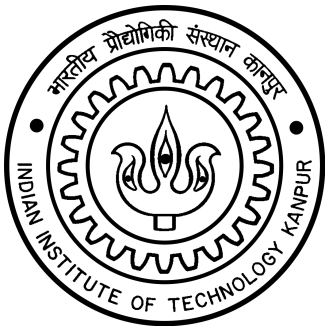
PHY473A: Project Report
Observing phase transitions in 2D Ising model

Adityaa Bajpai (150056)
Navya Gupta (150440)

Under the guidance of: **Prof. MK Verma**



Contents

1 The Ising Model

The Ising model [Hua87] captures the dynamics of a lattice of magnetic dipole moments. The model consists of lattice sites labelled by i, and each lattice site is occupied by a spin s_i which may take the value $\pm\frac{1}{2}$. The Hamiltonian for the system is given by:

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

where s, J and h_i denote the spin configuration, coupling constant and magnetic field at the ith location respectively.

2 Metropolis Algorithm

Physically, we are given a lattice of randomly oriented and interacting spins at a given temperature (with no external magnetic field, for simplicity). The objective is to evolve this system for a sufficiently long time and to arrive at the equilibrium configuration dictated by the canonical ensemble. The most popular way to simulate this evolution in a computer is the Metropolis algorithm [Met+53]. The key steps involved in this algorithm are:

1. Start with the first site on the lattice.
2. If the system loses energy after flipping the spin at this site, then flip it.
3. If the system gains energy ΔE after flipping the spin at this site, flip it with the probability $e^{-\frac{\Delta E}{k_B T}}$.
4. Move to the next spin and repeat this process. Once all spin sites have been visited once, go back to the first spin site and repeat the process all over again. But how many times should we repeat this cycle? In this case, we have arrived at the number by trial-and-error. However, more complicated algorithms like the Propp-Wilson algorithm [Häg02] are capable of determining the exact number of iterations needed.

3 Our implementation

Some salient features about our implementation are given below. Then full code can be found in the appendix.

- The function `E_cal()` calculates the change in energy post flipping for every spin on the lattice and stores them in an array `dE[ ]`.
- The function `E_cor()` is called every time a spin is flipped. It corrects the energies of the spin under consideration and its nearest neighbours, stored in the array dE[ ]. This is used instead of simply calling `E_cal()` because most of the spins from that redundant calculation.
- The function `Eq()` repetitively calculates the energies for all the spins and then flips under the conditions mentioned in the previous section.
- The code is run on a 30 X 30 lattice and is evolved at 100 temperatures between the temperatures 1.5- to 3.5-J/k.
- The process of flipping spins is performed over 100 iterations to get the lattice to equilibrium, we arrived at this value by trial and error as stated in the previous section.
- Magnetization and Internal Energy is averaged over 100 iterations of the function `Eq()`.

