Computational Physics - Exercise 4: Monte Carlo methods

Torben Steegmann Chih-Chun Kuo

Introduction

The Monte Carlo method is a computational technique used to approximate numerical solutions through random sampling. It's particularly useful for problems with many variables or states where deterministic methods are impractical, for example the percolation problem, fluid flow problems and problems in statistical physics.

The Ising model is used in statistical physics to describe the behavior of magnetic spins in a material. However, if we have N spins in the Ising model, we will have 2^N possible spin configurations (states), which means it is inefficient to analyze by brute force. Therefore, Monte Carlo simulations can be computationally more efficient than deterministic methods for these kind of large systems.

To simulate the Ising model using Monte Carlo methods, we employ a technique called the Metropolis Monte Carlo algorithm. In this approach, the system evolves through a series of random changes in spin configurations, and the acceptance of these changes is determined probabilistically based on the change in energy of the system. By repeating this process many times and averaging over the results, one can obtain statistical information about the behavior of the system, for example the average energy of a spin.

Simulation model and method

Background The Ising model is a mathematical representation of ferromagnetism in statistical physics. This model comprises a lattice, where each lattice point represents the magnetic dipole moment of atomic spins. These spins can be oriented either 'up' (+1) or 'down' (-1). The general Hamiltonian of the Ising model can be written as:

$$H = -\sum_{<ij>} J_{ij} S_i S_j - \mu \sum_j h_j S_j.$$

Here, < ij > denotes any two adjacent points on the lattice configuration, S denotes the spin for each point, μ represents the magnetic moment, and h_j represents an external magnetic field. To simplify the simulation, we assume h=0 and J=1. Therefore, the model used in our simulation becomes

$$H = -\sum_{<ij>} S_i S_j.$$

From the representation we know that if we have N lattice points, we will have 2^N states, which means it would not be time-efficient to explore all possible configurations exhaustively to get statistical quantity such as average energy or average specific heat.

Therefore, we adapt the Metropolis Monte Carlo (MMC) method to simulate it more efficiently. According to the theory of Markov chains, a stationary state is always attainable after a sufficient number of iterations of the Markov process. Markov processes describe stochastic systems where future states depend solely on the current state. Therefore, by conducting our simulations over a substantial number of steps, the Markov chain gradually converges to its equilibrium state. From this stabilized state, we can derive estimations for statistical quantities, including average energy or specific heat, by analyzing the sampled states. The definitions and simulation approximations of the quantities as results of our simulation are as follows:

1. The internal energy,

$$U = \frac{\sum_{\{S_1,\ldots,S_N\}} E e^{-\beta E}}{\sum_{\{S_1,\ldots,S_N\}} e^{-\beta E}} \approx \frac{1}{\#\Omega} \sum_{\{S_1,\ldots,S_N\} \in \Omega} E(S_1, \ldots, S_N),$$

where E represents the energy of a state, β is the inverse of the temperature T and Ω is the set of configurations generated by the MMC algorithm.

2. The specific heat C at temperature $T = \frac{1}{\beta}$ is given by

$$\begin{aligned} C &= \beta^2 \left[\frac{\sum_{\{S_1,\dots,S_N\}} E^2 e^{-\beta E}}{\sum_{\{S_1,\dots,S_N\}} e^{-\beta E}} - U^2 \right] \\ &\approx \beta^2 \left[\frac{1}{\#\Omega} \sum_{\{S_1,\dots,S_N\} \in \Omega} E^2(S_1, \dots, S_N) - U^2 \right]. \end{aligned}$$

3. The average magnetization for the 2D-Lattice is:

$$\begin{aligned} M &= \frac{\sum_{\{S_{1,1},\dots,S_{N,N}\}} \sum_{i,j=1}^{N} S_{i,j} e^{-\beta E}}{\sum_{\{S_{1,1},\dots,S_{N,N}\}} e^{-\beta E}} \\ &= \frac{\sum_{\{S_{1,1},\dots,S_{N,N}\} \in \Omega} \sum_{i,j=1}^{N} S_{i,j}}{\#\Omega} \end{aligned}$$

For the 1D-Lattice, we can then compare the internal energy per spin and specific heat per spin to their theoretical values:

$$\frac{U}{N} = -\frac{N-1}{N} \tanh \beta$$

$$\frac{C}{N} = \frac{N-1}{N} \left(\frac{\beta}{\cosh \beta} \right)^2$$

For the 2D-Lattice, we can compare the resulting magnetization with the theoretical results.

$$\frac{M}{N^2} = \begin{cases} (1 - \sinh^{-4} 2\beta)^{\frac{1}{8}} & \text{if } T < T_C \\ 0 & \text{if } T > T_C \end{cases}$$

Implementation The implementations for the 1D and 2D lattices generally work quite similar. We begin by defining the the lattice and sample sizes. Then we generate the lattice with random spins of ± 1, after which we can already calculate the initial energy of this system according to the given formulas. The next step offers the choice of starting with relaxation steps, where we flip spins without taking the energy of the state of the system into account in our final averaging yet. This can achieve more accurate results but for our 1D-lattice we omit this step as starting from a high temperature and gradually decreasing it while using the configuration of the previous temperature as the starting configuration of the next temperature already achieves great accuracy. On the other hand we do use $N_{\text{samples}} \cdot 0.1$ relaxation steps for the 2D lattice. Each step in the MMC we pick a random spin, and calculate its overall weight contribution to the system. We calculate 'E_prime' as the energy difference it would make if we did flip that spin. Next, we calculate a value q as $q = e^{-\beta \cdot E_{\text{prime}}}$. We compare this q to a random value between 0 and 1. The flip is thus more likely to happen for negative values of E_{prime} and it is more likely the smaller E_{prime} is. Simultaneously, it is also more likely to happen for smaller Temperatures, as $\beta = \frac{1}{T}$. We repeat this as many times as we have spins in the lattice, each time choosing a new random spin. At the end of each sample we add the current value for E to E_{sum} and E^2 to $E_{\text{sum squared}}$, except if it is a relaxation step. We repeat this process N_{samples} amount of times and divide the resulting E_{sum} and $E_{\text{sum squared}}$ by N_{samples} to get the average over all samples. We calculate the values of U, C and M and move on to the next temperature where we continue with the same spin configuration to avoid long relaxation periods. We repeat this process for each combination of lattice and sample sizes we want to plot. Finally we compute some theoretical values for comparison and then plot the graphs.

