Mini Project 2: Metropolis algorithm
Mathematical Foundation for Scientific Computing FTN0469

Li Ju

November 30, 2023

1 Introduction

The Ising model, named after the physicist Ernst Ising, is a mathematical model in statistical mechanics. Initially proposed in 1925 to describe ferromagnetism, the model's simplicity makes it a fundamental tool for understanding more complex systems in statistical physics and other computational studies[Cipra, 1987].

The Ising model consists of discrete variables known as spins, which can take on values of $+1$ or -1. These spins are arranged on a lattice, representing the atoms of a ferromagnetic material. The spins interact with their nearest neighbors and the external magnetic field. The Hamiltonian of a configuration is determined by these interactions and is given by:

$$H(s) = -J \sum_{i,j} s_i s_j - B \sum_i s_i \tag{1}$$

where J is the interaction strength between neighbouring pins, s_i denotes the spin value at point i and B represents the external magnetic field. $\sum_{i,j}$ denotes the sum over all neighbouring pin pairs and $\sum_i$ denotes the sum over all pins.

The equilibrium distribution of the system follows Boltzman distribution, i.e. $\Pr(s) \propto \exp(-\frac{H(s)}{k_B T})$, where k_B is the Boltzman constant and T is the absolute temperature[Newman and Barkema, 1999].

A key feature of the Ising model is its ability to exhibit phase transitions. With no external magnetic field, the two-dimensional square lattice Ising model undergoes a phase transition at a critical temperature, separating a low-temperature ferromagnetic phase (where spins are aligned) from a high-temperature paramagnetic phase (where spins are disordered). The magnetization of the Ising model is given by

$$M = \frac{1}{N} \sum_i s_i \tag{2}$$

where N denotes the number of pins in a lattice.

In this project, we implement the Ising model using the Metropolis algorithm, to simulate the behavior of the system under various temperature conditions. Then we discuss the detection of equilibrium of the simulation, the measurement of estimations of interest and the estimation of statistical errors. To improve the efficiency of the simulation, we parallelize the simulation with checkerboard scheme. Finally, We tackle the model's behavior near the critical temperature to observe and characterize the phase transition from ordered to disordered states. The main codes are deferred to the appendix and the full codes can be found at GitHub. The animation for the phase transition can be found here.

2 Implementation

The implementation of the Metropolis algorithm for Ising model follows such configurations:

- No external magnetical field: $B = 0$
- Unit interaction: $J = 1$
- Periodic boundary condition
- Unit Boltzman constant: $k_B = 1$

The pseudo-code for the implementation is shown as Algorithm 1.

Algorithm 1 Metropolis Algorithm for 2D Ising Model

procedure IsingMetropolis(N, T, n_{steps})
  Initialize lattice of size $N \times N$ with random spins (hot start) or ones (cold start)
  for step $= 1$ to n_{steps} **do**
    Choose a random spin at position $i, j \in \{N\}$
    $\Delta E \leftarrow$ Compute Delta Energy at (i, j)
    if $\Delta E \leq 0$ **then**
      Flip the spin
    else
      $r \leftarrow$ random number from $\mathcal{U}(0, 1)$
      if $r < \exp(-\Delta E/T)$ **then**
        Flip the spin
      end if
    end if
    Compute magnetization M and E
  end for
end procedure

The code for the implementation is deferred to the Appendix A.1.

3 Results & Discussion

In this section, we report our simulation results, including the detection of equilibrium, discussion of the measurement, and the behaviours of the phase transition.

3.1 Equilibrium Detection

To detect the equilibrium stage of the simulation, the magnetization and energy are monitored. For each specific temperature T, we run the simulation with three random seeds. By watching the three simulation reaching the same approximately constants, we deduce that the equilibrium is reached. Here we report 3 temperatures $T \in \{1.0, 2.0, 4.0\}$ as examples, shown in Figure 1. For $T = 1.0$ and $T = 2.0$, the lattice is initialized with state of $T = \infty$, and for $T = 4.0$, we initialize the lattice state as $T = 0$. Here time is measured in Monte Carlo steps per site (i.e. 10^4 iterations for a 100×100 lattice).

