Becs-114.1100 Computational Science – exercise round 10

Kunal Ghosh, 546247

December 1, 2015

1 Solution 1

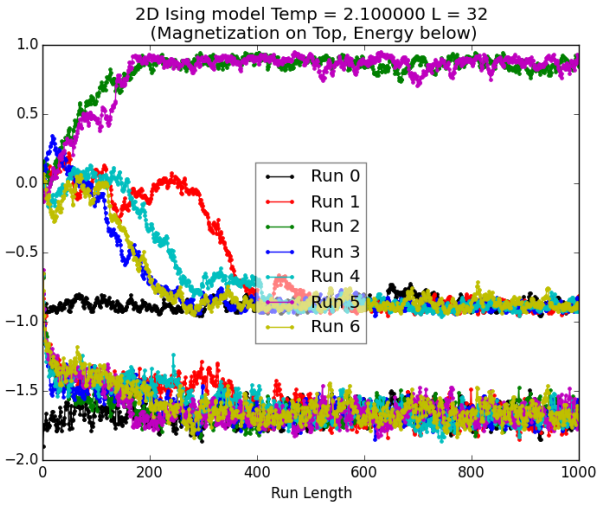
In this exercise, we are experimenting with the Ising model, a mathematical model of ferromagnetism given by Ernst Ising. The premise is that, each cell (site) of the L x L lattice is an atom the spin of which can be +ve or -ve (we are simulating that by +1 and -1 states in the L x L matrix).
Monte-carlo simulations are then used to simulate the change in energy and magnetization of the (simulated) ferro-magnet as the temperature changes.
From theoretical results we know that for temperatures below 2.265 the ferro-magnets equilibriate to either positive or negative magnetization. At temperatures above 2.265 (eg. 3.5 in our case) the object exhibits para-magnetism and is disordered at equilibrium.
Finally at the Critial temperature (2.265 in this case) the ferro-magnet goes through phase transition from ferro-magnetism to para-magnetism and fluctuates between positive and negative magnetization staying constant at a particular magnetization for long periods of time.
NOTE: In the experiments below, run 0 through 6 have been executed with
seeds 5555,6000,7000,8000,9000,10000,11000 respectively.

1.1 Checking ground state energy

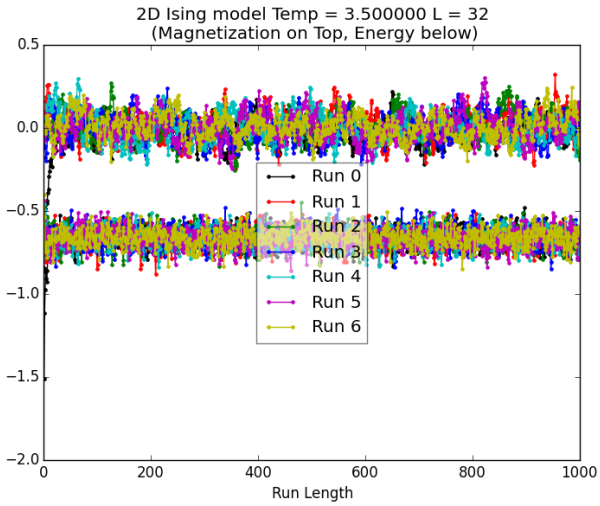
Setting all the spins to -1 we see that the ground state energy of a 32 x 32 Lattice is -2048.0 which is as expected.

1.2 Estimating suitable equilibriation time

Jugdging from the plots below and table in section 1.5.1
Equlibriation time for T = 2.1 : around 500 MCS
Equlibriation time for T = 3.5 : around 200 MCS



(a) T = 2.1



(b) T = 3.5

Figure 1: Magnetization and energy plots at T = 2.1 and T = 3.5 for a 32 x 32 lattice. Time is measured on the X axis in MCS (Montecarlo Simulation) units.

In the plots above, run 0 is started with the lattice having -1 at all the sites. This indicates the ground state where the enery is the least and the object is the most magnetized (magnetization just above -1). This is the state we expect all the simulation runs to converge to at equilibrium. (Some of the simulation runs converge to just below +1 which is another ground state which we have not plotted separately.)
The plot to the left is of the object at T = 2.1 at which it is still ferro-magnetic hence some simulation runs assume positive or negative magnetization at equilibrium.
The plot to the right is of the object at T = 3.5 at which it assumes para-magnetism and it exhibits almost no magnetization (magnetization oscillating about 0, as observed in the plots).

