PHY 410
Homework Assignment 10

Han Wen

Person No. 50096432

December 5, 2014

Abstract

The goal of this assignment is to get more familiar with random process and method, as well as its application on random walk and statistical mechanics.

Contents

1 Problem 1 **2**
- 1.1 Description . . . 2
- 1.2 Numerical result and analysis . . . 2

2 Problem 2 **4**
- 2.1 Description . . . 4
- 2.2 Result . . . 4

3 Problem 3 **8**
- 3.1 Description . . . 8
- 3.2 Result . . . 8
  - 3.2.1 a . . . 8
  - 3.2.2 b . . . 8
  - 3.2.3 c . . . 8

A Appendix **11**
- A.1 python code . . . 11

1 Problem 1

1.1 Description

Modify rwalk.cpp or rwalk.py to simulate a random walk on a 2-dimensional square lattice and a 3-dimensional cubic lattice. Measure the diffusion constant D in 2 and 3 dimensions and compare with the 1-dimensional walk and theoretical expectations.

1.2 Numerical result and analysis

Theoretically, for example in one dimension, assume for each step the displacement will be $\pm\delta$, thus the position after N steps can be expressed as:

$$x^2(N) = [x(N-1) \pm \delta]^2 = x^2(N+1) \pm 2\delta x(N-1) + \delta^2$$

Thus the average will be:

$$\langle x^2(N)\rangle = \langle x^2(N-1)\rangle \pm 2\delta x(N-1) + \delta^2$$

the middle term is zero, therefore we have:

$$\langle x^2(N)\rangle = \langle x^2(N-1)\rangle + \delta^2$$

consequently:

$$\langle x^2(N)\rangle = N\delta^2$$

In 2 and 3 dimensions it is nothing but the summation in each individual direction. In my case, for each step the displacement in x, y, z direction is 1, therefore the diffusion constants for 1, 2, 3 dimensions are 0.5, 1, 1.5.

Modify the code to simulate for 2 and 3 dimensions, for 2 dimensions, since the random number generator is not a perfect one and limited by the number of steps, when using 1000 ensembles and 1000 steps the result is 1.017 ± 0.001 Fig. 1 , while using 10000, 10000 setting, the result is 0.9990 ± 0.0001very close to theoretical value. Fig 2

For 3 dimensions: for 1000 walkers 1000 steps, the result is 1.499 ± 0.001 Fig. 3

Since it is a random process, each time the result will be different and roughly fluctuating around the theoretical value, I was quite lucky in this trial, thus I stopped here.