It has been observed that the time required to reach equilibrium varies with different temperatures and initial states. Nevertheless, it can be generally stated that equilibrium is typically attained within an order of magnitude of 10^7. Additionally, it is observed that both energy and magnetization fluctuate more at higher temperatures due to the increased likelihood of accepting energetically unfavorable spin configurations.

3.2 Measurement

After a simulation reaches the equilibrium, we can compute the magnetization and energy by averaging E and M at each step. To identify the number of iterations we need to average to get a proper estimation of the values, correlation time τ with respected to energy and magnetization is tackled. Here we visualize the autocorrelation at $T \in \{1.0, 2.0, 3.0\}$ for both magnetization and energy. The analysis is conducted on $5e7$ samples generated after equilibrium with the maximum correlation time $\tau = 2000$.

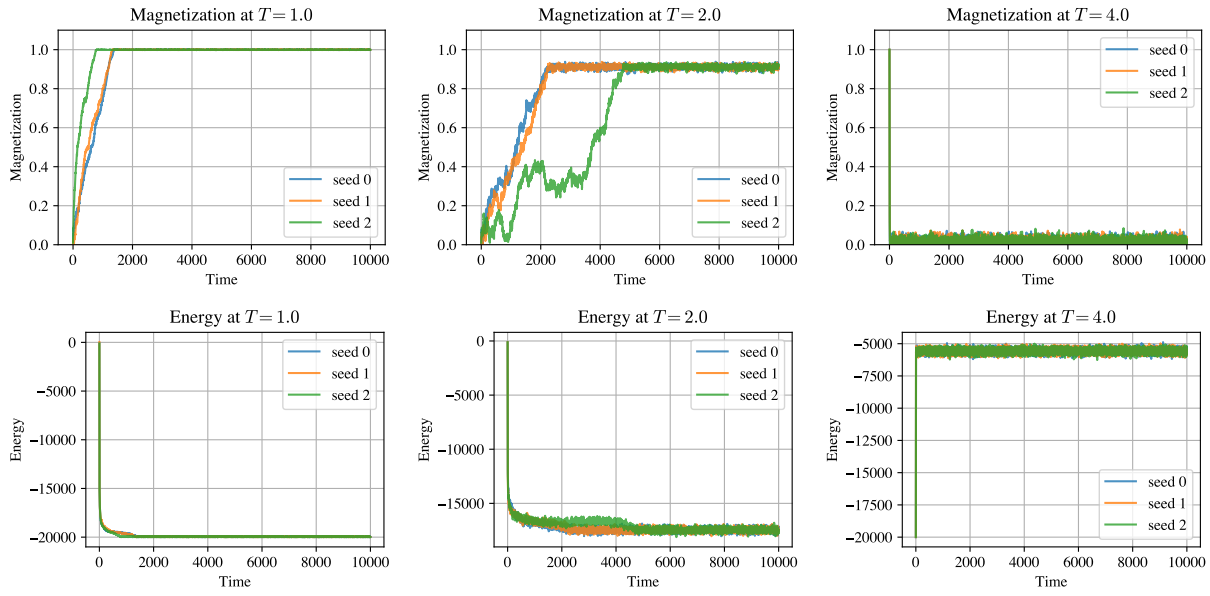

Figure 1: Equilibrium detection: Magnetization and energy are monitored to identify the process of equilibration.

