Using Monte-Carlo and Metropolis methods

2D Ising model simulation

Nováki Lóránt

Further information:
Neptun code: TK3EJ9
Submission date: January 8, 2026

1 Introduction

During my experiment, I was studying the properties of the 2D Ising model, in different magnetic fields. The 2D Ising model is an exceptional tool in studying the behaviors of large scale systems owing to its simplicity and stochastic behavior.

2 Experimental setup and tasks

2.1 Experimental setup

I performed the simulation with the help of a `C++` code, that I have developed. In order to combine the `C++` language's fast calculating properties for computing heavy tasks, and to use the prewritten visual modules `Python 3.11.8`, I have used `pybind11` which I used to wrap the `C++` code into a Python module. The `C++` code simulating the system can be seen in section 6. Although the `C++` performs the more basic calculations, like magnetic field, hamiltonian in each time step, the time averageing calculations after reaching steady state, and the fittings were too implemented in python. This code is handed in alongside this document.

For the programming part, I have used `Visual Studio Code`, and a `12th Gen Intel(R) Core(TM) i3-12100F, 3.30 GHz` processor, sometimes a less powerful one. As it turns out, computation power is not the main challenge in the completion of these tasks. Especially for a 10x10 lattice.

2.2 Tasks

For completion I was given these tasks:

1. Use a square lattice of size $L \times L = 10 \times 10$ with periodic boundary conditions.
2. The Hamiltonian is the following:
$$H = -K \sum_{i,j} \sigma_i \sigma_j - h \sum_i \sigma_i$$
Set the parameters as $K = 1$, $k_B T = 5$.
3. Assign each lattice site j a random spin $\sigma_j = \pm 1$.
4. A Monte Carlo time step consists of $L \times L$ elementary steps in which a spin is chosen randomly and is flipped with the Metropolis probabilities.
5. Measure the Monte Carlo correlation time τ from the exponential decay of the average magnetization starting from the ferromagnetic "all spins up" configuration.
6. Modify the external field h in the range $h \in [0, 10]$ and plot the $M(h)$ curve.
7. Measure the susceptibility using
$$\chi = \left. \frac{\partial M}{\partial h} \right|_{h=0}.$$
Hint: Use small fields.
8. Verify the susceptibility from the fluctuations of the magnetization:
$$\chi = \frac{\langle M^2 \rangle - \langle M \rangle^2}{k_B T}.$$

Their solution can be found in section 4.

3 Methods

The Hamiltonian of the general 2D Ising model has the general form of the following:

$$H(\sigma) = -\sum_{\langle ij \rangle} J_{ij} \sigma_i \sigma_j - \mu \sum_j B_j \sigma_j \tag{3.1}$$

For an isotropic system we can use a non-element dependent coupling constant J, and for a homogenous field, we can also use a constant B. Normalizing the Hamiltonian by $k_B T$ we get a scaleless form, where $K = J/k_B T$ and $h = \mu B / k_B T$:

$$\mathcal{H}(\sigma) = -K \sum_{\langle ij \rangle} \sigma_i \sigma_j - h \sum_j \sigma_j \equiv \frac{H}{k_b T} \tag{3.2}$$

This function will determine the energy of the system in a computer simulation. To determine where the system will be in the next timestep, I have used the Metropolis algorithm. This works the following way: The algorithm chooses a spin σ_i in the lattice. Then it determines whether changing its state to $-\sigma_i$ would satisfy the detailed balance with the following acceptance probability $A(\mu, \nu)$:

$$A(\mu, \nu) = \begin{cases} e^{-\beta(H_\nu - H_\mu)}, & \text{if } H_\nu - H_\mu > 0, \\ 1 & \text{otherwise.} \end{cases} \tag{3.3}$$

where the μ, ν states in our case represent the $\sigma_i \to -\sigma_i$ state transition. This acceptance probability can be phrased more concisely in our case:

$$A(\sigma_i \to \sigma_i) = e^{-\beta(\Delta H)} \quad \text{if } \Delta H > 0 \tag{3.4}$$

Which means, the system will only NOT change state with the acceptance probability defined above. In each timestep we choose N spins with uniform probability, and decide with the acceptance probability whether to change it.

