CO28, Ferromagnetism

Arnav Prasad

Wadham College, Oxford

January 2019

1 Abstract

We implement the Ising Model of magnetism via the Metropolis algorithm to simulate the probabilistic nature of ferromagnets and their approach to equilibrium. We also study the temperature dependence of magnetisation and phase transition at the critical temperature.

2 Theory

2.1 Introduction

In this practical, we will study *ferromagnetism* using the Ising model and Monte Carlo methods of computation implemented via the Metropolis algorithm.

Magnetism is an inherently quantum phenomenon. Electron spin and its associated magnetic moment are both important factors in the theory of magnetism. Ferromagnetism arises when a lot of such spins align in the same direction to form a macroscopic magnetic moment [1].

The alignment of electron spins is governed by two competing factors: energy minimisation and entropy maximisation [2]. Temperature plays a key role in the balance of the two - below a certain temperature, the ferromagnet has a macroscopic magnetic moment, regardless of the external field applied. However, above this critical temperature, the material loses its magnetism. This is an example of a *phase transition* [3].

We would like to use this model to study how these systems arise, the role that probability and randomness play, how the system is affected by an external magnetic field, and most importantly, the effect of temperature on such a system and its magnetism.

2.2 The Ising Model

The two-dimensional Ising Model describes a ferromagnet as an $N \times N$ square lattice, where each element represents an electron with a certain spin. We will assume that the spin can only be "up" or "down". So, the spin of the ith element, $S_i = \pm 1$.

The energy of a certain configuration is given by:

$$E = -J \sum_{i,j} S_i S_j + B \sum_i S_i \tag{1}$$

Here, the first sum is over the four nearest neighbours of an electron, and the second is over all the sites in the lattice [3].

J is known as the exchange constant and for a ferromagnet, $J > 0$. B is the external magnetic field applied.

From the first term in equation (1), we can see that it is energetically favourable for neighbouring electrons to have spins in the same direction.

For this program, we will be using *periodic boundary conditions.* Therefore, the boundary points that do not have four immediate nearest neighbours will "interact" with the points that are on the geometric opposite end of the lattice. In this way, we can represent the 2D lattice as a 3D torus.

Along with energy considerations, we also have to think about the entropy of the system (there are many more ways the system can be "disordered"). We find that at low temperatures, the ordered configuration is preferred at low temperatures, whereas the disordered configuration is preferred at high temperatures (a more detailed discussion is given in [4]).

For an infinite square lattice, it has been shown [3] that a phase transition takes place at $J/k_BT \approx 0.44$. For a finite square lattice, the sharpness of this critical point will be blurred.

According to statistical mechanics, the probability of a particular configuration is proportional to the Boltzmann factor, $\exp(\frac{-E}{k_BT})$ [3]. This proportionality encapsulates the role of temperature and probability in this system.

2.3 Monte Carlo Method

The Monte Carlo method is a simple but powerful way of simulating such a system using randomness. The Metropolis algorithm is used to sweep through the system and determine the spin at a particular point.

- First, we initialise the $N \times N$ lattice with random values that are either 1 or -1.
- Define a function `sweep()` that iterates over N^2 randomly chosen elements of the lattice, and for each one, computes the energy of the current configuration, and the energy if the spin is flipped.

  If the energy of the flipped configuration is lower, then the spin of that electron is flipped. Otherwise it is flipped with a probability given by:

  $$P(\text{spin flips}) = \exp\left(\frac{-(E_{\text{next}} - E_{\text{current}})}{k_BT}\right) \tag{2}$$

  In the code, this is implemented by comparing $P(\text{spin flips})$ to a randomly generated number in $[0, 1]$.
- We then conduct a number of sweeps (this number is stored in the variable `iterations` in the code) of the full lattice and store the information we wish to analyse in lists.

2.4 Analysis

We then use this model to collect data about the simulated ferromagnet. In particular, we are interested in:

- The *total magnetic moment* (used interchangably with "*magnetisation*" here), defined by $M = \sum_i S_i$ as a function of number of sweeps.

  [NB The magnetisation is "normalised" such that $M = 1$ corresponds to the maximum magnetic moment (all the spins pointing up) and $M = 0$ corresponds to no macroscopic magnetic moment (all the spins cancel out)]
- The *cumulative magnetic moment* (ie. the sum of all the magnetic moments till a point).

