

Monte-Carlo Simulations of the 2-D Ising Model

Prateek Mehta and Hui Li


December 13, 2014

Contents

1	A brief note about this document	1
2	A reliable random number generator	1
3	2-D Ising Model Simulator	2
4	Average Spin Trajectories	5
5	Magnetization and Susceptibility	6
5.1	Submission to the queue	6
5.2	Magnetization	6
5.3	Susceptibility	7
6	Finite Size Scaling	8
6.1	Job Submission	8
6.2	Fitting	9
7	Correlation Function	10
8	A little bit about Umbrella Sampling	11
9	Flipping behavior of a single spin	12

1 A brief note about this document

This document was prepared with Emacs [orgmode](#). All the code was written and executed within the org-document and the results were captured in place. It has been exported into a latex pdf You can get the original org-file and all the other relevant files at this git repo: <https://github.com/prtkm/ising-monte-carlo>.

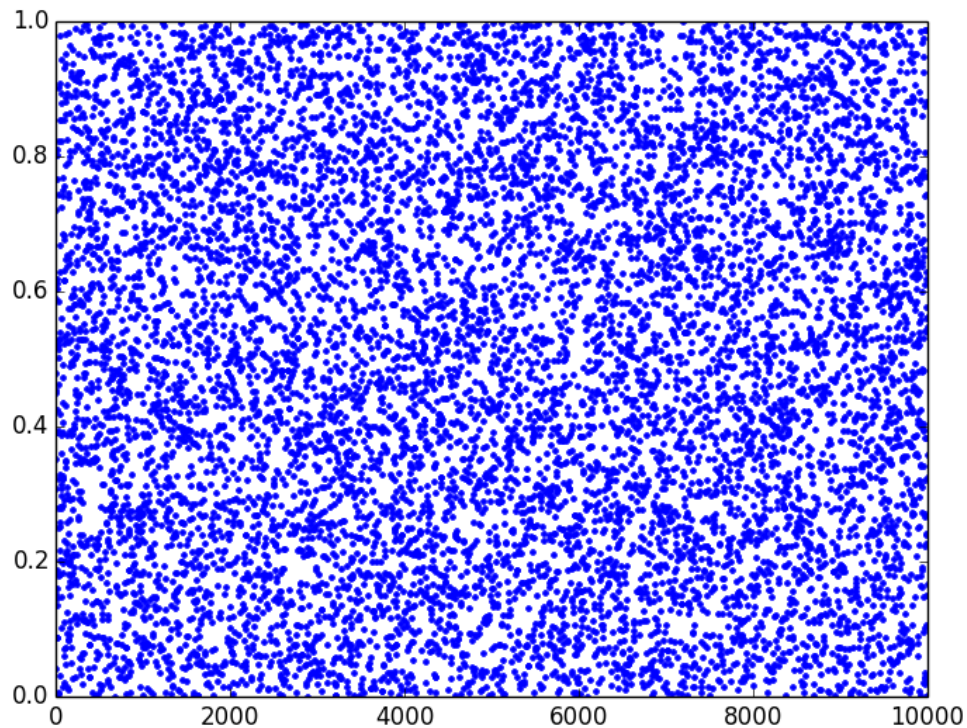
If you are reading the pdf you can view the source by clicking here:  (Mac Preview does not read this attachment).

2 A reliable random number generator

For reliable Monte-Carlo simulations, we need a good random number generator. We shall use Python's `numpy.random`. It uses the *Mersenne twister* pseudo-random number generator, so we should expect to get a uniform random distribution. We test this here. Let us generate a 10000 random numbers and plot them.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 plt.plot(range(10000), np.random.random(10000), 'r.')
```

```
5 plt.savefig('images/numpy-random.png')
6 plt.show()
```



That looks pretty uniform!

3 2-D Ising Model Simulator

The main ideas to simulate the 2-D Ising Model using Metropolis Monte Carlo are as follows:

- We create a $n \times n$ lattice with a random spin configuration
- For the purposes of our Monte Carlo simulation we start off by randomly flipping a spin and calculate the change in energy, ΔE
- If ΔE is negative, we accept the new configuration and move to the next step
- If ΔE is positive, we select a random number between 0 and 1, and accept the configuration only if the number is less than $e^{-\Delta E/k_B T}$. This is the Metropolis technique. This saves computation time by selecting the more probable configurations.
- We use periodic boundary conditions. This effectively reduces the geometry of the problem to a torus.
- To simulate over a range of temperatures, we will submit jobs to the queue system for every temperature. This is useful, especially for large lattice sizes where a lot of simulations are required to reach convergence. We have found that at large lattices flipping one spin per step is pretty inefficient (for a 1000×1000 lattice, we are practically flipping one spin in a million!). A better approach than ours is probably needed in such cases, maybe flipping multiple spins at the same time.

Here is our main script that does all the work. We write this out to our system for use later.