1.3 Simulation for T = 2.265

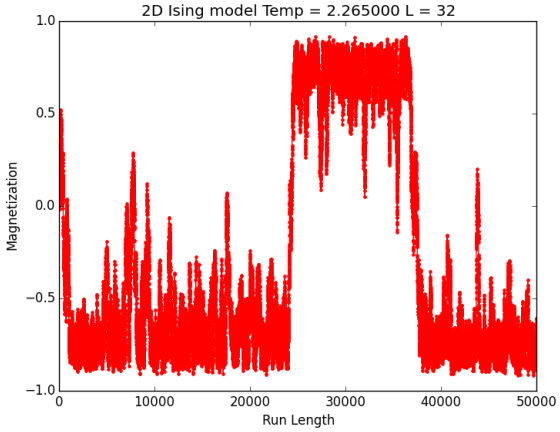


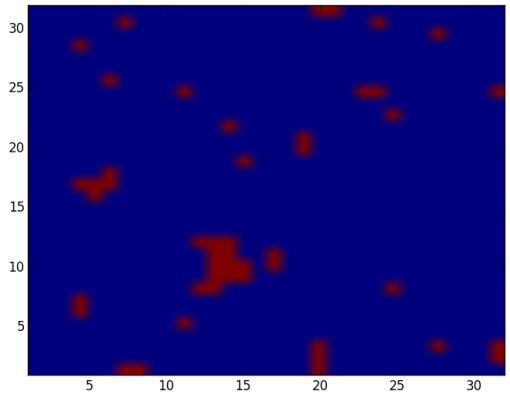
Figure 2: Plot of the magnetization at T = 2.265. Simulation run for 50,000 MCS

What is happening ?
T = 2.265 is the critical temperature at which the model undergoes **"phase transition"** and keeps fluctuating between magnetization = +1 and magnetization = -1 but staying constant at one magnetization for long periods of time (MCS).
Effect of this behavior on time needed to perform measurements at this temperature ?
At this temperature since the model keeps fluctuating (after long periods of stability) it causes any equilibrium calculation which considers magnetisation values only a few 100 MCS, falsely report that the model has reached equlibrium and fail to catch the flip in magnetization. Hence we must run the simulation for very a long time (50,000 MCS in this case) to figure out that the model is indeed going through phase transition and has not yet equilibriated.
Additionally, following the discussions on the topic [here] and [here] this phenomenon is called **"critical slowdown"** which causes the correlation length [ref](Time after which the deviations from the mean, stop being similar) to diverge. This means, we need to perform really long simulation runs to observe changes in correlation (change in magnetization). To avoid this, it is advisable to use either multiple simulation runs and average over them, to identify equilibrium, when using a local update scheme (as in this exercise) or use cluster update schemes of Montecarlo simulations.

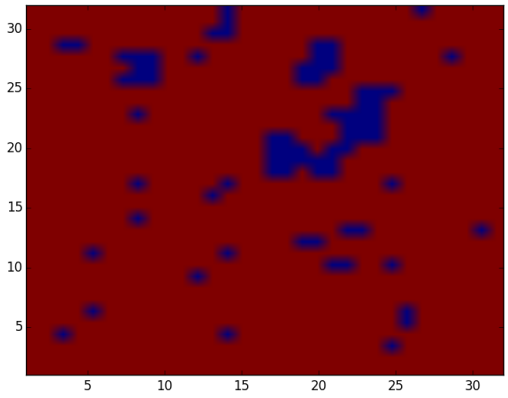
1.4 Lattice Visualizations

In the plots below we visualize the lattice after the simulations have reached equilibrium. Blue patches indicate magnetization of -1 and Red patches indicate magnetization of +1.