  We start storing the cumulative magnetic moment after a certain number of iterations (set as `eqstate` in the code). This is done to avoid the effect of the initial state.
- The average "final" magnetisation as a function of J/k_BT. This will help us better observe the phase transition that takes place at $J/k_BT \approx 0.44$.

3 Results

3.1 Magnetisation data as a function of Monte Carlo steps

The data was collected and analysed for $N = 30$, with `iterations = 1000` and `eqstate = int(iterations/3)`. This was done for three cases:

- $\frac{J}{k_B T} = 0.6$, $\frac{B}{k_B T} = 0.01$

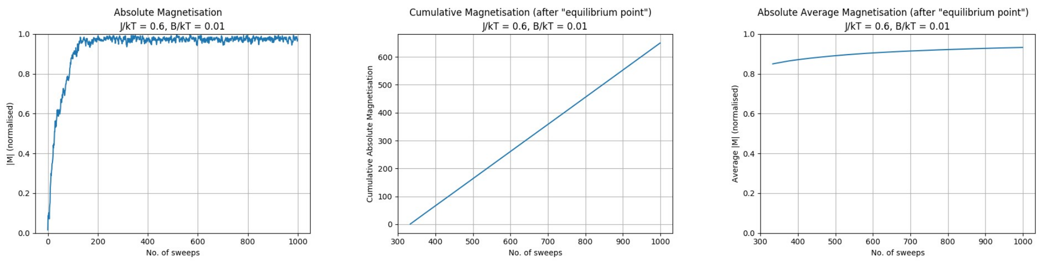

Figure 1: (Left) Absolute Magnetisation, (Centre) Cumulative Magnetisation, (Right) Absolute Average Magnetisation, all plotted against the number of Monte Carlo steps

These values correspond to a relatively low temperature. As we expect from our model, the magnetisation $|M|$ quickly reaches values close to 1 (all spins almost parallel), and doesn't vary a lot around there.

The reason it is always varying is due to the fact that the quantum mechanical effects of randomness and probability are constantly at play, regardless of energy considerations.

The second and third graphs reflect the fact that the magnetisation remains fairly constant once the effects initial state have gone.

- $\frac{J}{k_B T} = 0.4$, $\frac{B}{k_B T} = 0.01$

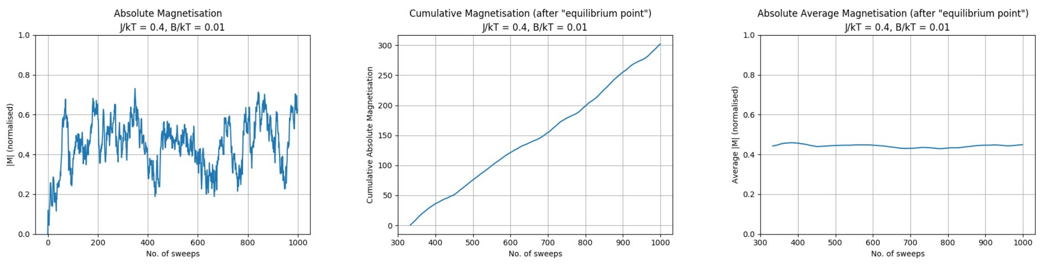

Figure 2: (Left) Absolute Magnetisation, (Centre) Cumulative Magnetisation, (Right) Absolute Average Magnetisation, all plotted against the number of Monte Carlo steps

This value of $\frac{J}{k_B T}$ is close to the phase transition of the ferromagnet, as expected. We can see that $|M|$ takes a wide range of values and never settles near a point. This is what we would expect when the temperature is close to the critical temperature.

The second and third graphs are also not as smooth as in the previous case.