```

1  from __future__ import division
2  import numpy as np
3
4  def init_lattice(n):
5
6      '''Create a n*n lattice with random spin configuration'''
7
8      lattice = np.random.choice([1, -1], size=(n, n))
9      return lattice
10
11
12  def deltaE(S0, Sn, J, H):
13
14      '''Energy difference for a spin flip'''
15
16      return 2 * S0 * (H + J * Sn)
17
18
19  def ising(n=200,
20           nsteps=500000,
21           H=0,
22           J=1,
23           T=1,
24           count_spins = False,
25           countij = [1,1],
26           correlation=False,
27           corr_ij=[0,0],
28           corr_r=1):
29
30      '''Ising Model Simulator. If count_spins = True, only flipping behavior of 1 site is studied.'''
31
32      lattice = init_lattice(n)
33      energy = 0
34      energies = []
35      spins = []
36      spin = np.sum(lattice)
37      icount, jcount = countij
38      counted_spins = [lattice[icount, jcount]]
39      counted_intervals = []
40      icorr, jcorr = corr_ij
41      Sis = []
42      SiSjs = []
43
44      for step in xrange(nsteps):
45
46          i = np.random.randint(n)
47          j = np.random.randint(n)
48
49          # Periodic Boundary Condition
50          Sn = lattice[(i - 1) % n, j] + lattice[(i + 1) % n, j] + \
51              lattice[i, (j - 1) % n] + lattice[i, (j + 1) % n]
52
53          dE = deltaE(lattice[i, j], Sn, J, H)
54
55          if dE < 0 or np.random.random() < np.exp(-dE/T):
56              lattice[i, j] = -lattice[i, j]
57              energy += dE
58              energies.append(energy)
59              # Note that the spin is collected at every step
60              spin += 2*lattice[i, j]
61
62          if count_spins:
63              ispin = lattice[icount, jcount]
64              if ispin != counted_spins[-1]:
65                  counted_spins.append(ispin)
66                  counted_interval = step - sum(counted_intervals)
67
68                  counted_intervals.append(counted_interval)
69
70          if correlation:
71              Sn_corr = lattice[(icorr - corr_r) % n, jcorr] + lattice[(icorr + corr_r) % n, jcorr] + \
72                  lattice[icorr, (jcorr - corr_r) % n] + lattice[icorr, (jcorr + corr_r) % n]
73              Si = lattice[icorr, jcorr]
74              SiSj = Si * Sn_corr / 4.0
75              Sis.append(Si)
76              SiSjs.append(SiSj)
77
78          spins.append(spin)
79
80      if correlation:

```

```

81         return Sis, SiSjs
82
83     if count_spins:
84         return counted_spins, counted_intervals
85
86     return lattice, energies, spins
87
88
89 def ising1000(n=1000, nsteps=10000000000, H=0, J=1, T=1):
90
91     '''Ising Model Simulator. Special case for very large lattices.
92     To reduce some memory usage:
93     spin is added to the array every 1000 steps.
94     Energies are not returned.
95     Still pretty inefficient!
96     '''
97
98     lattice = init_lattice(n)
99     energy = 0
100
101     spins = []
102     spin = np.sum(lattice)
103     for istep, step in enumerate(xrange(nsteps)):
104
105         i = np.random.randint(n)
106         j = np.random.randint(n)
107
108         # Periodic Boundary Condition
109         Sn = lattice[(i - 1) % n, j] + lattice[(i + 1) % n, j] + \
110             lattice[i, (j - 1) % n] + lattice[i, (j + 1) % n]
111
112         dE = deltaE(lattice[i, j], Sn, J, H)
113
114         if dE < 0 or np.random.random() < np.exp(-dE/T):
115             lattice[i, j] = -lattice[i, j]
116             energy += dE
117             spin += 2*lattice[i, j]
118         if istep % 1000 == 0:
119             spins.append(spin)
120     return lattice, spins
121
122
123 def write_job_script(wd='.', n=10, s=1000, i=1, T=1., nprocs=1, pe='smp', name = 'batch', q = 'long'):
124     '''
125     This is a function that writes a script to submit MC jobs
126     '''
127     py_file = '/afs/crc.nd.edu/user/p/pmehta1/ising-monte-carlo/spins.py'
128     script='''#!/bin/bash
129     $$ -N {0}
130     $$ -pe {1} {2}
131     $$ -q {3}
132     $$ -cwd
133     '''format(name, pe, nprocs, q)
134
135     if nprocs > 1:
136         cmd = 'mpirun -np $NSLOTS python'
137         script+='{6} {5} -n {0} -s {1} -i {2} -t {3} -w {4}'.format(n, s, i, T, wd, py_file, cmd)
138
139     else:
140         script+='python {5} -n {0} -s {1} -i {2} -t {3} -w {4}'.format(n, s, i, T, wd, py_file)
141
142     with open('{0}/qscript'.format(wd), 'w') as f:
143         f.write(script)
144
145 def run_job(wd):
146     '''
147     Submit job to the queue
148     '''
149     import os
150     from subprocess import Popen, PIPE
151     cwd = os.getcwd()
152     os.chdir(wd)
153     p = Popen(['qsub', 'qscript'], stdout=PIPE, stderr=PIPE)
154     out, err = p.communicate()
155
156     if out == '' or err != '':
157         raise Exception('something went wrong in qsub:\n\n{0}'.format(err))
158     jobid = out.split()[2]
159     f = open('jobid', 'w')
160
161     f.write(jobid)

```