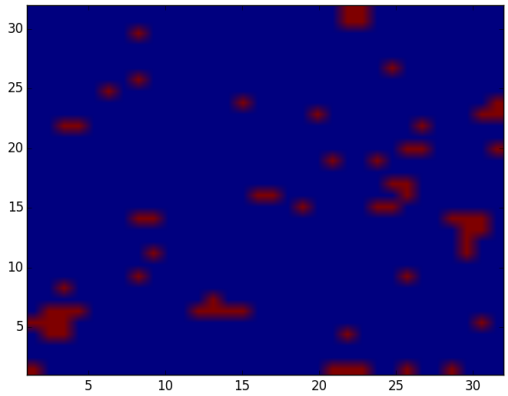
1.4.1 T = 2.1



(a) Run 0, lattice initialized to -1, and simulations converges to just above -1



(b) Run 2, Random Initialization, converges to just below 1.0

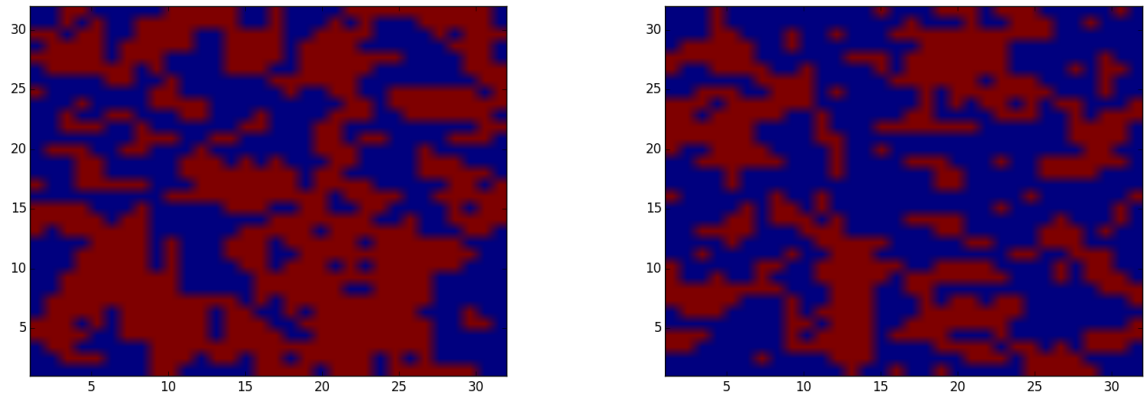


(c) Run 3, Random Initialization, converges to just above -1.0

Figure 3: Plots drawn from lattice configurations after 700 iterations for T = 2.1 (Blue = -1, Red = +1 Magnetization)

Since the object gets magnetized at equilibrium for T = 2.1, we observe that some simulation runs show the object to be positively magnetised (more red) and some negatively magnetised (more blue). The ground state with lattice initialized to -1 is seen at negatively magnetised in figure (3.a) above.

1.4.2 T = 3.5



(a) Run 0, lattice initialized to -1, and simulations converges to 0 magnetization

(b) Run 2, Random Initialization, stays at 0.0 magnetization

Figure 4: Plots drawn from lattice configurations after 700 iterations for T = 3.5 (Blue = -1, Red = +1 Magnetization)

At T = 3.5 since the object attains paramagnetism, it is seen to have almost equal patches of positive and negative magnetisation rendering the overall magnetisation close to zero. This is observed also for the ground state (lattice initialized to -1) seen in figure (4.a) above.

1.4.3 T = 2.265

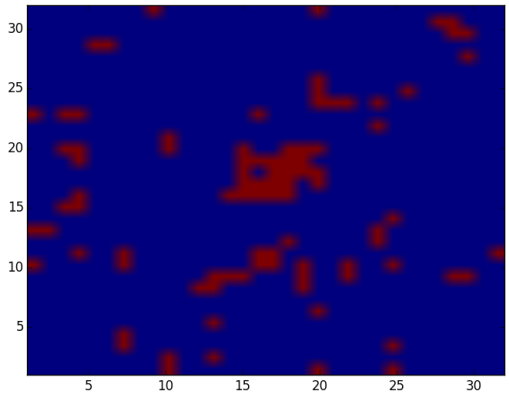


Figure 5: Lattice of T = 2.265 when convergence is just above -1.0. This plot from drawn from a configuration after 50,000 MCS (Blue = -1, Red = +1 Magnetization)

At T = 2.265 since the object is going through phase transition, it is seen to exhibit either positive or negative magnetization for extended periods of time. This lattice visualization shows the object to have -ve magnetization after 50,000 MCS. The corresponding python code can be found at 2

1.5 Values of Averages

In the table below we see that, at

Temperature = 2.1 the mean magnetization is -ve but close to zero, that's expected, since 4 runs have converged to -1 and only 2 to +1 so they don't quite cancell out each other. The mean of the absolute magnetizations is quite close to 1 which is also inline with the theoretical result, that at equilibrium the magnetizations converge to -1 or +1 for temperatures below critical temperature.

For Temperature = 3.5, the mean magnetization is quite close to zero and so is the mean of the absolute magnetization, these are again inline with theoretical results, according to which, for temperatures above critical temperature, the spins are disordered resulting in a mean magnetization which is close to zero.

For Temperature = 2.265, the critical temperature, the magnetization keeps flipping between -1 and +1 so depending on our simulation run, we would get the mean slightly above zero or below zero because some +ve and -ve values in the sequence would cancell and the remaining values would get divided by the number of measurements which is quite large. Also the mean of absolute magnetizations is close to 0.75 because most of the values are between |0.5| and |1| (as can be seen in the 1.3) so the mean of them should be around 0.75 and our result is quite close to that. If we take longer runs, the value of <|m|> should converge to 0.75

Listing 1: Average magnetization values

```
For Temp = 2.1 <m> = -0.298180989583 err = 0.0149983573735 AND <|m|> = 0.872975260417 err = 0.000696132850481
For Temp = 3.5 <m> = 0.00534993489583 err = 0.00114563542299 AND <|m|> = 0.0633504231771 err = 0.00069438550247
For Temp = 2.265 <m> = -0.272830034722 err = 0.00295765007233 AND <|m|> = 0.659539149306 err = 0.000857419301299
```