In this paper, I also refer to the magnetization of the system, which is:

$$M = \sum_i \sigma_i = N \cdot m \tag{3.5}$$

where m is the average magnetization.

4 Results

In this section I will show my results and solutions to tasks given. I will often refer to the simulator, which is a program that can be found in section 6.

Task 1: Lattice creation and boundary conditions

In the simulator's constructor (PYBIND11_MODULE) I have given the default value of rows and columns 10, as asked, but I did not want to have them as constant, to leave my program scalable.

To have a periodic condition, I have defined the function "getSpin" which will satisfy the boundary condition when called.

Task 2: Define the Hamiltonian

I have defined the Hamiltonian according to 3.2. equation. As before, I have set the default parameter values as it has been asked to $K = 1$ and $k_B T = 5$.

Task 3: Initialize the spins

The "initialize" function takes care of this task. It accepts 3 commands:

1. The "UP" command initializes each lattice j with value $\sigma_j = +1$.
2. The "DOWN" command initializes each lattice j with value $\sigma_j = -1$.
3. The "RANDOM" command initializes each lattice j with a random value $\sigma_j = \pm 1$

Task 4: Monte Carlo time step

I have successfully implemented the Metropolis algorithm in the function "Metropolis". It is used in both "run_external_output" (outputting state records in a text file) and "run_numpy_output" (returning state records as a numpy array) which functions run the system for the defined runtime in "time_steps".

In each timestep, simulator calls the Metropolis function on N lattice sites (probably some lattice sites will be called more than once, and some never), and then the simulator stores the energy and average magnetisation, in the vectors "energy_record" (callable with "get_energy_record") and in "magnetization_record" (callable with "get_magnetization_record") respectively.

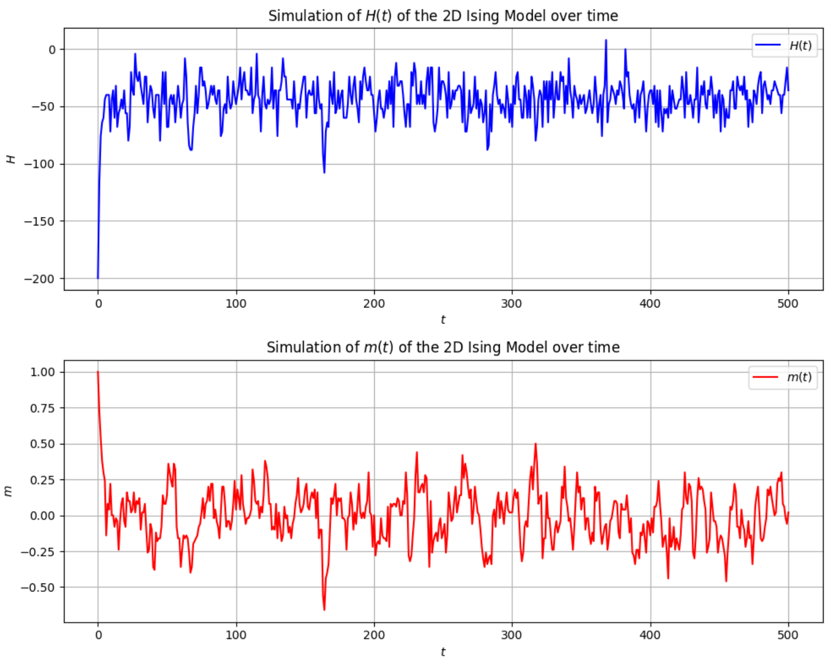

Figure 4.1: Behaviour of the energy and the average magnetisation in time, simulated from the "UP" initialstate

During the implementation I had a small confusion, because notation used in class and in this task were a different, because based on lecture notation, the given $\mathcal{H}$ function would be the unit less function, so there would be no need for division in the Metropolis algorithm by the k_BT factor. But if I calculate with that function, it would not leave the ordered state. So I have assumed, it is the original H function, and $K = J$, $h = \mu B$.

Task 5: Correlation time measurement

To measure the correlation time of the system, I have initialised the system from the "UP" position. I have plotted both the average magnetisation, and the energy against time, and fitted both of them with exponential functions.