```

162     f.close()
163     os.chdir(cwd)
164     return out.strip()

```

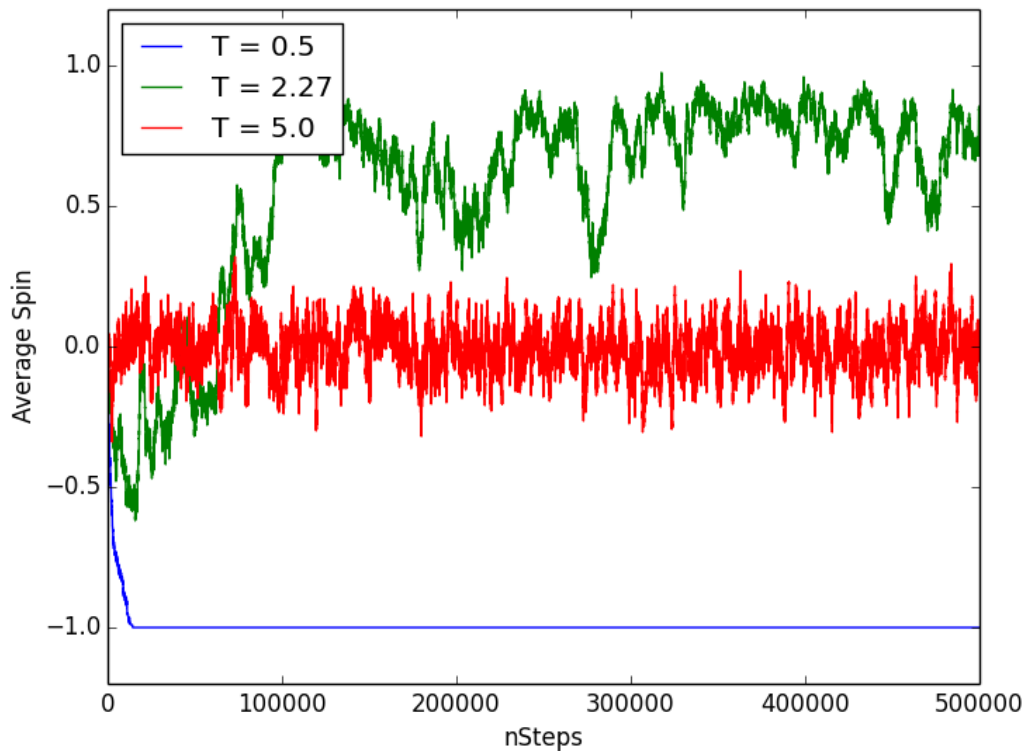
4 Average Spin Trajectories

Now we plot the average spin trajectories at three different temperatures. The spin is collected at every step of the Monte Carlo simulation, regardless of whether we accepted the flip or not.

```

1  import matplotlib.pyplot as plt
2  from ising import *
3
4  temperatures = [0.5, 2.27, 5.0]
5
6  for T in temperatures:
7
8      lattice, energies, spins = ising(n=20, nsteps = 500000, T=T)
9      spins = np.array(spins) / 20. ** 2
10     plt.plot(range(len(spins)), spins, label = 'T = {}'.format(T))
11 plt.legend(loc = 'best')
12 plt.xlabel('nSteps')
13 plt.ylabel('Average Spin')
14 plt.ylim(-1.2, 1.2)
15 plt.savefig('images/average-spin.png')
16 plt.show()

```



This looks pretty much like what one would expect. At a low temperature the average spin per site is 1, meaning that all the points in the lattice have the same spin. At a temperature close to Onsager's T_c , the system has an intermediate spin, and at a high temperature, the system has no net spin.

5 Magnetization and Susceptibility

In this section, we calculate the magnetization and susceptibility at different lattice sizes and temperatures.

5.1 Submission to the queue

To simulate things for multiple temperatures and lattice sizes, we submitted jobs to the queueing system. This is the python script that we will execute in the queue.

