statistical expectations, it is natural for them to have *statistical error* in their values. In the cases of the mean energy and magnetization this is simply the standard deviation. The situation with the heat capacity is a bit more complicated, since its value is derived from the energy. In the case when it is calculated as $C = \frac{dE}{dT}$, the relative error is given by the relative error in E divided by dT, $R_C = \frac{R_E}{dT}$. In the second case when the heat capacity is calculated as $C = \frac{\sigma_E^2}{k_B T^2}$, the uncertainty can be estimated using the so-called *bootstrap method*. This technique consists in taking several subsets of the energy values, from which the heat capacity has been obtained, and calculating a separate heat capacity from each. The standard deviation of these values then provides an estimate of the error.

The main sources of *systematic errors* are τ_{eq} and τ_{corr}. The negative effects from these are minimized by taking safe assumptions for their values.

2. Implementation

In this section, a brief outline of the computer implementation of the problem is given. The full source code listing is included in the Appendix.

The foundations of the Ising model and the Metropolis algorithm are implemented in the class *Lattice*. As its name suggests, it represents an $N \times N$ lattice of spins. The most important method in this class is *flip*. It realizes a single step of the Metropolis algorithm and updates the values of the energy and magnetization as suggested in the previous section.

Next, the essence of all subsequent simulations is implemented in the source file *simulation.cc*. The class *Simulation* builds records of all quantities of interest, which are used during some of the calculations and eventually also to produce various plots. The two key methods are *simulate* and *evolve_in_T*. The first one realizes a time evolution of a given Ising system. It constitutes a central building block for the subsequent levels of the simulation and also produces the results for the first task from the project instructions handout. The second method unites the major part of the contents of the source file. As its name suggests, it realizes consecutive simulations of Ising systems in a given range of temperatures and analyses their behaviour to answer the rest of the project tasks. As a whole, the implementation has tried to follow closely the key steps described in the preliminary analysis. More detailed information about the individual methods is given in the comments of the source code.

Finally, the *main.cc* source file establishes the interface for communication with the program. It offers five input options. The first three are related to the project tasks and the last two are to carry out performance tests.

Performance The performance of the program was examined with the aid of two tests. The first one investigated the scaling of the time evolution simulation with the size of the system. It consisted in running systems of increasing size, each for 1000 time steps, and recording the CPU time needed for each. The result from this test is shown in Figure 2. The

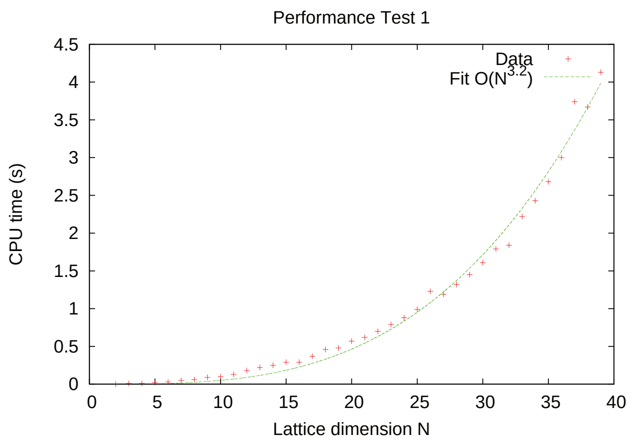

Figure 2. Result from the first performance test.

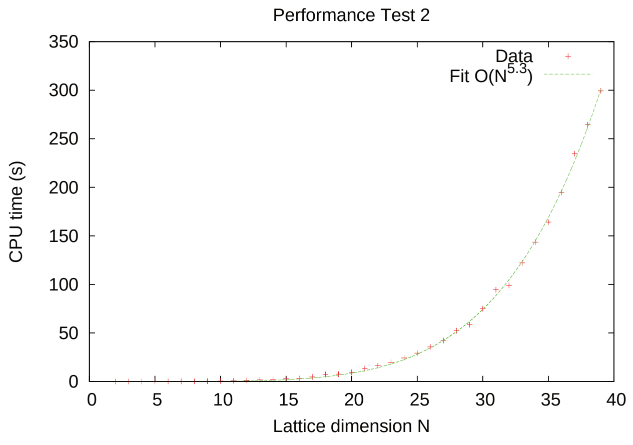

Figure 3. Result from the second performance test.

scaling corresponds to $O(n^{1.6})$ where $n = N^2$ is the number of spins in the system. The implementation originally aimed at achieving $O(n)$ scaling with $O(1)$ scaling of the individual algorithm steps. As it turns out, this has not been fully achieved and the reason is believed to be hidden in the scaling of the steps, which somehow exhibit sublinear scaling. Nevertheless, $O(n^{1.6})$ still allows for satisfactory simulations to be achieved. The second test investigated the scaling in the case when the simulation is run repeatedly over a certain temperature range. The result is shown in Figure 3 and it exhibits $O(n^{2.6})$ scaling. This agrees well with the expectations because the additional power of n has to come from the adaptive time for each simulation, which reflects the fact that the equilibration time is longer for a larger system.

3. Results and Discussion

In this section, I present and discuss the relevant results related to the five tasks from the project manual. The discussion is organized in separate subsections, each corresponding to a specific task.

Task 1. Time evolution of the energy and magnetization was investigated for a 50×50 lattice in a range of initial temperatures and with both starting conditions, $T = 0$ and $T \to \infty$. Representative subsets of the results are shown in Figure 4 and Figure 5. In the first one, it can be seen how quickly the systems' energy reaches equilibrium values. However, attention should be drawn to the middle curve, corresponding to $T = 2.0$. In this case the system initially falls into a state of metastable equilibrium and much later, at a time around 1800, it manages to reach the global minimum of its energy. Nevertheless, all systems manage to achieve equilibrium in less than the assumed equilibration time, which in this case is 2500. In the second figure, it is shown how the magnetization of the same systems evolves with time. The three cases, for which $T < T_C$, reach equilibrium values of non-zero magnetization, while the other two, for which $T > T_C$, equilibrate at zero average magnetization. The big fluctuations in the system at $T = 2.5$ are most probably due to its nearness to the critical state. All results are in agreement with theoretical predictions, which suggests that the simulation works correctly.

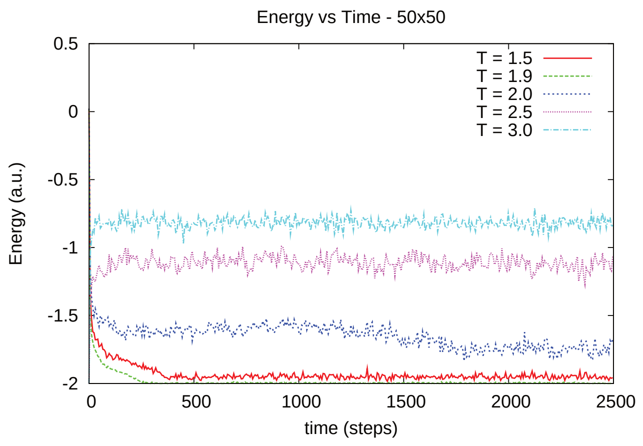

Figure 4. Time evolution of the energy per site for a 50×50 lattice.