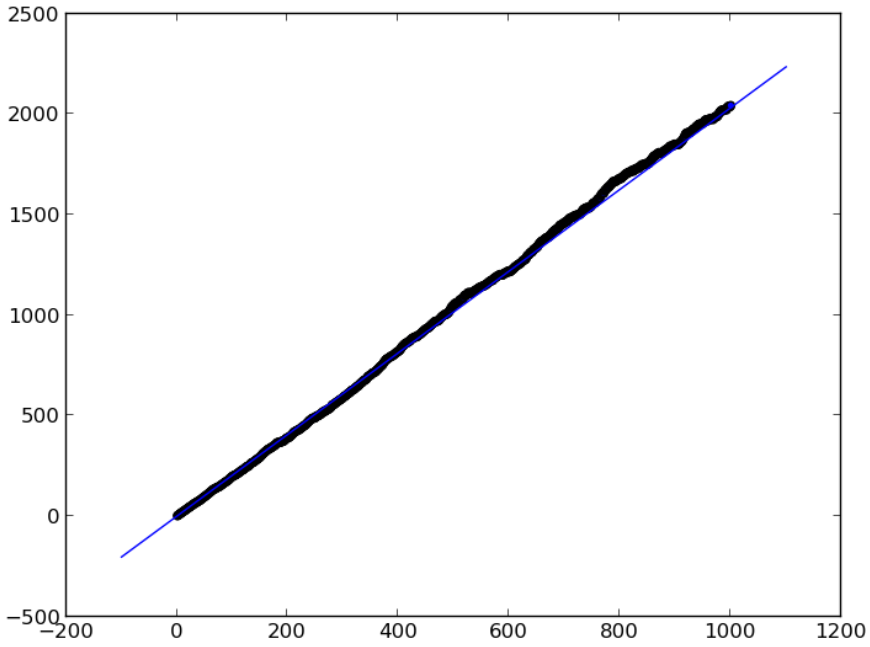

Figure 1: 2d average of $displacement^2$ vs. steps, 1000 walkers

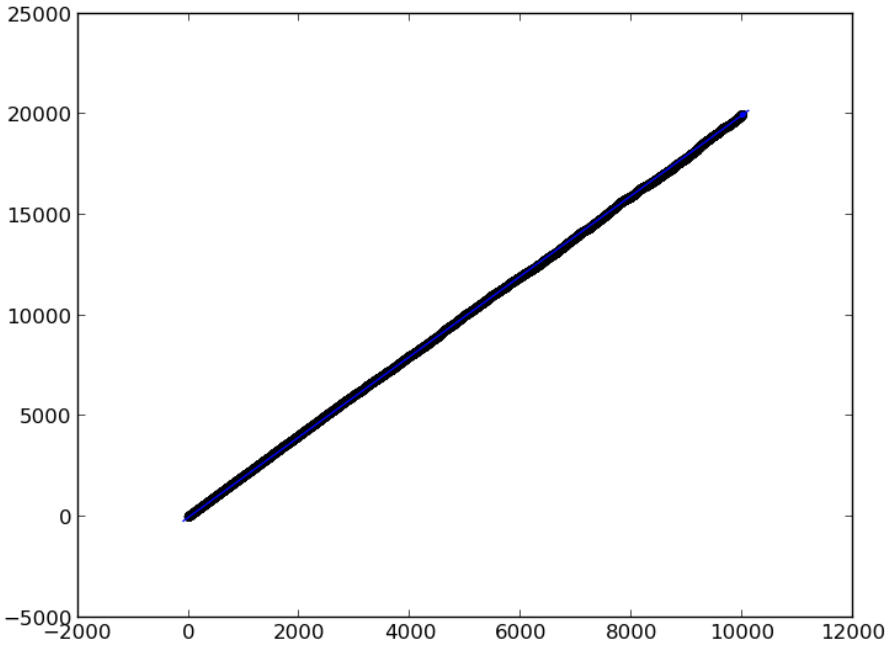

Figure 2: 2d average of $displacement^2$ vs. steps, 10000 walkers

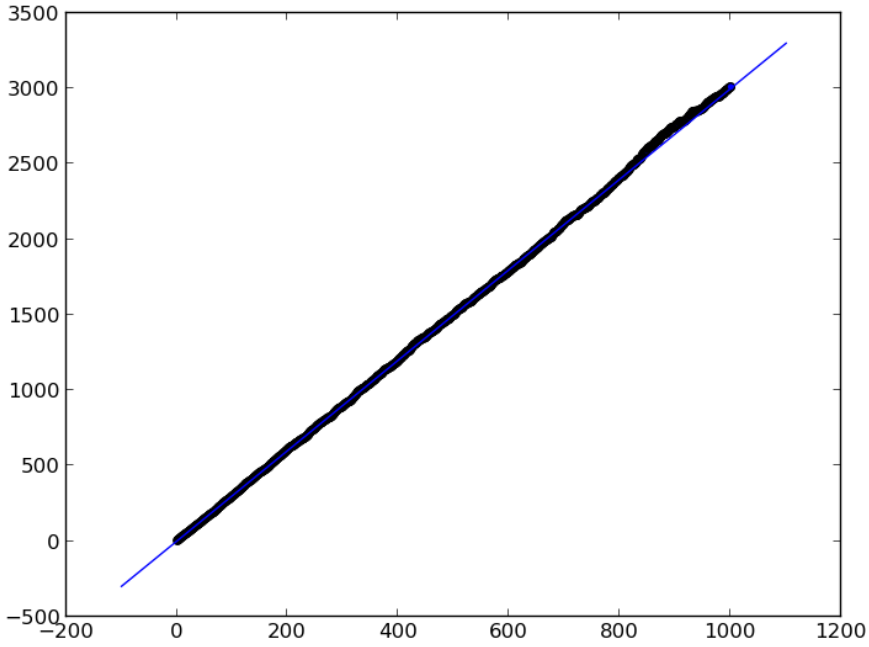

Figure 3: 3d average of $displacement^2$ vs. steps, 1000 walkers

2 Problem 2

2.1 Description

Measure and plot the average magnetization $\langle |M| \rangle$ and Heat Capacity C of the Ising model as a function of temperature T for three different lattice sizes. Use the data to identify the transition temperature T_C .

2.2 Result

I used the lattice of size 10x10, 50x50, 100x100, using 500 MC steps.(This is a fair number for 10x10 yet not sufficient for larger grids, however, limited by the performance even using c++, I chose this number, we can see below for large grids the result is not so obvious)

For 10x10: Fig. 4 For 50x50: Fig. 5 For 100x100: Fig. 5

We can see for larger lattice, since I didn't use enough number of steps, the pattern is not as obvious as that of 10x10, yet we can still see the shape of it. To better evaluate the transition temperature, I applied FFT and got rid of the high frequency waves, as you can see in this plot: 7 8 9.

After fft, we can see for 10x10, 50x50, 100x100, the $T_c = 2.62, 2.22, 2.23$ We can see later in the third problem, when the lattice is infinitely large, approximately the $T_c = 2.27$, my result here is really close, considering the limit on the size and the random nature of the algorithm. I am going to use chi-square fit later in problem 3