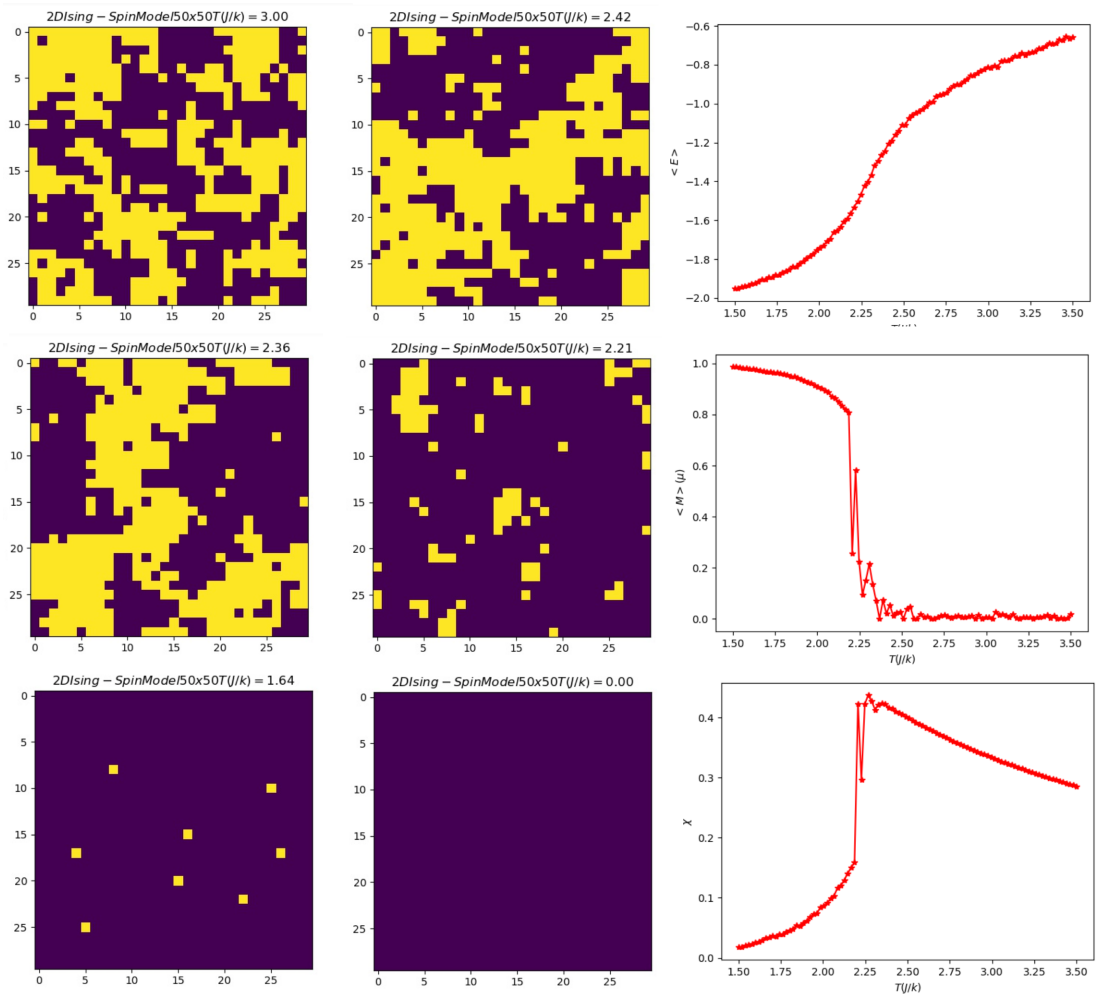


Figure 1: Results of our simulation for various temperatures (left) and (top to bottom on right) plots of average energy per site, average magnetization per site and susceptibility as a function of temperature for ferromagnetic coupling.

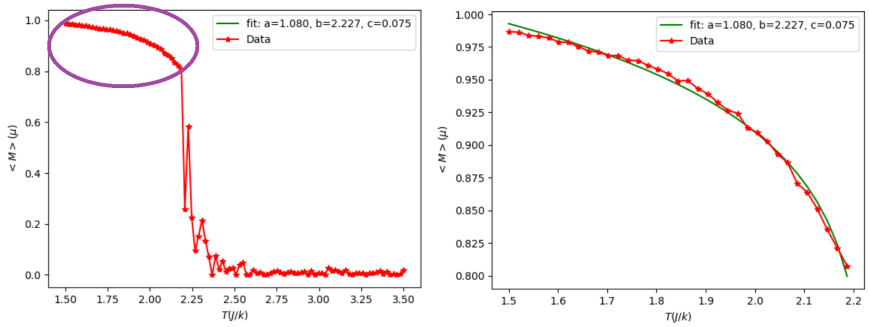


Figure 2: Plot of magnetization per site as a function of time: Theoretical fit and experimental data.

4 Results and discussion

We ran our code for the 30×30 lattice for $J = 1$ (ferromagnetic coupling) and $J = -1$ (anti-ferromagnetic coupling) for various values of the temperature, and found that the results agree very well with theoretical