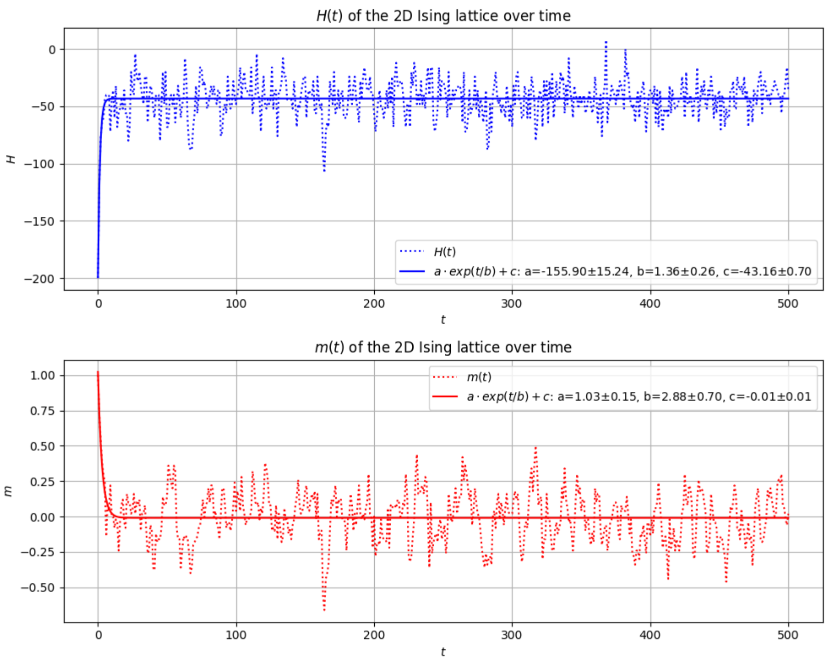

Figure 4.2: Exponential fit on the datapoints previously seen on fig 4.1.

For the Monte Carlo correlation time I have measured:

$$\tau_H = (1.3620 \pm 0.2628)\ s$$
$$\tau_m = (2.8782 \pm 0.7024)\ s$$

Task 6: External field dependence

I have modified the external field h in the range $h \in [0, 10]$ and plotted the $M(h)$ curve (see fig 4.3).

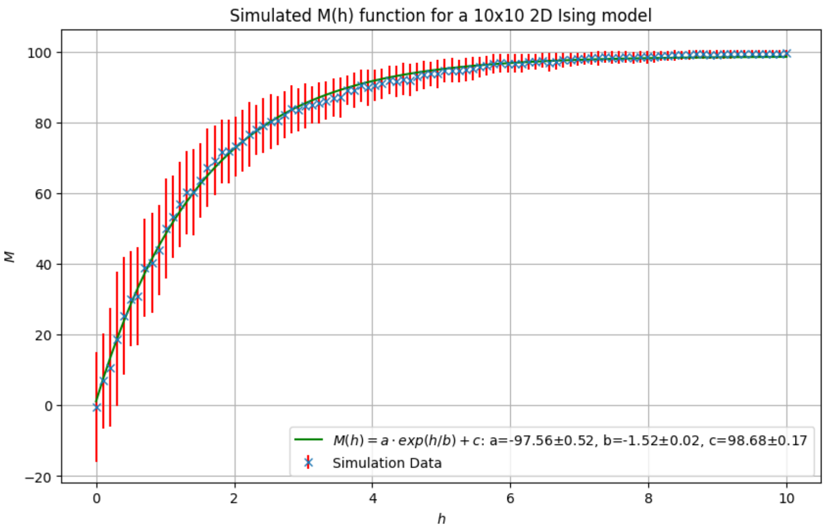

Figure 4.3: The $M(h)$ curve in the range $h \in [0, 10]$

It can be seen the the exponential function fits it quite well.

