Ising Model Project Assignment PHM6420

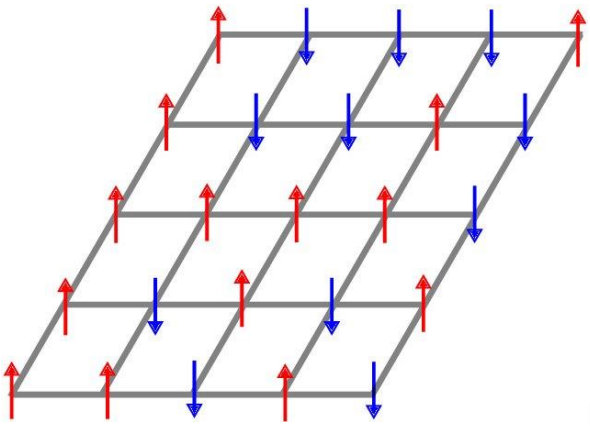
Tob31@aber.ac.uk | 11th December2023

Contents

Explanation of Science

The Ising model or Ising-Lenz model, proposed by Ernst Ising in his 1924 PhD thesis, the problem was given to him by his supervisor Wilhelm Lenz.

It is a mathematical model of ferromagnetism in statistical mechanics, the model simulates ferromagnetism and exhibits a phase transition between ferromagnetic ordered states and paramagnetic disordered states. The model consists of discrete variables named spins which can binarily be either +1 or -1 alternatively “up” or “down”. The spins are arranged on a grid/lattice, each spin can only interact with its closest neighbors on the lattice.



[fig1]

In this project the Lattice structure is a two-dimensional (2d) grid, however it is possible to simulate this model in Three dimensions(3d) or potentially more. The energy of the system is defined by the Hamiltonian which for the Ising model is given by:

$$E = -J \sum_{<i,j>} S_i S_j - H \sum_i S_i$$

Where J is the interaction energy of the neighboring spins, H is the external magnetic field, S_i and S_j are the spins at I and j respectively. The summation of <i,j> ($\sum_{<i,j>} S_i S_j$)represents the summation of all pairs of neighboring spins I and j. The summation of I is the summation of all the spins ($\sum s[i,j]$ and represents the interaction of each spin with the external magnetic field. Since this project uses a 2d grid the Hamiltonian becomes:

$$E = -J \sum_{i,j} s[i,j]s[i+1,j] - J \sum_{i,j} s[i,j]s[i,j+1] - H \sum_{i,j} s[i,j]$$

Where each term indexes to the neighboring pairs for each x and y direction respectively.

To simulate the Ising model Monte Carlo methods are used, these methods use random number/sampling to obtain the numerical results. These methods are used in physics for simulating systems where the analytical solution is too challenging or impossible to obtain, they can be used to study the properties of many body systems like solids, liquids, and gasses under set conditions for example simulating a radiation environment.

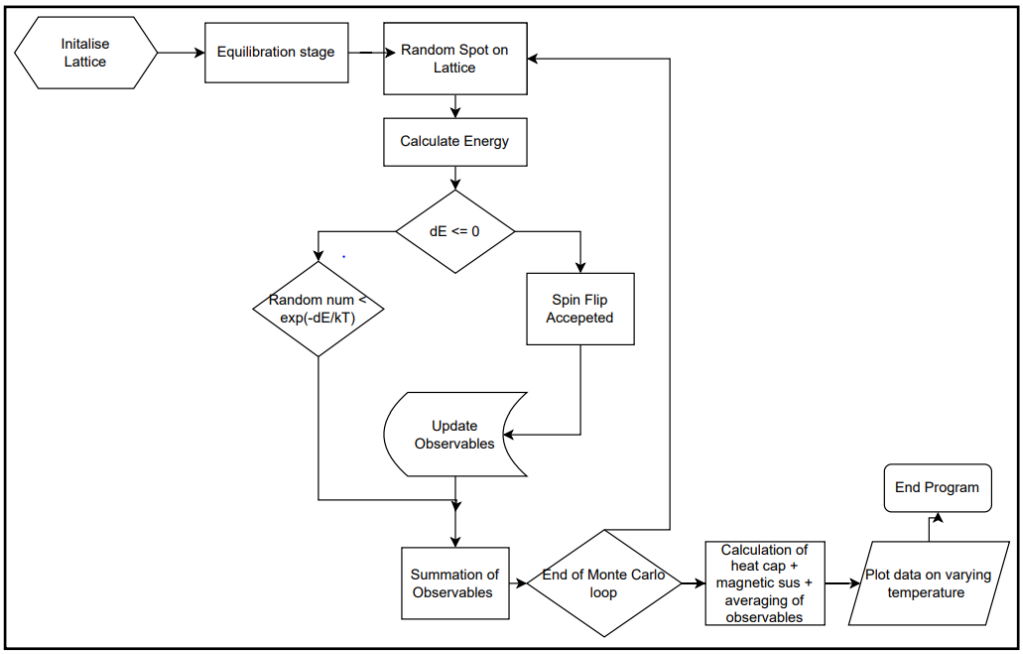
For this project the Monte Carlo Metropolis Algorithm is used.