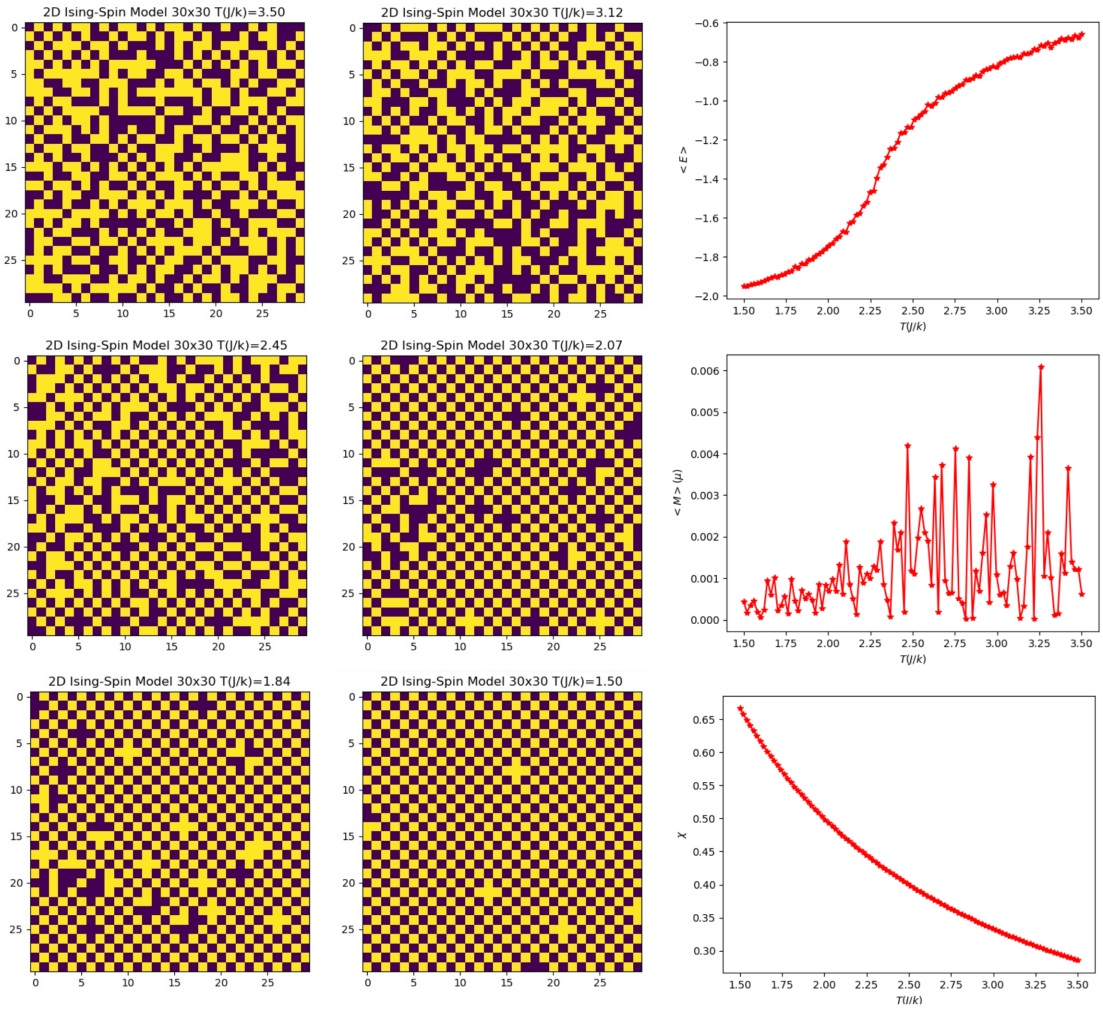


Figure 3: Results of our simulation for various temperatures (left) and (top to bottom on right) plots of average energy per site, average magnetization per site and susceptibility as a function of temperature for antiferromagnetic coupling.

predictions.

In figure 1, we see that the system becomes more and more ordered as the temperature is lowered. Further, a clear phase transition is seen in the plots of energy per site, magnetization per site and susceptibility as a function of temperature. In figure 2, we have fitted the magnetization per site $< m(T) >$ as a function of temperature T to the theoretical curve:

$$< m(T) >= a\Big(1 - \frac{T}{b}\Big)^c \tag{2}$$

From the fit, we find that $T_c = b = 2.227$-J/k, which is within 1.8% of the theoretical value of 2.269-J/k.

Figure 3 shows the results for antiferromagnetic coupling. We clearly see the characteristic antiferromagnetic spin pattern emerge as we lower the temperature. The phase transition is, however, not seen so clearly in the graphs for magnetization per sitea and susceptibility. This is because the total value of magnetization is close to zero due to the antiferromagnetic coupling.

5 Future work

Using our code we can perform numerous other simulations. Particularly, it would be interesting to see:

- What happens to T_c we increase the lattice size/ change its shape?
- How do free and periodic boundary conditions affect the physics of the system?
- What happens if we turn on an external magnetic field?
- Can we observe spontaneous symmetry breaking in our simulation?
- What happens to the correlation functions as we increase temperature?
- What happens if we extend the interaction to more neighbours?

We will try to answer these questions in our future work.

6 Acknowledgments

We extend a heartfelt thanks to Prof. MK Verma for teaching us this enjoyable course and for offering us with an opportunity to work on this project. We are also grateful to all the TAs for their guidance and support.