Task 7 and 8: Susceptibility from derivative and from fluctuations

I have measured the susceptibility using two methods:

$$\chi_\partial = \left.\frac{\partial M}{\partial h}\right|_{h=0} = 56.613 \pm 2.497$$

and

$$\bar{\chi} = \frac{\langle M^2 \rangle - \langle M \rangle^2}{k_B T} = 56.99 \pm 16.88$$

The derivative, was based on the previously fitted $M(h)$ curve, with the assumption, that the theoretized exponantial approximation is a good choice when trying to find the continious function describing $M(h)$ (which, at least on this interval seems to hold, and therefore usable to find the derivative).

As for the second formula, both averages are time averages of the magnetization at of course different magnetic fields, on time intervals where the system reached statistical equilibrium.

Fortunatly, both of them are inside each other's error bounds. χ_∂ was derived by calculating the averages of the magnetisation values for a longer period of time (500 time steps) and by varying $h \in [-0.1, 0.1]$. On the resulting datapoints I have performed a linear fit as can be seen on fig 4.4. The error of $\bar{\chi}$ was estimated by the standard deviation of the simulated M values.

Out of curiosity I have compared χ_∂, and $\bar{\chi}$ on the scale $h \in [0, 10]$. For $\bar{\chi}$ I have used the datapoints used in the previous task, and χ_∂ I have calculated the derivate of the fitted function $M(h) = a \cdot exp(h/b) + c$. The result (fig 4.5.) showed promising overlaps between the two curves.

5 Discussion

The results of the simulation demonstrate that the Monte Carlo method combined with the Metropolis algorithm is capable of reproducing the expected physical behavior of the 2D Ising model under external magnetic fields.

From the correlation time analysis (Task 5), both the magnetization and the energy showed exponential decay starting from the ferromagnetic initial condition. The measured values of τ_H and τ_m are within the same order of magnitude, which indicates that both observables provide a consistent characterization of the system's

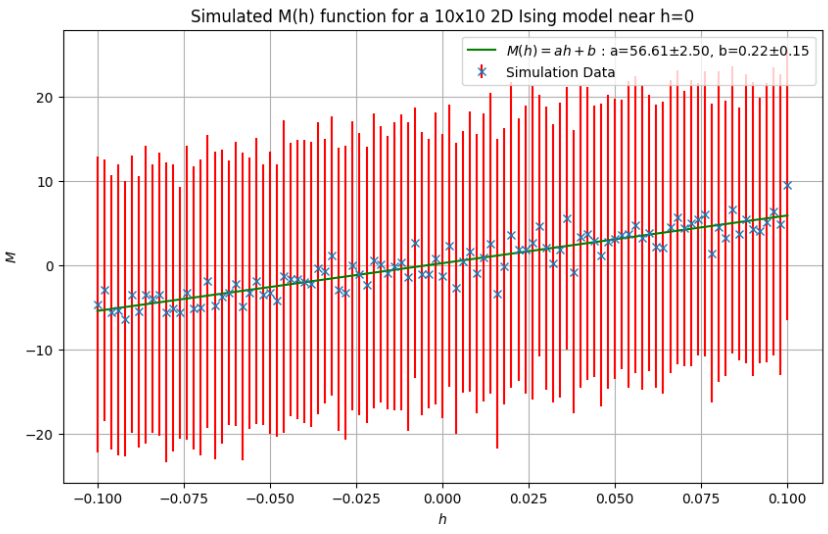

Figure 4.4: χ_∂ derivation with linear fit, performed on the averages of the magnetisation values for a longer period of time (500 time steps) and by varying $h \in [-0.1, 0.1]$.