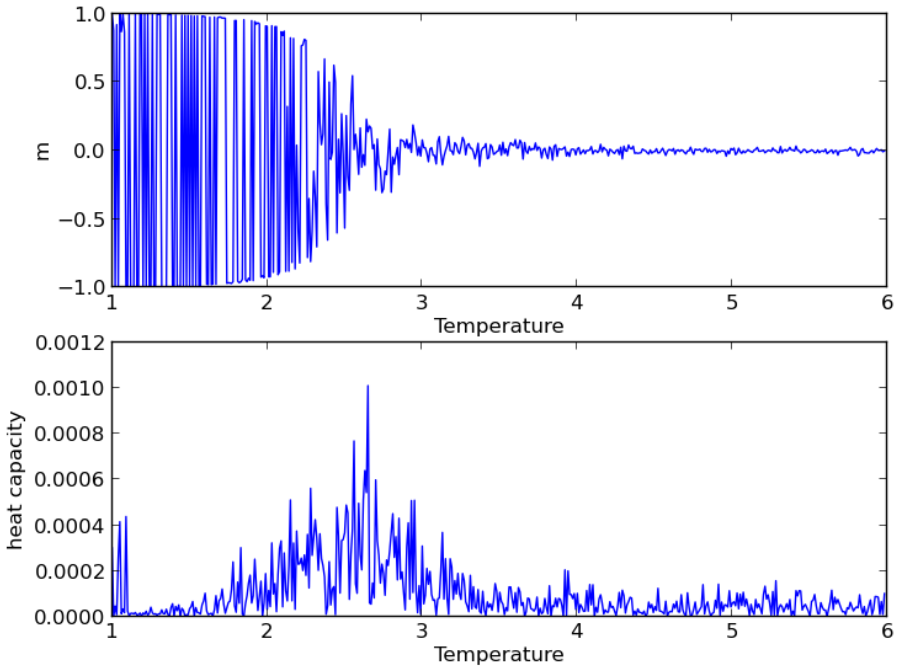

Figure 4: 10x10 lattice m and c vs. T

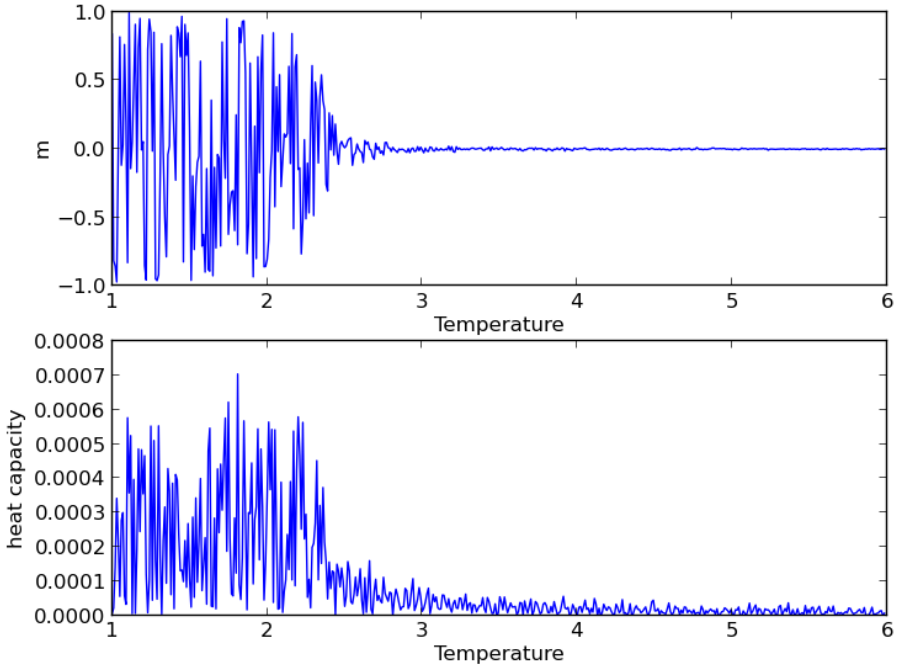

Figure 5: 50x50 lattice m and c vs. T

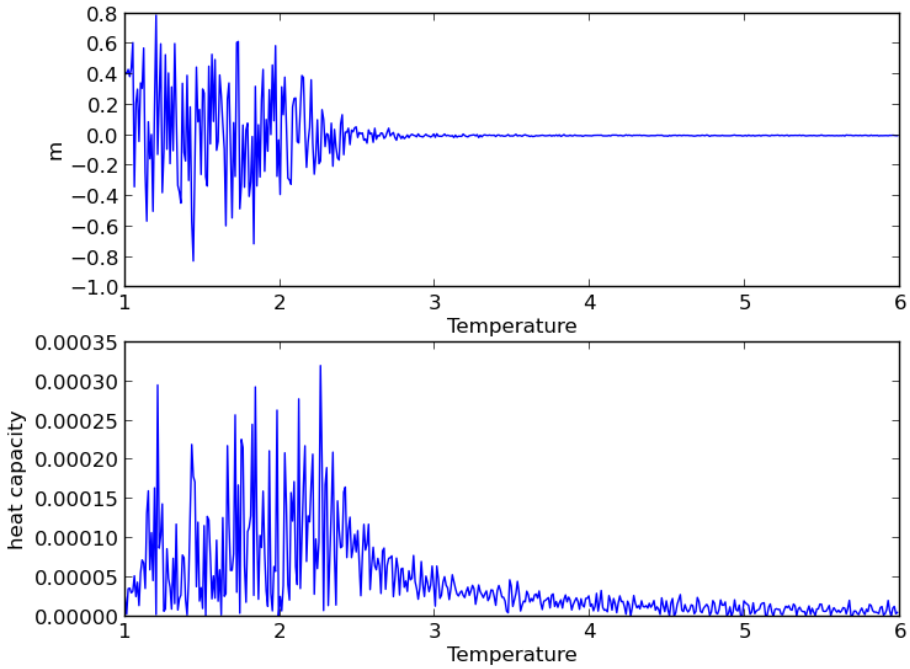

Figure 6: 100x100 lattice m and c vs. T

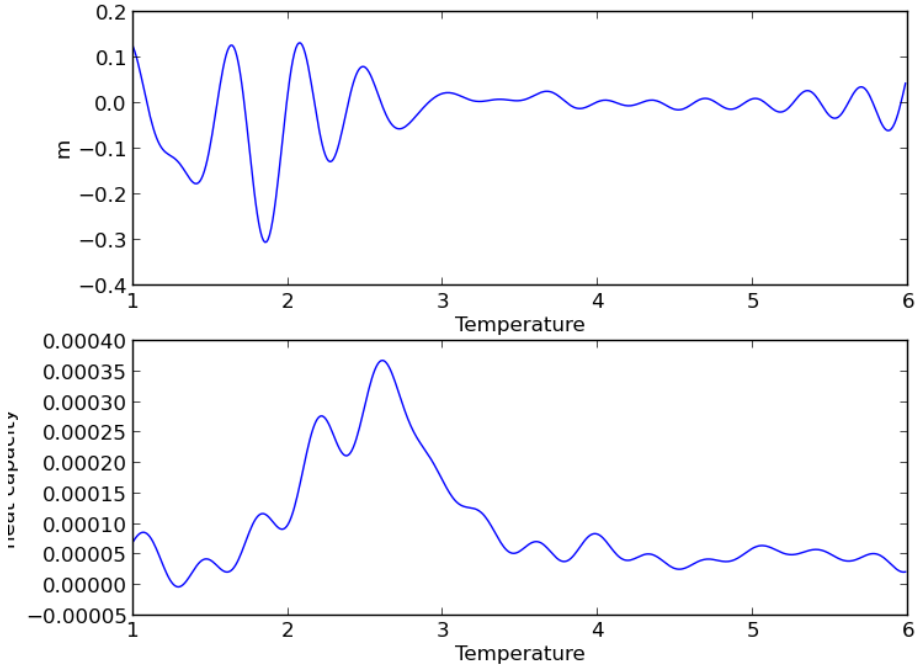

Figure 7: 10x10 fft lattice m and c vs. T

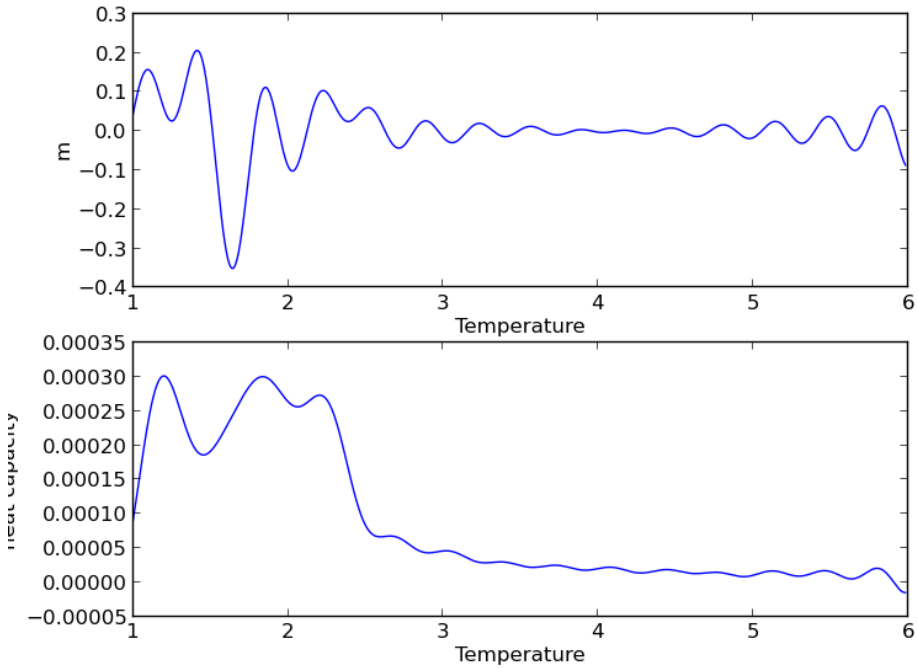