```
1  #!/usr/bin/env python
2  import os
3  from ising import ising, ising1000
4  import sys, getopt
5  opts, args = getopt.getopt(sys.argv[1:], 'n:s:t:w')
6
7  for key, val in opts:
8
9      if key == '-n': n = int(val)
10     elif key == '-s': nsteps = int(val)
11     elif key == '-t': T = float(val)
12     elif key == '-i': index = int(val)
13     elif key == '-w': wd = str(val)
14
15 if n < 500:
16     lattice, energies, spins = ising(n=n, nsteps=nsteps, T=T)
17 else:
18     lattice, spins = ising1000(n=n, nsteps=nsteps, T=T)
19
20 with open(os.path.join(wd, 'temp-{1}.out'.format(wd, index)), 'w') as f:
21     for i, spin in enumerate(spins):
22         if i % 1000 == 0:
23             f.write("{0}\t{1}\n".format(i, spin))
```

We submit jobs here for multiple lattice sizes and temperatures.

```
1  from ising import *
2  import os
3
4  Ns = [10, 20, 50, 100, 1000] # System Size
5  T_Tcs = np.linspace(0.5, 1.7, 30) # T/Tc
6  Tc = 2.268 # Onsager's Tc
7
8  for n in Ns:
9      for i, T_Tc in enumerate(T_Tcs):
10         T = T_Tc*Tc
11         wd = 'magnetization/size-{0}/temp-{1}'.format(n, i)
12         if not os.path.exists(wd):
13             os.makedirs(wd)
14         if n != 1000:
15             write_job_script(wd=wd, n=n, s= n * 1000000, T=T, i=i)
16         else:
17             write_job_script(wd=wd, n=n, s= n * 1000000, T=T, i=i, nprocs = 1, q='long')
18         run_job(wd)
```

5.2 Magnetization

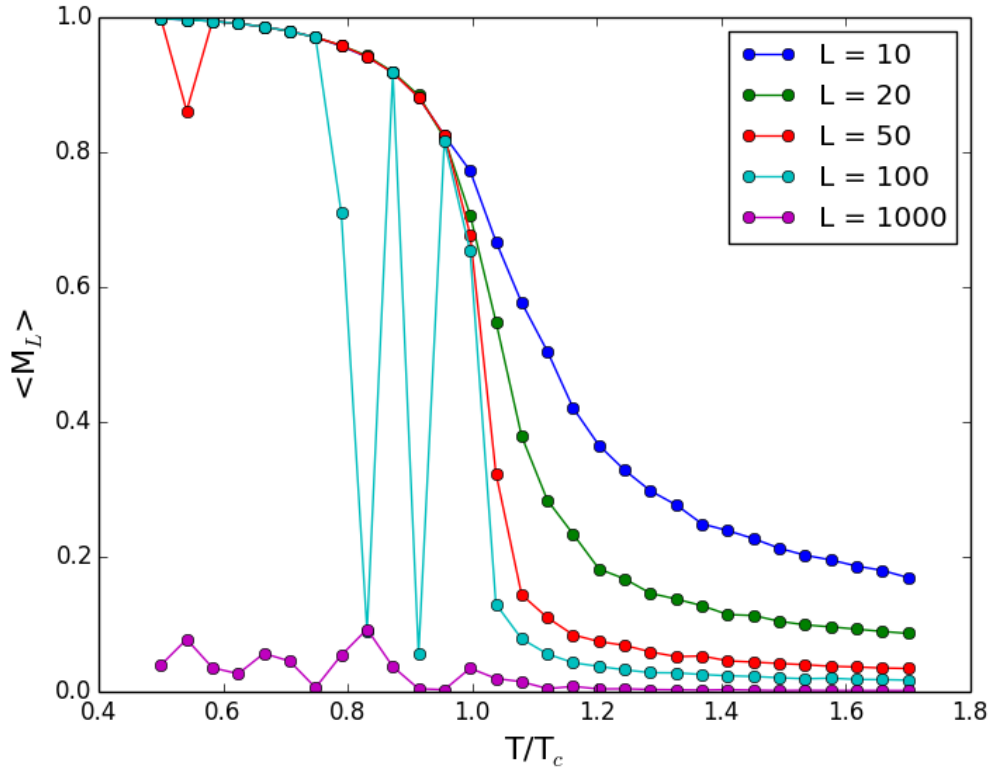
Here we plot the magnetization. We see that for larger lattice sizes, the system has not reached equilibrium and the data is very noisy. Otherwise the plot matches what is known, i.e, full magnetization at low temperatures, disorder at higher temperatures, with a transition at the critical temperature. The transition is sharper at larger lattice sizes.