- $\frac{J}{k_B T} = 0.2$, $\frac{B}{k_B T} = 0.01$

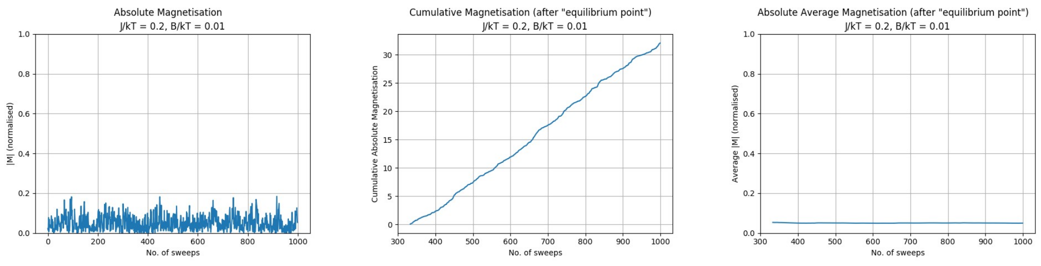

Figure 3: (Left) Absolute Magnetisation, (Centre) Cumulative Magnetisation, (Right) Absolute Average Magnetisation, all plotted against the number of Monte Carlo steps

These values correspond to a relatively high temperature (higher than the critical temperature). We can see that $|M|$ takes values close to zero, indicating the absence of a macroscopic magnetic moment. This is again what we expect, as the magnetization breaks down at temperatures higher than the critical temperature.

The second and third graphs also reflect this fact. We also note the drastic difference in the scale of the second graph.

3.2 Temperature dependence of magnetisation

The data was again collected and analysed for $N = 30$, with `iterations = 1000` and `eqstate = int(iterations/3)`. This was done for three different values of $\frac{B}{k_B T}$.

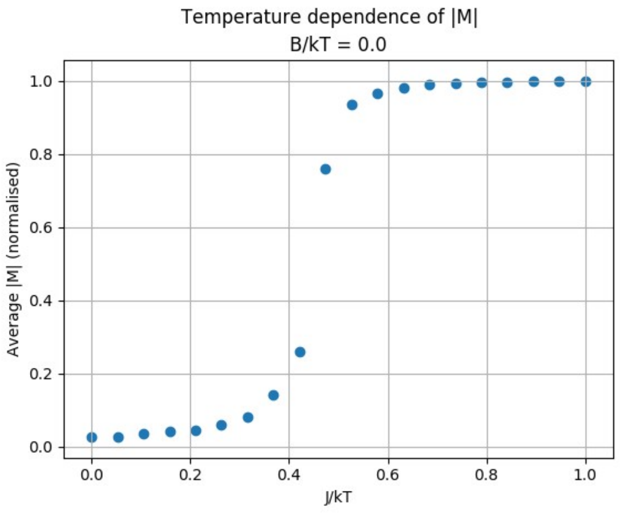

Figure 4: $|M|$ vs $\frac{J}{k_B T}$ for $\frac{B}{k_B T} = 0.0$

This time, the final value of the average absolute magnetisation after 1000 Monte Carlo steps was plotted against $\frac{J}{k_B T}$ for 20 linearly spaced values of $\frac{J}{k_B T}$ between 0 and 1.

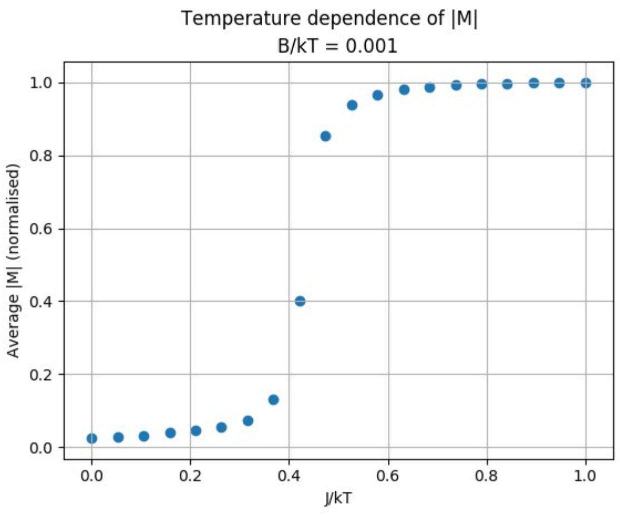

Figure 5: $|M|$ vs $\frac{J}{k_B T}$ for $\frac{B}{k_B T} = 0.001$

These graphs clearly demonstrate the phase transitions that take place around $J/k_B T \approx 0.44$. For values of $J/k_B T$ smaller than this, there is almost no magnetisation, whereas for larger values, the magnetisation is very close to the maximum value.