Figure 8: 50x50 fft lattice m and c vs. T

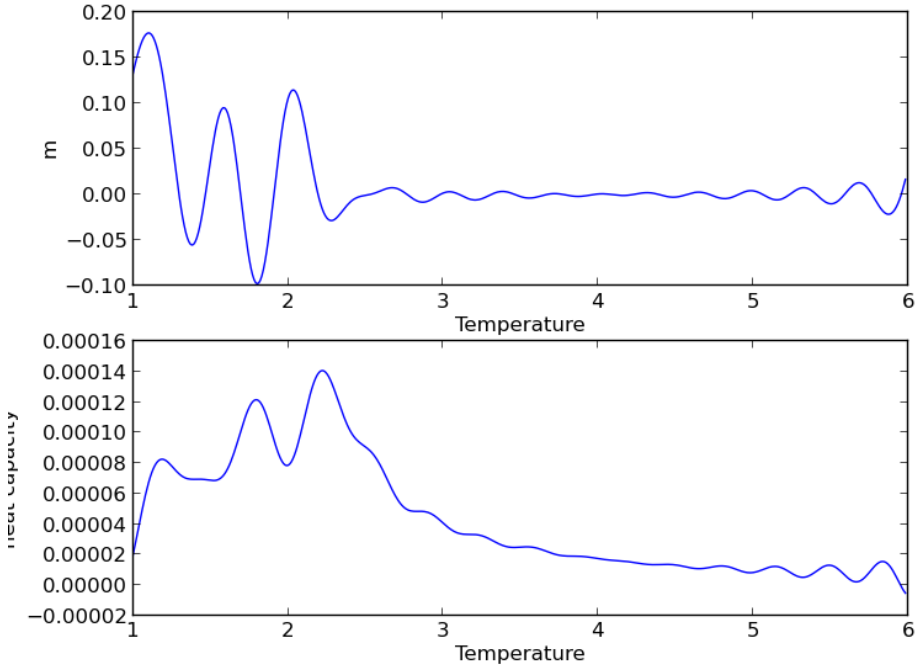

Figure 9: 100x100 fft lattice m and c vs. T

Problem 3 (PHY 505 ONLY) : Consider the Ising model for a 2-d simplified ferromagnet. In the thermodynamic limit $N \to \infty$ and zero magnetic field $H = 0$, the magnetization per spin satisfies :