Task 2. The temperature dependence of the mean energy and magnetization was investigated for 30×30, 40×40 and 50×50 lattices. A large number of plots was produced for each case and these can be found in the project directory. Here, only key plots related to the 50×50 system are presented. In Figure 6 and Figure 7, it can be seen how the mean energy and magnetization of the simulated system depend on temperature with the corresponding uncertainties in their values. Both plots indicate the presence of a transition at a temperature somewhere between 2.2 and 2.5. This is in agreement with the theory, which predicts a transition at $T_C = 2.27$. The behaviour of the system in the region around T_C exhibits larger fluctuations and this naturally leads to greater uncertainties in the calculated mean values, as observed in the figures. The temperature dependence of the errors in the mean energy and magnetization are presented in Figure 8 and Figure 9,

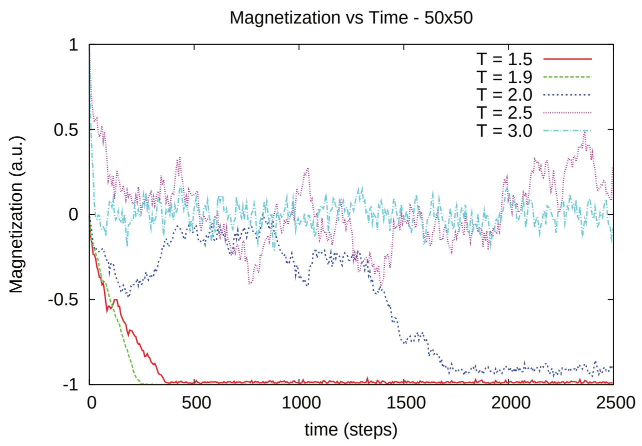

Figure 5. Time evolution of the magnetization per site for a 50×50 lattice.

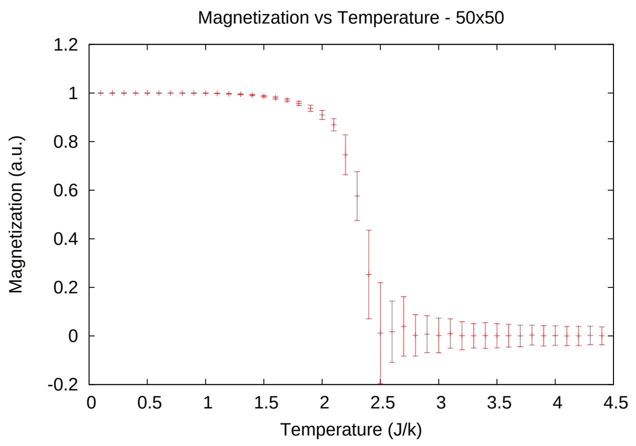

Figure 7. Temperature dependence of the magnetization per site for a 50×50 lattice.

respectively.

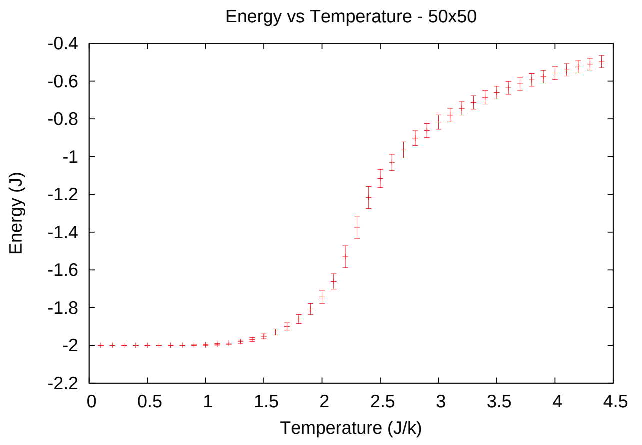

Figure 6. Temperature dependence of the energy per site for a 50×50 lattice.

Task 3. The heat capacity as a function of temperature was examined in two ways for 30×30, 40×40 and 50×50 lattices. It was calculated once as $C = \frac{dE}{dT}$ and a second time using $C = \frac{\sigma_E^2}{k_B T^2}$. The resulting plots from each calculation for the 50×50 lattice are presented in Figure 10 and Figure 11, respectively. Both figures show the same qualitative and quantitative features with the main difference being in the uncertainties of the individual values. As expected, the first approach experiences larger errors due to random fluctuations in the simulation. The shapes of the plots allow for a better estimate of T_C, which is characterized by the discontinuity in the heat capacity. The critical temperature was estimated for each of the two cases giving $T_C = 2.3 \pm 0.1$ and $T_C = 2.25 \pm 0.04$. These values agree well with Onsager's theoretical result. Again, as in the cases of the mean energy and magnetization, the calculated heat capacity exhibits larger uncertainties near

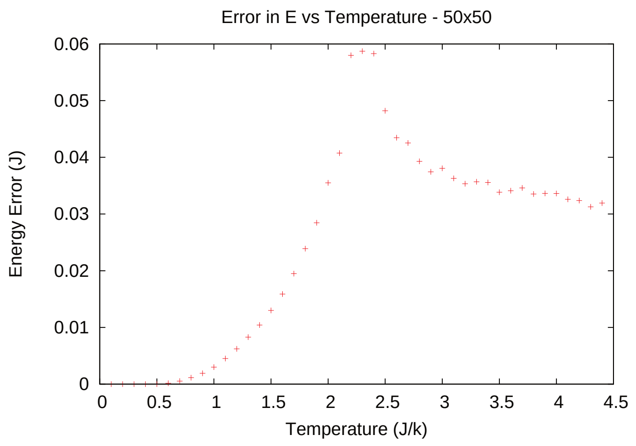

Figure 8. Temperature dependence of the error in the energy per site for a 50×50 lattice.

T_C. The relevant plots of this trait are presented in Figure 12 and Figure 13.

Task 4. The power-law behaviour of the magnetization near the critical temperature, which is predicted by theory, was investigated by simulating the 50×50 system in this region with a smaller temperature step and plotting a logarithmic graph, which is expected to be a straight line with a slope corresponding to the critical exponent. The resulting plot is presented in Figure 14. The best fit line gives $\beta = 0.101 \pm 0.005$. The theoretically expected value is $\beta = 0.125$. The slight discrepancy between the two values comes from the fact that the given error is only due to the spread of the points, while each point carries a significant additional error. If this is taken into account, the theoretical expectation falls inside the error boundaries.

Task 5. In the last stage of the project, the effects of non-zero magnetic field, $H \neq 0$, were investigated. The resulting plots are shown in Figure 15, Figure 16 and Figure 17. As

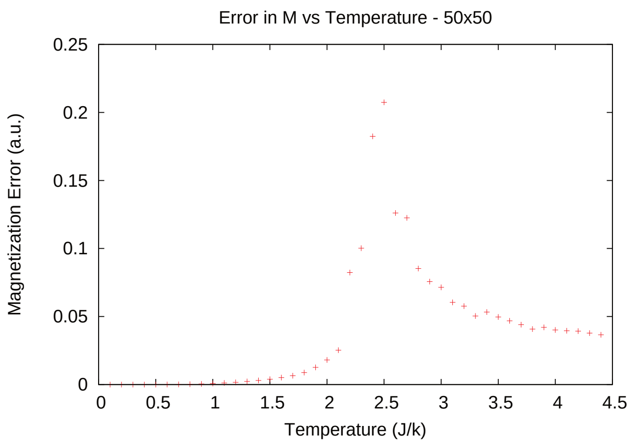

Figure 9. Temperature dependence of the error in the magnetization per site for a 50×50 lattice.

