Becs-114.1100 Computational Science – exercise round 9

Srikanth Gadicherla, 546195 December 1, 2015

Problem 1

1(a) The ground energy state for all spins up was found out to be -2048. The corresponding code is present in Appendix A (i)

Ising Model :

The Ising model was simulated for three different temperature .

1(b)For temperatures , 5 short runs of 1000 MCS each were simulated where each run was initialized randomly with different seed of random number. The Figure 1 shows the plot of internal energy per site and magnetisation per site for run 2 versus time(mcs).

From Figure 2, the magnetisation per site plot versus time (mcs) for 5 runs is plotted. So the equilibrium time for temperature is given below.

Equilibrium for T = 2.1 is 600 MCS.

Equilibrium for T = 3.5, from Figure 2 it shows that the magnetisation for all the runs wavers near zero, so the equilibrium time was considered at 100 MCS.

At the equilibrium temperature, the internal energy per site for temperature(2.1) below critical temperature is near -2 and for temperature (3.5) above critical temperature, it is wavering a at a point below zero. Here it is near -0.7

The code for it is given in Appendix A (ii)

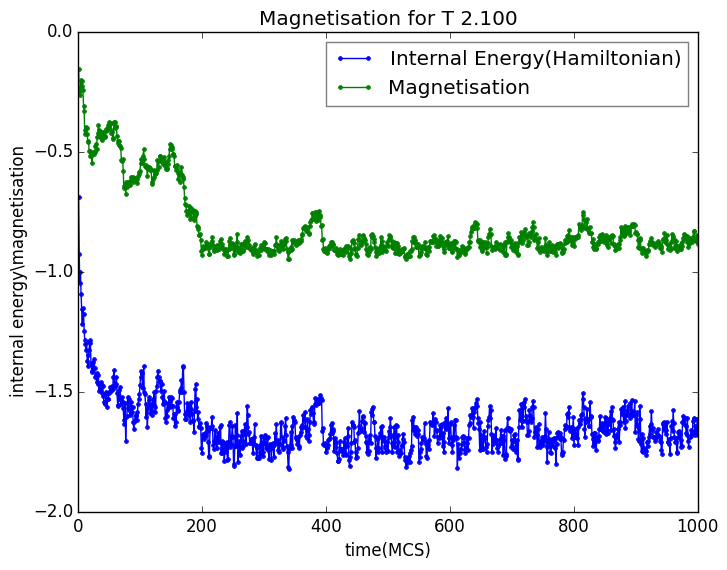
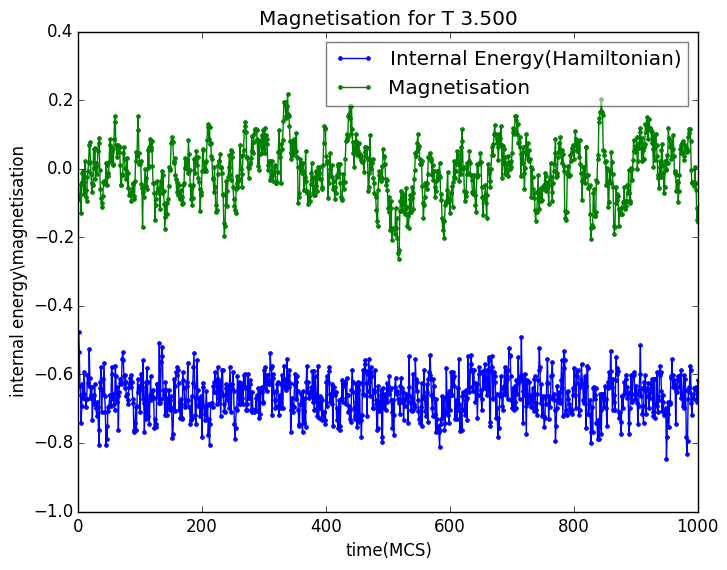


Figure 1: The internal energy (u) per site and magnetization (m) per site for Run 2.