$$m = \lim_{N\to\infty} \frac{\langle \sum_i s_i \rangle}{N} = \begin{cases} \left[1 - \left\{\sinh\left(\frac{2J}{k_B T}\right)\right\}^{-4}\right]^{1/8}, & \text{for } T \le T_c \\ 0, & \text{for } T > T_c \end{cases}$$

Close to the critical temperature, the magnetization per spin satisfies :

$$m \sim (T_c - T)^{\beta} ,$$

where β is a critical exponent. Onsager's formula shows that $\beta = 1/8$ for the 2-D Ising model.

a) Recall that sudden reversals of the magnetization occur from time to time in systems of finite time. From the program "`ising`" in Lecture 32, what is a reasonable value of L to be chosen to maintain one single domain instead of flipping throughout? Show a plot of the average magnetization per spin.

b) Recall from class that the average magnetization per spin (m) can be estimated using the above formula in the thermodynamic limit. Using the above value of L from Problem 1a, what is the minimum number of Monte Carlo steps that should be taken to ensure proper coverage in the "metropolis" algorithm? Why?

c) Using the values for L and number of steps from a and b respectively, run the "ising" program for H=0., and vary T from 2.0 to 2.5 to compute the average magnetization $\langle |M| \rangle$ as a function of temperature T . Use the data to identify the transition temperature T_c using the above relationship for m. Hint : you may find it useful to use one of the chi-squared fitting functions from earlier in the semester (Lecture 2).

Figure 10: p3 description, sorry I am a little lazy

3 Problem 3

3.1 Description

3.2 Result

3.2.1 a

To theoretically determine the proper number of L require probability theory and using Gaussian distribution. Yet that might be too troublesome here, after several trials I found L=30 should be a reasonable number. Here is the plot: 11

3.2.2 b

I firstly found all the Boltzmann factors are:0.026347980814448734
37.95357249735125
0.16232061118184818
6.160647084304639
1.0
1.0
6.160647084304639
0.16232061118184818
37.95357249735125
0.026347980814448734
According the the Metropolis algorithm, firstly we have fifty fifty chance to have a spin=1 or 0(s_i), for spin=0, the factor will always be 0 and this trial will not be used. For spin=1, after multiplying with Boltzmann factor, we can see six out of nine B-factors is larger than 1 and when times them, the trial will be effective. For other three, we can roughly think the chance the trial can be used is $0.11(\frac{1}{3}\times 0.1623\times 2+\frac{1}{3}\times 0.026)$. Thus, the chance one trial can be used is $0.5\times(\frac{2}{3}+0.11) = 0.388$. A

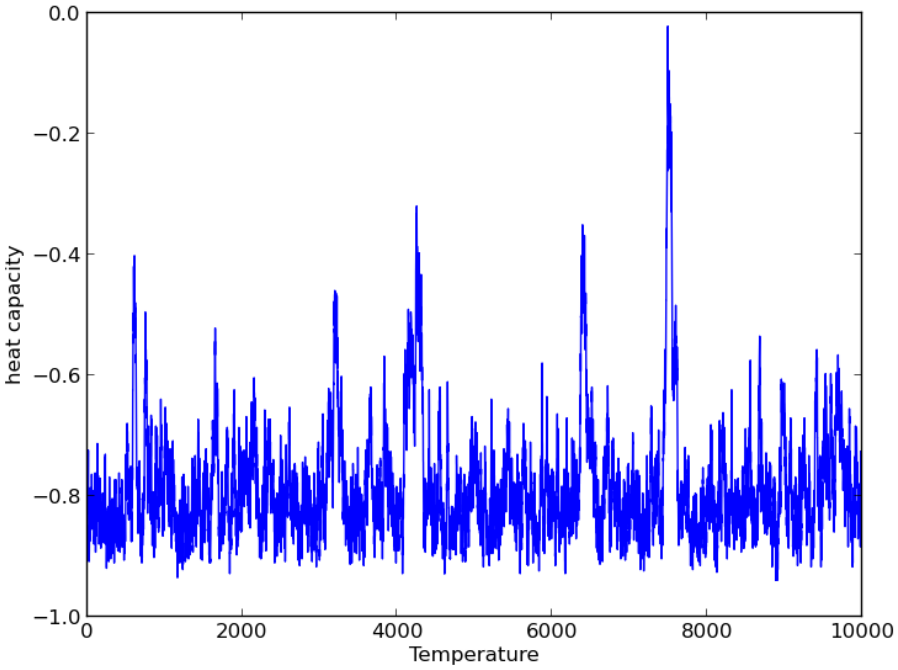

Figure 11: 30x30 average magnetization vs. steps

fair number of steps should satisfy that the expectation flipping number for individual spin greater than 1. Giving $N > L * L/0.388 = 2318$ I will say 2500 steps for simplicity.