Simulation results

1-D result

The result of our simulation is plotted in graphs Figure 1 - Figure 4. Figure 1 and Figure 2 depict the average energy per spin compared to the theoretical result of $N = 1000$. As the theoretical result does not change drastically between different lattice sizes we do not include other lattice sizes for clarity. Figure 3 and Figure 4 depict the average heat of a spin at a given temperature. Again we omit displaying different theoretical graphs for clarity. Generally we can observe more accurate results the more samples we take. We also observe smoother graphs the bigger the lattice is.

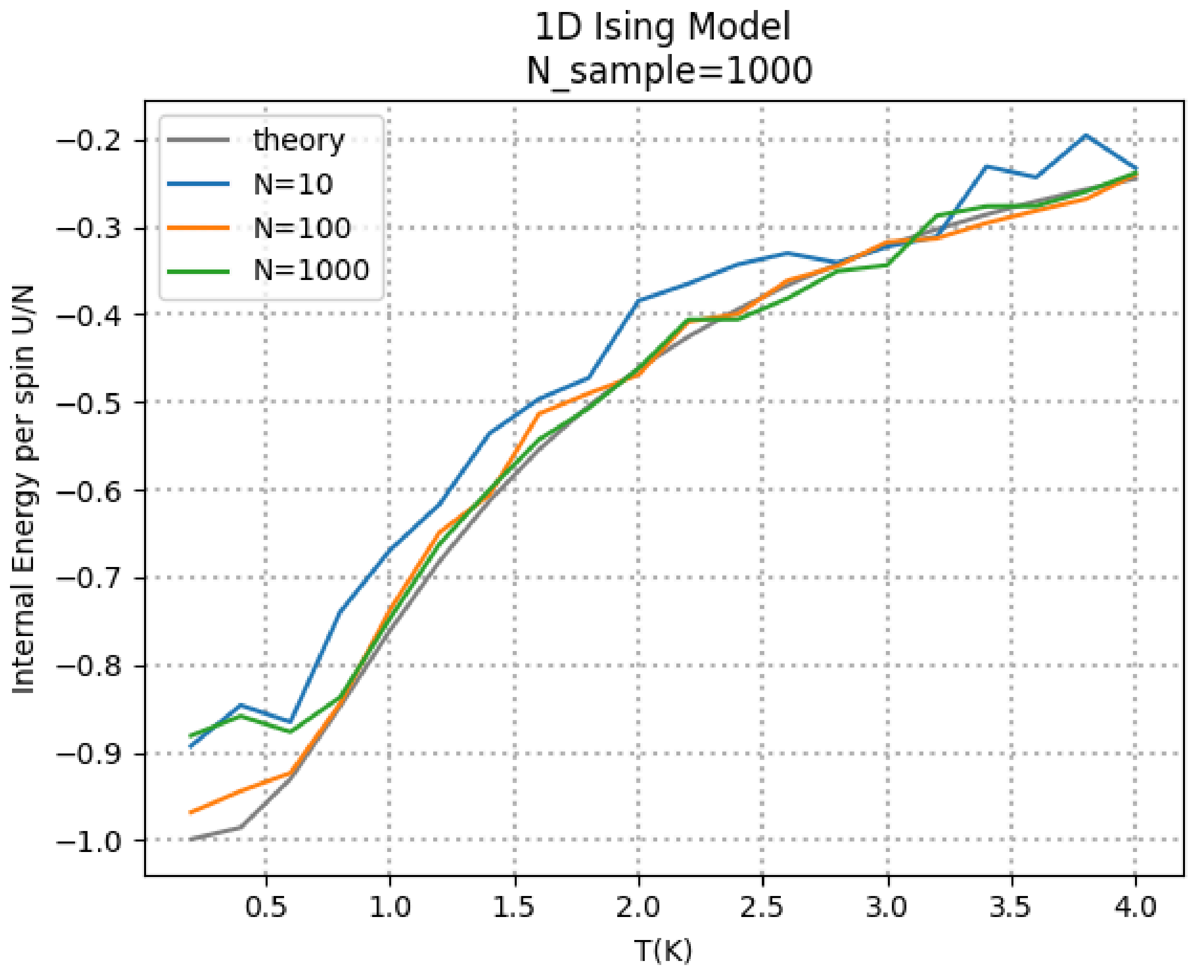

Figure 1: The average energy of a spin depending on the temperature in the Ising Model for different 1D-lattice sizes N and $N_{\text{Sample}} = 1000$. The x-axis displays the temperature of the configuration from 0.2 to 4.0 while the y-axis measures the average energy per spin $\frac{U}{N}$. We see strong fluctuations especially for small N.