```
1  from __future__ import division
2  import matplotlib.pyplot as plt
3  from ising import *
4  import os
5
6  Ns = [10, 20, 50, 100, 1000] # System Size
7  T_Tcs = np.linspace(0.5, 1.7, 30) # T/Tc
8  Tc = 2.268 # Onsager's Tc
```

```

9
10 for n in Ns:
11     avgspins = []
12     for i, T_Tc in enumerate(T_Tcs):
13         T = T_Tc*Tc
14         indices, spins = np.loadtxt('magnetization/size-{0}/temp-{1}/temp-{1}.out'.format(n,i), unpack =True)
15         spins = spins[int(len(spins)/2):]
16         avgspin = np.sum(np.abs(spins)) / n ** 2 / len(spins)
17         avgspins.append(avgspin)
18     plt.plot(T_Tcs, avgspins, 'o-', label = 'L = {0}'.format(n))
19
20 plt.xlabel('T/T_{c}', fontsize = 16)
21 plt.ylabel('<M_L>', fontsize = 16)
22 plt.legend()
23 plt.savefig('images/magnetization.png')
24 plt.show()

```



5.3 Susceptibility

Susceptibility is the second derivative of the energy and measures the extent to which the lattice will be magnetized. It is discontinuous at the critical temperature. We see that the peak gets sharper, and T/T_c gets closer to unity with increasing lattice size. This means that larger lattices give better approximations of Onsager's T_c , and an infinite lattice would have exact resemblance. The data looks very noisy at high temperatures!

```

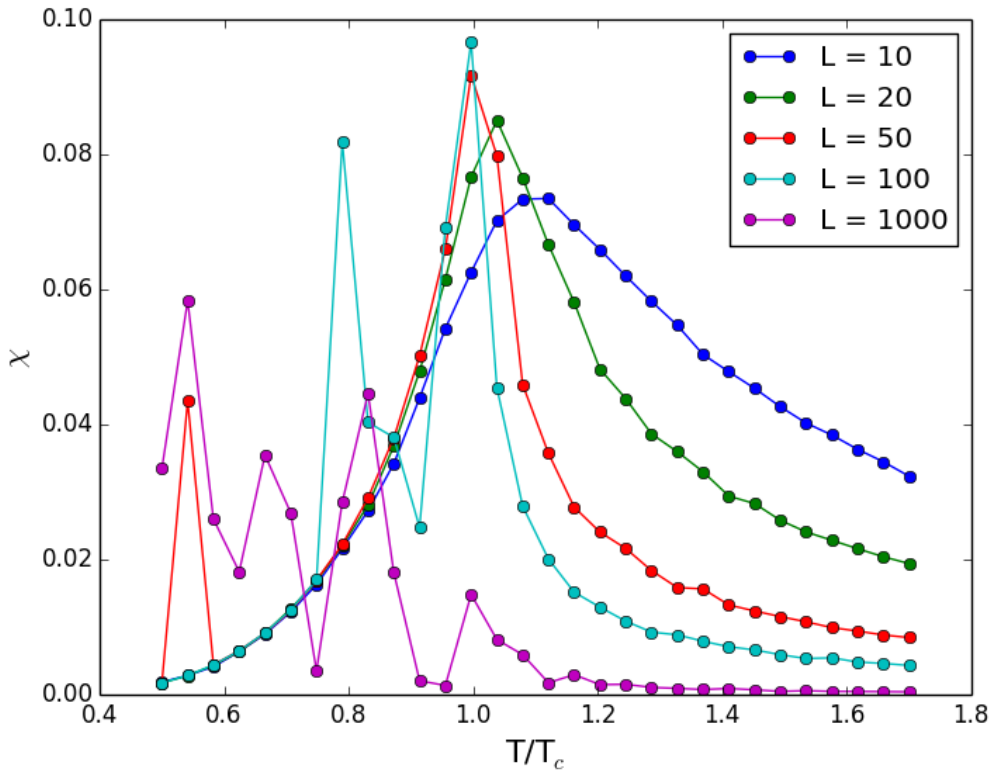
1 from __future__ import division
2 import matplotlib.pyplot as plt
3 from ising import *
4 import os
5
6 Ns = [10, 20, 50, 100, 1000] # System Size
7 T_Tcs = np.linspace(0.5, 1.7, 30) # T/Tc
8 Tc = 2.268 # Onsager's Tc
9

```