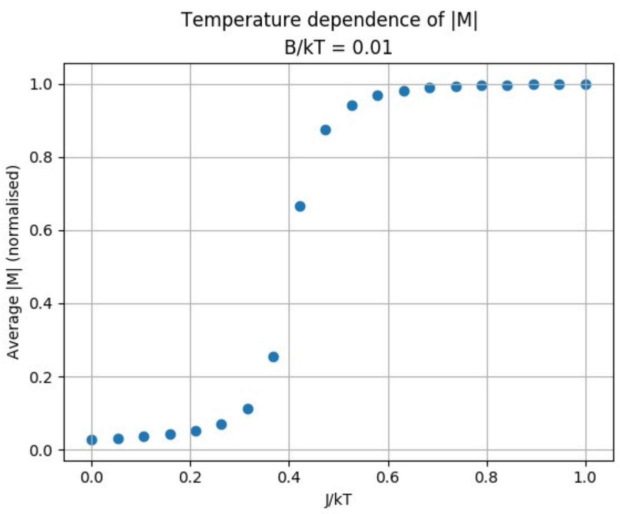

Figure 6: $|M|$ vs $\frac{J}{k_B T}$ for $\frac{B}{k_B T} = 0.01$

We can also see that a higher value of the external field, B, corresponds to a higher value of M around the phase transition. This is because an external field also tries to align the magnetic dipoles in its direction.

4 Conclusion

The implementation of the Ising Model for ferromagnetism using the Metropolis algorithm proved to be a simple yet accurate model for a phenomenon that is, in reality, a lot more complex than the model suggests. The results matched favourably with the theoretical predictions.

This model does, however, have a few drawbacks. It uses a "brute force" approach to simulating this model, and the large number of iterations and sweeps result in the programs taking quite a long time to run. The finite size of lattice also leads to some error in the simulated results.

In conclusion, the Ising Model is an interesting and insightful introduction to phase transitions and a probabilistic QM-based theory of magnetism. It is a very practical and helpful example of Monte Carlo methods, and provides an intuitive understanding of the trade-off between energy minimisation and entropy maximisation through temperature.

References

[1] N. J. Giordano and H. Nakanishi, *Computational Physics.* Prentice-Hall, 2006.

[2] J. Kotze, *Introduction to Monte Carlo methods for an Ising Model of a Ferromagnet.* 2008.

[3] *CO28 Ferromagnetism Lab Script.* Oxford Physics Practical Course, 2018.

[4] S. J. Blundell and K. M. Blundell, *Concepts in Thermal Physics.* Oxford Univ. Press, 2010.