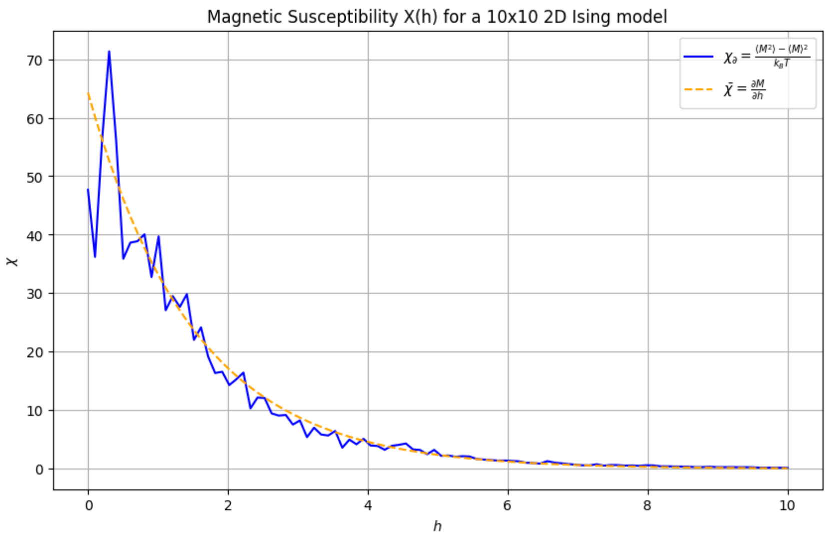

Figure 4.5: Comparison of χ_∂, and $\bar{\chi}$ on the scale $h \in [0, 10]$

relaxation dynamics. The longer correlation time of the magnetization suggests that spin alignment responds more slowly to fluctuations compared to the total energy, which is consistent with physical intuition.

The $M(h)$ curve (Task 6) clearly shows a monotonic increase in the magnetization as the external field h is strengthened. The exponential fit captures this behavior well, reflecting that even in a finite-size lattice, the magnetization aligns strongly with the applied field. Although no sharp phase transition is observed due to the limited lattice size (10×10), the qualitative behavior matches expectations.

The susceptibility analysis (Tasks 7 and 8) provides further confirmation of the validity of the simulation. The derivative method and the fluctuation–dissipation relation produced results consistent within their respective error bounds:

$$\chi_\partial = 56.61 \pm 2.50, \qquad \bar{\chi} = 56.99 \pm 16.88.$$

The larger uncertainty in the fluctuation-based method can be attributed to statistical noise and finite sampling time. Nevertheless, the agreement between the two approaches highlights the correctness of both the simulation algorithm and the statistical analysis.

Overall, the simulation reproduced the expected thermodynamic properties of the 2D Ising model, even within the limitations of a relatively small lattice. The main sources of error are finite-size effects and limited sampling time. Increasing the lattice size and simulation length would reduce statistical noise and better approximate the thermodynamic limit. Despite these limitations, the results confirm the consistency between theoretical predictions and numerical experiments, and demonstrate the effectiveness of Monte Carlo methods in statistical physics.