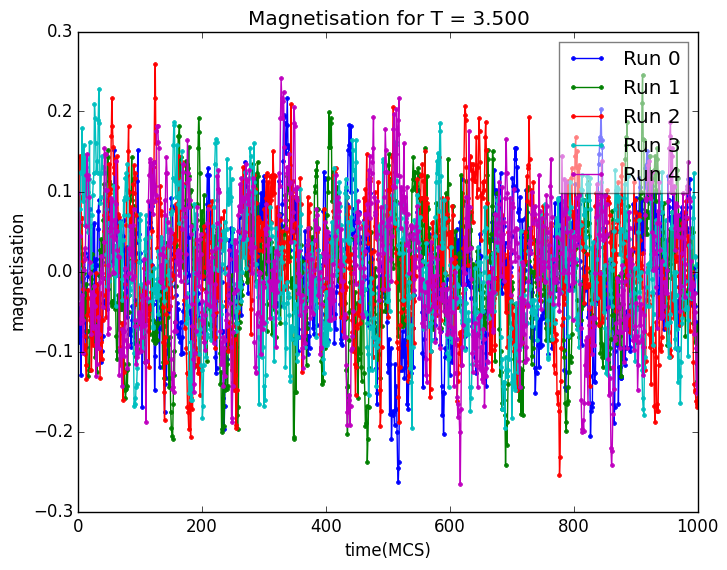
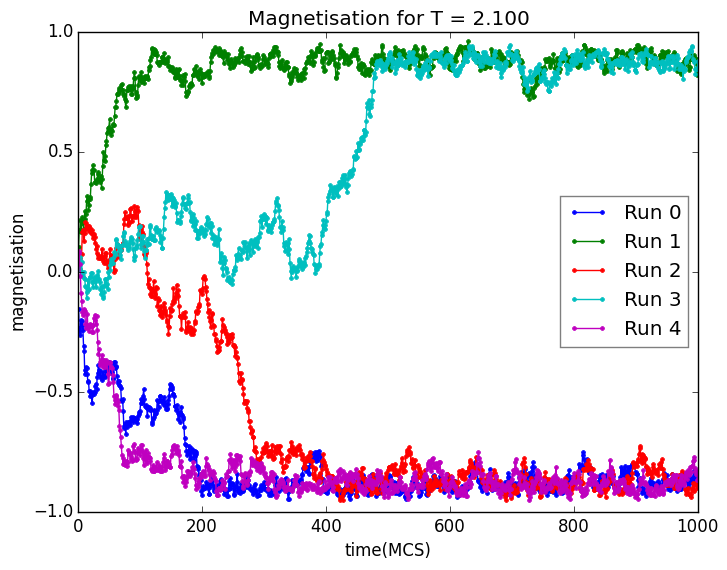


Figure 2: Magnetization (m) per site for all runs.

1(c) The Figure 3(a) shows the internal energy and magnetisation(per site). Notable observation is that the system reaches equilibrium pretty fast(here within few 100 MCS). The internal energy is below -1 and here wavers around -1.5. The magnetisation for both the runs is abruptly changing from end of (-1,1). This oscillation affects the estimation of equilibrium time. That's the reason it was recommended to make a simulation for 50,000 MCS and more independent runs. Here, 2 runs were simulated.

The code for this given in Appendix A (iii)