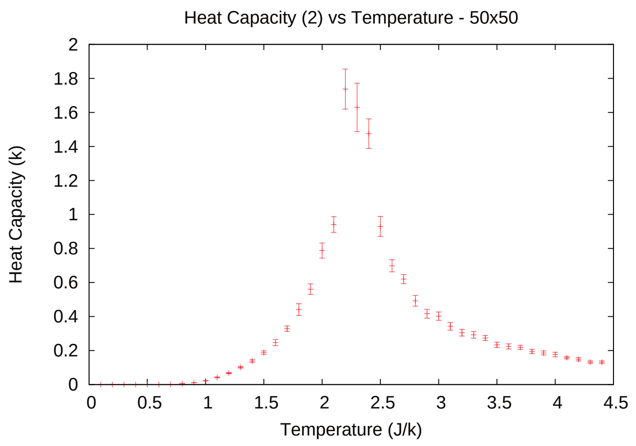

Figure 11. Temperature dependence of the heat capacity (2) per site for a 50×50 lattice.

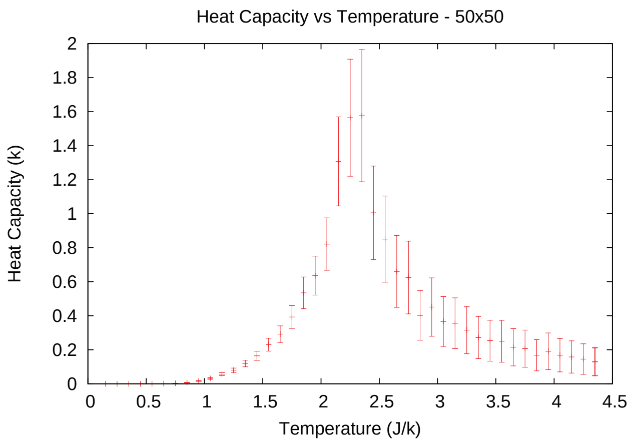

Figure 10. Temperature dependence of the heat capacity (1) per site for a 50×50 lattice.

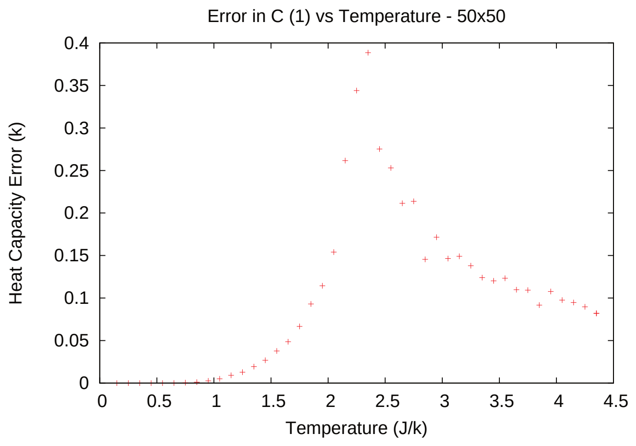

Figure 12. Temperature dependence of the error in the heat capacity (1) per site for a 50×50 lattice.

expected, the presence of a magnetic field decreases the total energy, reduces the sharpness of the phase transition and increases the value of the critical temperature.

4. Conclusion

To summarise, this project created an implementation of the 2D Ising model, using the Metropolis algorithm, which was then employed to investigate the energy, magnetization and heat capacity of the simulated systems. The produced results agreed well with Onsager's theoretical solution.

References

[1] Lars Onsager. Crystal statistics. i. a two-dimensional model with an order-disorder transition. *Phys. Rev.*, 65:117, 1944.

[2] Barry M. McCoy and Tai Tsun Wu. *The Two-dimensional Ising Model*. Harvard University Press.

[3] M. E. J. Newman and G. T. Barkema. *Monte Carlo Methods in Statistical Physics*. Clarendon Press, Oxford.

[4] Steven. E. Koonin. *Computational Physics*. Benjamin/Cummings.

[5] Cristian C. Bordeianu Rubin H. Landau, Manuel J. Paez. *Computational Physics*. Wiley-VCH.

[6] J. M. Thijssen. *Computational Physics*. Cambridge University Press.

[7] Tao Pang. *An Introduction to Computational Physics*. Cambridge University Press.

[8] William T. Vetterling Brian P. Flannery William H. Press, Saul A. Teukolsky. *Numerical Recipies in C, The Art of Scientific Computing*. Cambridge University Press.

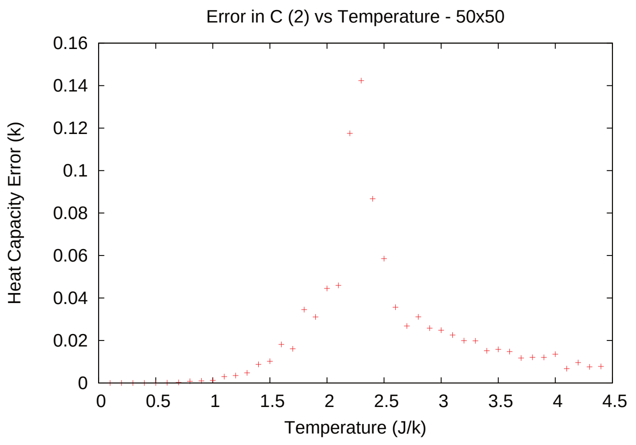

Figure 13. Temperature dependence of the error in the heat capacity (2) per site for a 50×50 lattice.

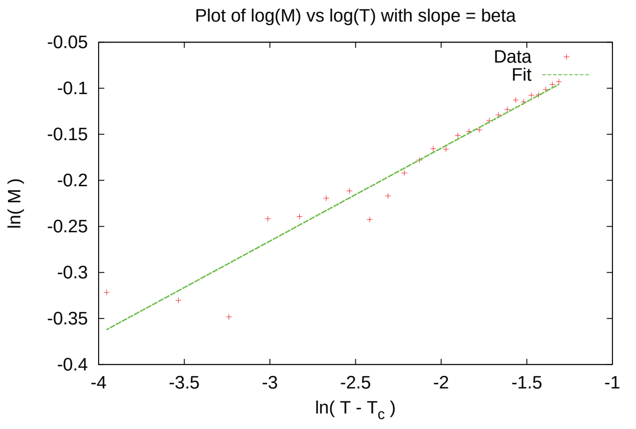

Figure 14. Graph of the logarithm of the magnetization per site against the logarithm of the difference between the temperature and the critical temperature for a 50×50 lattice. The best fit line has slope $\beta = 0.101 \pm 0.005$ that corresponds to the critical exponent in the theoretical power-law prediction $M \propto (T_C - T)^{\beta}$.