6 Simulation program

!!! Disclaimer: The program can only be used with `pybind11` **!!!**

```cpp
#include <iostream>
#include <vector>
#include <fstream>
#include <cstdlib>
#include <ctime>
#include <cmath>
#include <random>

// Pybind11 headers
#include <pybind11/pybind11.h>
#include <pybind11/numpy.h>

namespace py = pybind11;
using namespace std;

class MCIsing {
    public:
    const string file_name;
    string state;
    const int rows, cols;
    const double K, k_BT, h;
    vector<vector<int>> grid;
    vector<double> energy_record;
    vector<double> magnetization_record;
    int seed_random;
    mt19937 mt;
        MCIsing(int r, int c, string state = "RANDOM", double K = 1, double k_BT = 5, double h =
        0, int seed = time(0), string file_name_ = "output.txt")
        : rows(r), cols(c),
          grid(r, vector<int>(c, 0)), K(K), k_BT(k_BT), h(h), state(state), file_name(file_name_),
          mt(seed) {
        initialize(state);
    }

    void initialize(string state = "RANDOM") {
        energy_record.clear();
        magnetization_record.clear();
        for (int i = 0; i < rows; ++i)
            for (int j = 0; j < cols; ++j) {
                if (state == "UP") grid[i][j] = 1;
                else if (state == "DOWN") grid[i][j] = -1;
                else grid[i][j] = (mt() % 2 == 0) ? 1 : -1;
            }
        energy_record.push_back(H());
        magnetization_record.push_back(M());
    }

    int getSpin(int i, int j) {
        if (i < 0) i = rows - 1;
        else if (i >= rows) i = 0;
        if (j < 0) j = cols - 1;
        else if (j >= cols) j = 0;
        return grid[i][j];
    }

    void step() {
        for (int i = 0; i < rows * cols; ++i) {
            int r = mt() % rows;
            int c = mt() % cols;
            Metropolis(r, c);
        }
    }

    void display(ofstream& file) {
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                if (j == cols - 1) file << grid[i][j];
                else file << grid[i][j] << "\t";
            }
            file << '\n';
        }
    }

    void run_external_output(int time_steps) {
        ofstream file(file_name);
        if (!file.is_open()) {
            cerr << "Error opening file: " << file_name << endl;
            return;
        }
        for (int t = 0; t < time_steps; ++t) {
            step();
            display(file);
            energy_record.push_back(H());
            magnetization_record.push_back(M());
        }
        file.close();
    }

    py::array_t<int> run_numpy_output(int time_steps) {
        auto result = py::array_t<int>({time_steps, rows, cols});
        auto states = result.mutable_unchecked<3>();
        for (int t = 0; t < time_steps; ++t) {
            step();
            // Copy the current state of the grid into the numpy array
            for (int i = 0; i < rows; ++i) {
                for (int j = 0; j < cols; ++j) {
                    states(t, i, j) = grid[i][j];
                }
            }
            energy_record.push_back(H());
            magnetization_record.push_back(M());
        }
        return result;
    }

    py::array_t<double> get_energy_record() {
        auto result = py::array_t<double>(energy_record.size());
        auto r = result.mutable_unchecked<1>();
        for (size_t i = 0; i < energy_record.size(); ++i) {
            r(i) = energy_record[i];
        }
        return result;
    }

    py::array_t<double> get_magnetization_record() {
        auto result = py::array_t<double>(magnetization_record.size());
        auto r = result.mutable_unchecked<1>();
        for (size_t i = 0; i < magnetization_record.size(); ++i) {
            r(i) = magnetization_record[i];
        }
        return result;
    }

```

```cpp
    double H() {
        int adjacentSum = 0;
        int totalSum = 0;
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                adjacentSum += grid[i][j] * (getSpin(i+1, j) + getSpin(i-1, j) + getSpin(i, j+1) +
                getSpin(i, j-1));
                totalSum += grid[i][j];
            }
        }
        return -K * adjacentSum / 2.0 - h * totalSum;
    }

    double M() {
        int totalSum = 0;
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                totalSum += grid[i][j];
            }
        }
        return (double)totalSum / (rows * cols);
    }

    void Metropolis(int i, int j) {
        int relevantSum = getSpin(i+1, j) + getSpin(i-1, j) + getSpin(i, j+1) + getSpin(i, j-1);
        double deltaE = 2 * grid[i][j] * (K * relevantSum + h);
        uniform_real_distribution<double> dist(0.0, 1.0);
        if (deltaE <= 0 || dist(mt) < exp(-deltaE / k_BT)) {
            grid[i][j] *= -1;
        }
    }

    private:

};


PYBIND11_MODULE(simulator, m) {
    m.doc() = "Monte Carlo Ising Model Simulator using Metropolis Algorithm";
    py::class_<MCIsing>(m, "MCIsing")
        .def(py::init<int, int, string, double, double, double, int, string>(),
             py::arg("rows") = 10,
             py::arg("cols") = 10,
             py::arg("state") = "RANDOM",
             py::arg("K") = 1,
             py::arg("k_BT") = 5,
             py::arg("h") = 0,
             py::arg("seed") = time(0),
             py::arg("file_name") = "output.txt",
             "Initialize the MCIsing simulator with given parameters.\n\nParameters:\n----------
             \nrows : int\ncols : int\nstate : str\nK : float\nk_BT : float\nh : float\nseed :
             int\nfile_name : str\n")
        .def("step", &MCIsing::step)
        .def("initialize", &MCIsing::initialize, py::arg("state") = "RANDOM")
        .def("run_numpy_output", &MCIsing::run_numpy_output, py::arg("time_steps"))
        .def("run_external_output", &MCIsing::run_external_output, py::arg("time_steps"))
        .def("get_energy_record", &MCIsing::get_energy_record)
        .def("get_magnetization_record", &MCIsing::get_magnetization_record);
}
```