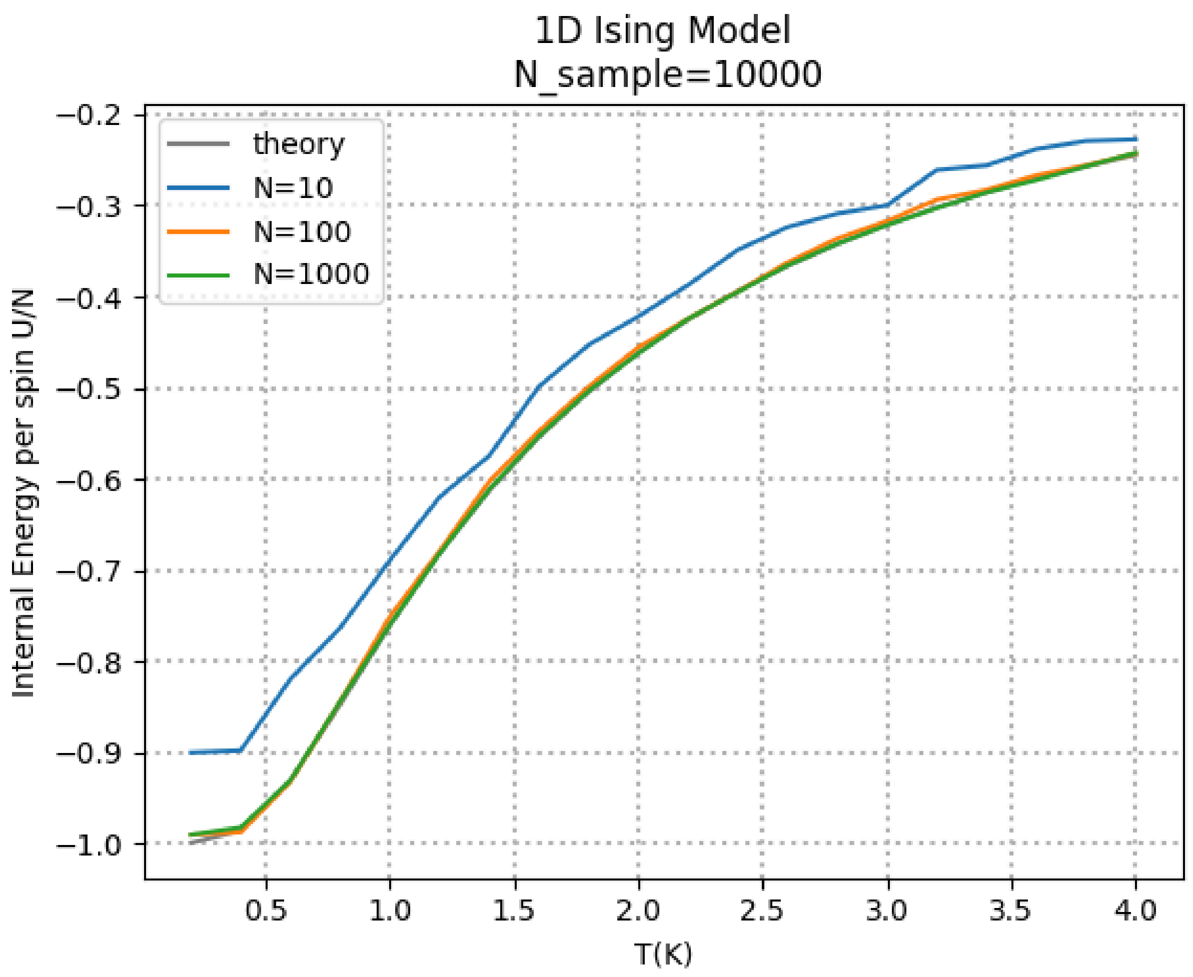

Figure 2: The average energy of a spin depending on the temperature in the Ising Model for different 1D-lattice sizes N and $N_{\text{Sample}} = 10000$. The x-axis displays the temperature of the configuration from 0.2 to 4.0 while the y-axis measures the average energy per spin $\frac{U}{N}$. We see smoother curves than for $N_{\text{Sample}} = 1000$ especially for larger N.