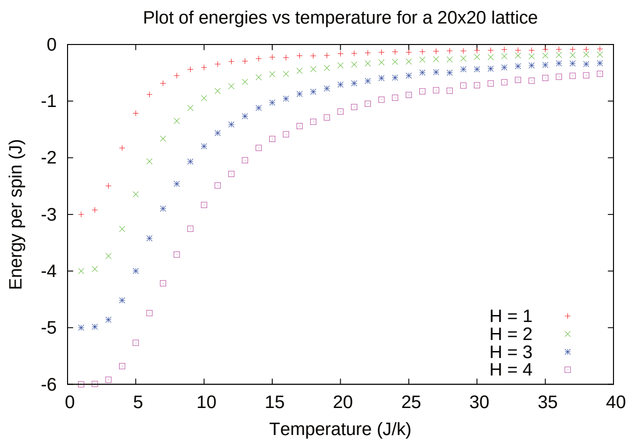

Figure 15. Temperature dependence of the energy per site for a 20×20 lattice in a range of $H \neq 0$.

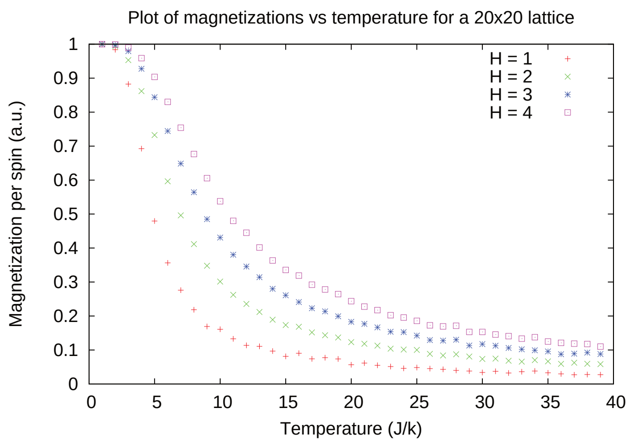

Figure 16. Temperature dependence of the magnetization per site for a 20×20 lattice in a range of $H \neq 0$.

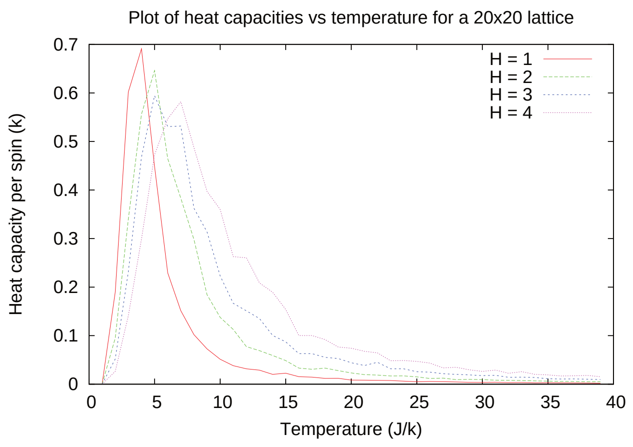

Figure 17. Temperature dependence of the heat capacity per site for a 20×20 lattice in a range of $H \neq 0$.

Code/lattice.hh

```
// lattice.hh

#ifndef LATTICE_HH
#define LATTICE_HH

#include <vector>

using namespace std;

// assume that Boltzman constant k_B = 1 (natural units)
const long J = 1; // exchange energy
const long mu = 1; // magnetic moment

class Lattice
{
        private:
                // spin state of the lattice with
                // each spin being either +1 or -1
                vector<long> m_state;

                long m_N; // dimendion of the N x N lattice
                long m_E; // energy of the current state
                long m_M; // magnetization of the current state
                double m_T; // temperature
                double m_H; // magnetic field

                // Array with all possible values of the acceptance
                // probability - exp(-deltaE / kT). In the absence of a
                // magnetic field there are only five possible values of deltaE
                // from which only three are positive. In the presence of a
                // magnetic field, however, the possible values of deltaE
                // become 10. This still allows to create an array with all
                // possible values of the acceptance probability and save the
                // evaluation of an exponent at every flip step.
                double exp_vals[10];

        public:
                //constructors

                // The constructors also initialize the energy and
                // magnetization for the initial state. The consequential
                // changes are handled by the flip method
                Lattice (long N, double T, double H);
                Lattice (long N, double T, vector<long> state, double H);

                // setters and getters

                void SetT (double T); // the only setter we need
                void SetH (double H);

                vector<long> GetState ();
                long GetN ();
                long GetE ();
                long GetM ();
                double GetT ();
                double GetH ();

                // Methods

                // Function giving the initial energy of the lattice
                long initial_energy (vector<long> data, long N);

                // This method will be used to carry out a single flip-step
                // from the Metropolis algorithm. The call to this method asks
                // for a specific spin to be flipped and the method flips it
                // only if it agrees with the conditions specified in the
                // Metropolis algorithm. The method also updates the values
                // of the energy and magnetization once the spin is flipped.
                void flip (long row, long col, double p);
};

#endif
```