3.2.3 c

Using the parameters above and I applied two chi-squared fits in two small ranges symmetric about a estimation of T_c, then find the intersect of these two lines. I found $T_c = 2.272$, very close to the predicted value. Fig. 12

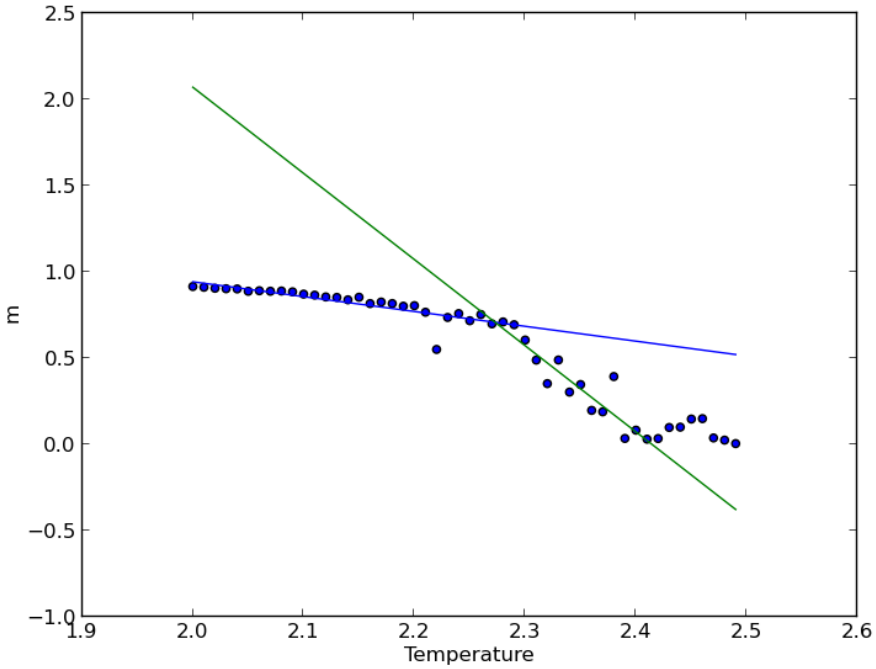

Figure 12: 30x30 average magnetization vs. temperature

Acknowledgements

I discussed this assignment with my classmates and used material from the cited references, but this write-up is my own.

References

[1] PHY 410-505 Webpage, `http://www.physics.buffalo.edu/phy410-505`.