- The algorithm starts with an initial configuration of spins,
- A spin is then randomly selected.
- The change in energy (ΔE) is then calculated should the spin be flipped.
- If $\Delta E \leq 0$ or the probability $e^{-\Delta^{E/KbT}}$ > a random number, then the flip is accepted. Where T is the temperature and Kb is the Boltzmann's constant (for this project Kb = 1 for reduced units). Otherwise, the flip is not accepted.

As the system is run it is allowed to reach thermal equilibrium. Physical observables like magnetization, magnetic susceptibility, energy, and specific heat capacity can be calculated and plotted as functions of temperature and displayed to show the behavior of the ferromagnet under increasing temperature reaching a paramagnetic state, at the critical temperature the model exhibits a phase transition displaying this ferromagnetic behavior.

The purpose for simulating the Ising model is to show how the system behaves near a phase transition and studying the Ising model can help researchers/students learn more about computational physics and statistical mechanics, the Ising model also has applications in other fields like finance, biology, engineering, and artificial intelligence. Possible extensions could include applying this model to a project in one of these other fields and/or simulating the lattice in 3D.

The program can be described as the flow chart below.



Description of Numerical Method

The program for this method is broken down into various functions, which are used to calculate physical observables, save, and display the plots to the screen.

```
import numpy as np
import matplotlib.pyplot as plt
import time
import datetime

kB = 1  # Boltzmann constant in reduced units
```

These are the imports used for the project, numpy is a mathematical module. Matplotlib is used for plotting data visually. Datetime and time are used for getting and manipulating the current date/time.

Kb is the Boltzmann constant in reduced units, it is set as a global variable because it is a constant and is used throughout the program.

Main

```
if __name__ == '__main__':
    # input data for size and
    size = 10  # 10x10 lattic
    steps = 20000000  # Monte
    # curie temperature shoul

    # run simulation and plot
    plotModel(size, steps)
```

The main function here is for starting the program by calling the relevant function to plot the data. This project produced various plots the final most accurate plot created was a 10x10 lattice with 20 million steps which took 8.7 hours to compute.

This function is also used to set the input data for the model that will be plotted. Size 10, steps 20 million.

Simulate

Simulate is the main function which sets up the Monte Carlo simulation of the Ising model, it initializes the spins on the lattice perform the Monte Carlo metropolis algorithm with equilibration and calculates and updates the quantitative values for the physical observables of the system.

The function parses in the steps of the Monte Carlo, the size, the temperature and sets J to 1 and H to 0.

```
def simulate(steps, size, temperature, J=1., H=0):
    spins = np.random.choice([-1, 1], size=(size, size)) #generates the random spins on the latice
    # initalized summation variables as floats to avoid overflow errors
    sum_energy = 0.
    sum_energy_squared = 0.
    sum_magnetization = 0.
    sum_magnetization_squared = 0.

    # setting the initial calculations for energy and magnetization
    energy = hamiltonian(J, H, spins)
    magnetization = np.sum(spins)

    # Equilibration
    for step in range(steps//10):  # bias removal step at 10% of the number of steps
        spins, energy, magnetization = montecarloMetStep(size, temperature, spins, J, H, energy, magnetization)
```

The initialization stage comes first, the random spins are generated on the lattice and the variables to hold the summations of the quantitative values like energy and magnetization are initialized as floats to avoid overflow errors. The energy and magnetization is also initially calculated before the Equilibration stage, to allow for further deduction in bias and proper updating of the quantitative values.