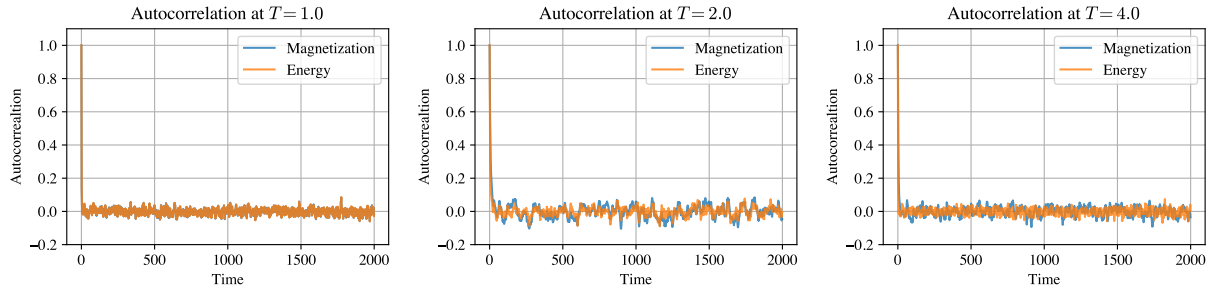

Figure 2: Autocorrelation of energy and magnetization of different lags.

It is observed that The autocorrelation indicates that for T away from the critical temperature, a small correlation time τ is enough to provide valid estimation. For simulations at temperatures near the critical temperature, greater correlation times τ are generally required. For the three simulations with $T = 1.0, 2.0, 3.0$, the correlation time can be chosen as $\{500, 1000, 500\} \times 10^4$, respectively.

3.3 Error Estimation

To estimate the statistical error of our estimations, we use the following equation for correlated samples:

$$\sigma = \sqrt{\frac{1 + 2\tau/\Delta t}{t - 1}(\overline{m^2} - \overline{m}^2)} \tag{3}$$

where τ is the correlation time, Δt is the interval for sampling, t is the number of total samples, $\overline{m^2}$ is the mean of all squared sample, and $\overline{m}^2$ is the square of the mean value of all samples [Newman and Barkema, 1999]. Here with our examples of $T = \{1.0, 2.0, 4.0\}$, we have $\tau = \{5, 10, 5\} \times 10^6$, $t = 5000$ and $\Delta t = 10^4$. The estimated energy and magnetization is shown in Table 1.

T	$\hat{M}$	$\hat{E}$
1.0	0.9993 ± 0.0002	-19972.2744 ± 6.8018
2.0	0.9117 ± 0.0055	-17461.7784 ± 106.8997
4.0	0.0004 ± 0.0092	-5569.6768 ± 75.0108

Table 1: Estimated energy and magnetization of simulations at $T \in \{1.0, 2.0, 3.0\}$ from 5×10^3 samples.

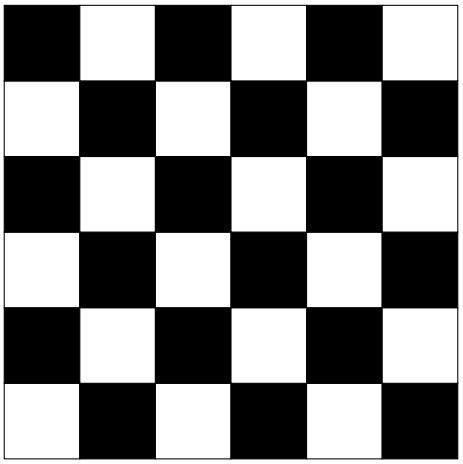

Figure 3: An illustration of checkerboard scheme for paralleling Metropolis algorithm for Ising model simulation.

3.4 Parallelization

The Metropolis algorithm for Ising model simulation discussed above is sequential, with only one spin is flipped per step. Although for each step, the computational cost is cheap, with few float operations involved except the generation of random numbers, sequential updates with single spin are generally inefficient. For modern computers, both CPU and GPU have implemented Single Instruction Multiple Data (SIMD) paradigm, but the standard Metropolis algorithm for Ising model does not take advantages of the feature. Observing that the computation for each step is local (only related to points' nearest neighbours), it is possible to parallelize the simulation with checkerboard scheme [Weigel, 2018].