A sweep() function

```
"""
Sweep Function
CO28: Ferromagnetism

Arnav Prasad
Wadham College, Oxford

Created on Thu Nov 29 15:54:25 2018
"""
```

```python
import numpy as np

def energy(Si, Sj, JkT):
    """
    Calculates energy (divided by kT) at a point due to spin of a neighbouring
         particle

    """

    E = -JkT * (Si*Sj)

    return E

def sweep(S_init, JkT, BkT):
    """
    Performs a full sweep of the configuration, given the values of J/kT and B
        /kT

    Input:
        -S_init: (NxN) array of the initial configuration
        -JkT, BkT: The values of J/kT and B/kT

    Output:
        -S_next: (NxN) array of the next configuration after a full sweep

    """
    N = np.shape(S_init)[0]

    S = np.copy(S_init)

    #iterate over N^2 elements (full sweep of the NxN lattice)
    for a in range(N*N):

        #Pick random row of the lattice
        i = np.random.randint(0,N)

        #Pick random column of the lattice
        j = np.random.randint(0,N)

        #We now have a random (i,j) component of the lattice
        #We can now calculate the change in the energy of the configuration if
             its spin flips, and accordingly change the
        #configuration of the lattice (do this N^2 times for a full sweep)

        S_flipped = np.copy(S) #create a copy of S #np.copy() is VERY
            important
        S_flipped[i,j] = -S_flipped[i,j] #copy of S but with the (i,j)
            component's spin flipped

        #Setting up the periodic boundary conditions

        #top row
        if i == 0:

            #top left corner
            if j == 0:

                #energy in current configuration
                Ecurrent = energy(S[i,j], S[i+1, j], JkT) + energy(S[i,j], S[i
                    , j+1], JkT) + energy(S[i,j], S[N-1, j], JkT) + \
                            energy(S[i,j], S[i, N-1], JkT) + BkT * np.sum(S)
                #energy of configuration if spin flips
                Enext = energy(-S[i,j], S[i+1, j], JkT) + energy(-S[i,j], S[i,
                     j+1], JkT) + energy(-S[i,j], S[N-1, j], JkT) + \
                           energy(-S[i,j], S[i, N-1], JkT) + BkT * np.sum(
                              S_flipped)

            #top right corner
            elif j == N-1:

                #energy in current configuration
                Ecurrent = energy(S[i,j], S[i+1, j], JkT) + energy(S[i,j], S[i
                    , j-1], JkT) + energy(S[i,j], S[N-1, j], JkT) + \
                           energy(S[i,j], S[i, 0], JkT) + BkT * np.sum(S)

                #energy of configuration if spin flips
                Enext = energy(-S[i,j], S[i+1, j], JkT) + energy(-S[i,j], S[i,
                     j-1], JkT) + energy(-S[i,j], S[N-1, j], JkT) + \
                           energy(-S[i,j], S[i, 0], JkT) + BkT * np.sum(
                              S_flipped)

            #rest of the top row
            else:

                #energy in current configuration
                Ecurrent = energy(S[i,j], S[i+1, j], JkT) + energy(S[i,j], S[i
                    , j+1], JkT) + energy(S[i,j], S[N-1, j], JkT) + \
                           energy(S[i,j], S[i, j-1], JkT) + BkT * np.sum(S)

                #energy of configuration if spin flips
                Enext = energy(-S[i,j], S[i+1, j], JkT) + energy(-S[i,j], S[i,
                     j+1], JkT) + energy(-S[i,j], S[N-1, j], JkT) + \
                           energy(-S[i,j], S[i, j-1], JkT) + BkT * np.sum(
                              S_flipped)

        #bottom row
        elif i == N-1:

            #bottom left corner
            if j == 0:

                #energy in current configuration
                Ecurrent = energy(S[i,j], S[i-1, j], JkT) + energy(S[i,j], S[i
                    , j+1], JkT) + energy(S[i,j], S[i, N-1], JkT) + \
                           energy(S[i,j], S[0, j], JkT) + BkT * np.sum(S)

                #energy of configuration if spin flips
                Enext = energy(-S[i,j], S[i-1, j], JkT) + energy(-S[i,j], S[i,
                     j+1], JkT) + energy(-S[i,j], S[i, N-1], JkT) + \
                           energy(-S[i,j], S[0, j], JkT) + BkT * np.sum(
                              S_flipped)

            #bottom right corner
            elif j == N-1:

                #energy in current configuration
                Ecurrent = energy(S[i,j], S[i-1, j], JkT) + energy(S[i,j], S[i
                    , j-1], JkT) + energy(S[i,j], S[0, j], JkT) + \
                           energy(S[i,j], S[i, 0], JkT) + BkT * np.sum(S)

                #energy of configuration if spin flips
```

```
            Enext = energy(-S[i,j], S[i-1, j], JkT) + energy(-S[i,j], S[i,
                 j-1], JkT) + energy(-S[i,j], S[0, j], JkT) + \
                        energy(-S[i,j], S[i, 0], JkT) + BkT * np.sum(
                            S_flipped)

        #rest of the bottom row
        else:

            #energy in current configuration
            Ecurrent = energy(S[i,j], S[i-1, j], JkT) + energy(S[i,j], S[i
                , j-1], JkT) + energy(S[i,j], S[0, j], JkT) + \
                        energy(S[i,j], S[i, j+1], JkT) + BkT * np.sum(S)

            #energy of configuration if spin flips
            Enext = energy(-S[i,j], S[i-1, j], JkT) + energy(-S[i,j], S[i,
                 j-1], JkT) + energy(-S[i,j], S[0, j], JkT) + \
                        energy(-S[i,j], S[i, j+1], JkT) + BkT * np.sum(
                            S_flipped)

    #leftmost column
    elif j == 0:

        #energy in current configuration
        Ecurrent = energy(S[i,j], S[i-1, j], JkT) + energy(S[i,j], S[i, N
            -1], JkT) + energy(S[i,j], S[i+1, j], JkT) + \
                    energy(S[i,j], S[i, j+1], JkT) + BkT * np.sum(S)

        #energy of configuration if spin flips
        Enext = energy(-S[i,j], S[i-1, j], JkT) + energy(-S[i,j], S[i, N
            -1], JkT) + energy(-S[i,j], S[i+1, j], JkT) + \
                    energy(-S[i,j], S[i, j+1], JkT) + BkT * np.sum(
                        S_flipped)

    #rightmost column
    elif j == N-1:

        #energy in current configuration
        Ecurrent = energy(S[i,j], S[i-1, j], JkT) + energy(S[i,j], S[i, j
            -1], JkT) + energy(S[i,j], S[i+1, j], JkT) + \
                    energy(S[i,j], S[i, 0], JkT) + BkT * np.sum(S)

        #energy of configuration if spin flips
        Enext = energy(-S[i,j], S[i-1, j], JkT) + energy(-S[i,j], S[i, j
            -1], JkT) + energy(-S[i,j], S[i+1, j], JkT) + \
                    energy(-S[i,j], S[i, 0], JkT) + BkT * np.sum(S_flipped)

    else:

            #energy in current configuration
            Ecurrent = energy(S[i,j], S[i+1, j], JkT) + energy(S[i,j], S[i
                , j+1], JkT) + energy(S[i,j], S[i-1, j], JkT) + \
                        energy(S[i,j], S[i, j-1], JkT) + BkT * np.sum(S)

            #energy of configuration if spin flips
            Enext = energy(-S[i,j], S[i+1, j], JkT) + energy(-S[i,j], S[i,
                 j+1], JkT) + energy(-S[i,j], S[i-1, j], JkT) + \
                        energy(-S[i,j], S[i, j-1], JkT) + BkT * np.sum(
                            S_flipped)

    #Ratio of weight factors
    r = np.exp(Ecurrent-Enext) #factor of 1/kT is included in Ecurrent and
         Enext
    #Change in energy is negative
    if r >= 1:
        S[i,j] = -S[i,j] #spin flips

    #Change in energy is positive
    else:
        #spin flips with probability r, so we compare r to a random number
             in [0,1]
        if r >= np.random.rand():
            S[i,j] = -S[i,j] #spin flips

S_next = np.copy(S)

return S_next
```