The Equilibration stage runs the Monte Carlo metropolis simulation step for 10% of the step count, this allows the system to reach a state that is stable and representative of a system in thermal equilibrium. The data is not recorded in this stage, but the spins are updated.

```
# Main Monte Carlo simulation
for step in range(steps):  # full steps loop
    spins, energy, magnetization = montecarloMetStep(size, temperature, spins, J, H, energy, magnetization)

    # data collection
    sum_energy += energy
    sum_energy_squared += energy ** 2
    sum_magnetization += magnetization
    sum_magnetization_squared += magnetization ** 2
```

```
# data averaging from collected data, multiplies by the inverse to save on computation
inverseSteps = 1/steps
avg_energy = sum_energy * inverseSteps
avg_magnetization = sum_magnetization * inverseSteps
```

```
# calling functions to calculate heat capacity and magnetic susceptibility
heatcapacity = calculateHeatCapacity(inverseSteps, temperature, avg_energy, sum_energy_squared)
MagneticSus = calculateMagneticSus(inverseSteps, temperature, avg_magnetization, sum_magnetization_squared)
return avg_energy, avg_magnetization, heatcapacity, MagneticSus
```

The main Monte Carlo simulation is run after the equilibration stage, this will run the monte Carlo metropolis function for the full amount of specified steps and update the values for spins, energy and magnetization at each step and then sums up the data and the data squared. The function montecarlometStep takes in the size of the lattice, the current temperature, the previous spins, J, H and the previous energy and magnetization. The function then returns the new spins, energy and magnetization.

Out of the loop the averages for summed energy and magnetization are calculated, which would be the sum / steps, however since the division operator is many times slower than the multiplication operator and the average will need to be calculated 4 times, the inverse of the steps is pre calculated so that the values for the averages can be determined with a much faster multiplication operator.

Separate functions to calculate the heat capacity and magnetic susceptibility, they parse in the value for the inverse steps, the current temperature, the averages for energy/magnetization and the sum for energy/magnetization squared. Once these are complete the simulate function returns the values for average energy, average magnetization, heat capacity and magnetic susceptibility.

calculateHeatCapacity

This function calculates the specific heat capacity of the system given the equation.

$$C = \frac{\partial U}{\partial T} = \frac{1}{k_B T^2}\left(\langle E^2 \rangle - \langle E \rangle^2\right)$$

```
# function to calculate the heat capacity of the system
def calculateHeatCapacity(inversesteps, temperature, avg_energy, sum_energy_squared):
    avg_energy_squared = sum_energy_squared * inversesteps
    heatcapacity = (avg_energy_squared - avg_energy ** 2) / (kB * temperature ** 2)
    return heatcapacity
```

The inverse of the steps is used to calculate the average of the energy squared, the equation is then used to calculate the heat capacity and then returned.

calculateMagneticSus

This function calculates Magnetic susceptibility of the system given the equation.

$$\chi = \frac{\partial M}{\partial H} = \frac{1}{k_B T}\left(\langle S^2 \rangle - \langle S \rangle^2\right)$$

```
# function to calculate the magnetic sus
def calculateMagneticSus(inversesteps, temperature, avg_magnetization, sum_magnetization_squared):
    avg_magnetization_squared = sum_magnetization_squared * inversesteps
    MagneticSus = (avg_magnetization_squared - avg_magnetization ** 2) / (kB * temperature)
    return MagneticSus
```

The inverse of the steps is used to calculate the average of the magnetization squared, the equation is then used to calculate the magnetic susceptibility and then returned.

monetcarloMetStep

The function performs a single Monte Carlo step using the metropolis algorithm.

```
def montecarloMetStep(size, temperature, spins, J, H, Eng, Mag):
    i, j = np.random.randint(0, size, 2) # randomly select the spin
    dEng = dEnergy(spins, i, j, size, J, H) # call change in energy calcul

    #Metropolis algorithm (previous version had **if dEng = 0**)
    if dEng <= 0 or np.random.rand() < np.exp(-dEng / (kB * temperature)):
        spins[i, j] *= -1 # flip the spin
        # update the energy and magnetization
        Eng += dEng
        Mag += 2*spins[i,j]
    return spins, Eng, Mag
```

A random spin on the lattice is selected and the change in energy is calculated by calling the dEnergy function.