The table above reports average M = $\langle m \rangle$ and average abs(M) = $\langle |m| \rangle$ values separately for each temperature. (magnetisation values have been taken from each run as follows):
For T = 2.1 : Equilibriation time is taken as 500
For T = 3.5 : Equilibriation time is taken as 200
For T = 2.265 : Equilibriation time is taken as 5000

Mean magnetisation $\langle m \rangle$ is calculated by taking a mean of magnetisation values after equilibriation time for all run.
Mean abs magnetisation $\langle |m| \rangle$ is calculated by taking a mean of absolute values of magnetisation after equilibriation time for all run.
Error values are calculated using the following equation.

$$\sigma_m = \sqrt{\frac{1}{n-1}(\langle m^2 \rangle - \langle m \rangle^2)} \tag{1}$$

The corresponding python code can be found at 3

1.5.1 The data below has mean values from each run of T=2.1 and T=3.5 (Raw data for reference ONLY)

The below tables report average M = $\langle m \rangle$ and average abs(M) = $\langle |m| \rangle$ values separately for each run. The averages were calculated by taking magnetization values from the past 100 MCS and averaging them to get average M. The average of the absolute values was calculated to get the average abs(M). The **convergence criteria** was for the error (difference between two successive averages) to be less than 0.01

Listing 2: Average Magnetization values for T 2.1

```
# Lattice initialized to -1 and then MC simulations were run
Temp = 2.1 Run 0 Convergence after 200 MCS at M = -0.876015625, M_err = 0.00283203125
Temp = 2.1 Run 0 Convergence after 200 MCS at abs(M) = 0.876015625, abs(M)err = 0.00283203125
Temp = 2.1 Run 0 Final Data after 1000 MCS at abs(M) = 0.879296875, abs(M)err = 0.0109765625, abs(M) = 0.879296875, abs(M)err = 0.0109765625

Temp = 2.1 Run 1 Convergence after 700 MCS at M = -0.89130859375, M_err = 0.0034765625
Temp = 2.1 Run 1 Convergence after 700 MCS at abs(M) = 0.89130859375, abs(M)err = 0.0034765625
Temp = 2.1 Run 1 Final Data after 1000 MCS at abs(M) = 0.90140625, abs(M)err = 0.02017578125, abs(M) = 0.90140625, abs(M)err = 0.02017578125

Temp = 2.1 Run 2 Convergence after 600 MCS at M = 0.884375, M_err = 0.00033203125
Temp = 2.1 Run 2 Convergence after 600 MCS at abs(M) = 0.884375, abs(M)err = 0.00033203125
Temp = 2.1 Run 2 Final Data after 1000 MCS at abs(M) = 0.84498046875, abs(M)err = 0.02615234375, abs(M) = 0.84498046875, abs(M)err = 0.02615234375

Temp = 2.1 Run 3 Convergence after 800 MCS at M = -0.885546875, M_err = 0.00876953125
Temp = 2.1 Run 3 Convergence after 800 MCS at abs(M) = 0.885546875, abs(M)err = 0.00876953125
Temp = 2.1 Run 3 Final Data after 1000 MCS at abs(M) = 0.886640625, abs(M)err = 0.0027734375, abs(M) = 0.886640625, abs(M)err = 0.0027734375

Temp = 2.1 Run 4 Convergence after 600 MCS at M = -0.8751171875, M_err = 0.00986328125
Temp = 2.1 Run 4 Convergence after 600 MCS at abs(M) = 0.8751171875, abs(M)err = 0.00986328125
Temp = 2.1 Run 4 Final Data after 1000 MCS at abs(M) = 0.88275390625, abs(M)err = 0.00666015625, abs(M) = 0.88275390625, abs(M)err = 0.00666015625

# Run 5 did not satisfy convergence criteria even after 1000 MCS
Temp = 2.1 Run 5 Final Data after 1000 MCS at abs(M) = 0.86546875, abs(M)err = 0.012421875, abs(M) = 0.86546875, abs(M)err = 0.012421875

Temp = 2.1 Run 6 Convergence after 800 MCS at M = -0.87662109375, M_err = 0.00427734375
Temp = 2.1 Run 6 Convergence after 800 MCS at abs(M) = 0.87662109375, abs(M)err = 0.00427734375
Temp = 2.1 Run 6 Final Data after 1000 MCS at abs(M) = 0.87595703125, abs(M)err = 0.026640625, abs(M) = 0.87595703125, abs(M)err = 0.026640625
```