Figure 3 visualizes a 6×6 lattice as an illustration. It leads to no side effects to attempt to flip spins at all black (or white) at the same time. For each sweep of spin updates, we just need to sequentially update black and white points in two steps and thus, parallel threads of $N/2$ can be achieved, where N denotes the number of spins. The implementation of the parallelization is deferred to Appendix A.2.

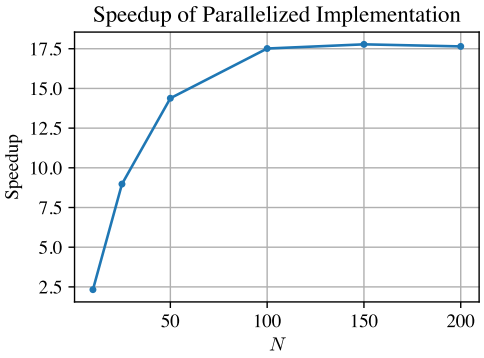

Figure 4: Speedup of parallelized implementation over single-spin implementation

With checkerboard parallelization using `NumPy` with CPU, we report the speedup of parallelized implementation over single-spin implementation for lattices of different sizes $\{10 \times 10, 25 \times 25, 50 \times 50, 100 \times 100, 150 \times 150, 200 \times 200\}$. The results is plotted in Figure 4. It is observed that the parallelized implementation consistently brings speedup over single-spin implementaiton. The bigger size the lattice is of, the more speedup can be obtained from checkerboard parallelization. Additionally, the parallelized implementation can be easily extended to GPU, where even more acceleration could be obtained.

3.5 Phase Transition

With methods to detect equilibrium, to measure values and to estimate error of estimations, we start to simulate the phase transition behaviour of the system. Considering that the critical temperature of the system is analytiacally computed as $T = 2.269$, we stetup a temperature grid ranging from 1.0 to 3.5, with dense points between 2.0 to 2.5. The magnetization of the system at different temperature is plotted in Figure 5.

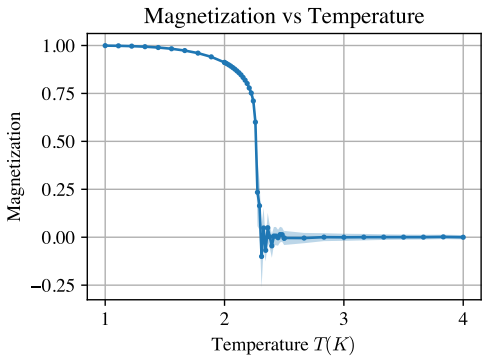

Figure 5: Phase transition of the system

It is observed that a significant drop of magnetization occur between $T = 2.0$ to $T = 2.5$, indicating the phase transition of the system. The critical temperature observed from the experiment is consistent with theoretical analysis. Additionally, for points close to the critical temperature, larger standard deviation of estimations are observed as well (denoted by shadowed areas), which is consistent with our discussion in Section 3.2.

4 Conclusion

In this study, we have implemented the Ising model using the Metropolis algorithm, providing a exploration of its dynamical behavior under various temperature conditions. Key highlights of our investigation include the detection of equilibrium states, measurements of physical properties, and the error estimation of the measurements. Above the these discussions, we examined the system behaviours of the phase transition near the critical temperature. The observed critical phenomena, particularly the significant fluctuations in magnetization and energy, align well with theoretical results, underscoring the robustness and accuracy of our simulation approach. Furthermore, the use of checkerboard parallelization has significantly improved computational efficiency, demonstrating the potential of modern computational methods in tackling complex statistical physics problems.

References

Barry A Cipra. An introduction to the ising model. *The American Mathematical Monthly*, 94(10):937–959, 1987.

Mark EJ Newman and Gerard T Barkema. *Monte Carlo methods in statistical physics*. Clarendon Press, 1999.

Martin Weigel. Monte carlo methods for massively parallel computers. In *Order, Disorder and Criticality: Advanced Problems of Phase Transition Theory*, pages 271–340. World Scientific, 2018.