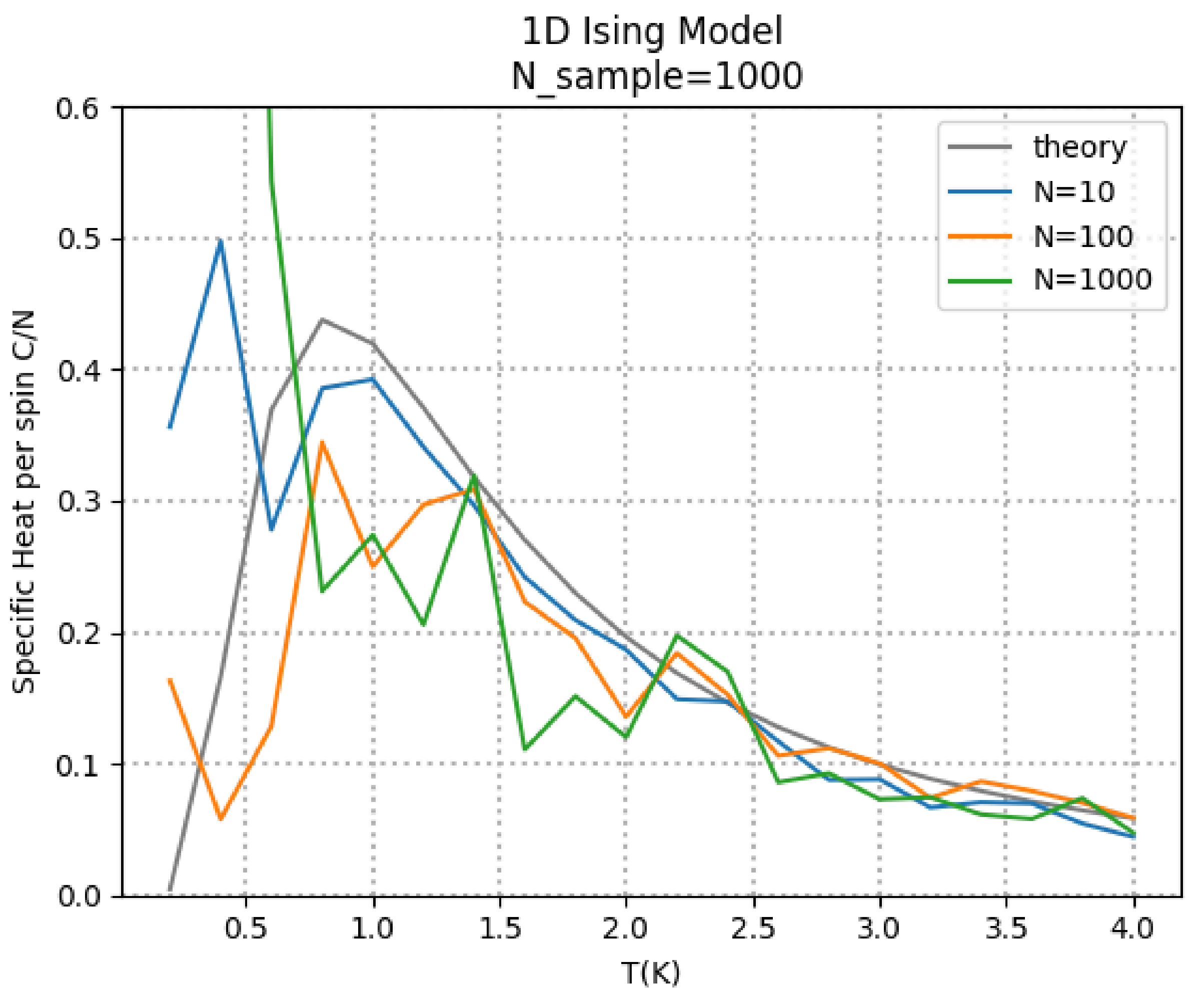

Figure 3: The average heat of a spin depending on the temperature in the Ising Model for different 1D-lattice sizes N and $N_{\text{Sample}} = 1000$. The x-axis displays the temperature of the configuration from 0.2 to 4.0 while the y-axis measures the average energy per spin $\frac{C}{N}$. We see strong fluctuations in the curves which are introduced by the inherent randomness of the simulation method.

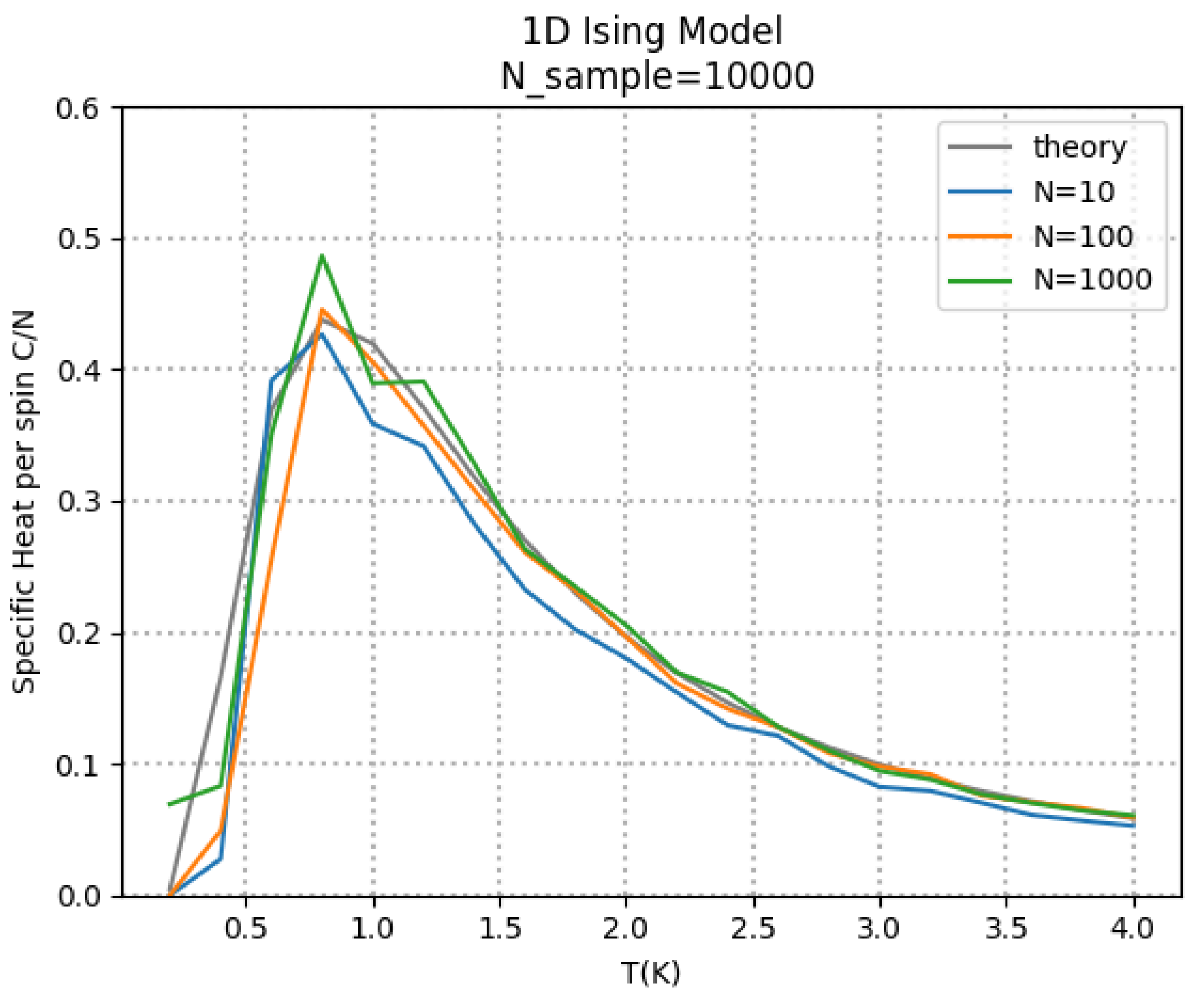

Figure 4: The average heat of a spin depending on the temperature in the Ising Model for different 1D-lattice sizes N and $N_{\text{Sample}} = 10000$. The x-axis displays the temperature of the configuration from 0.2 to 4.0 while the y-axis measures the average energy per spin $\frac{C}{N}$. As a higher sample size counteracts the randomness we see Smooth Curves that follow the theoretical result closely.