Listing 3: Average Magnetization values for T 3.5

```
# Lattice initialized to -1 and then MC simulations were run
Temp = 3.5 Run 0 Convergence after 300 MCS at abs(M) = 0.05380859375, abs(M)err = 0.0001171875
Temp = 3.5 Run 0 Convergence after 600 MCS at M = -0.03533203125, M_err = 0.008359375
Temp = 3.5 Run 0 Final Data after 1000 MCS at abs(M) = 0.053984375, abs(M)err = 0.01162109375, abs(M) = 0.053984375, abs(M)err = 0.01162109375

Temp = 3.5 Run 1 Convergence after 100 MCS at M = 0.02720703125, M_err = 0.00447265625
Temp = 3.5 Run 1 Convergence after 300 MCS at abs(M) = 0.05716796875, abs(M)err = 0.009609375
Temp = 3.5 Run 1 Final Data after 1000 MCS at abs(M) = 0.08421875, abs(M)err = 0.0171484375, abs(M) = 0.08421875, abs(M)err = 0.0171484375

Temp = 3.5 Run 2 Convergence after 700 MCS at abs(M) = 0.06861328125, abs(M)err = 0.0042578125
Temp = 3.5 Run 2 Final Data after 1000 MCS at abs(M) = 0.05421875, abs(M)err = 0.03365234375, abs(M) = 0.05421875, abs(M)err = 0.03365234375

Temp = 3.5 Run 3 Convergence after 100 MCS at M = 0.0115625, M_err = 0.0028125
Temp = 3.5 Run 3 Convergence after 200 MCS at abs(M) = 0.06232421875, abs(M)err = 0.00533203125
Temp = 3.5 Run 3 Final Data after 1000 MCS at abs(M) = 0.058125, abs(M)err = 0.00541015625, abs(M) = 0.058125, abs(M)err = 0.00541015625

Temp = 3.5 Run 4 Convergence after 400 MCS at abs(M) = 0.0665234375, abs(M)err = 0.0051953125
Temp = 3.5 Run 4 Final Data after 1000 MCS at abs(M) = 0.05048828125, abs(M)err = 0.013203125, abs(M) = 0.05048828125, abs(M)err = 0.013203125

Temp = 3.5 Run 5 Convergence after 100 MCS at abs(M) = 0.0578515625, abs(M)err = 0.00736328125
Temp = 3.5 Run 5 Final Data after 1000 MCS at abs(M) = 0.0575, abs(M)err = 0.0240625, abs(M) = 0.0575, abs(M)err = 0.0240625

Temp = 3.5 Run 6 Convergence after 500 MCS at M = 0.0059375, M_err = 0.00458984375
Temp = 3.5 Run 6 Convergence after 500 MCS at abs(M) = 0.0428125, abs(M)err = 0.00302734375
Temp = 3.5 Run 6 Final Data after 1000 MCS at abs(M) = 0.07537109375, abs(M)err = 0.02443359375, abs(M) = 0.07537109375, abs(M)err = 0.02443359375
```