```

10 for n in Ns:
11     avgspins = []
12     Xs = []
13     for i, T_Tc in enumerate(T_Tcs):
14         T = T_Tc*Tc
15         indices, spins = np.loadtxt('magnetization/size-{0}/temp-{1}/temp-{1}.out'.format(n,i), unpack =True)
16         spins = spins[int(len(spins)/2):]
17         avgspin = np.sum(np.abs(spins)) / n ** 2 / len(spins)
18         X = np.abs(np.sum(((np.abs(spins) / n ** 2) ** 2)) \
19                     / len(spins) - avgspin) / T
20         avgspins.append(avgspin)
21         Xs.append(X)
22     plt.plot(T_Tcs, Xs, 'o-', label = 'L = {0}'.format(n))
23 plt.xlabel('T/T_{c}', fontsize = 16)
24 plt.ylabel('$\chi$', fontsize = 16)
25 plt.legend()
26 plt.savefig('images/susceptibility.png')
27 plt.show()

```



6 Finite Size Scaling

We know that in the infinite size limit, all observables behave as $O \propto |T - T_c|^\alpha$. To estimate the value of T_c and the critical exponent for the magnetization (B), from our simulations, we will do a non linear fit of our data to this relation. For a system size of 50 let us run a few simulations around the critical point so that we can get a good fit it to our scaling relation.

6.1 Job Submission

Here is a script that submits a 100 jobs at different temperatures around the critical point.

```

1 import matplotlib.pyplot as plt
2 from ising import *

```

```

3 import os
4
5 Ns = [50] # System Size
6 T_Tcs = np.linspace(0.9, 1.1, 100) # T/Tc
7 Tc = 2.268 # Onsager's Tc
8
9 for n in Ns:
10     for i, T_Tc in enumerate(T_Tcs):
11         T = T_Tc*Tc
12         wd = 'finite-size-scaling/size-{0}/temp-{1}'.format(n, i)
13         if not os.path.exists(wd):
14             os.makedirs(wd)
15         write_job_script(wd=wd, n=n, s= n * 1000000, T=T, i=i)
16         run_job(wd)

```

6.2 Fitting

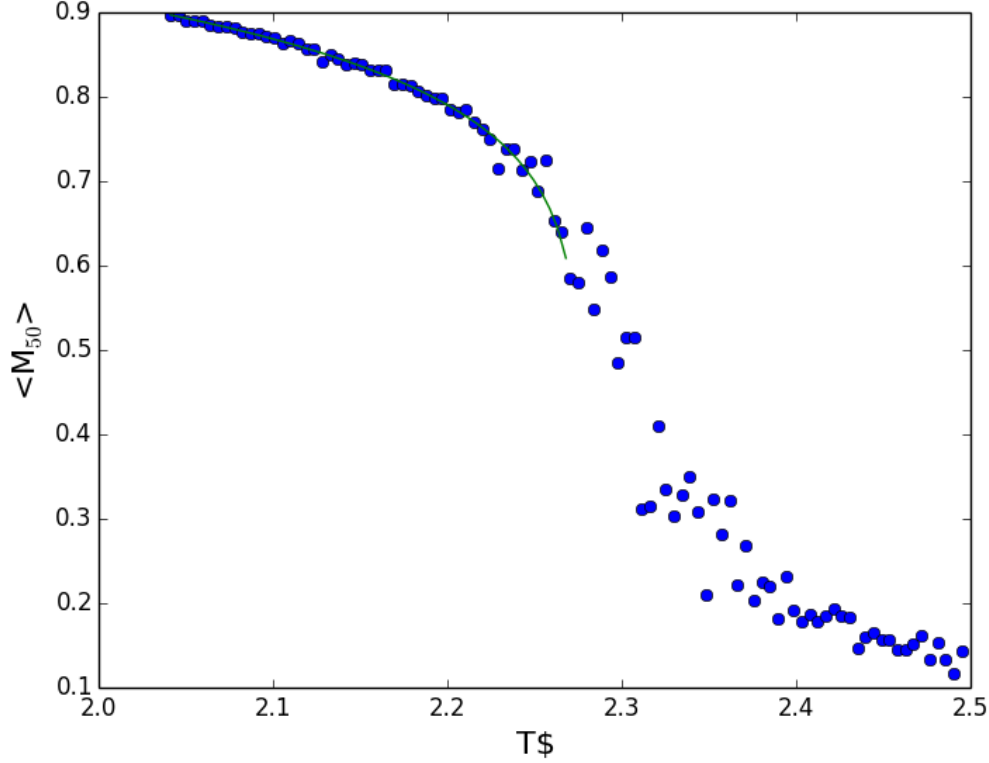
Now we are ready to perform the fitting. We select 40 values on the left of the expected Onsager value for T_c and do a nonlinear fit.