Code/lattice.cc

```
// lattice.cc

#include <cmath>
#include "lattice.hh"

using namespace std;

// Function to apply the periodic boundary condition to an index,
// taking the index and the maximum value, which the index can take
long periodicBC (long index, long max_index)
{
        long x = abs(index);

        if (index >= 0)
        return index % max_index;
        else
        return (max_index - (x % max_index));
}

// Function returning the [row][col] element of a matrix
// with periodic b.c.
long& element (long row, long col, vector<long> data)
{
        long N = (long) sqrt(data.size());
        return data.at( periodicBC(row, N) * N + periodicBC(col, N));
}

// Function giving the initial energy of the lattice
long Lattice::initial_energy (vector<long> data, long N)
{
        long E = 0;

        // nearest neighbour interactions
        for (long i = 0; i < N; i++)
        {
                for (long j = 0; j < N; j++)
                {
                        E -= J * element(i, j, data) * ( element(i, j+1, data)
                                 + element(i+1, j, data) );
                }
        }

        // interaction with external magnetic field
        if (m_H != 0)
        {
                for (long k = 0; k < N*N; k++)
                {
                        E -= mu * m_H * data.at(k);
                }
        }

        return E;
}

// Function giving the initial magnetization of the lattice
long initial_magn (vector<long> data, long N)
{
        long M = 0;

        for (long i = 0; i < N*N; i++)
        {
                M += data.at(i);
        }

        return M;
}

// constructors

Lattice::Lattice (long N, double T, double H)
        : m_state(N*N, 1), m_N(N), m_E(0), m_M(0), m_T(T), m_H(H)
{
        // initialize energy
        m_E = initial_energy( m_state, m_N );

        // initialize magnetization
        m_M = initial_magn( m_state, m_N );

        // Initialize the array of possible values of the
        // acceptance probability. The first 5 values correspond to
        // the case when deltaM<0 and the next 5 to when deltaM>0

        /* Possible values of deltaE
         * i    deltaE1          deltaE2          Neighbours
         * 0    -4*2*J+2*mu*H    -4*2*J-2*mu*H    (all spins -1)
         * 1    -2*2*J+2*mu*H    -2*2*J-2*mu*H    (3 spins -1)
         * 2     0*2*J+2*mu*H     0*2*J-2*mu*H    (2 spins -1)
         * 3    +2*2*J+2*mu*H    +2*2*J-2*mu*H    (3 spins +1)
         * 4    +4*2*J+2*mu*H    +4*2*J-2*mu*H    (all spins +1)
         *
         * ( 2*i - 4 ) corresponds to the possible values of the sum of
         * the four nearest neighbours
         *
         * */

        for (long i = 0; i < 5; i++)
        {
                long deltaE1 = ( 2*i - 4 ) * 2*J + 2*mu*m_H;
                long deltaE2 = ( 2*i - 4 ) * 2*J - 2*mu*m_H;

                exp_vals[i] = exp( -deltaE1 / m_T );
                exp_vals[i+5] = exp( -deltaE2 / m_T );
        }
}

Lattice::Lattice (long N, double T, vector<long> state, double H)
        : m_state(state), m_N(N), m_E(0), m_M(0), m_T(T), m_H(H)
{
        // initialize energy
        m_E = initial_energy( m_state, m_N );

        // initialize magnetization
        m_M = initial_magn( m_state, m_N );

        // Initialize the array of possible values of the
        // acceptance probability. The first 5 values correspond to
        // the case when deltaM<0 and the next 5 to when deltaM>0

        /* Possible values of deltaE
         * i    deltaE1         deltaE2         Neighbours
         * 0    -4*2*J+2*mu*H   -4*2*J-2*mu*H   (all spins -1)
         * 1    -2*2*J+2*mu*H   -2*2*J-2*mu*H   (3 spins -1)
         * 2     0*2*J+2*mu*H    0*2*J-2*mu*H   (2 spins -1)
         * 3    +2*2*J+2*mu*H   +2*2*J-2*mu*H   (3 spins +1)
         * 4    +4*2*J+2*mu*H   +4*2*J-2*mu*H   (all spins +1)
         *
         * ( 2*i - 4 ) corresponds to the possible values of the sum of
         * the four nearest neighbours
         *
         * */

        for (long i = 0; i < 5; i++)
        {
                long deltaE1 = ( 2*i - 4 ) * 2*J + 2*mu*m_H;
                long deltaE2 = ( 2*i - 4 ) * 2*J - 2*mu*m_H;

                exp_vals[i] = exp( -deltaE1 / m_T );
                exp_vals[i+5] = exp( -deltaE2 / m_T );
        }
}

// setters and getters

void Lattice::SetT (double T) { m_T = T; }

void Lattice::SetH (double H) { m_H = H; }

vector<long> Lattice::GetState () { return m_state; }

long Lattice::GetN () { return m_N; }

long Lattice::GetE () { return m_E; }

long Lattice::GetM () { return m_M; }

double Lattice::GetT () { return m_T; }

double Lattice::GetH () { return m_H; }

// Methods
/*
long& operator Lattice::() (long row, long col)
{
        return element(row, col, m_state);
}
* */

void Lattice::flip (long row, long col, double p)
{
        bool flipped = 0;

        // Assign a variable with the value of the element, which
        // is being considered for flipping, and the sum of its
        // nearest neighbours in order to save repeated
        // evaluations of the function element().
        long s = element(row, col, m_state);
        long sum_nn = element(row, col - 1, m_state) +
                                element(row, col + 1, m_state) +
                                element(row - 1, col, m_state) +
                                element(row + 1, col, m_state);

        // deltaM is equal to -2 times the spin of the element
        // before it gets flipped.
        long deltaM = -2 * s;

        // deltaE is equal to 2J times the spin of the element
        // before it gets flipped times the sum of the spins
        // of its nearest neighbours
        long deltaE = 2 * J * s * sum_nn;

        // in presence of external magnetic field the value
        // of deltaE is additionally modified
        if ( m_H != 0 )
        {
                deltaE -= mu * m_H * deltaM;
        }

        // decide whether to flip the spin or not, based on the
        // conditions of the Metropolis algorithm

        if ( deltaE < 0 )
        {
                flipped = 1;
                // flip the spin element
                m_state[ periodicBC(row, m_N) * m_N + periodicBC(col, m_N)] *= -1;
        }
        else if ( ((exp_vals[ (s*sum_nn + 4)/2 ] > p) && ( deltaM <= 0 )) ||
                ((exp_vals[ (s*sum_nn + 4)/2 + 5 ] > p) && ( deltaM >= 0)) )
        {
                flipped = 1;
                // flip the spin element
                m_state[ periodicBC(row, m_N) * m_N + periodicBC(col, m_N)] *= -1;
        }

        // update energy and magnetization of the system
        if ( flipped )
        {
                m_E += deltaE;
                m_M += deltaM;
        }
}
```

Code/simulation.hh

```
// simulation.hh

#ifndef SIMUL_HH
#define SIMUL_HH

#include "lattice.hh"


// Function to extract the quilibrium region from a vector array
// of E or M values. This means to get only the second half of the
// values.
vector<long> extract_equilibrium ( vector<long> data);

// Using the function random_uniform() from /ex/prng5.cc
// to produce random numbers in the interval [0,1] with
// uniform distrubution, using <gsl/gsl_rng.h> from the
// GSL Library. The function is used with its default
// seed in this program.
double random_uniform( unsigned long int seed);

// Function to choose n random values from a vector array.
// This function is used in the implementation of the bootstrap
// method when calculating the error in C = simga^2/kT^2.
vector<long> choose_random ( long n, vector<long> data );

// Function to calculate the mean of an array of integral values
double mean_value(vector<long> data);

// Function to calculate the standard deviation squared
double sigma_sqr_value_int( vector<long> data );

// Function to calculate the standard deviation squared
```

```
double sigma_sqr_value_double( vector<double> data );

// Function to create T=inf initial conditions
vector<long> random_init (long N);

// Function to print a picture of the current spin state
// which to be used then for visualization
void take_picture ( vector<long> state , long at_time , double at_T );

// Function to carry out a whole time step of N^2 flips
void metropolis ( Lattice& latt );

class Simulation
{
        private:

                vector<long> M_values; // values of the magnetization
                vector<long> E_values; // values of the energy

                vector<double> T_values; // values in the range [T_min, T_max]

                vector<double> H_values; // values in the range [H_min, H_max]

                vector<double> M_mean_values; // values of the mean magnetization
                vector<double> M_mean_errors; // errors of the mean magnetization

                vector<double> E_mean_values; // values of the mean energy
                vector<double> E_mean_errors; // errors of the mean energy

                vector<double> C_values_I; // heat capacity values C = dE/dT
                vector<double> C_errors_I; // errors in C = dE/dT
                vector<double> T_values_I; // T values for C = dE/dT

                vector<double> C_values_II; // values of C = sigma^2/kT^2
                vector<double> C_errors_II; // errors of C = sigma^2/kT^2
                vector<double> T_values_II; // T values for C = sigma^2/kT^2

                // Function to simulate the system lattice for max_time steps
                void simulate ( Lattice& lattice , long max_time );

        public:

                // Methods

                // Function to print the data for the current simulation.
                void print_data ( long time_step );

                // Function to print the data for the current simulation
                // per site of the lattice.
                void print_data_per_site ( long time_step , long N );

                // Function to print the the mean values of M and E over some
                // temperature range.
                void print_T_data ();

        // Function to print the the mean values of M and E over some
        // temperature range.
        void print_T_data_per_site ( long N );

        // Function to carry out a simulation of a system with
        // dimension N and temperature T for a number of time steps
        // max_time. The bool init indicates whether the initial
        // configuration of the spins should correspond to T = 0
        // (i.e. all spins in the same direction) or to T -> inf
        // (i.e. all spins are randomly oriented). The two cases
        // correspond to bool = 0 and 1 respectively.
        void evolve_system ( long N, double T, long max_time, bool init );

        // Function to calculate the mean values of the equilibrium
        // energy and magnetization for the current simulation.
        // It assummes that the second half of the interval max_time
        // corresponds to equilibrium.
        void calc_means ();

        // Function to calculate the error in the value of C, which
        // has been calculated by C = dE/dT. The absolute error is
        // simply twice the absolute error of E divided by dT.
        void calc_error_C_I ( double E_mean, double dE, double dT );

        // Function to calculate the error in the value of C, which
        // has been calculated by C = sigma^2/kT^2. The bootstrap
        // method is used to achieve this. (see more in report)
        void calc_error_C_II ( double T );

        // Function to calculate the heat capacity using the first
        // formula from the instruction handout - C = dE/dT. It uses
        // the stored values of the mean energies and their
        // corresponding temperatures.
        void calc_C_I ();

        // Function to calculate the heat capacity using the second
        // formula from the instruction handout - C = sigma^2/kT^2.
        // It uses the stored values of the mean energies and their
        // corresponding temperatures.
        void calc_C_II ( double T );

        // Function to carry out a series of simulations of a system
        // for different temperatures in a range [T_min, T_max] with
        // a step of T_step. For every individual simulation the mean
        // energy and magnetization of the equilibrium state are
        // measured.
        void evolve_in_T ( long N, double T_min, double T_max, double
           T_step, double H );


};


#endif
```