The Metropolis step uses an if statement to check if the energy is accepted, the spin is flipped, and the energy and magnetization is updated. The requirement for this acceptance is the energy being less than or equal to 0 or a random number being larger than the exponent of the negative energy /

the Boltzmann constant * Temperature. The values are then returned whether they're updated or not.

dEnergy

This function calculates the change in energy of the system at each step when called.

```
def dEnergy(spins, i, j, size, J, H):
    # getting values from the nearest cells in the lattice with pe
    up = spins[(i - 1) % size, j]
    down = spins[(i + 1) % size, j]
    left = spins[i, (j - 1) % size]
    right = spins[i, (j + 1) % size]

    dEng = 2 * spins[i, j] * (J * (up + down + left + right) + H)
    return dEng
```

The values of the neighboring spins on the lattice are determined using the modulus of the size to achieve the periodic boundary conditions for the system.

The change in energy, should the flip be accepted, is then calculated, and returned in the metropolis step.

Hamiltonian

This function calculates the total energy for the system at each step when called.

```
def hamiltonian(J, H, spins):
    #peridoic boundary conditions using np.roll. np.roll is less memory efficent but is faster than other methods of calculating ΔEnergy
    energy = (-J * (np.sum(spins * np.roll(spins, 1, axis=0)) + np.sum(spins * np.roll(spins, 1, axis=1)))) -(H * np.sum(spins))
    return energy
```

The equation below is implemented here using the np.roll function from the imported numpy library, it allows for the most time efficient periodic boundary conditions implementation for this Hamiltonian, however it is less memory efficient than using the modulus operator method, this is because np.roll creates a copy of the lattice.

$$E = -J\sum_{i,j} s[i,j]s[i+1,j] - J\sum_{i,j} s[i,j]s[i,j+1] - H\sum_{i,j} s[i,j]$$

plotModel

This function is for plotting and visualizing the data results from the simulation, it runs the simulation over increasing temperature, 60 datapoints from 1-4^{0}c.

```python
# visual data plotting function
def plotModel(size, steps, J=1, H=0):
    start_time = time.time() # getting the start time
    # initalized empty lists to append the values collected
    energies = []
    magnetizations = []
    heat_capacities = []
    magnetic_susceptibilities = []
    temperatures = []

    tempRange = np.linspace(1.0, 4.0, 60)  # temp range from 1-4, creates 60 data points

    for temp in tempRange:
        energy, magnetization, heat_capacity, magnetic_sus = simulate(steps, size, temp, J, H)
        # adds the data points to the corresponding lists
        energies.append(energy)
        magnetizations.append(abs(magnetization))
        heat_capacities.append(heat_capacity)
        magnetic_susceptibilities.append(magnetic_sus)
        temperatures.append(temp)
```

The start time is obtained, to increase the efficiency of method these lines relating to time should be removed. However, they provide valuable insight as to how fast the method is thus have been left in the documentation.

Empty lists of the physical observables are initialized. The range of 60 data point temperatures from 1-4^{o}c are created using np.linspace.

The simulation is run in a for loop through the temp range data points and the observables are then appended to their respective lists. The absolute value for magnetization is taken here.

```python
plt.figure(figsize=(12, 10))

# Plot energy
plt.subplot(2, 2, 1)
plt.plot(temperatures, energies, 'o-')
plt.title('Energy')
plt.xlabel('Temperature')
plt.ylabel('Energy')

# Plot magnetization
plt.subplot(2, 2, 2)
plt.plot(temperatures, magnetizations, 'o-')
plt.title('Magnetization')
plt.xlabel('Temperature')
plt.ylabel('Magnetization')

# Plot heat capacity
plt.subplot(2, 2, 3)
plt.plot(temperatures, heat_capacities, 'o-')
plt.title('Heat Capacity')
plt.xlabel('Temperature')
plt.ylabel('Heat Capacity')

# Plot magnetic susceptibility
plt.subplot(2, 2, 4)
plt.plot(temperatures, magnetic_susceptibilities, 'o-')
plt.title('Magnetic Susceptibility')
plt.xlabel('Temperature')
plt.ylabel('Magnetic Susceptibility')
```

The image on the left shows simply how the plots are created named, labeled, and set to the values obtained using the matplotlib imported library. The temperature ranges are on the x axis and the physical observables ranges are on the y axis.

```
end_time = time.time() # getting the end time
print(fr'This program took: {end_time - start_time} seconds to run') # displaying the

# formating, saving and displaying the plotted data to the screen
plt.tight_layout()
plotname = datetime.datetime.now().strftime('%Y-%m-%d_%H-%M-%S')
plt.savefig(fr'C:\Users\Tommy\PycharmProjects\IsingMonteCarlo\output\{plotname}.png')
plt.show()
```