7 Appendix: Our code

```
import numpy as np
from scipy import constants
import matplotlib.pyplot as plt
import time
strt=time.clock() # to time the code

#All the numbers that you will need to change are here
n=30  #size
m=100
i_avg=100 #averaging iterations
i_eq=(10**2) #iteration to reach equilibrium
J=1      #coupling constant

#Initialization of arrays
Cv=np.zeros(m)
X=np.zeros(m)     #susceptibility
M=np.zeros(m)
M_2=np.zeros(m)   #<M^2>
E=np.zeros(m)
E_2=np.zeros(m)   #<E^2>
S=np.ones((n,n,m))
dE=np.zeros((n,n,m))
t=np.linspace(1.5,3.5,m)   #Temp(J/k)

#filling the spin array with 0,1 randomly
for i in range(0,n):
        for j in range(0,n):
                S[i,j]=np.random.randint(0,1)
```

```
# Calculates the energy in each state of S[] at tempt[k]
def calc_E(k):
        for i in range(0,n):
                for j in range(0,n):
                        dE[i,j,k]=8*J*(S[i,j,k]-0.5)*(S[(i+1)%n,j,k]+S[(i-1)%n,j,k]+
                        S[i,(j+1)%n,k]+S[i,(j-1)%n,k]-2)
        return

# Corrects the dE of the nearest neighbours after every flip
def E_cor(i0,j0,k):

        dE[i0,j0,k]=8*J*(S[i0,j0,k]-0.5)*(S[(i0+1)%n,j0,k]+S[(i0-1)%n,j0,k]+S[i0,(j0+1)%n,k]+
        S[i0,(j0-1)%n,k]-2)
        i=i0+1
        j=j0
        dE[i%n,j,k]=8*J*(S[i%n,j,k]-0.5)*(S[(i+1)%n,j,k]+S[(i-1)%n,j,k]+S[i%n,(j+1)%n,k]+
        S[i%n,(j-1)%n,k]-2)
        i=i0-1
        j=j0
        dE[i%n,j,k]=8*J*(S[i%n,j,k]-0.5)*(S[(i+1)%n,j,k]+S[(i-1)%n,j,k]+S[i%n,(j+1)%n,k]+
        S[i%n,(j-1)%n,k]-2)
        i=i0
        j=j0+1
        dE[i,j%n,k]=8*J*(S[i,j%n,k]-0.5)*(S[(i+1)%n,j%n,k]+S[(i-1)%n,j%n,k]+S[i,(j+1)%n,k]+
        S[i,(j-1)%n,k]-2)
        i=i0
        j=j0-1
        dE[i,j%n,k]=8*J*(S[i,j%n,k]-0.5)*(S[(i+1)%n,j%n,k]+S[(i-1)%n,j%n,k]+S[i,(j+1)%n,k]+
        S[i,(j-1)%n,k]-2)



        return


#flips spin
def flip(i,j,k):
        if(S[i,j,k]==1.0):
                S[i,j,k]=0.0
        else:
                S[i,j,k]=1.0

        return

#performs the flipings untill equilibrium is achieved
def Eq(T,k):
        for l in range(0,i_eq):
                calc_E(k)
                for i in range(0,n):
                        for j in range(0,n):
                                if(dE[i,j,k]<0):
                                        flip(i,j,k)
                                        E_cor(i,j,k)
```

```
                                else:
                                        r=np.random.random()
                                        p=np.exp(-dE[i,j,k]/T)
                                        if(r<=p):
                                                flip(i,j,k)
                                                E_cor(i,j,k)
        return


########## calculating <M>, <E>, Cv, susceptibility ############

for k in range(0,m):
        for q in range(0,i_avg):
                Eq(t[k],k)

                for i in range(0,n):
                        for j in range(0,n):
                                M[k]=M[k]+2*S[i,j,k]-1
                                M_2[k]=M_2[k]+(2*S[i,j,k]-1)**2
                                E[k]=E[k]+dE[i,j,k]

        M[k]=abs(M[k]/((n**2)*i_avg))
        E[k]=-E[k]/((n**2)*i_avg*4)
        M_2[k]=M_2[k]/((n**2)*i_avg)
        X[k]=(1/t[k])*(M_2[k]-(M[k]**2))

        print (float((k+1)*100)/m), '%'  #prints the percentage of the code completed

#saving data Will be used for fitting and calculation of Tc


np.save('Dising_X_f',X)
np.save('Dising_E_f',E)
np.save('Dising_M_f',M)
np.save('Dising_S_f',S)



h=int(time.clock()-strt)/(60*60)
m=int((time.clock()-strt)%(60*60))/60
s=((time.clock()-strt)%(60*60))%60
print'time=',h,'hrs',m,'min',s,'sec'
```

References

[Met+53] Nicholas Metropolis et al. “Equation of state calculations by fast computing machines”. In: *The Journal of Chemical Physics* 21.6 (1953), pp. 1087–1092.

[Hua87] Kerson Huang. *Statistical mechanics*. New York: Wiley, 1987.

[Häg02] Olle Häggström. *Finite Markov Chains and Algorithmic Applications*. Cambridge University Press, 2002.