Code/simulation.cc

```
// simulation.cc

#include <iostream>
#include <cmath>
#include "gsl/gsl_rng.h"
#include "simulation.hh"

using namespace std;

// Function to extract the quilibrium region from a vector array
// of E or M values. This means to get only the second half of the
// values.
vector<long> extract_equilibrium ( vector<long> data)
{
        long size = data.size();
        vector<long> result ( data );

        result.erase ( result.begin(), result.begin() + 2*size/3 );

        return result;
}

// Using the function random_uniform() from /ex/prng5.cc
// to produce random numbers in the interval [0,1] with
// uniform distrubution, using <gsl/gsl_rng.h> from the
// GSL Library. The function is used with its default
// seed in this program.
double random_uniform( unsigned long int seed = 0 )
{
  static gsl_rng* rng = 0;

  if ( rng == 0 ) rng = gsl_rng_alloc( gsl_rng_default );
  if ( seed != 0 ) gsl_rng_set( rng, seed );

  return gsl_rng_uniform( rng );
}

// Function to choose n random values from a vector array.
// This function is used in the implementation of the bootstrap
// method when calculating the error in C = simga^2/kT^2.
vector<long> choose_random ( long n, vector<long> data )
{
        long size = data.size();
        long el = 0;
        vector<long> result;

        for ( long i = 0; i < n; i++ )
        {
                el = (long) ( size * random_uniform() );
                result.push_back( data.at( el ) );
        }

        return result;
}
```

```

// This function calculates the mean of an array of integral values
double mean_value(vector<long> data)
{
        long sum = 0;
        long nt = data.size();

        for (long i = 0; i < nt; i++) sum += data.at(i);

        return ((double)sum)/nt;
}

// Function to calculate the standard deviation squared
double sigma_sqr_value_int( vector<long> data )
{
        /*
        long sum = 0, ssum = 0;
        long nt = data.size();

        for (long i = 0; i< nt; i++)
        {
                long el = data.at(i);

                sum += el;
                ssum += el*el;
        }

        double sigma = ((double)ssum)/nt - (((double)sum)/nt)*(((double)sum)/nt);

        return fabs( sigma );
        * */
        double mean = mean_value ( data );
        long size = data.size();

        double sigma_sqr = 0;

        for ( int i = 0; i < size; i++ )
        {
                long el = data.at(i);
                sigma_sqr += ( el - mean ) * ( el - mean );
        }

        sigma_sqr /= size;

        return sigma_sqr;
}

// Function to calculate the standard deviation
double sigma_sqr_value_double( vector<double> data )
{
        double sum = 0, ssum = 0;
        long nt = data.size();

        for (long i = 0; i< nt; i++)
        {
                double el = (double)data.at(i);

                sum += el;
                ssum += el*el;
        }

        double sigma = ssum/nt - (sum/nt)*(sum/nt);

        return sigma;
}

// Function to create T=inf initial conditions
vector<long> random_init (long N)
{
        vector<long> rand;
        double r = 0.0;

        for (long i = 0; i < N*N; i++)
        {
                r = random_uniform();

                if ( r < 0.5 ) rand.push_back( -1 );
                else rand.push_back( 1 );
        }

        return rand;
}

// Function to print a picture of the current spin state
// which to be used then for visualization
void take_picture ( vector<long> state, long at_time, double at_T )
{
        long N = (long) ( sqrt( (double)state.size() ) );

        cout << "System size: " << N << " x " << N << ", at time: "
                 << at_time << ", at temperature: " << at_T << endl;

        for ( long i = 0; i < N; i++ )
        {
                for ( long j = 0; j < N; j++ )
                {
                        cout<< state.at( i*N + j ) <<" ";
                }

                cout << endl;
        }

        cout << endl;
}

// a whole time step of N^2 flips
void metropolis ( Lattice& latt )
{
        long ran_row = 0; // to carry a random row number
        long ran_col = 0; // to carry a ranodm col number
        double p = 0.0; // to carry a random number to decide flipping
        long N = latt.GetN(); // dimension of lattice

        for (long i = 0; i < (N*N); i++)
        {
                ran_row = (long) ( random_uniform() * (double)N );
                ran_col = (long) ( random_uniform() * (double)N );
                p = random_uniform();

                latt.flip(ran_row, ran_col, p);
        }
}

// Function to simulate the system lattice for max_time steps
void Simulation::simulate ( Lattice& lattice, long max_time )
{
        long M = 0;
        long E = 0;

        M_values.clear();
        E_values.clear();

        for (long t = 0; t < max_time; t++)
        {
                M = lattice.GetM();
                E = lattice.GetE();

                /* This snippet was used to print lattice configurations
                 * for visualization. Not needed for ant other activities.

                if ( ( t == 0) || ( t == max_time / 1000 ) ||
                        ( t == max_time / 100 ) || ( t == max_time / 10 )
                        || ( t == (max_time - 1) ) )
                {
                        double T = lattice.GetT();
                        take_picture ( lattice.GetState(), t, T );
                }
                * Note: This must be used only if no other print options
                *                  are specified in the main program.
                * */

                M_values.push_back( M );
                E_values.push_back( E );

                metropolis( lattice );
        }
}

// Function to print the data for the current simulation
void Simulation::print_data ( long time_step )
{
        long size = E_values.size();

        for ( long t = 0; t < size; t += time_step )
        {
                cout<<t<<" "<<E_values.at(t)<<" "<<M_values.at(t)<<endl;
        }
}

// Function to print the data for the current simulation
// per site of the lattice.
void Simulation::print_data_per_site ( long time_step, long N )
{
        long size = E_values.size();

        for ( long t = 0; t < size; t += time_step )
        {
                cout<<t<<" "<<((double)E_values.at(t))/(N*N)<<
                                 " "<<((double)M_values.at(t))/(N*N)<<endl;
        }
}

// Function to print the mean values of M and E over some
// temperature range.
void Simulation::print_T_data ()
{
        long size = T_values.size();

        // T_values_I, C_values_I, C_errors_I have one element less, so
        // we can just add the last one once more to make the
        // printing easier
        T_values_I.push_back( T_values_I.at(size-2) );
        C_values_I.push_back( C_values_I.at(size-2) );
        C_errors_I.push_back( C_errors_I.at(size-2) );

        for ( long i = 0; i < size; i++ )
        {

                cout<< T_values.at(i) <<" "                        // 1)
                        << E_mean_values.at(i) <<" "               // 2)
                        << E_mean_errors.at(i) <<" "               // 3)
                        << fabs(M_mean_values.at(i)) <<" "         // 4)
                        << M_mean_errors.at(i) <<" "               // 5)
                        << T_values_I.at(i) <<" "                  // 6)
                        << C_values_I.at(i) <<" "                  // 7)
                        << C_errors_I.at(i) <<" "                  // 8)
                        << T_values_II.at(i) <<" "                 // 9)
                        << C_values_II.at(i) <<" "                 // 10)
                        << C_errors_II.at(i) <<endl;               // 11)

        }
}

// Function to print the mean values of M and E per site over some
// temperature range.
void Simulation::print_T_data_per_site ( long N )
{
        long size = T_values.size();

        // T_values_I, C_values_I, C_errors_I have one element less, so
        // we can just add the last one once more to make the
        // printing easier
        T_values_I.push_back( T_values_I.at(size-2) );
        C_values_I.push_back( C_values_I.at(size-2) );
        C_errors_I.push_back( C_errors_I.at(size-2) );