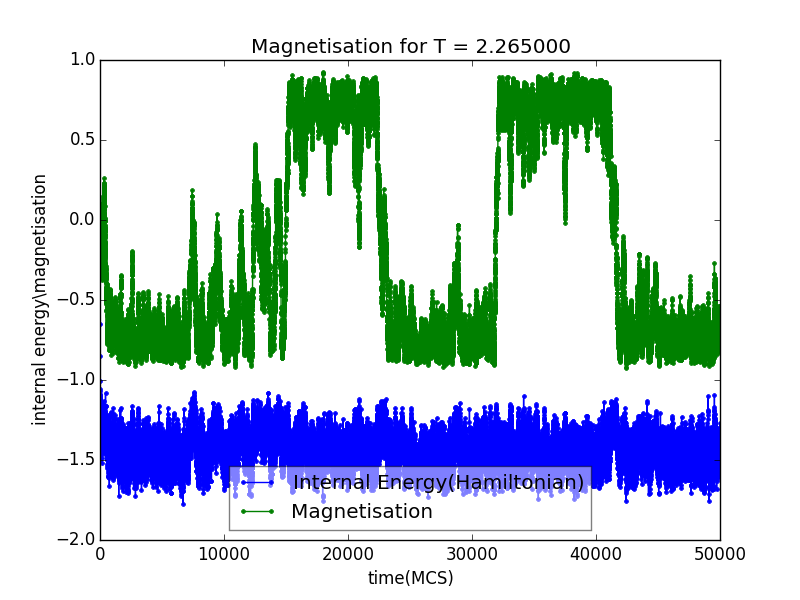
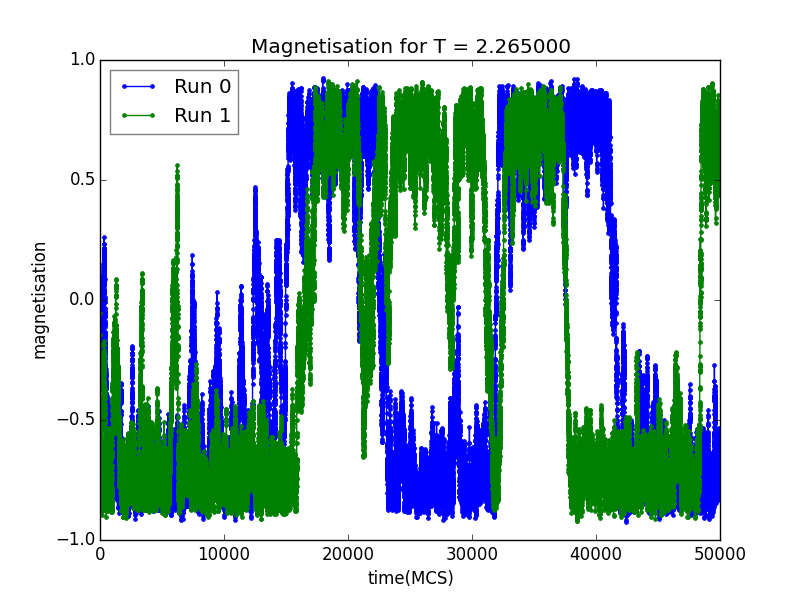


Figure 3: (a)The internal energy (u) per site and magnetization (m) per site for Run 2.(b) The magnetisation curve for temperature 2.265

1(d) The equilibrium snapshot for Temperatures 2.1, 3.5 and 2.265 is given below. For temperature 2.1(below critical temperature), as seen from magnetisation curves the magnetisation for different runs is either near +1 or -1, so it can inferred that equilibrium is also between -1 and 1. Here it is near -1. For temperature 3.5, which is above the critical temperature the equilibrium state is pretty random and the figure confirms it. For critical temperature 2.265, the equilibrium wavers between extremes, it is here near to +1.

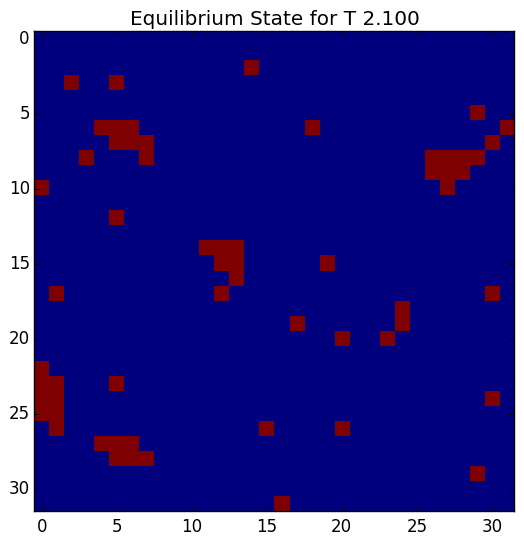
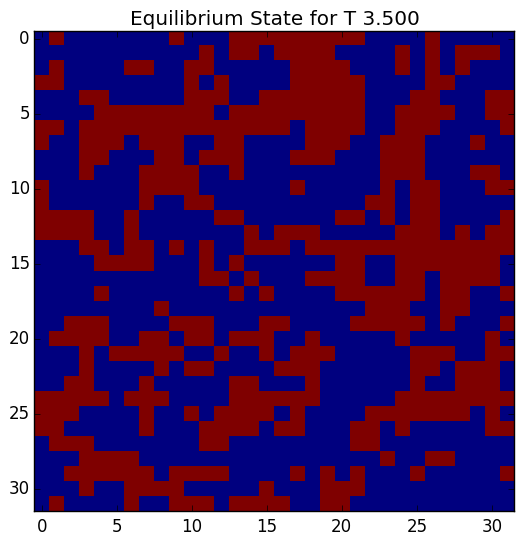
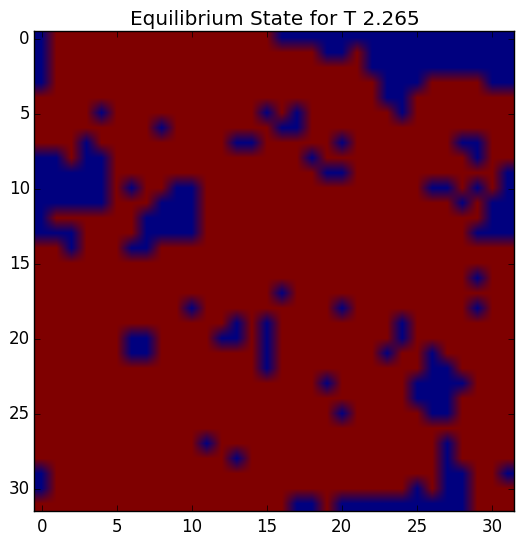