2-D result

We plot the 2-D lattice results similar to the 1D-lattice ones. In Figure 5 we can see that we get better results, the bigger the lattice and sample sizes are. For small lattice sizes, even small changes can have a big impact on the system, making the resulting graph unsmooth.

In Figure 6 we can see that the average heat spikes at about $T = 2.2$. We also see that the average heat around that values is steadily at above 0 for most graphs. The bigger the lattice is, the bigger the smaller the energy at lower temperatures is going to be. This results in the exponential heat decrease near $T = 0.5$.

In Figure 7 we see that we follow the theoretical results closely for most configurations with the only two obvious outliers being $N = 10, N_{\text{Samples}} = 1000$ and $N = 100, N_{\text{Samples}} = 1000$. The former does not get as close to a value $\frac{M}{N^2}$ as other configurations. This can be explained by the small lattice and sample size. The latter configuration does not converge towards 1 but -1 instead. This is not an error, as the absolute value tells us whether or not the spins align. The sign of the value only dictates the direction of the alignment. In our simulation this alignment is random. Another notable curve is that of $N = 100, N_{\text{Samples}} = 10000$. This curve follows the analytical result exceptionally well.

It is also worth mentioning that these accuracy improvements came at the cost of computational speed. Larger lattice and sample sizes are much slower to compute than their smaller counterparts. Given that we can see in Figure 7 that the lattice reaches a state of equilibrium we can see that further increases to sample sizes would lead to diminishing returns in accuracy but further increase runtime.

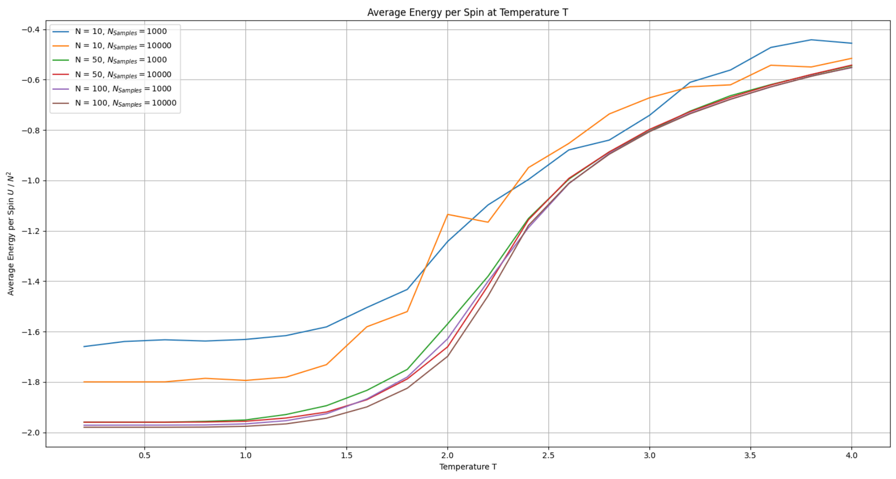

Figure 5: The average energy of a spin depending on the temperature in the Ising Model for different 2D-lattice sizes N and different sampling sizes N_{sample}. The x-axis displays the temperature of the configuration from 0.2 to 4.0 while the y-axis measures the average energy per spin $\frac{U}{N^2}$

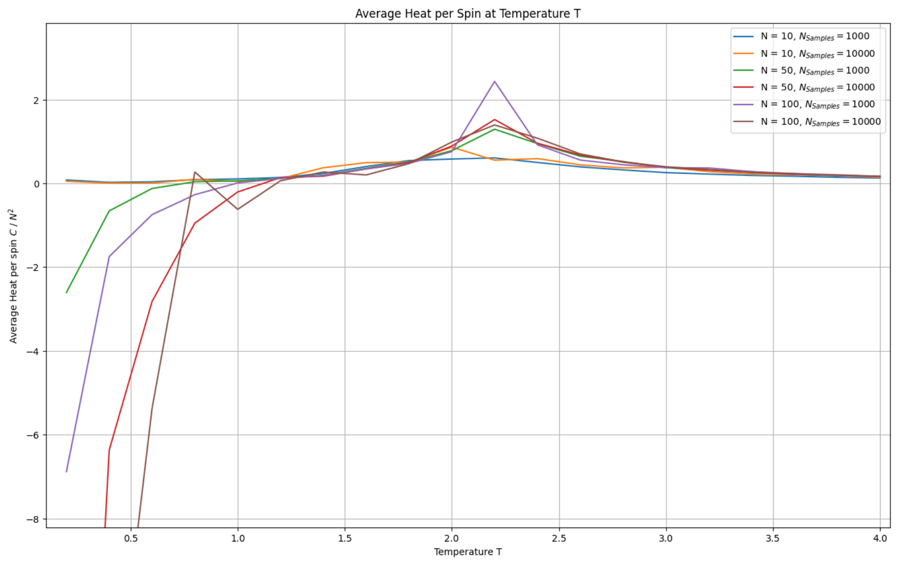

Figure 6: The average Heat of a spin depending on the temperature in the Ising Model for different 2D-lattice sizes N and different sampling sizes N_{sample}. The x-axis displays the temperature of the configuration from 0.2 to 4.0 while the y-axis measures the average energy per spin $\frac{C}{N^2}$.