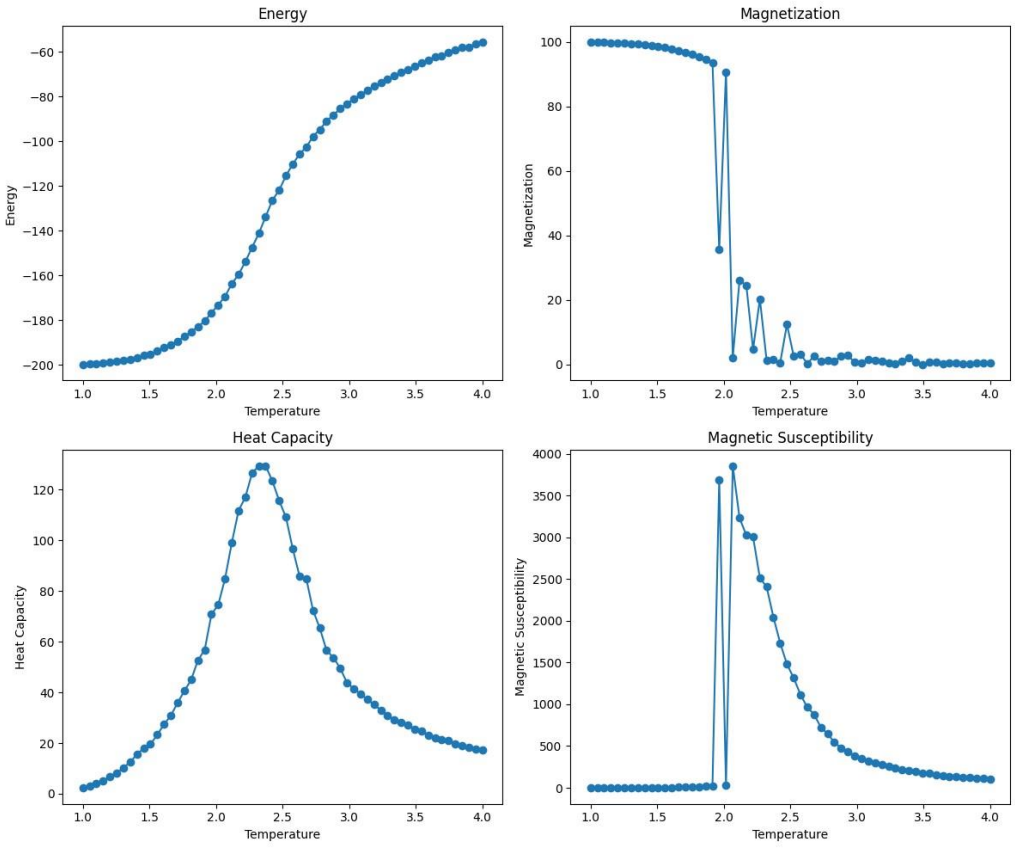
As discussed above to increase the efficiency of method these lines highlighted in red should be taken out.

The plot is then formatted, named the current date and time, displayed and saved to an output directory.