Figure 4: The equilibrium snapshots for temperature T = 2.1, 3.5, 2.265 (Red +1, Blue -1)

1(e) Below is the table for the mean magnetisation and absolute mean magnetisation for all the temperatures. The values of magnetisation were taken after the system was equilibrated. For the temperature 2.265, the equilibrium time was taken as 1000.

|  |  |  |
| --- | --- | --- |
| Temperature | Mean magnetisation <m> | Absolute mean magnetisation <|m|> |
| 2.1 | -0.17556 | 0.87456 |
| 2.265 | -0.16291 | 0.63908 |
| 3.5 | -0.00057 | 0.06465 |

The mean magnetizations for all the temperature is expected to be near zero, and the same is obtained in the simulation. For 3.5 temperature, the mean magnetisation is very near to zero, as the magnetization always wavers near zero and the same goes to absolute mean magnetisation it is near to zero.

For the temperature 2.1, the absolute mean magnetisation is expected to be near one as it reaches equilibrium for all the runs. The result obtained proves it.

For temperature 2.265, as the magnetisation states moved between [-1,1], so the absolute mean magnetisation should be near 0.5 and the result obtained is 0.63908, which is near the expected value.

Appendix A

(i) Python source code for Problem 1(a).

Code as per the instructions given in the exercise.

--------------------------------------------------------------------------------------------------------------------------------------

import numpy as np  
from IPython.core.debugger import Tracer  
  
len = 32  
lattice = np.ones((len,len))  
t=np.ones((len+1,len+1))  
  
sum\_ = 0  
*##Tracer()()*for i in range(len):  
 for j in range(len):  
 sum\_ += t[i,j]\*t[i,j+1]+t[i,j]\*t[i+1,j]  
  
print -sum\_  
print -sum\_/(len\*len)

(ii)Python source code for Problem 1(b).

Code as per the instructions given in the exercise.

--------------------------------------------------------------------------------------------------------------------------------------

from \_\_future\_\_ import division  
import pylab  
import numpy as np  
import math  
import matplotlib.pyplot as plt  
from IPython.core.debugger import Tracer  
import pickle  
  
  
*# Simple 1 dimensional lattice with periodic boundary  
#  
# This is an incomplete example and only to give you some ideas of the  
# possible ways for implementing the lattice and Monte Carlo simulation!*class PLattice:  
 def \_\_init\_\_(self, length,seed):  
 self.seed = seed  
 np.random.seed(self.seed)  
 *# create an array storing the values on the lattice and set them  
 # to initial\_value  
 #self.lattice = -np.ones((5,5))  
 #t=np.ones((33,33))* self.lattice = np.matrix.round(np.random.rand(length,length))  
 self.lattice[self.lattice == 0] = -1  
 self.length = length  
 *# compute the energy of the initial configuration* self.energy = self.compute\_energy()  
 self.magnetisation = self.compute\_magnetisation()  
 *#self.seed = 5555  
 # ... more variables  
  