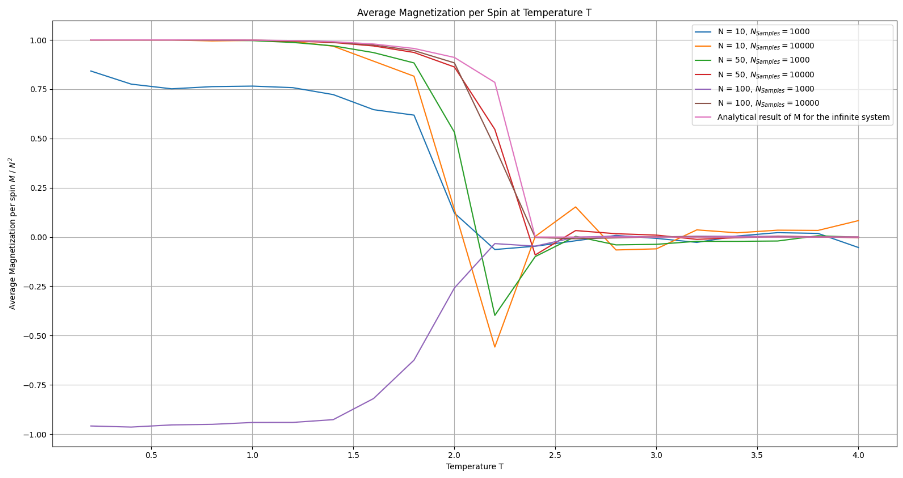

Figure 7: The average magnetization of a spin depending on the temperature in the Ising Model for different 2D-lattice sizes N and different sampling sizes N_{sample}. The x-axis displays the temperature of the configuration from 0.2 to 4.0 while the y-axis measures the average energy per spin $\frac{M}{N^2}$.

Discussion

During the relaxation processes, the configuration attempts to align the spins to the same orientation to minimize the energy and reach the equilibrium state. However, due to thermal fluctuations, implemented by our random process for selecting the index for flipping, some random flips disrupt the system, consequently increasing the energy. This phenomenon is more likely to occur at higher temperatures. Hence, we observe in both the 1D and 2D cases that the average energy is higher at higher temperatures.

We can also observe how thermal fluctuations affect the results of specific heat and magnetization from the plot. At around T=2.3, we observe a rapid rise in energy, a sharp drop in magnetization, and a peak in specific heat. This value corresponds to the critical temperature

$$T_C = \frac{2}{\ln\left(1+\sqrt{2}\right)} \approx 2.27.$$

When $T > T_C$, thermal fluctuations dominate the system and prevent it from aligning the spins to the same orientation. Therefore, the energy rises, and the magnetization transitions from being more organized (+1, −1) to being more chaotic (0). During the phase transition from $T < T_C$ to $T > T_C$, the system needs to consume a lot of energy to counteract the change in spins. Hence, we observe a peak in the specific heat.

In conclusion, in this simulation we observed that we can closely replicate analytical results of statistical physics with the MMC algorithm. We saw that the accuracy follows from the lattice and sample sizes. While increasing them increases the run time, it also increases the accuracy. Through the magnetization we also observed that the lattice reaches a state of Equilibrium given enough samples. This shows that increasing the sample sizes to arbitrary values yields diminishing returns for accuracy, while increasing the computational runtime.