A Appendix

A.1 python code

The following python code was used to obtain the results in this report:

```
#include "cptstd.hpp"
#include "linalg.hpp"
#include "random.hpp"
#include <cmath>
using namespace cpt;


class Ising {
public:

  Ising(double iJ=1.0, int iL=10, int iN=100, double iT=2.0, double iH=0.0) :
    J(iJ), L(iL), N(iN), T(iT), H(iH), s(L,L), Lx(L), Ly(L)
  {
    s = Matrix<int,2>(Lx, Ly);
    for (int i = 0; i < Lx; i++)
      for (int j = 0; j < Ly; j++)
        s[i][j] =  rng.rand() < 0.5 ? +1 : -1;    // hot start
    compute_boltzmann_factors();
    steps = 0;
  }



  void compute_boltzmann_factors()
  {
    for (int i = -8; i <= 8; i += 4) {
      w[i + 8][0] = exp( - (i * J - 2 * H) / T);
      w[i + 8][2] = exp( - (i * J + 2 * H) / T);
    }
  }

  bool metropolis_step()
  {
    // choose a random spin
    int i = int(Lx * rng.rand());
    int j = int(Ly * rng.rand());

    // find its neighbors using periodic boundary conditions
    int iPrev = i == 0 ? Lx-1 : i-1;
    int iNext = i == Lx-1 ? 0 : i+1;
    int jPrev = j == 0 ? Ly-1 : j-1;
    int jNext = j == Ly-1 ? 0 : j+1;
```

```
    // find sum of neighbors
    int sumNeighbors = s[iPrev][j] + s[iNext][j] + s[i][jPrev] + s[i][jNext];
    int delta_ss = 2*s[i][j]*sumNeighbors;

    // ratio of Boltzmann factors
    double ratio = w[delta_ss+8][1+s[i][j]];
    if (rng.rand() < ratio) {
      s[i][j] = -s[i][j];
      return true;
    } else return false;
  }

  double acceptanceRatio;

  void one_monte_carlo_step_per_spin ( ) {
    int accepts = 0;
    for (int i = 0; i < N; i++)
      if (metropolis_step())
        ++accepts;
    acceptanceRatio = accepts/double(N);
    ++steps;
  }

  double magnetizationPerSpin ( ) {
    int sSum = 0;
    for (int i = 0; i < Lx; i++)
      for (int j = 0; j < Ly; j++) {
        sSum += s[i][j];
      }
    return sSum / double(N);
  }

  double energyPerSpin ( ) {
    int sSum = 0, ssSum = 0;
    for (int i = 0; i < Lx; i++)
      for (int j = 0; j < Ly; j++) {
        sSum += s[i][j];
        int iNext = i == Lx-1 ? 0 : i+1;
        int jNext = j == Ly-1 ? 0 : j+1;
        ssSum += s[i][j]*(s[iNext][j] + s[i][jNext]);
      }
    return -(J*ssSum + H*sSum)/N;
  }


protected :

  Random rng;                        // random number generator
```

```
  double J;                          // ferromagnetic coupling
  int L, Lx, Ly;                     // number of spins in x and y
  int N;                             // number of spins
  Matrix<int,2> s;                   // the spins
  double T;                          // temperature
  double H;                          // magnetic field

  double w[17][3];                   // Boltzmann factors

  int steps;                         // steps so far


};

int main (int argc, char *argv[]) {

  int Lx, Ly, N;
  double T,Th, H, c, e1,e2,e3,m1,m2;
     cout << " Two-dimensional Ising Model - Metropolis simulation\n"
          << " ---------------------------------------------------\n"
          << " Enter number of spins L in each direction: ";
     cin >> Lx;
     Ly = Lx;
     N = Lx*Ly;
//      cout << " Enter temperature T: ";
//      cin >> T;
     cout << " Enter magnetic field H: ";
     cin >> H;
     cout << " Enter number of Monte Carlo steps: ";
     int MCSteps;
     cin >> MCSteps;

     T=1.0;
     Th=0.01;
     int Nloop=500;
     e1=0.0;
     e2=0.0;
     e3=0.0;
     ofstream dataFile;
     dataFile.open("1isingcpp.data");
     for (int count=0; count<Nloop; count++ ) {



            Ising ising(1.0, Lx, N, T, H);
```

```
int thermSteps = int(0.2 * MCSteps);
cout << " Performing " << thermSteps
     << " steps to thermalize the system ..." << flush;
for (int s = 0; s < thermSteps; s++)
     ising.one_monte_carlo_step_per_spin();

cout << " Done\n Performing production steps ..." << flush;
double mAv = 0, m2Av = 0, eAv = 0, e2Av = 0;

for (int s = 0; s < MCSteps; s++) {
     ising.one_monte_carlo_step_per_spin();
     double m = ising.magnetizationPerSpin();
     double e = ising.energyPerSpin();
     mAv += m; m2Av += m * m;
     eAv += e; e2Av += e * e;
     //dataFile << m << '\t' << e << '\n';
}
//dataFile.close();
mAv /= MCSteps; m2Av /= MCSteps;
eAv /= MCSteps; e2Av /= MCSteps;
e3=e2;
e2=e1;
e1=eAv;
m2=m1;
m1=mAv;


if (count==1) {
  c = abs(e2-e1)/2*Th;
  dataFile <<  T-Th << '\t' << m2 << '\t'<< c << '\n';
}

else if(count>1 and count < Nloop-1) {
  c = abs(e1-e3)/2*Th;
  dataFile << T-Th << '\t' << m2 << '\t' << c << '\n';
      cout<< T << '\t' << m1 << '\t' << c << '\n';
}

else if(count==Nloop-1) {
  c = abs(e1-e3)/2*Th;
  dataFile << T-Th << '\t' << m2 << '\t' << c << '\n';
  c = abs(e1-e2)/Th;
  dataFile << T << '\t' << m1 << '\t' << c << '\n';
  cout<< T << '\t' << m1 << '\t' << c << '\n';
}
```

```
//            cout << " \n Magnetization and energy per spin written in file "//
                  //<< " \"ising.data\"" << endl;
              //cout << " <m> = " << mAv << " +/- " << sqrt(m2Av - mAv*mAv) << endl;
              //cout << " <e> = " << eAv << " +/- " << sqrt(e2Av - eAv*eAv) << endl;
              T +=Th;
}

dataFile.close();
}
```

```
import math
import random
import matplotlib.pyplot as plt
from cpt import *

class Ising :
    def __init__(self, J=1.0, L=10, N=100, T=2., H=0.) :

        self.J = J                          # spin-spin coupling += ferro, -= antiferr
        self.L_x = L; self.L_y = L          # number of spins in x and y
        self.N = N                          # total number of spins
        self.s = []                         # L_x x L_y array of spin values
        self.T = T                          # Temperature
        self.H = H                          # magnetic field

        self.w = []                         # Boltzmann factors at fixed T and H
        self.steps = 0                      # Monte Carlo steps so far