2 Appendix A

Python source code for 1.

```
from __future__ import division
from itertools import combinations
import cPickle as pickle
import sys

import pylab
import numpy as np

class PLattice:
    def __init__(self, dimension, initial_value, temperature, seed=None):
        # create an array storing the values on the lattice and set them
        # to initial_value
        if seed is not None:
            np.random.seed(seed)
            rand_vals = np.random.random(dimension)*2-1 # scale the rand numbers between -1,1
            self.lattice = np.floor(rand_vals) + np.ceil(rand_vals)
        else:
            np.random.seed(5555)
            self.lattice = initial_value * pylab.ones(dimension)

        self.dimension = dimension # dimension = (row,col)
        self.length = np.prod(self.dimension)

        self.row_max = self.dimension[0]
        self.col_max = self.dimension[1]
        self.J = 1
        self.temp = temperature
        self.indices = np.asarray([zip(np.ones(self.col_max).astype(int)*r,np.arange(self.col_max)) for r in xrange(self.row_max)])
        l,b,h = self.indices.shape
        self.indices = self.indices.reshape(l*b,h)

        # compute the energy and magnetization of the initial configuration
        self.energy = self.compute_energy()
        self.magnetization = self.compute_magnetization()

    def __get_indices__(self, idx):
        # the modulus operator implements the periodic boundary
        # (may not be the most efficient way but it's ok for this...)
        # one should check that negative values of idx behave also as expected
        # idx in this case is a Tuple OR a List of Tuples
        # List of Tuples would allow us to vectorize operations.
        retVal = None
        if isinstance(idx,np.ndarray) or isinstance(idx,list):
            new_rows,new_cols = np.asarray(zip(*idx))
            new_rows = new_rows % self.row_max
            new_cols = new_cols % self.col_max
            retVal = np.asarray(zip(new_rows, new_cols))
        elif isinstance(idx,tuple):
            new_row = idx[0] % self.row_max
            new_col = idx[1] % self.col_max
            retVal = (new_row, new_col)
        else:
            raise ValueError("Only list of Tuples or Tuples accepted")
        return retVal

    # see below the flip method and the flip example in the main-part on how
    # __getitem__ and __setitem__ work
    def __get_left_idx(self,idx):
        retVal = None
        if isinstance(idx,np.ndarray) or isinstance(idx,list):
            rows,cols = np.asarray(zip(*idx))
            cols = (cols - 1) % self.col_max
            retVal = np.asarray(zip(rows,cols))
        elif isinstance(idx,tuple):
            retVal = (idx[0],(idx[1]-1) % self.col_max)
        else:
            raise ValueError("Only list of Tuples or Tuples accepted")
        return retVal

    def __get_right_idx(self,idx):
        retVal = None
        if isinstance(idx,np.ndarray) or isinstance(idx,list):
            rows,cols = np.asarray(zip(*idx))
            cols = (cols + 1) % self.col_max
            retVal = np.asarray(zip(rows,cols))
        elif isinstance(idx,tuple):
            retVal = (idx[0],(idx[1]+1) % self.col_max)
        else:
            raise ValueError("Only list of Tuples or Tuples accepted")
        return retVal

    def __get_top_idx(self,idx):
        retVal = None
        if isinstance(idx,np.ndarray) or isinstance(idx,list):
            rows,cols = np.asarray(zip(*idx))
            rows = (rows - 1) % self.row_max
            retVal = np.asarray(zip(rows,cols))
        elif isinstance(idx,tuple):
            retVal = ((idx[0]-1) % self.row_max,idx[1])
        else:
            raise ValueError("Only list of Tuples or Tuples accepted")
        return retVal

    def __get_bottom_idx(self,idx):
        retVal = None
```

```
        if isinstance(idx,np.ndarray) or isinstance(idx,list):
            rows,cols = np.asarray(zip(*idx))
            rows = (rows + 1) % self.row_max
            retVal = np.asarray(zip(rows,cols))
        elif isinstance(idx,tuple):
            retVal = ((idx[0]+1) % self.row_max,idx[1])
        else:
            raise ValueError("Only list of Tuples or Tuples accepted")
        return retVal

    def __getitem__(self, idx):
        idxes = self.__get_indices__(idx)
        retVal = None
        if isinstance(idx,np.ndarray) or isinstance(idx,list):
            retVal = self.lattice[zip(*idxes)]
        elif isinstance(idx,tuple):
            retVal = self.lattice[idxes]
        else:
            raise ValueError("Only list of Tuples or Tuples accepted")
        return retVal

    def __setitem__(self, idx, val):
        # same here
        self.lattice[self.__get_indices__(idx)] = val

    def flip(self, idx, compute_energy=True):
        # this is equal to self[idx] = -1 * self[idx]
        # self[idx] causes call to either __getitem__ or __setitem__ (see below)
        self[idx] *= -1
        if compute_energy:
            self.energy = self.compute_energy()

    def compute_magnetization(self):
        return np.sum(self.lattice)

    def get_energy(self):
        return self.energy

    def get_energy_per_site(self):
        return np.true_divide(self.get_energy(),self.length)

    def get_magnetization(self):
        return self.magnetization

    def get_magnetization_per_site(self):
        return np.true_divide(self.get_magnetization(),self.length)

    def compute_energy(self):
        # compute the energy here and return it
        # get values of the right cell and the bottom cell and do this for each cell.
        # this ensure that an i,j is not indexed twice
        right_indices = self.__get_right_idx(self.indices)
        bottom_indices = self.__get_bottom_idx(self.indices)
        right_vals = self[right_indices]
        bottom_vals = self[bottom_indices]
        cell_vals = self[self.indices]
        np.add(right_vals, bottom_vals, out=bottom_vals)
        np.multiply(bottom_vals,cell_vals,out=cell_vals)
        return np.sum(cell_vals) * -1

    def __is_flip_accepted(self, idx):
        retVal = None
        deltaE = 2 * self[idx] * ( self[self.__get_bottom_idx(idx)]
                                 + self[self.__get_top_idx(idx)]
                                 + self[self.__get_left_idx(idx)]
                                 + self[self.__get_right_idx(idx)] )
        if deltaE <= 0:
            retVal = True
        else:
            w = np.exp((-1 * deltaE)/self.temp) # kB = 1
            if np.random.random() < w:
                retVal = True
            else:
                retVal = False
        return retVal,deltaE

    def do_montecarlo(self):