Appendix

```cpp
#include <cmath>
#include <iostream>
#include <vector>

#define WITHOUT_NUMPY
#include "matplotlibcpp.h"

double const EulerConstant = std::exp(1.0);

using namespace std;

// helper struct
struct results
{
    std::vector<float> U;
    std::vector<float> M;
    std::vector<float> C;
};
namespace task_1
{
// main simulation for the 1D-Lattice
results simulation(double N, double N_Samples)
{
    results r;
    // make random spin config
    std::vector<float> spins(N);
    for (auto& element : spins)
    {
        element = -1 + 2 * (std::rand() % 2);
    }
    // calc E of configuration
    auto E = 0;
    for (int j = 0; j < N - 1; ++j)
        E += spins[j] * spins[j + 1];
    E *= -1;

    // iterate over temperature, double inacc doesnt matter here
    double T = 4.0;
    for (; T > 0; T -= 0.2)
    {
        // recalc beta
        float Beta = 1 / T;
        float E_sum = 0;
        float E_sum_squared = 0;
        // calc for one temperature
        for (int k = 0; k < N_Samples; ++k)
        {
            for (int i = 0; i < N; ++i)
            {
                 int j = (int)((static_cast<float>(rand()) / (static_cast<float>(RAND_MAX))) *
(N - 1));
                int E_prime = 0;
                int weight_prior = 0;
                int weight_after = 0;
                // special weighting for first and last element
                if (j == 0)
                {
                    weight_prior = spins[j] * spins[j + 1];
                    weight_after = (-1 * spins[j]) * spins[j + 1];
                }
                else if (j == N - 1)
                {
                    weight_prior = spins[j - 1] * spins[j];
                    weight_after = spins[j - 1] * (-1 * spins[j]);
                }
                else
                {
                    weight_prior = spins[j - 1] * spins[j] + spins[j] * spins[j + 1];
                     weight_after = spins[j - 1] * (-1 * spins[j]) + (-1 * spins[j]) * spins[j
+ 1];
                }
                E_prime = -weight_after + weight_prior;
                auto q = std::pow(EulerConstant, -1 * Beta * E_prime);
                if (q > (static_cast<float>(rand()) / (static_cast<float>(RAND_MAX))))
                {
                    spins[j] *= -1;
                    E = E + E_prime;
                    continue;
                }
            }
            // add E to sum
            E_sum += E;
            E_sum_squared += (E * E);
        }
        // average over all sample iterations
        double U = E_sum / N_Samples;
        double C = (Beta * Beta) * ((E_sum_squared / N_Samples) - (U * U));
        // average over all spins
        U /= N;
        C /= N;
        // add to result to plot later
        r.U.emplace_back(U);
        r.C.emplace_back(C);
    }
    return r;
}


void task_1()
{
    // run simulations for different lattice and sample sizes
    results r1 = simulation(10, 1000);
    results r2 = simulation(10, 10000);
    results r3 = simulation(100, 1000);
    results r4 = simulation(100, 10000);
    results r5 = simulation(1000, 1000);
    results r6 = simulation(1000, 10000);

    // calc theoretical values
    std::vector<float> T_plot;
    std::vector<float> U_theory_plot;
    std::vector<float> C_theory_plot;
    double T = 4.0;
    for (; T > 0; T -= 0.2)
    {
        double beta = 1.0 / T;
        T_plot.emplace_back(T);
        double U_Theory = -((1000. - 1.) / (1000.)) * std::tanh(beta);
          double C_Theory = ((1000. - 1.) / (1000.)) * ((beta / std::cosh(beta)) * (beta /
std::cosh(beta)));
        U_theory_plot.emplace_back(U_Theory);
        C_theory_plot.emplace_back(C_Theory);
    }
    // plot it out
    namespace plt = matplotlibcpp;
    plt::plot(T_plot, r1.U, {{"label", "N = " + std::to_string((int)10) + ", $N_{Samples} = $"
+ std::to_string((int)1000)}});
    plt::plot(T_plot, r2.U, {{"label", "N = " + std::to_string((int)10) + ", $N_{Samples} = $"
+ std::to_string((int)10000)}});
    plt::plot(T_plot, r3.U, {{"label", "N = " + std::to_string((int)100) + ", $N_{Samples} = $"
+ std::to_string((int)1000)}});
    plt::plot(T_plot, r4.U, {{"label", "N = " + std::to_string((int)100) + ", $N_{Samples} = $"
+ std::to_string((int)10000)}});
    plt::plot(T_plot, r5.U, {{"label", "N = " + std::to_string((int)1000) + ", $N_{Samples} =
$" + std::to_string((int)1000)}});
    plt::plot(T_plot, r6.U, {{"label", "N = " + std::to_string((int)1000) + ", $N_{Samples} =
$" + std::to_string((int)10000)}});
    plt::plot(T_plot, U_theory_plot, {{"label", "U_Theory for N = 1000"}});
    // plot settings
    plt::title("Average Energy per Spin at Temperature T");
    plt::xlabel("Temperature T");
    plt::ylabel("Average Heat per spin $U$ / $N$");
    plt::grid(true);
    plt::legend();
    plt::show();

    plt::plot(T_plot, r1.C, {{"label", "N = " + std::to_string((int)10) + ", $N_{Samples} = $"
+ std::to_string((int)1000)}});
    plt::plot(T_plot, r2.C, {{"label", "N = " + std::to_string((int)10) + ", $N_{Samples} = $"
+ std::to_string((int)10000)}});
    plt::plot(T_plot, r3.C, {{"label", "N = " + std::to_string((int)100) + ", $N_{Samples} = $"
+ std::to_string((int)1000)}});
    plt::plot(T_plot, r4.C, {{"label", "N = " + std::to_string((int)100) + ", $N_{Samples} = $"
+ std::to_string((int)10000)}});
    plt::plot(T_plot, r5.C, {{"label", "N = " + std::to_string((int)1000) + ", $N_{Samples} =
$" + std::to_string((int)1000)}});
    plt::plot(T_plot, r6.C, {{"label", "N = " + std::to_string((int)1000) + ", $N_{Samples} =
$" + std::to_string((int)10000)}});
    plt::plot(T_plot, C_theory_plot, {{"label", "C_Theory for N = 1000"}});
    // plot settings
    plt::title("Average Heat per Spin at Temperature T");
    plt::xlabel("Temperature T");
    plt::ylabel("Average Heat per spin $C$ / $N^2$");
    plt::grid(true);
    plt::legend();
    plt::show();
}
} // namespace task_1

namespace task_2
{
// main simulation loop for 2D-Lattice
results simulation(double N, double N_Samples)
{
    results r;
    double N_Relaxation = N_Samples * 0.1;
    double M = 0;
    // make random spin config
    std::vector<std::vector<int>> spins(N, std::vector<int>(N));
    for (auto& row : spins)
    {
        for (auto& element : row)
        {
            element = -1 + 2 * (std::rand() % 2);
            M += element;
        }
    }
    // calc E of configuration
    int E_left = 0;
    for (int i = 0; i < N - 1; ++i)
        for (int j = 0; j < N; ++j)
            E_left += spins[i][j] * spins[i + 1][j];
    int E_right = 0;
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N - 1; ++j)