        self.acceptance_ratio = 0           # accepted steps / total number of steps

        # create spin lattice and set spin randomly up or down (hot start)
        for i in range(self.L_x):
            self.s.append( [ ] )
            for j in range(self.L_y):
                self.s[i].append(random.choice( (-1, 1) ))
        self.compute_Boltzmann_factors()
        self.steps = 0



    def compute_Boltzmann_factors(self):
        self.w = []
        for m in range(5):
            self.w.append( [] )
            sum_of_neighbors = -4 + 2 * m
            for n in range(2):
                s_i = -1 + 2 * n
                factor = math.exp( -2.0 * (self.J * sum_of_neighbors + self.H) * s
                self.w[m].append(factor)
```

```
    def Metropolis_step_accepted(self):

        # choose a random spin
        i = random.randrange(self.L_x)
        j = random.randrange(self.L_y)

        # find the sum of neighbors assuming periodic boundary conditions
        sum_of_neighbors = ( self.s[(i-1)%self.L_x][j] + self.s[(i+1)%self.L_x][j]
                             self.s[i][(j-1)%self.L_y] + self.s[i][(j+1)%self.L_y]

        # access ratio of precomputed Boltzmann factors,
        ratio = self.w[2 + int(sum_of_neighbors/2)][int((1 + self.s[i][j])/2)]

        # apply the Metropolis test
        if ratio > 1.0 or ratio > random.random():
            self.s[i][j] = -self.s[i][j]
            return True
        else:
            return False



    def one_Monte_Carlo_step_per_spin(self):
        accepts = 0
        for n in range(self.N):
            if self.Metropolis_step_accepted():
                accepts += 1
        self.acceptance_ratio = accepts / float(self.N)
        self.steps += 1

    def magnetization_per_spin(self):

        s_sum = 0.0
        for i in range(self.L_x):
            for j in range(self.L_y):
                s_sum += self.s[i][j]
        return s_sum / float(self.N)

    def energy_per_spin(self):

        s_sum = 0.0
        ss_sum = 0.0
        for i in range(self.L_x):
            for j in range(self.L_y):
                s_sum += self.s[i][j]
```

```
                ss_sum += self.s[i][j] * (self.s[(i+1)%self.L_x][j] + self.s[i][(j-
        return -(self.J * ss_sum + self.H * s_sum) / float(self.N)

print " Two-dimensional Ising Model - Metropolis simulation"
print " ---------------------------------------------------"
L = int(input(" Enter number of spins L in each direction: "))
#T = float(input(" Enter temperature T: "))
H = float(input(" Enter magnetic field H: "))
MC_steps = int(input(" Enter number of Monte Carlo steps: "))


def my_range(start, end, step):
    while start <= end:
        yield start
        start += step

Tout=[]
mav=[]
eav=[]
hcout=[]
T=2.0
Nl=50
Th=0.01
fp=open("p3data","w")
for count in range(Nl):
    print count
    ising = Ising(L=L, N=L*L, T=T, H=H)

    therm_steps = int(0.2 * MC_steps)


    print " Performing", therm_steps, "thermalization steps ..."
    for i in range(therm_steps):
        ising.one_Monte_Carlo_step_per_spin()

    print " Done ... Performing production steps ..."
    m_av = 0.0; m2_av = 0.0; e_av = 0.0; e2_av = 0.0
    #data_file = open("isingT.data", "w")
    for i in range(MC_steps):
        ising.one_Monte_Carlo_step_per_spin()
        m = ising.magnetization_per_spin()
        e = ising.energy_per_spin()
        m_av += m
        m2_av += m**2
        e_av += e
        e2_av += e**2
        #data_file.write(repr(m) + "\t" + repr(e) + "\n")
    #data_file.close()
```

```
    print " M/spin and E/spin values written in isingT.data"
    m_av /= float(MC_steps)
    m2_av /= float(MC_steps)
    e_av /= float(MC_steps)
    e2_av /= float(MC_steps)
    mav.append(abs(m_av))
    eav.append(e_av)
    Tout.append(T)
    s = '{0:8.4f} {1:8.4f}\n'.format(T,abs(m_av))
    fp.write(s)
    T += Th
#     if count==1 :
#         c = (eav[1]-eav[0])/Th
#         print c
#         hcout.append(c)                                    #hcout[0]
#     elif count>1 and count < Nl-1:
#         c = (eav[count]-eav[count-2])/Th*2
#         print c
#         hcout.append(c)                                    #hcout[count-1]
#     elif count == Nl-1:
#         c = (eav[count]-eav[count-2])/Th*2
#         print c
#         hcout.append(c)
#         c = (eav[count]-eav[count-1])/Th
#         print c
#         hcout.append(c)                                    #hcout[N-1] hcout[N-2]
#print " <m> =", m_av, "+/-", math.sqrt(m2_av - m_av**2)
#print " <e> =", e_av, "+/-", math.sqrt(e2_av - e_av**2)
#print ising.w


print Tout,len(Tout)
print mav,len(mav)
#print hcout,len(hcout)
#plt.subplot(2, 1, 1)
plt.plot(Tout,mav)
plt.xlabel('Temperature')
plt.ylabel('m')

#plt.subplot(2,1,2)
#plt.plot(Tout,hcout)
#plt.xlabel('Temperature')
#plt.ylabel('heat capacity')

plt.show()

import matplotlib.pyplot as plt
import math
```

```
from numpy import array, real
from fft import fft, fft_power, ifft

filename = "l100.data"
file = open(filename, "r")
lines = file.readlines()
T=[]
m=[]
c=[]

for line in lines:
    if line !='\n' :
        words = line.split()
        tv,mv,cv = [float(s) for s in words]
        T.append(tv)
        m.append(mv)
        c.append(cv)

N=len(m)
log2N = math.log(N,2)
if log2N - int(log2N) > 0.0 :
    print 'Padding with zeros!'
    pads = [0.0] * (pow(2, int(log2N)+1) - N)    #now paddle with 0s
    ma = m + pads
    ca = c + pads
    N = len(m)
    print len(m)
M = fft(ma)
C = fft(ca)


maxfreq = 15
# Now smooth the data
for i in range(maxfreq, len(M)-maxfreq ) :
    M[i] = complex(0,0)
    C[i] = complex(0,0)

cp = ifft(C)
mp = ifft(M)
cpr = real(cp)
mpr = real(mp)

mf = mpr[0:len(m)]
cf = cpr[0:len(c)]


plt.subplot(2, 1, 1)
plt.plot(T,mf)
```

```
plt.xlabel('Temperature')
plt.ylabel('m')

plt.subplot(2,1,2)
plt.plot(T, cf)
plt.xlabel('Temperature')
plt.ylabel('heat capacity')

plt.show()
```