        # we need max_row * max_col random indices
        rand_row_indices = np.random.randint(low=0,high=self.row_max,size=self.length)
        rand_col_indices = np.random.randint(low=0,high=self.col_max,size=self.length)
        idxes = zip(rand_row_indices,rand_col_indices)
        for idx in idxes:
            result, deltaE = self.__is_flip_accepted(idx)
            if result: # Flip Accepted
                self.flip(idx,compute_energy=False)
                self.energy += deltaE
                self.magnetization += 2 * self[idx]

    def print_lattice(self):
        import pprint
        pprint.pprint(self.lattice)

    def get_lattice(self):
        return self.lattice

def plot_lattice(lattice,fileName):
    from mpl_toolkits.mplot3d import Axes3D
    import matplotlib.pyplot as plt
    from matplotlib import cm
    fig = plt.figure()
    x,y = lattice.shape
    X = np.arange(0, x, 1)
    Y = np.arange(0, y, 1)
```

```
    X, Y = np.meshgrid(X, Y)
    R = lattice[X,Y]
    surf = plt.imshow(R,origin='lower', aspect='auto', extent=(1,x,1,y))
    plt.savefig(fileName+".png")
    plt.close()

if __name__ == "__main__":
    # create the lattice object
    l = PLattice((32,32), -1,temperature = 2.265, seed=None)

    # print the energy
    print l.energy

    # print the values of the lattice at the left neighbor, current index and
    # right neighbor to check that periodic boundary works...

    # Code to check periodic boundary
    # for i in xrange(l.col_max):
    #     print "Col = {} -- ".format(i),
    #     print l[0,i-1], l[0,i], l[0,i+1]

    # for i in xrange(l.row_max):
    #     print "Row = {} -- ".format(i),
    #     print l[i-1,0], l[i,0], l[i+1,0]

    # here's how the monte carlo simulation could be implemented
    # you need to use e.g. lists to keep track of the energy etc.
    # at each iteration...
    runlength = 1000
    lattice_shape = (32,32)
    energies_per_temp = []
    magnetizations_per_temp = []
    lattices_per_temp = []
    xvals = range(1,runlength+1)
    seeds = [None,6000,7000,8000,9000,10000,11000]
    colours = ["k","r","g","b","c","m","y"]

    # For checking equlibriation
    m_old,m_new = -200,200 # Mean magnetization
    m_abs_old, m_abs_new = -200,200 # Mean abs magnetization

    for temp in [2.1,3.5]:
        energies = []
        magnetizations = []
        lattices = []
        pylab.figure()
        for idx,val in enumerate(zip(seeds,colours)):
            run_seed,color = val
            energy = []
            magnetization = []
            l = PLattice(lattice_shape, -1, temperature = temp, seed=run_seed)
            for i in xrange(1, runlength+1):
                # ... keep track of the interesting quantities
                # print the progress in long runs
                l.do_montecarlo()
                energy.append(l.get_energy_per_site())
                magnetization.append(l.get_magnetization_per_site())
                if i % 100 == 0:
                    # print "Temp = %f , Seed = %d , %d MCS completed." % (temp, run_seed if run_seed is not None else -1, i)
                    m_old, m_new = m_new, np.mean(magnetization[-100:])
                    m_abs_old, m_abs_new = m_abs_new, np.mean(np.abs(magnetization[-100:]))
                    err_m, err_abs = np.abs(m_old-m_new) , np.abs(m_abs_old - m_abs_new)
                    if err_m < 0.01:
                        print "Temp = {} Run {} Convergence after {} MCS at M = {}, M_err = {}".format(temp, idx,i,m_new,err_m)
                    if err_abs < 0.01:
                        print "Temp = {} Run {} Convergence after {} MCS at abs(M) = {}, abs(M)err = {}".format(temp,idx,i,m_abs_new,err_abs)

            print "Temp = {} Run {} Final Data after {} MCS at abs(M) = {}, abs(M)err = {}, abs(M) = {}, abs(M)err =
                  {}".format(temp,idx,i,m_abs_new,err_abs,m_abs_new,err_abs)
            energies.append(energy)
            magnetizations.append(magnetization)
            lattices.append(l.get_lattice())
            plot_lattice(lattices[-1],"%d_run_%d"%(int(temp),idx))

            pylab.plot(xvals,energy,marker=".",c=color,label="Run %d" % idx)
            pylab.plot(xvals,magnetization,marker=".",c=color)
        pylab.legend(framealpha=0.5,loc=10)
        pylab.xlabel("Run Length")
        pylab.title("2D Ising model Temp = %f L = %d \n (Magnetization on Top, Energy below)" % (temp,lattice_shape[0]))
        pylab.savefig("energyVsmagnetization_%d.png"%int(temp))
        pylab.show()
        for idx,val in enumerate(zip(magnetizations,colours)):
            m,c = val
            pylab.plot(xvals, m, marker=".",c=c,label="Run %d" % idx)
        pylab.legend(framealpha=0.5,loc=10)
        pylab.xlabel("Run Length")
        pylab.ylabel("Magnetization")
        pylab.title("2D Ising model Temp = %f L = %d" % (temp,lattice_shape[0]))
        pylab.savefig("magnetization_%d.png"%int(temp))
        pylab.show()
        energies_per_temp.append(energies)
        magnetizations_per_temp.append(magnetizations)
        lattices_per_temp.append(lattices)

    with open('data.pkl', 'wb') as dat_dmp_file:
        pickle.dump([energies_per_temp, magnetizations_per_temp, lattices_per_temp], dat_dmp_file)

    temp = 2.265
    runlength = 50000
    xvals = range(1,runlength+1)
    pylab.figure()
    energies = []
    magnetizations = []
```

```
    lattices = []


    for idx,val in enumerate(zip([seeds[1]],[colours[1]])):
        run_seed,color = val
        energy = []
        magnetization = []
        l = PLattice(lattice_shape, -1, temperature=temp, seed=run_seed)
        for i in xrange(1, runlength+1):
            l.do_montecarlo()
            energy.append(l.get_energy_per_site())
            magnetization.append(l.get_magnetization_per_site())
            if i % 1000 == 0:
                print "Temp = %f , Seed = %d , %d MCS completed." % (temp, run_seed if run_seed is not None else -1, i)
        energies.append(energy)
        magnetizations.append(magnetization)
        lattices.append(l.get_lattice())
        plot_lattice(lattices[-1],"50000_lattice")
        # pylab.plot(xvals,energy,marker=".",c=color,label="Run %d" % idx)
        try:
            pylab.plot(xvals,magnetization,marker=".",c=color)
        except:
            pass

    with open('data50000.pkl', 'wb') as dat_dmp_file:
        pickle.dump([energies, magnetizations], dat_dmp_file)
        #pickle.dump(magnetizations, dat_dmp_file)

    pylab.legend(framealpha=0.5,loc=10)
    pylab.xlabel("Run Length")
    pylab.ylabel("Magnetization")
    pylab.title("2D Ising model Temp = %f L = %d" % (temp,lattice_shape[0]))
    pylab.savefig("magnetization5000_%d.png"%int(temp))
    pylab.show()
```