A Code

A.1 Single-spin Implementation

```python
# Metropolis Monte Carlo simulation of the 2D Ising model
import numpy as np
from dataclasses import dataclass


@dataclass
class Simulator:
    N: int
    T: float
    random_seed: int
    lattice: np.ndarray = None
    cold_start: bool = False
    J: int = 1

    def __post_init__(self):
        # set the random seed
        np.random.seed(self.random_seed)

        # initialize the lattice
        if self.lattice is None:
            self.lattice = self.init_lattice(self.N, self.cold_start)

        # precompute exponentials to speed up the simulation
        self.delta_energy_prob_dict = {
            delta_energy: np.exp(-float(delta_energy) / self.T)
            for delta_energy in range(-8, 9, 4)
        }

    # compute the change in energy if the spin at (i, j) is flipped
    def compute_delta_energy(self, i, j):
        N = self.N
        up = self.lattice[(i - 1) % N, j]
        down = self.lattice[(i + 1) % N, j]
        left = self.lattice[i, (j - 1) % N]
        right = self.lattice[i, (j + 1) % N]

        delta_energy = 2*self.J * self.lattice[i, j] * (up+down+left+right)
        return delta_energy

    # compute the probability of flipping the spin
    def compute_flip_prob(self, delta_energy):
        return self.delta_energy_prob_dict[delta_energy]

    def mc_step(self, i, j):
        # compute the change in energy
        delta_energy = self.compute_delta_energy(i, j)

        # compute the probability of flipping the spin
        flip_prob = self.compute_flip_prob(delta_energy)
```

```
        # flip the spin with probability flip_prob
        if np.random.rand() < flip_prob:
            self.lattice[i, j] *= -1
            delta_mag_sum = 2*self.lattice[i, j]
        else:
            delta_energy = 0
            delta_mag_sum = 0

        return delta_energy, delta_mag_sum

    def run(self, num_sweeps: int, verbose: bool = True):
        n_iter = num_sweeps * self.N**2

        # initialize the arrays to store the magnetization and lattice
        magnetization_array = np.zeros(num_sweeps)
        energy_array = np.zeros(num_sweeps, dtype=np.int32)
        lattice_array = np.zeros((num_sweeps, self.N, self.N), dtype=np.int8)

        # generate random indices
        ijs = np.random.randint(0, self.N, size=(n_iter, 2))

        # initialize the metrics
        energy = self.compute_energy(self.lattice, self.J)
        mag_sum = self.lattice.sum()

        # run the simulation
        for iter_i, (i, j) in enumerate(ijs):
            # step the lattice
            delta_energy, delta_mag_sum = self.mc_step(i, j)

            # update the metrics
            energy += delta_energy
            mag_sum += delta_mag_sum

            # save the lattice every sweep
            if iter_i % (self.N**2) == 0:
                current_sweep = iter_i // (self.N**2)
                magnetization_array[current_sweep] = (mag_sum / self.N**2)
                energy_array[current_sweep] = energy
                lattice_array[current_sweep] = self.lattice.astype(np.int8)

                if verbose and current_sweep % 100 == 0:
                    print(f"Sweep {current_sweep}: E={energy}, " +
                          f"M={mag_sum / self.N**2}", flush=True)
        return magnetization_array, energy_array, lattice_array

    # function to calculate the energy of the lattice
    # utilize periodic boundary conditions
    @staticmethod
    def compute_energy(lattice, J):
        up = np.roll(lattice, 1, axis=0)
        down = np.roll(lattice, -1, axis=0)
        left = np.roll(lattice, 1, axis=1)
        right = np.roll(lattice, -1, axis=1)
```

```
        energy = -J/2 * np.sum(lattice * (up+down+left+right))
        return energy

    # initialize the lattice
    @staticmethod
    def init_lattice(N, cold_start: bool = False):
        if cold_start:
            lattice = np.ones((N, N))
        else:
            lattice = np.random.choice(
                [-1, 1], size=(N, N))
        return lattice
```