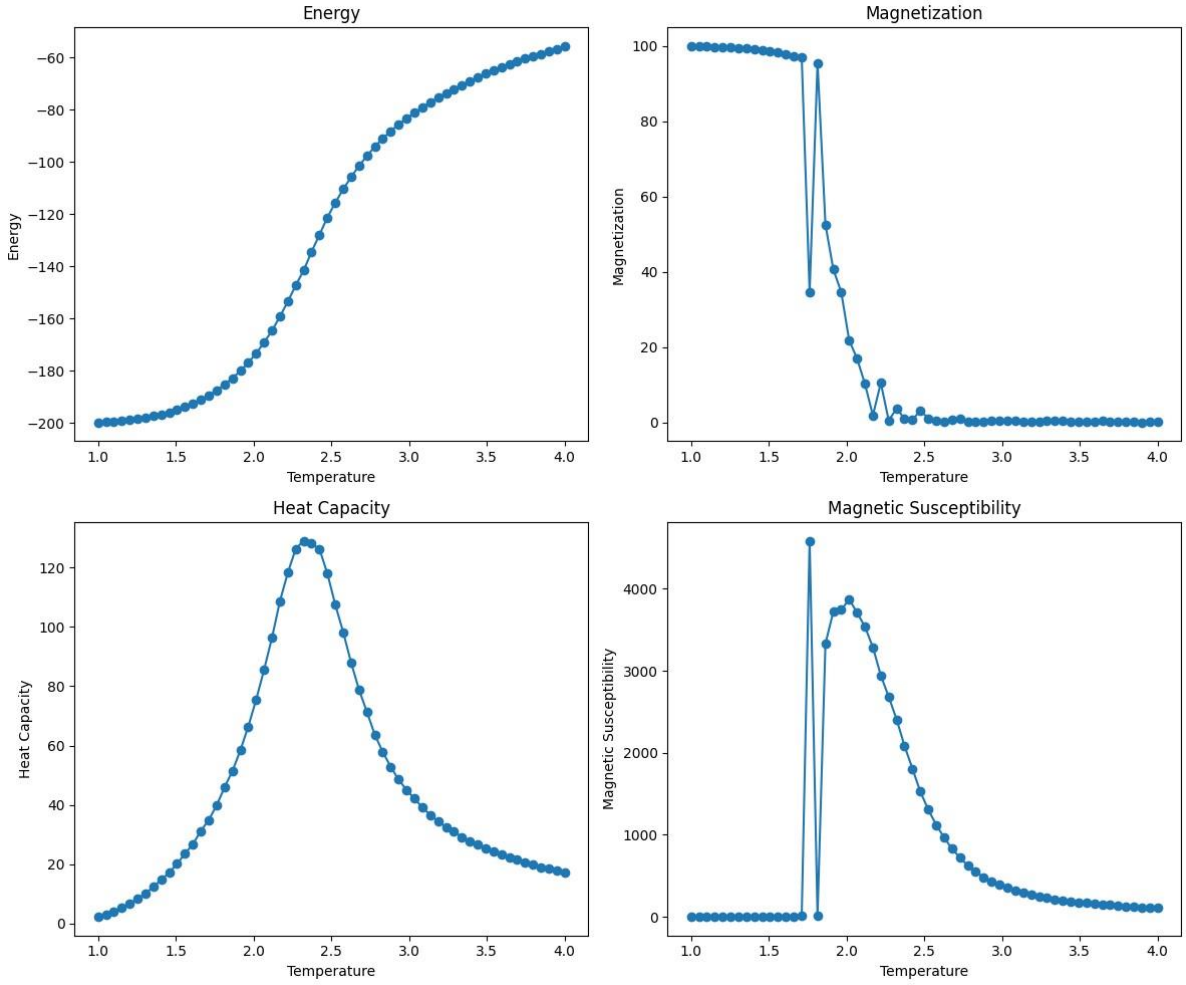
Discussion of Results

The results of the Monte Carlo simulation of the Ising model are plots of energy, magnetization, heat capacity and magnetic susceptibility as functions of temperature. The plots of these results display the theoretical expectations about the behavior of the Ising model. The units for temperature in these plots are in degrees Celsius.

The plot below was the result of a 10x10 lattice for 2 million steps, which took 1.4 hours to finish running.



Similarly, the plot below was the result of a 10x10 lattice for 20 million steps, which took 8.7 hours to finish running. When compared to the plot above it is noticeable smoother and therefore potentially a better representation of the physical behavior of the Ising model.



These physical observables above display the following behaviors:

- Energy: As the temperature of the system increases so does the energy of the system, the higher temperature in the system increases the disorder of the spins in the system as spins get flipped at random, this is known as thermal agitation.

- Magnetization: The plot shows the absolute value of magnetization, at low temperatures the magnetization in the system is high, suggesting that the spins in the system are mostly aligned. However, as the temperature increases the magnetization heavily decreases to zero. Where the temperature at which the magnetization drops to zero indicates the value for the Curie temperature. At approximately 2.35^{θ}c, this behavior indicates the phase transition from a ferromagnetic state to a paramagnetic.

- Heat Capacity: The plot displays a clear peak at approximately 2.35^{θ}c, same as the indicated Curie temperature from the plot of magnetization. This peak is an example of a second order phase transition.