B Program for Section 3.1

```
"""
C028: Ferromagnetism

Study of approach to equilibrium state of magnetisation using the Ising model

Arnav Prasad
Wadham College, Oxford

Created on Thu Nov 29 15:07:09 2018
"""

import numpy as np
import matplotlib.pyplot as plt

from sweep import sweep

N = 30 #Elements in the sides of the square lattice

JkT = 0.6 #Value of J/kT
BkT = 0.01 #Value of B/kT

#Prepare lattice (NxN array) with randomly distributed values of plus/minus 1

S_init = np.random.rand(N,N) #create NxN array with random numbers in [0,1]

S = np.copy(S_init)

#loop through S and adjust numbers to be either 1 or -1
for i in range(N):
    for j in range(N):
        S[i,j] = (round(S[i,j]))
        if S[i,j] == 0:
            S[i,j] = -1

maxM = N*N #To normalise M such that |M|=1 corresponds to all spins pointing
    in the same direction

M = [] #Empty list to store magnetisation (normalised)
absM = [] #Empty list to store absolute value of magnetisation (normalised)
avgabsM = [] #Empty list to store average absolute value of magnetisation

iterations = 1000 #No. of sweeps we want to conduct
```

```
#To start after a few sweeps are done so that we are close to the equilibrium
    state
#(and to avoid the effect of the initial state)
eqstate = int(iterations/3)

#Set up empty lists for storing data after our chosen "equilibrium state"
eqsweep = [] #sweep no. after "equilibrium"
eqM = [] #absolute magnetisation value after "equilibrium"

#Do multiple sweeps and store the information related to magnetisation in a
    list
for i in range(iterations):
    S = sweep(S, JkT, BkT)

    M.append(np.sum(S)/maxM) #Store value of M after i sweeps
    absM.append(abs(np.sum(S)/maxM)) #Store value of |M| after i sweeps

    #For points after our chosen "equilibrium state"
    if i >= eqstate:

        eqsweep.append(i)
        eqM.append(abs(np.sum(S)/maxM))
        avgabsM.append(sum(absM)/len(absM))


#Plot(s) for studying approach to equilibrium

#Plot |M| vs no. of sweeps
plt.figure(1)
plt.plot(range(iterations), absM)
plt.grid(True)
plt.ylim(0,1)
plt.xlabel('No. of sweeps')
plt.ylabel('|M| (normalised)')
plt.suptitle('Absolute Magnetisation')
plt.title('J/kT = ' + str(JkT) + ', B/kT = ' + str(BkT))

#Plot cumulative sum of M vs no. of sweeps (after "equilibrium point")
plt.figure(2)
plt.plot(eqsweep, np.cumsum(eqM))
plt.suptitle('Cumulative Magnetisation (after "equilibrium point")')
plt.xlabel('No. of sweeps')
plt.ylabel('Cumulative Absolute Magnetisation')
plt.grid(True)
plt.title('J/kT = ' + str(JkT) + ', B/kT = ' + str(BkT))

#Plot average absolute magnetisation vs no. of sweeps
plt.figure(3)
plt.plot(eqsweep, avgabsM)
plt.grid(True)
plt.ylim(0,1)
plt.xlabel('No. of sweeps')
plt.ylabel('Average |M| (normalised)')
plt.suptitle('Absolute Average Magnetisation (after "equilibrium point")')
plt.title('J/kT = ' + str(JkT) + ', B/kT = ' + str(BkT))
```