 # see below the flip method and the flip example in the main-part on how   
 # \_\_getitem\_\_ and \_\_setitem\_\_ work* def \_\_getitem\_\_(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* return self.lattice[int(idx/self.length),int(idx % self.length)]  
  
 def \_\_setitem\_\_(self, idx, val):  
 *# same here* self.lattice[int(idx/self.length),int(idx % self.length)] = val  
  
 def index\_plus\_column(self,idx):  
  
 x = int(idx/self.length)  
 y =int(idx % self.length)  
  
 if y + 1 == self.length:  
 y = 0  
 else:  
 y = y + 1  
 *#Tracer()()* return self.lattice[x,y]  
  
  
 def index\_minus\_column(self,idx):  
 return self.lattice[int(idx/self.length),int(idx % self.length)-1]  
  
 def index\_plus\_row(self,idx):  
 idx = idx + self.length  
 idx %= self.length \*\* 2  
 return self.lattice[int(idx/self.length),int(idx % self.length)]  
  
 def index\_minus\_row(self,idx):  
 idx = idx - self.length  
 idx %= self.length \*\* 2  
 return self.lattice[int(idx/self.length),int(idx % self.length)]  
  
 def flip(self, idx):  
 *# this is equal to self[idx] = -1 \* self[idx]  
 # self[idx] causes call to either \_\_getitem\_\_ or \_\_setitem\_\_ (see below)* self[idx] \*= -1  
  
 def compute\_energy(self):  
 sum\_ = 0  
 *## Create the modified lattice as it is periodic boundary* temp = np.r\_[ self.lattice, [self.lattice[0]] ]  
 temp = np.c\_[ temp, temp[:,0] ]  
 temp[self.length,self.length]=0  
 *# compute the energy here and return it* for i in range(self.length):  
 for j in range(self.length):  
 sum\_ += temp[i,j]\*temp[i,j+1] + temp[i,j]\*temp[i+1,j]  
  
 return -sum\_  
  
 def compute\_magnetisation(self):  
 *#Tracer()()* return np.sum(self.lattice)  
  
 def delta\_energy(self,idx):  
  
 *#Tracer()()* return 2\*self[idx] \*(self.index\_plus\_column(idx) + self.index\_minus\_column(idx) \  
 + self.index\_plus\_row(idx) + self.index\_minus\_row(idx))  
  
 def update\_energy(self,energy):  
  
 *##Tracer()()* self.energy += energy  
  
 def update\_magnetisation(self,idx):  
 *#Tracer()()* self.magnetisation += (2 \* self.lattice[int(idx/self.length),int(idx % self.length)])  
  
 def do\_montecarlo\_step(self,temperature):  
  
  
  
 *#np.random.seed(self.seed)  
 # implement here and remove pass* config = np.matrix.round((self.length\*\*2-1)\*(np.random.rand(self.length\*\*2,1)))  
 *#print "Unique Config = {}".format(np.unique(config))* for i in range(self.length\*\*2):  
 *#Tracer()()* temp\_energy = self.delta\_energy(int(config[i]))  
 if temp\_energy <= 0:  
 *##l[config[i]] = -1\*l[config[i]] ##Flip the state* self.flip(int(config[i]))  
 self.update\_energy(temp\_energy)  
 self.update\_magnetisation(int(config[i]))  
 elif np.random.random() < math.exp(-temp\_energy/temperature) : *## if r < w accept the change  
 ##l[config[i]] = -1\*l[config[i]]* self.flip(int(config[i]))  
 self.update\_energy(temp\_energy)  
 self.update\_magnetisation(int(config[i]))  
  
 return (self.energy\*1.0/self.length\*\*2,self.magnetisation\*1.0/self.length\*\*2)  
 *# ... more methods*if \_\_name\_\_ == "\_\_main\_\_":  
 *# create the lattice object* length =32  
 *##Tracer()()* temperature = [2.1,3.5]  
 *#temperature = [2.1]  
 #temperature = [2.265]* mcs\_step = 1000  
  
 seed = [1000,2000,3000,4000,5000,6000]  
  
 mcs\_step\_runs = 5  
 *##print l.energy* mean =[]  
 mean\_abs =[]  
 mcs\_linspace = np.linspace(1,mcs\_step,mcs\_step)  
  
 mean\_magnetisation = np.zeros((mcs\_step\_runs))  
  