            E_right += spins[i][j] * spins[i][j + 1];
    int E = -E_left - E_right;

    // Iterate over Temperature
    double T = 4.0;
    for (; T > 0; T -= 0.2)
    {
        // reset values
        float Beta = 1 / T;
        float E_sum = 0;
        float E_sum_squared = 0;
        double M_sum = 0;
        // calc for one temperature
        for (int k = 0; k < N_Samples + N_Relaxation; ++k)
        {
            for (int n = 0; n < N * N; ++n)
            {
                int i = (int)((static_cast<float>(rand()) / (static_cast<float>(RAND_MAX))) *
(N - 1));
                int j = (int)((static_cast<float>(rand()) / (static_cast<float>(RAND_MAX))) *
(N - 1));

                int weight_prior = 0;
                int weight_after = 0;
                if (i + 1 < N)
                {
                    weight_prior -= spins[i][j] * spins[i + 1][j];
                    weight_after -= -spins[i][j] * spins[i + 1][j];
                }
                if (i - 1 >= 0)
                {
                    weight_prior -= spins[i - 1][j] * spins[i][j];
                    weight_after -= spins[i - 1][j] * -spins[i][j];
                }
                if (j + 1 < N)
                {
                    weight_prior -= spins[i][j] * spins[i][j + 1];
                    weight_after -= -spins[i][j] * spins[i][j + 1];
                }
                if (j - 1 >= 0)
                {
                    weight_prior -= spins[i][j - 1] * spins[i][j];
                    weight_after -= spins[i][j - 1] * -spins[i][j];
                }
                int E_prime = -weight_prior + weight_after;

                auto q = std::pow(EulerConstant, -1 * Beta * E_prime);
                if (q > (static_cast<float>(rand()) / (static_cast<float>(RAND_MAX))))
                {
                    spins[i][j] *= -1;
                    E += E_prime;
                    M += 2 * spins[i][j];
                }
            }
            if (k < N_Relaxation)
                continue;
            E_sum += E;
            E_sum_squared += (E * E);
            M_sum += M;
        }
        // average over all sample size
        auto U = E_sum / N_Samples;
        auto C = (Beta * Beta) * ((E_sum_squared / N_Samples) - (U * U));
        M_sum /= N_Samples;
        // average over spins
        M_sum /= N * N;
        U /= (N * N);
        C /= (N * N);
        r.U.emplace_back(U);
        r.C.emplace_back(C);
        r.M.emplace_back(M_sum);
    }
    return r;
}

void task_2()
{
    // run simulation for different lattice and sample sizes
    results r1 = simulation(10, 1000);
    results r2 = simulation(10, 10000);
    results r3 = simulation(50, 1000);
    results r4 = simulation(50, 10000);
    results r5 = simulation(100, 1000);
    results r6 = simulation(100, 10000);

    // calc theory values
    std::vector<float> T_plot;
    std::vector<float> M_theory_plot;
    double T = 4.f;
    for (; T > 0; T -= 0.2)
    {
        auto beta = 1 / T;
        T_plot.emplace_back(T);
        double M_Theory = 0;
        if (T < ((2) / (std::log1p(std::sqrt(2)))))
        {
            M_Theory = std::pow(1.0 - std::pow(std::sinh(2.0 * beta), -4.0), (1.0 / 8.0));
        }
        M_theory_plot.emplace_back(M_Theory);
    }

    // plot it out
    namespace plt = matplotlibcpp;
    plt::plot(T_plot, r1.U, {{"label", "N = " + std::to_string((int)10) + ", $N_{Samples} = $"
+ std::to_string((int)1000)}});
    plt::plot(T_plot, r2.U, {{"label", "N = " + std::to_string((int)10) + ", $N_{Samples} = $"
+ std::to_string((int)10000)}});
    plt::plot(T_plot, r3.U, {{"label", "N = " + std::to_string((int)50) + ", $N_{Samples} = $"
+ std::to_string((int)1000)}});
    plt::plot(T_plot, r4.U, {{"label", "N = " + std::to_string((int)50) + ", $N_{Samples} = $"
+ std::to_string((int)10000)}});
    plt::plot(T_plot, r5.U, {{"label", "N = " + std::to_string((int)100) + ", $N_{Samples} = $"
+ std::to_string((int)1000)}});
    plt::plot(T_plot, r6.U, {{"label", "N = " + std::to_string((int)100) + ", $N_{Samples} = $"
+ std::to_string((int)10000)}});
    // plot settings
    plt::title("Average Energy per Spin at Temperature T");
    plt::xlabel("Temperature T");
    plt::ylabel("Average Energy per Spin $U$ / $N^2$");
    plt::grid(true);
    plt::legend();
    plt::show();

    plt::plot(T_plot, r1.C, {{"label", "N = " + std::to_string((int)10) + ", $N_{Samples} = $"
+ std::to_string((int)1000)}});
    plt::plot(T_plot, r2.C, {{"label", "N = " + std::to_string((int)10) + ", $N_{Samples} = $"
+ std::to_string((int)10000)}});
    plt::plot(T_plot, r3.C, {{"label", "N = " + std::to_string((int)50) + ", $N_{Samples} = $"
+ std::to_string((int)1000)}});
    plt::plot(T_plot, r4.C, {{"label", "N = " + std::to_string((int)50) + ", $N_{Samples} = $"
+ std::to_string((int)10000)}});
    plt::plot(T_plot, r5.C, {{"label", "N = " + std::to_string((int)100) + ", $N_{Samples} = $"
+ std::to_string((int)1000)}});
    plt::plot(T_plot, r6.C, {{"label", "N = " + std::to_string((int)100) + ", $N_{Samples} = $"
+ std::to_string((int)10000)}});
    // plot settings
    plt::title("Average Heat per Spin at Temperature T");
    plt::xlabel("Temperature T");
    plt::ylabel("Average Heat per spin $C$ / $N^2$");
    plt::grid(true);
    plt::legend();
    plt::show();

    plt::plot(T_plot, r1.M, {{"label", "N = " + std::to_string((int)10) + ", $N_{Samples} = $"
+ std::to_string((int)1000)}});
    plt::plot(T_plot, r2.M, {{"label", "N = " + std::to_string((int)10) + ", $N_{Samples} = $"
+ std::to_string((int)10000)}});
    plt::plot(T_plot, r3.M, {{"label", "N = " + std::to_string((int)50) + ", $N_{Samples} = $"
+ std::to_string((int)1000)}});
    plt::plot(T_plot, r4.M, {{"label", "N = " + std::to_string((int)50) + ", $N_{Samples} = $"
+ std::to_string((int)10000)}});
    plt::plot(T_plot, r5.M, {{"label", "N = " + std::to_string((int)100) + ", $N_{Samples} = $"
+ std::to_string((int)1000)}});
    plt::plot(T_plot, r6.M, {{"label", "N = " + std::to_string((int)100) + ", $N_{Samples} = $"
+ std::to_string((int)10000)}});
      plt::plot(T_plot,  M_theory_plot,  {{"label",  "Analytical  result  of  M  for  the  infinite
system"}});
    // plot settings
    plt::title("Average Magnetization per Spin at Temperature T");
    plt::xlabel("Temperature T");
    plt::ylabel("Average Magnetization per spin $M$ / $N^2$");
    plt::grid(true);
    plt::legend();
    plt::show();
}
} // namespace task_2

int main()
{
    std::srand(5378);
    task_1::task_1();
    task_2::task_2();
    return 0;
}
```