        for ( long i = 0; i < size; i++ )
        {

        cout<< T_values.at(i) <<" "                                      // 1)
                << E_mean_values.at(i) / (N*N)<<" "                      // 2)
                << E_mean_errors.at(i) / (N*N)<<" "                      // 3)
                << fabs(M_mean_values.at(i)) / (N*N)<<" "                // 4)
                << M_mean_errors.at(i) / (N*N)<<" "                      // 5)
                << T_values_I.at(i) <<" "                                // 6)
                << C_values_I.at(i) / (N*N)<<" "                         // 7)
                << C_errors_I.at(i) / (N*N)<<" "                         // 8)
                << T_values_II.at(i) <<" "                               // 9)
                << C_values_II.at(i) / (N*N) <<" "                       // 10)
                << C_errors_II.at(i) / (N*N)<<endl;                      // 11)

        }
}

// Function to calculate the mean values of the equilibrium
// energy and magnetization for the current simulation.
// It assummes that the second half of the interval max_time
// corresponds to equilibrium.
void Simulation::calc_means ()
{
        // equilibrium regions of E and M
        vector<long> eq_M_values ( extract_equilibrium( M_values ) );
        vector<long> eq_E_values ( extract_equilibrium( E_values ) );

        // calculate the means and add them to the corresponding vectors
        double mean_M = mean_value ( eq_M_values );
        double mean_E = mean_value ( eq_E_values );

        // calculate the errors in the means
        double error_M = sqrt ( sigma_sqr_value_int( eq_M_values ) );
        double error_E = sqrt ( sigma_sqr_value_int( eq_E_values ) );

        // record the mean values
        M_mean_values.push_back ( mean_M );
        E_mean_values.push_back ( mean_E );

        // record the errors in the means
        M_mean_errors.push_back ( error_M );
        E_mean_errors.push_back ( error_E );
}

// Function to calculate the error in the value of C, which
// has been calculated by C = dE/dT. The relative error is
// simply the relative error of E divided by dT.
void Simulation::calc_error_C_I ( double E_mean, double dE, double dT )
```

```
{
        // equilibrium region of E values
        vector<long> eq_E_values ( extract_equilibrium( E_values ) );

        // absolute error in the equilibrium value of E
        double sigma = sqrt( sigma_sqr_value_int( eq_E_values ) );

        // relative error in E
        double R_E = sigma / E_mean;

        // relative error in C
        double R_C = ( R_E / dT );

        // absolute error in C
        double error = R_C * ( dE / dT );

        C_errors_I.push_back( error );
}

// Function to calculate the error in the value of C, which
// has been calculated by C = sigma^2/kT^2. The bootstrap
// method is used to achieve this. (see more in report)
void Simulation::calc_error_C_II ( double T )
{
        // equilibrium region of E values
        vector<long> eq_E_values ( extract_equilibrium( E_values ) );

        // number of resamplings to be done
        const long N_resample = 20;

        // 1/fraction of the whole array to be used at each resampling
        const long fraq = 4; // i.e. 1/4 of the size of the array

        // size of the array to be resampled
        long size = eq_E_values.size();

        // number of entries corresponding to the fraction specified above
        long n = size / fraq;

        // vector array to contain the calculated C values after each
        // resampling cycle
        vector<double> c_values;

        // do the resamplings
        for ( long i = 0; i < N_resample; i++ )
        {
                vector<long> sample ( choose_random( n, eq_E_values ) );
                double sigma_sqr = sigma_sqr_value_int(sample);

                c_values.push_back( sigma_sqr / (T*T) );
        }

        double sigma = sqrt ( sigma_sqr_value_double( c_values ) );

        C_errors_II.push_back( sigma );
}


// Function to calculate the heat capacity using the first
// formula from the instruction handout - C = dE/dT. It uses
// the stored values of the mean energies and their
// corresponding temperatures. The method consists in taking
// the difference of two neighbouring energies and dividing
// it by the difference of the two corresponding neighbouring
// temperatures. The resulting value is the heat capacity at the
// average of the two temperatures.
void Simulation::calc_C_I ()
{
        long size = T_values.size();

        double dE = 0;
        double dT = 0;

        double E_mean = 0;

        for ( long i = 0; i < (size - 1); i++ )
        {
                dE = fabs(E_mean_values.at(i+1) - E_mean_values.at(i));
                dT = T_values.at(i+1) - T_values.at(i);

                C_values_I.push_back( dE/dT );
                T_values_I.push_back( (T_values.at(i+1) + T_values.at(i)) / 2 );

                E_mean = ( E_mean_values.at(i+1) + E_mean_values.at(i) ) / 2;

                calc_error_C_I( E_mean, dE, dT );
        }
}

// Function to calculate the heat capacity using the second
// formula from the instruction handout - C = sigma^2/kT^2.
// It uses the stored values of the mean energies and their
// corresponding temperatures.
void Simulation::calc_C_II ( double T )
{
        vector<long> eq_E_values ( extract_equilibrium ( E_values ) );

        double sigma_sqr = sigma_sqr_value_int( eq_E_values );

        C_values_II.push_back ( sigma_sqr / ( T*T ) );
        T_values_II.push_back ( T );

        calc_error_C_II ( T );
}

// Function to carry out a simulation of a system with
// dimension N and temperature T for a number of time steps
// max_time. The bool init indicates whether the initial
// configuration of the spins should correspond to T = 0
// (i.e. all spins in the same direction) or to T -> inf
// (i.e. all spins are randomly oriented). The two cases
// correspond to bool = 0 and 1 respectively.
void Simulation::evolve_system ( long N, double T, long max_time, bool init )
{
        Lattice lattice (N, T, 0);

        if ( init ) lattice = Lattice( N, T, random_init(N), 0 );

        simulate ( lattice, max_time );
}

// Function to carry out a series of simulations of a system
// for different temperatures in a range [T_min, T_max] with
// a step of T_step. For every individual simulation the mean
// energy and magnetization of the equilibrium state are
// measured. Also, the heat capacity is estimated in the two
// ways suggested in the instruction handout.
//NOTE: At every temperature step, the simulated system is
// initialized with the equilibrium lattice state of the system
// from the previous step. This helps for a much faster achievement
// of equilibrium compared to the case where the initial state
// is T=0 or T->inf at every step.
void Simulation::evolve_in_T ( long N, double T_min, double T_max, double T_step,
    double H )
{
        // The time to reach equilibrium should not be more than N*N
        // because by that time all spins will have flipped at least once
        // on average. Experimentation shows that this time is usually
        // less than N*N/2 and if you add the fact that at each step
        // the starting configuration is the equilibrium state of the
        // previous step, max_time = N*N should be more than enough to
        // have equilibrium at least during the second half of it.
        // The final version of the code even uses 2.5 N*N.

        long max_time = 2.5 * N * N; // equilibrium during the second half

        long steps = (long)( (T_max - T_min) / T_step ); // number of steps

        Lattice lattice ( N, T_min, H ); // initialize ( try random_init(N) )

        T_values.clear(); // to be safe

        for ( long i = 0; i < steps; i++ )
        {
                double T = T_min + i*T_step; // step forward

                // use the lattice configuration reached by the system
                // from the previous iteration to achieve equilibrium faster

                lattice = Lattice( N, T, lattice.GetState(), H );

                simulate ( lattice, max_time );

                // take note of the current T