 *## Do MC for given mcs\_step* for j in temperature:  
 *#Tracer()()* mcs\_energy = []  
 mcs\_magnetisation = []  
 file\_mag = open('C:\Python27\CS\Exercise10\FigureFor1b\Magnetisation T %1.3f.txt' % j, 'wb')  
 file\_energy = open('C:\Python27\CS\Exercise10\FigureFor1b\Energy T %1.3f.txt' % j, 'wb')  
  
 pylab.figure()  
  
 for k in range(mcs\_step\_runs):  
  
 l = PLattice(length,seed[k])  
  
 mcs\_e =[]  
 mcs\_m=[]  
  
 for i in range(mcs\_step):  
  
 *#Tracer()()* e,m = l.do\_montecarlo\_step(j)  
 *#print m* mcs\_e.append(e); mcs\_m.append(m)  
  
 if i % 100 == 0:  
 print "%d MCS completed for T %1.3f - Run = %d" % (i,j,k)  
  
 mcs\_energy.append(mcs\_e)  
 mcs\_magnetisation.append(mcs\_m)  
  
  
 pylab.plot(mcs\_linspace,mcs\_magnetisation[k],marker='.',label='Run %d' % k)  
 pylab.legend(framealpha=0.5,loc=0)  
 plt.title('Magnetisation for T = %1.3f' % j)  
 pylab.xlabel("time(MCS)")  
 pylab.ylabel("magnetisation")  
 *#Tracer()()* if (i == mcs\_step-1) & (k == 2):  
 if j == temperature[0]:  
 lattice\_temp1 = l.lattice  
 elif j == temperature[1]:  
 lattice\_temp2 = l.lattice  
 *#Tracer()()* if j == temperature[0]:  
 mag\_1 = np.asarray(mcs\_magnetisation)  
 mag\_1\_temp = mag\_1[:,600:]  
 mag\_1\_temp\_abs = np.absolute(mag\_1\_temp)  
 mean.append(np.mean(np.mean(mag\_1\_temp,axis=1)))  
 mean\_abs.append(np.mean(np.mean(mag\_1\_temp\_abs,axis=1)))  
  
 if j == temperature[1]:  
 mag\_2 = np.asarray(mcs\_magnetisation)  
 mag\_2\_temp = mag\_2[:,100:]  
 mag\_2\_temp\_abs = np.absolute(mag\_2\_temp)  
 mean.append(np.mean(np.mean(mag\_2\_temp,axis=1)))  
 mean\_abs.append(np.mean(np.mean(mag\_2\_temp\_abs,axis=1)))  
  
 pylab.savefig('C:\Python27\CS\Exercise10\FigureFor1b\MagnetisationRuns T %1.3f.png' % j,bbox\_inches='tight')  
  
  
 pylab.figure()  
 *#pl.ylim((95.7, 124.4))* pylab.plot(mcs\_linspace,mcs\_energy[0],marker='.',label='Internal Energy(Hamiltonian)')  
 pylab.plot(mcs\_linspace,mcs\_magnetisation[0],marker='.',label='Magnetisation')  
 pylab.legend(framealpha=0.5,loc=0)  
 plt.title('Magnetisation for T %1.3f' % j)  
 pylab.xlabel("time(MCS)")  
 pylab.ylabel("internal energy\magnetisation")  
 pylab.savefig('C:\Python27\CS\Exercise10\FigureFor1b\Energy&Magnetisation for T %1.3f.png' % j,bbox\_inches='tight')  
  
 pylab.figure()  
 plt.imshow(lattice\_temp1, interpolation='nearest')  
 plt.title('Equilibrium State for T %1.3f' % temperature[0])  
 pylab.savefig('C:\Python27\CS\Exercise10\FigureFor1b\Lattice for T %1.3f.png' % temperature[0],bbox\_inches='tight')  
  
 pylab.figure()  
 plt.imshow(lattice\_temp2, interpolation='nearest')  
 plt.title('Equilibrium State for T %1.3f' % temperature[1])  
 pylab.savefig('C:\Python27\CS\Exercise10\FigureFor1b\Lattice for T %1.3f.png' % temperature[1],bbox\_inches='tight')  
  
 print 'Mean for T %1.3f is %1.5f and for T %1.3f is %1.5f' % (temperature[0],mean[0],temperature[1],mean[1])  
 print 'Mean of absolute m values for T %1.3f is %1.5f and for T %1.3f is %1.5f' % (temperature[0],mean\_abs[0],temperature[1],mean\_abs[1])

(iii)Python source code for Problem 1c.

Code as per the instructions given in the exercise.