- Magnetic Susceptibility: Similarly, but different to the plot for magnetization, the magnetic susceptibility has a large peak at approximately the Curie temperature. At low temperatures the magnetic susceptibility is zero, as the temperature increase to the curie temperature it sharply increases then slowly decreases. This peak indicates a high response to the external magnetic field H.

Considerations

Due to the finite size of the lattice in this project sharper transitions can be observed whereas a real system would exhibit a smoother transition.

Conclusion

This Monte Carlo simulation has yielded results that are consistent with the theoretical predictions of the Ising model, displaying the model's accuracy in predicting the physical behavior of ferromagnetic materials. The implementation of an equilibration stage was key to removing bias from the system, equal to 10% of the total steps.

As the main simulation was allowed to compute for 20 million steps, taking approximately 8.7 hours, was key to achieving highly accurate results. The sharp peaks around for magnetization, heat capacity and magnetic susceptibility observed in the simulation display the phase transition in the ferromagnet and the value for the Curie temperature can be observed.

In conclusion, this project has successfully simulated the expected physical behavior of the Ising model and reinforces the ability of using Monte Carlo methods to solve complex systems.

References

[Fig1], Sascha Wald, "2D Ising model on a square lattice", ResearchGate, Sep 2017. [Online]. Available: https://www.researchgate.net/publication/321920877_Thermalisation_and_Relaxation_of_Quantum_Systems/figures?lo=1

S. P. Singh, "The Ising Model: Brief Introduction and Its Application", in Solid State Physics - Metastable, Spintronics Materials and Mechanics of Deformable Bodies - Recent Progress, S. Sivasankaran, P. K. Nayak, and E. Günay, Eds. IntechOpen, 2020. [Online]. Available: https://www.intechopen.com/chapters/71210.

J. Kotze, "Introduction to Monte Carlo methods for an Ising Model of a Ferromagnet", arXiv:0803.0217, Mar. 2008. [Online]. Available: https://arxiv.org/abs/0803.0217

Appendix (Source Code)

Page 1/2

```python
import numpy as np
import matplotlib.pyplot as plt
import time
import datetime

kB = 1  # Boltzmann constant in reduced units

# main simulation function
def simulate(steps, size, temperature, J=1., H=0):
    spins = np.random.choice([-1, 1], size=(size, size)) #generates the random spins on the lattice
    # initalized summation variables as floats to avoid overflow errors
    sum_energy = 0.
    sum_energy_squared = 0.
    sum_magnetization = 0.
    sum_magnetization_squared = 0.

    # setting the initial calculations for energy and magnetization
    energy = hamiltonian(J, H, spins)
    magnetization = np.sum(spins)

    # Equilibration
    for step in range(steps//10):_# bias removal step at 10% of the number of steps
        spins, energy, magnetization = montecarloMetStep(size, temperature, spins, J, H, energy, magnetization)

    # Main Monte Carlo simulation
    for step in range(steps):_# full steps loop
        spins, energy, magnetization = montecarloMetStep(size, temperature, spins, J, H, energy, magnetization)

        # data collection
        sum_energy += energy
        sum_energy_squared += energy ** 2
        sum_magnetization += magnetization
        sum_magnetization_squared += magnetization ** 2

    # data averaging from collected data, multiplies by the inverse to save on computation
    inverseSteps = 1/steps
    avg_energy = sum_energy * inverseSteps
    avg_magnetization = sum_magnetization * inverseSteps

    # calling functions to calculate heat capacity and magnetic susceptibility
    heatcapacity = calculateHeatCapacity(inverseSteps, temperature, avg_energy, sum_energy_squared)
    MagneticSus = calculateMagneticSus(inverseSteps, temperature, avg_magnetization, sum_magnetization_squared)
    return avg_energy, avg_magnetization, heatcapacity, MagneticSus

# function to calculate the heat capacity of the system
def calculateHeatCapacity(inversesteps, temperature, avg_energy, sum_energy_squared):
    avg_energy_squared = sum_energy_squared * inversesteps
    heatcapacity = (avg_energy_squared - avg_energy ** 2) / (kB * temperature ** 2) # C = dU/dT = (1/kb T**2)*(([E**2)-([E)**2)]
    return heatcapacity

# function to calculate the magnetic sus
def calculateMagneticSus(inversesteps, temperature, avg_magnetization, sum_magnetization_squared):
    avg_magnetization_squared = sum_magnetization_squared * inversesteps
    MagneticSus = (avg_magnetization_squared - avg_magnetization ** 2) / (kB * temperature) # X = dM/dH = (1/kb T)*(([E**2)-([E)**2)]
    return MagneticSus

# Monte Carlo with Metropolis step function
def montecarloMetStep(size, temperature, spins, J, H, Eng, Mag):
    i, j = np.random.randint(0, size, 2)_# randomly select the spin
    dEng = dEnergy(spins, i, j, size, J, H)_# call change in energy calculation function

    #Metropolis algorithm (previous version had  **if dEng > 0**)
    if dEng <= 0 or np.random.rand() < np.exp(-dEng / (kB * temperature)): # statement to check if the move should be accepted
            spins[i, j] *= -1_# flip the spin
            # update the energy and magnetization
            Eng += dEng
            Mag += 2*spins[i,j]
    return spins, Eng, Mag

# function to calculate the change in energy
def dEnergy(spins, i, j, size, J, H):
    # getting values from the nearest cells in the lattice with periodic boundary conditions
    up = spins[(i - 1) % size, j]
    down = spins[(i + 1) % size, j]
    left = spins[i, (j - 1) % size]
    right = spins[i, (j + 1) % size]

    dEng = 2 * spins[i, j] * (J * (up + down + left + right) + H)_# change in energy from spin flip
    return dEng
```