A.2 Checkerboard Parallelization

```
# Metropolis Monte Carlo simulation of the 2D Ising model
# Parallelized version with checkerboard pattern
import numpy as np
from dataclasses import dataclass


@dataclass
class ParaSimulator:
    N: int
    T: float
    random_seed: int
    lattice: np.ndarray = None
    cold_start: bool = False
    J: int = 1

    def __post_init__(self):
        # set the random seed
        np.random.seed(self.random_seed)

        # initialize the lattice
        if self.lattice is None:
            self.lattice = self.init_lattice(self.N, self.cold_start)

        # initialize checkerboard patterns
        self.black_mask = np.zeros((self.N, self.N), dtype=np.int8)
        self.black_mask[::2, ::2] = 1
        self.black_mask[1::2, 1::2] = 1
        self.black_mask = self.black_mask.astype(bool)

        self.white_mask = np.ones((self.N, self.N), dtype=np.int8)
        self.white_mask[::2, ::2] = 0
        self.white_mask[1::2, 1::2] = 0
        self.white_mask = self.white_mask.astype(bool)

        # precompute exponentials to speed up the simulation
        self.de_prob_dict = {
            delta_energy: np.exp(-float(delta_energy) / self.T)
            for delta_energy in range(-8, 9, 4)
```

```
    }

def delta_energies_to_probs(self, delta_energies):
    vec_func = np.vectorize(
        lambda delta_energy: self.de_prob_dict[delta_energy])
    return vec_func(delta_energies)

def mc_step(self, mask):
    # get neighbours
    up = np.roll(self.lattice, 1, axis=0)[mask]
    down = np.roll(self.lattice, -1, axis=0)[mask]
    left = np.roll(self.lattice, 1, axis=1)[mask]
    right = np.roll(self.lattice, -1, axis=1)[mask]

    # compute energy changes
    delta_energies = 2*self.J * self.lattice[mask] * (up+down+left+right)

    # compute the probabilities of flipping masked spins
    probs = self.delta_energies_to_probs(delta_energies)

    # compute masks for spins to be flipped
    flip_mask = np.random.rand(*probs.shape) < probs

    # merge the masks
    mask = mask.copy().reshape(-1)
    mask[mask] = flip_mask
    mask = mask.reshape(self.N, self.N)

    # flip the spins
    self.lattice[mask] *= -1

def run(self, num_sweeps: int, verbose: bool = True):
    # initialize the arrays to store the magnetization and lattice
    magnetization_array = np.zeros(num_sweeps)
    energy_array = np.zeros(num_sweeps, dtype=np.int32)
    lattice_array = np.zeros((num_sweeps, self.N, self.N), dtype=np.int8)

    # run the simulation
    for current_sweep in range(num_sweeps):
        # step the lattice
        self.mc_step(self.black_mask)
        self.mc_step(self.white_mask)

        # compute the energy and magnetization
        energy = self.compute_energy(self.lattice, self.J)
        magnetization = self.lattice.mean()

        magnetization_array[current_sweep] = magnetization
        energy_array[current_sweep] = energy
        lattice_array[current_sweep] = self.lattice.astype(np.int8)

        if verbose and current_sweep % 100 == 0:
            print(f"Sweep {current_sweep}: E={energy}, M={magnetization}",
                  flush=True)
```

```
    return magnetization_array, energy_array, lattice_array

# function to calculate the energy of the lattice
# utilize periodic boundary conditions
@staticmethod
def compute_energy(lattice, J):
    up = np.roll(lattice, 1, axis=0)
    down = np.roll(lattice, -1, axis=0)
    left = np.roll(lattice, 1, axis=1)
    right = np.roll(lattice, -1, axis=1)

    energy = -J/2 * np.sum(lattice * (up+down+left+right))
    return energy

# initialize the lattice
@staticmethod
def init_lattice(N, cold_start: bool = False):
    if cold_start:
        lattice = np.ones((N, N))
    else:
        lattice = np.random.choice(
            [-1, 1], size=(N, N))
    return lattice
```