C Program for Section 3.2

```
"""
C028: Ferromagnetism
```

```
Study of temperature dependence of magnetisation using the Ising model

Arnav Prasad
Wadham College, Oxford

Created on Thu Nov 29 15:07:09 2018
"""

import numpy as np
import matplotlib.pyplot as plt

from sweep import sweep

N = 30 #Elements in the sides of the square lattice

JkTrange = np.linspace(0,1,20) #Values of J/kT
BkT = 0.01 #Value of B/kT

#Prepare lattice (NxN array) with randomly distributed values of plus/minus 1

S_init = np.random.rand(N,N) #create NxN array with random numbers in [0,1]

S = np.copy(S_init)

#loop through S and adjust numbers to be either 1 or -1
for i in range(N):
    for j in range(N):
        S[i,j] = (round(S[i,j]))
        if S[i,j] == 0:
            S[i,j] = -1

maxM = N*N #To normalise M such that |M|=1 corresponds to all spins pointing
    in the same direction

iterations = 1000 #No. of sweeps we want to conduct for each value of J/kT

#To start after a few sweeps are done so that we are close to the equilibrium
    state
#(and to avoid the effect of the initial state)
eqstate = int(iterations/3)

finalM = []

for JkT in JkTrange:

    M = [] #Empty list to store magnetisation (normalised)
    absM = [] #Empty list to store absolute value of magnetisation (normalised
        )
    avgabsM = [] #Empty list to store average absolute value of magnetisation

    #Do multiple sweeps and store the information related to magnetisation in
        a list
    for i in range(iterations):
        S = sweep(S, JkT, BkT)

        M.append(np.sum(S)/maxM) #Store value of M after i sweeps
        absM.append(abs(np.sum(S)/maxM)) #Store value of |M| after i sweeps

        #For points after our chosen "equilibrium state"
        if i >= eqstate:
            avgabsM.append(sum(absM)/len(absM))
    finalM.append(avgabsM[-1])

#Plot average |M| vs J/kT
plt.figure(1)
plt.scatter(JkTrange, finalM)
plt.grid(True)
plt.suptitle('Temperature dependence of |M|')
plt.ylabel('Average |M| (normalised)')
plt.xlabel('J/kT')
plt.title('B/kT = ' + str(BkT))
```