Page 2/2

```python
def hamiltonian(J, H, spins):
    #periodic boundary conditions using np.roll, np.roll is less memory efficent but is faster than other methods of calculating dEnergy
    energy = (-J * (np.sum(spins * np.roll(spins, 1, axis=0)) + np.sum(spins * np.roll(spins, 1, axis=1)))) -(H * np.sum(spins))
    return energy

# visual data plotting function
def plotModel(size, steps, J=1, H=0):
    start_time = time.time()_# getting the start time
    # initalized empty lists to append the values collected
    energies = []
    magnetizations = []
    heat_capacities = []
    magnetic_susceptibilities = []
    temperatures = []

    tempRange = np.linspace(1.0, 4.0, 60)  # temp range from 1-4, creates 60 data points

    for temp in tempRange:
        energy, magnetization, heat_capacity, magnetic_sus = simulate(steps, size, temp, J, H)_# calling the simulation
        # adds the data points to the corresponding lists
        energies.append(energy)
        magnetizations.append(abs(magnetization))
        heat_capacities.append(heat_capacity)
        magnetic_susceptibilities.append(magnetic_sus)
        temperatures.append(temp)

    plt.figure(figsize=(12, 10))

    # Plot energy
    plt.subplot(2, 2, 1)
    plt.plot(temperatures, energies, 'o-')
    plt.title('Energy')
    plt.xlabel('Temperature')
    plt.ylabel('Energy')

    # Plot magnetization
    plt.subplot(2, 2, 2)
    plt.plot(temperatures, magnetizations, 'o-')
    plt.title('Magnetization')
    plt.xlabel('Temperature')
    plt.ylabel('Magnetization')

    # Plot heat capacity
    plt.subplot(2, 2, 3)
    plt.plot(temperatures, heat_capacities, 'o-')
    plt.title('Heat Capacity')
    plt.xlabel('Temperature')
    plt.ylabel('Heat Capacity')

    # Plot magnetic susceptibility
    plt.subplot(2, 2, 4)
    plt.plot(temperatures, magnetic_susceptibilities, 'o-')
    plt.title('Magnetic Susceptibility')
    plt.xlabel('Temperature')
    plt.ylabel('Magnetic Susceptibility')

    end_time = time.time()_# getting the end time
    print(fr'This program took: {end_time - start_time} seconds to run')_# displaying the program run time

    # formating, saving and displaying the plotted data to the screen
    plt.tight_layout()
    plotname = datetime.datetime.now().strftime('%Y-%m-%d_%H-%M-%S')
    plt.savefig(fr'C:\Users\Tommy\PycharmProjects\IsingMonteCarlo\output\{plotname}.png')
    plt.show()

if __name__ == '__main__':
    # input data for size and number of steps
    size = 10  # 10x10 lattice
    steps = 200000  # Monte Carlo steps 31577seconds 8.7hours for 20000000 steps 5217seconds for 2000000, 7904
    # curie temperature should be around 2.5

    # run simulation and plotting
    plotModel(size, steps)
```