```

1 from __future__ import division
2 import numpy as np
3 np.set_printoptions(precision=3)
4 import matplotlib.pyplot as plt
5 from pycse import nlinfit
6 from ising import *
7
8 n = 50 # System Size
9 T_Tcs = np.linspace(0.9, 1.1, 100) # T/Tc
10 Tc = 2.268 # Onsager's Tc
11 outfile = 'finite-size-scaling/size-{0}/temp-{1}/temp-{1}.out'
12
13 avgspins = []
14
15 for i, T_Tc in enumerate(T_Tcs):
16     T = T_Tc*Tc
17     indices, spins = np.loadtxt(outfile.format(n,i), unpack =True)
18     spins = spins[int(len(spins)/2):]
19     avgspin = np.sum(np.abs(spins)) / n ** 2 / len(spins)
20     avgspins.append(avgspin)
21
22 # data
23 Ts = T_Tcs * Tc
24
25 # Let us fit the first 45 values on the left of Onsager's Tc
26 Ts2fit = Ts[0:40]
27 avgspins2fit = avgspins[0:40]
28
29 # Function to fit to
30 def M_fit(Ts, Tcinf, beta, a):
31
32     M = a * np.abs((-Ts + Tcinf) / Tcinf) ** beta
33     return M
34
35 # Initial guess
36 guess = [2.4, 0.1, 1]
37
38 pars, pint, SE = nlinfit(M_fit, Ts2fit, avgspins2fit, guess, alpha=0.05)
39 Tcinf, beta, a = pint
40
41 print 'T_{{c}} = {0:1.3f} (95% confidence interval = {1:1.3f} {2:1.3f})\n'.format(pars[0], pint[0][0], pint[0][1])
42 print '\beta = {0:1.3f} (95% confidence interval = [{1:1.3f} {2:1.3f}])'.format(pars[1], pint[1][0], pint[1][1])
43
44 # Plotting
45 Tfit = np.linspace(Ts2fit.min(), Tc)
46 plt.plot(Ts, avgspins, 'o')
47 plt.plot(Tfit, M_fit(Tfit, *pars))
48 plt.xlabel('T$', fontsize=16)
49 plt.ylabel('<M$_{50}$>', fontsize=16)
50 plt.savefig('images/finite-size-scaling.png')
51 plt.show()

```

$T_c = 2.275$ (95% confidence interval = 2.259 2.291])
 $\beta = 0.111$ (95% confidence interval = [0.096 0.127])



Looks like its a pretty good fit. We almost exactly reproduce the literature values!

7 Correlation Function

The correlation function is given by $\langle \sigma_i \sigma_j \rangle - \langle \sigma_i \rangle \langle \sigma_j \rangle$.

- Let us consider a spin in a 20×20 lattice, say, $[i, j]$.
- Now, at a given separation, r , we can have a spin in four directions, given by $[i, j + r]$, $[i, j - r]$, $[i + r, j]$, $[i - r, j]$.
- At this point we can calculate the correlation function for various values of r and at different temperatures.

Here is the code that does this.