3 Appendix B

Python source code for calculating means. 1.5

```
from __future__ import division
import cPickle as pickle
import numpy as np

temp_equilibrium = {"2.1":500,"3.5":200} # for T=2.1 and T=3.5 respectively
# load the pkl file for 2.1 and 3.5 temperatures
with open("data.pkl") as f:
    [energies_per_temp, magnetizations_per_temp, lattices_per_temp] = pickle.load(f)

# for each temperature
for idx,temp in enumerate(temp_equilibrium):
    mag_sum = 0
    mag_sq_sum = 0
    mag_abs_sum = 0
    mag_abs_sq_sum = 0
    equlib_time = temp_equilibrium[temp]

    # skip the values for run in ground state. hence [1:]
    magnetizations = magnetizations_per_temp[idx][1:]
    total_vals = 0
    for magnetization in magnetizations:
        mag_data = magnetization[equlib_time:]
        mag_sum += np.sum(mag_data)
        total_vals += len(mag_data)
        mag_sq_sum += np.sum(np.square(mag_data))
        mag_abs_sum += np.sum(np.abs(mag_data))
        mag_abs_sq_sum += np.sum(np.square(np.abs(mag_data)))

    mean = lambda x: x/total_vals
    mag_mean = mean(mag_sum)
    mag_err = np.sqrt((1.0/(total_vals-1))*(mean(mag_sq_sum) - np.square(mean(mag_sum))))
    abs_mag_mean = mean(mag_abs_sum)
    mag_abs_err = np.sqrt((1.0/(total_vals-1))*(mean(mag_abs_sq_sum) - np.square(mean(mag_abs_sum))))

    print "For Temp = {} <m> = {} err = {} AND <|m|> = {} err = {}".format(temp,mag_mean,mag_err,abs_mag_mean,mag_abs_err)

with open("data50000.pkl") as f:
    [energies, magnetizations] = pickle.load(f)

equilibrium = 5000
mag_dat = magnetizations[0][equilibrium:]
mag_dat_sq = np.square(mag_dat)
mag_err = np.sqrt((1.0/(len(mag_dat)-1))*(np.mean(mag_dat_sq) - np.square(np.mean(mag_dat))))

abs_mag_dat = np.abs(mag_dat)
abs_mag_dat_sq = np.square(abs_mag_dat)
abs_mag_err = np.sqrt((1.0/(len(mag_dat)-1))*(np.mean(abs_mag_dat_sq) - np.square(np.mean(abs_mag_dat))))


print "For Temp = 2.265 <m> = {} err = {} AND <|m|> = {} err = {}".format(np.mean(mag_dat), mag_err, np.mean(abs_mag_dat), abs_mag_err)
```