```

```
			T_values.push_back( T );

			// Calculate the heat capacity for the current T

			calc_C_II( T );

			// Measure mean energy and magnetization

			calc_means();
		}

		// Calculate heat capacity

		calc_C_I();
}
```

Code/main.cc

```
// main.cc

#include <ctime>
#include "simulation.hh"
#include "cavlib/string_util.hh"

//————————————————————————————————————————————————————————————————————————————//

void performance_test1 ( long N_min, long N_max, long N_step, double T, long
    max_time, bool init )
{
		using namespace std;

		clock_t begin, end; // to mark the begin and end times
		long steps = ( N_max - N_min ) / N_step; // number of steps

		for ( int i = 0; i < steps; i++ )
		{
			long N = N_min + i*N_step; // step forward
			cout << N << " ";
			begin = clock(); // take initial time

			Simulation sim;
			sim.evolve_system( N, T, max_time, init );

			end = clock(); // take final time
			double elapsed_secs = double(end - begin) / CLOCKS_PER_SEC;
			cout << elapsed_secs << endl;
		}
}

void performance_test2 ( long N_min, long N_max, long N_step,
							double T_min, double T_max, double
								T_step)
{
		using namespace std;

		clock_t begin, end; // to mark the begin and end times
```

```
        long steps = ( N_max - N_min ) / N_step; // number of steps

        for ( int i = 0; i < steps; i++ )
        {
                long N = N_min + i*N_step; // step forward
                cout << N << " ";
                begin = clock(); // take initial time

                Simulation sim;
                sim.evolve_in_T ( N, T_min, T_max, T_step, 0 );

                end = clock(); // take final time
                double elapsed_secs = double(end - begin) / CLOCKS_PER_SEC;
                cout << elapsed_secs << endl;
        }
}
//-------------------------------------------------------------------------//

int main( int argc, char* argv[])
{
        if ( cav::string_to_int(argv[1]) == 1 )
        {
        // To perform time evolution of a specified system. Related to
        // Task 1. from the instructions handout.
                long N = cav::string_to_int( argv[2] );
                double T = cav::string_to_double( argv[3] );
                long max_time = cav::string_to_int( argv[4] );
                bool init = (bool) cav::string_to_int( argv[5] );

                Simulation sim;

                sim.evolve_system( N, T, max_time, init );
                sim.print_data( 1 + max_time / 500 );

                // uncomment this if you want to print the same things
                // but with their values per spin
                //sim.print_data_per_site( 1 + max_time / 500, N );

        } else if ( cav::string_to_int(argv[1]) == 2 )
        {
        // To perform temperature evolution of a specified system.
        // Related to Tasks 2., 3. and 4. from the instructions
        // handout. Produce temperature dependence of various quantities.
                long N = cav::string_to_int( argv[2] );
                double T_min = cav::string_to_double( argv[3] );
                double T_max = cav::string_to_double( argv[4] );
                double T_step = cav::string_to_double( argv[5] );

                Simulation sim;

                sim.evolve_in_T ( N, T_min, T_max, T_step, 0 );
                sim.print_T_data ();

                // uncomment this if you want to print the same things
                // but with their values per spin
                //sim.print_T_data_per_site( N );

        } else if ( cav::string_to_int(argv[1]) == 3 )
        {
        // Same as the previous option, just this time you can also
        // specify a magnetic field H in the last parameter. Related to
        // Task 5. from the instructions handout.
                long N = cav::string_to_int( argv[2] );
                double T_min = cav::string_to_double( argv[3] );
                double T_max = cav::string_to_double( argv[4] );
                double T_step = cav::string_to_double( argv[5] );
                double H = cav::string_to_double( argv[6] );

                Simulation sim;

                sim.evolve_in_T ( N, T_min, T_max, T_step, H );
                sim.print_T_data ();

                // uncomment this if you want to print the same things
                // but with their values per spin
                //sim.print_T_data_per_site( N );

        } else if ( cav::string_to_int(argv[1]) == 4 )
        {
        // Performance test 1
                long N_min = cav::string_to_int( argv[2] );
                long N_max = cav::string_to_int( argv[3] );
                long N_step = cav::string_to_int( argv[4] );
                double T = cav::string_to_double( argv[5] );
                long max_time = cav::string_to_int( argv[6] );
                bool init = (bool) cav::string_to_int( argv[7] );

                performance_test1 ( N_min, N_max, N_step, T, max_time, init );

        } else if ( cav::string_to_int(argv[1]) == 5 )
        {
        // Performance test 2
                long N_min = cav::string_to_int( argv[2] );
                long N_max = cav::string_to_int( argv[3] );
                long N_step = cav::string_to_int( argv[4] );
                double T_min = cav::string_to_double( argv[5] );
                double T_max = cav::string_to_double( argv[6] );
                double T_step = cav::string_to_double( argv[7] );

                performance_test2 ( N_min, N_max, N_step, T_min, T_max, T_step );

        } else cout<<" Error: invalid input "<<endl;

        return 0;
}
```