--------------------------------------------------------------------------------------------------------------------------------------

from \_\_future\_\_ import division  
import pylab  
import numpy as np  
import math  
import matplotlib.pyplot as plt  
from IPython.core.debugger import Tracer  
import pickle  
  
  
*# Simple 1 dimensional lattice with periodic boundary  
#  
# This is an incomplete example and only to give you some ideas of the  
# possible ways for implementing the lattice and Monte Carlo simulation!*class PLattice:  
 def \_\_init\_\_(self, length,seed):  
 *# create an array storing the values on the lattice and set them  
 # to initial\_value  
 #self.lattice = -np.ones((5,5))  
 #t=np.ones((33,33))* self.lattice = np.matrix.round(np.random.rand(length,length))  
 self.lattice[self.lattice == 0] = -1  
 self.length = length  
 *# compute the energy of the initial configuration* self.energy = self.compute\_energy()  
 self.magnetisation = self.compute\_magnetisation()  
 self.seed = seed  
 *#self.seed = 5555  
 # ... more variables  
  
 # see below the flip method and the flip example in the main-part on how   
 # \_\_getitem\_\_ and \_\_setitem\_\_ work* def \_\_getitem\_\_(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* return self.lattice[int(idx/self.length),int(idx % self.length)]  
  
 def \_\_setitem\_\_(self, idx, val):  
 *# same here* self.lattice[int(idx/self.length),int(idx % self.length)] = val  
  
 def index\_plus\_column(self,idx):  
  
 x = int(idx/self.length)  
 y =int(idx % self.length)  
  
 if y + 1 == self.length:  
 y = 0  
 else:  
 y = y + 1  
 *#Tracer()()* return self.lattice[x,y]  
  
  
 def index\_minus\_column(self,idx):  
 return self.lattice[int(idx/self.length),int(idx % self.length)-1]  
  
 def index\_plus\_row(self,idx):  
 idx = idx + self.length  
 idx %= self.length \*\* 2  
 return self.lattice[int(idx/self.length),int(idx % self.length)]  
  
 def index\_minus\_row(self,idx):  
 idx = idx - self.length  
 idx %= self.length \*\* 2  
 return self.lattice[int(idx/self.length),int(idx % self.length)]  
  
 def flip(self, idx):  
 *# this is equal to self[idx] = -1 \* self[idx]  
 # self[idx] causes call to either \_\_getitem\_\_ or \_\_setitem\_\_ (see below)* self[idx] \*= -1  
  
 def compute\_energy(self):  
 sum\_ = 0  
 *## Create the modified lattice as it is periodic boundary* temp = np.r\_[ self.lattice, [self.lattice[0]] ]  
 temp = np.c\_[ temp, temp[:,0] ]  
 temp[self.length,self.length]=0  
 *# compute the energy here and return it* for i in range(self.length):  
 for j in range(self.length):  
 sum\_ += temp[i,j]\*temp[i,j+1] + temp[i,j]\*temp[i+1,j]  
  
 return -sum\_  
  
 def compute\_magnetisation(self):  
 *#Tracer()()* return np.sum(self.lattice)  
  
 def delta\_energy(self,idx):  
  
 *#Tracer()()* return 2\*self[idx] \*(self.index\_plus\_column(idx) + self.index\_minus\_column(idx) \  
 + self.index\_plus\_row(idx) + self.index\_minus\_row(idx))  
  
 def update\_energy(self,energy):  
  
 *##Tracer()()* self.energy += energy  
  
 def update\_magnetisation(self,idx):  
 *#Tracer()()* self.magnetisation += (2 \* self.lattice[int(idx/self.length),int(idx % self.length)])  
  
 def do\_montecarlo\_step(self,temperature):  
  
  
  
 *#np.random.seed(self.seed)  
 # implement here and remove pass* config = np.matrix.round((self.length\*\*2-1)\*(np.random.rand(self.length\*\*2,1)))  
 *#print "Unique Config = {}".format(np.unique(config))* for i in range(self.length\*\*2):  
 *#Tracer()()* temp\_energy = self.delta\_energy(int(config[i]))  
 if temp\_energy <= 0:  
 *##l[config[i]] = -1\*l[config[i]] ##Flip the state* self.flip(int(config[i]))  
 self.update\_energy(temp\_energy)  
 self.update\_magnetisation(int(config[i]))  
 elif np.random.random() < math.exp(-temp\_energy/temperature) : *## if r < w accept the change  
 ##l[config[i]] = -1\*l[config[i]]* self.flip(int(config[i]))  
 self.update\_energy(temp\_energy)  
 self.update\_magnetisation(int(config[i]))  
  
 return (self.energy\*1.0/self.length\*\*2,self.magnetisation\*1.0/self.length\*\*2)  
 *# ... more methods*if \_\_name\_\_ == "\_\_main\_\_":  
 *# create the lattice object* length =32  
 *##Tracer()()* temperature = [2.265]  
 mcs\_step = 50000  
  
 seed = [1000,2000,3000,4000,5000,6000]  
  
 mcs\_step\_runs = 2  
 *##print l.energy* mean =[]  
 mean\_abs =[]  
  
  
 mcs\_linspace = np.linspace(1,mcs\_step,mcs\_step)  
  
 *## Do MC for given mcs\_step* for j in temperature:  
 *#Tracer()()* mcs\_energy = []  
 mcs\_magnetisation = []  
 file\_mag = open('C:\Python27\CS\Exercise10\FigureFor1c\Magnetisation for T %1.3f.txt' % j, 'wb')  
 file\_energy = open('C:\Python27\CS\Exercise10\FigureFor1c\Energy for T %1.3f.txt' % j, 'wb')  
  
 pylab.figure()  
  
 for k in range(mcs\_step\_runs):  
 *#Tracer()()* l = PLattice(length,seed[k])  
  
 mcs\_e =[]  
 mcs\_m=[]  
  
 for i in range(mcs\_step):  
  
 *#Tracer()()* e,m = l.do\_montecarlo\_step(j)  
 *#print m* mcs\_e.append(e); mcs\_m.append(m)  
  
 if i % 100 == 0:  
 print "%d MCS completed for T %1.3f - Run = %d" % (i,j,k)  
  
 mcs\_energy.append(mcs\_e)  
 mcs\_magnetisation.append(mcs\_m)  
  
  
 pylab.plot(mcs\_linspace,mcs\_magnetisation[k],marker='.',label='Run %d' % k)  
 pylab.legend(framealpha=0.5,loc=0)  
 plt.title('Magnetisation for T = %f' % j)  
 pylab.xlabel("time(MCS)")  
 pylab.ylabel("magnetisation")  
  
 *#Tracer()()  
 #pickle.dump(mcs\_magnetisation, file\_mag)  
 #pickle.dump(mcs\_energy, file\_energy)* file\_mag.close()  
 file\_energy.close()  
  
 mag\_1 = np.asarray(mcs\_magnetisation)  
 mag\_1\_temp = mag\_1[:,1000:]  
 mag\_1\_temp\_abs = np.absolute(mag\_1\_temp)  
 mean.append(np.mean(np.mean(mag\_1\_temp,axis=1)))  
 mean\_abs.append(np.mean(np.mean(mag\_1\_temp\_abs,axis=1)))  
  
  
 pylab.savefig('C:\Python27\CS\Exercise10\FigureFor1c\MagnetisationRuns T %1.3f.png' % j)  
  
 pylab.figure()  
 *#pl.ylim((95.7, 124.4))* pylab.plot(mcs\_linspace,mcs\_energy[0],marker='.',label='Internal Energy(Hamiltonian)')  
 pylab.plot(mcs\_linspace,mcs\_magnetisation[0],marker='.',label='Magnetisation')  
 pylab.legend(framealpha=0.5,loc=0)  
 plt.title('Magnetisation for T = %f' % j)  
 pylab.xlabel("time(MCS)")  
 pylab.ylabel("internal energy\magnetisation")  
 pylab.savefig('C:\Python27\CS\Exercise10\FigureFor1c\Energy&Magnetisation for T %1.3f.png' % j)  
  
 pylab.figure()  
 plt.imshow(l.lattice)  
 plt.title('Equilibrium State for T %1.3f' % temperature[0])  
 pylab.savefig('C:\Python27\CS\Exercise10\FigureFor1c\Lattice for T %1.3f.png' % temperature[0],bbox\_inches='tight')  
  
 print 'Mean for T %1.3f is %1.5f' % (temperature[0],mean[0])  
 print 'Mean of absolute m values for T %1.3f is %1.5f' % (temperature[0],mean\_abs[0])