```

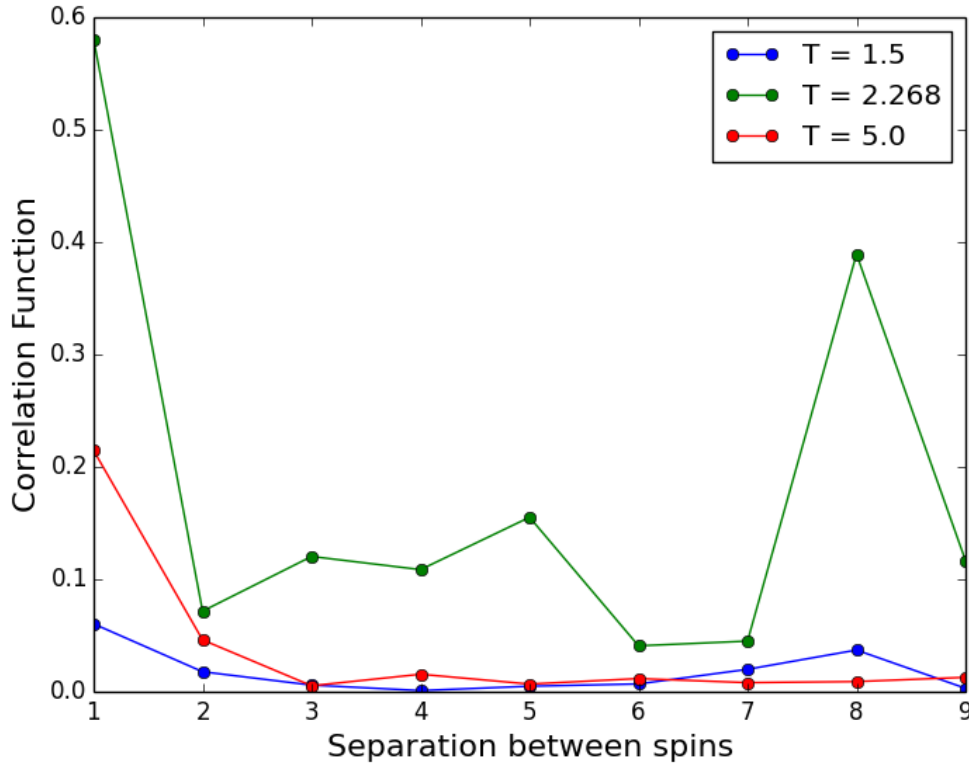
1  from __future__ import division
2  from ising import *
3  import matplotlib.pyplot as plt
4
5
6  n = 20 # Lattice size
7  nsteps = 1000000
8
9  temperatures = [1.5, 2.268, 5.0]
10 # Different values of the separation
11 rs = np.arange(1,10, 1)
12
13 for T in temperatures:
14     corr_funcs = []
15
16     for r in rs:
17         Sis, SiSjs = ising(n=n, nsteps=nsteps, correlation=True, corr_ij=[10,10], corr_r=r, T=T)
18         Si_avg = sum(Sis) / nsteps

```

```

19     SiSj_avg = sum(SiSjs) / nsteps
20     corr_func = np.abs(SiSj_avg - Si_avg ** 2)
21     corr_funcs.append(corr_func)
22     plt.plot(rs, corr_funcs, 'o-', label='T = {0}'.format(T))
23
24 plt.xlabel('Separation between spins', fontsize = 16)
25 plt.ylabel('Correlation Function', fontsize = 16)
26 plt.legend()
27 plt.savefig('images/correlation.png')
28 plt.show()

```



It looks like the correlation function decays pretty fast at low and high temperatures because the spin correlates the most with its nearest neighbours and there is no significant long range correlation. However, the correlation function maximum at the critical temperature and we see long range correlation, probably because of the net magnetization of the system.

8 A little bit about Umbrella Sampling

Umbrella sampling is a non-Boltzmann sampling technique commonly used in systems where the ergodic behavior is hindered by the energy landscape. For example if there is an energy barrier separating two configurations of the system, it might suffer from poor sampling if Metropolis Monte Carlo is used. This is because in the Metropolis sampling, since the probability of overcoming the barrier is low, configurations on either side of the barrier may be poorly sampled, or even unsampled, by the simulation. For example, the melting of a solid has a barrier for phase transition, and a Metropolis simulation might not adequately sample both the solid phase and the liquid phase.

In umbrella sampling, the Boltzmann weighting for Monte Carlo sampling is replaced by a potential chosen to cancel the influence of the energy barrier present, effectively forming a reference system with the barrier removed. The energy is biased like, $E' = E + W$, where W is 0 for the types of configurations we are interested in, but very large for those that we are not interested in. With umbrella sampling, the Monte-Carlo simulation only visits the states we are interested in.

We didn't have time to complete the problem in Chandler 6.10, and it is not shown here.

9 Flipping behavior of a single spin

We see that at high temperatures, the spin flips at short intervals. As we decrease the temperature, the spin hardly flips because the system becomes ordered.

```

1 from ising import *
2 import matplotlib.pyplot as plt
3 temperatures = np.linspace(1.7, 0.5, 6) * 2.26
4
5 ij = [2, 2]
6 for i, T in enumerate(temperatures):
7     counted_spins, counted_intervals = ising(n=10,
8                                             nsteps=1000000,
9                                             H=0,
10                                            J=1,
11                                            T=T,
12                                            count_spins=True,
13                                            countij=ij)
14     plt.subplot(3,2,i+1)
15     plt.hist(counted_intervals, 50)
16     plt.locator_params(nbins=4)
17     plt.title('T = {0}'.format(T))
18     plt.xlabel('Flip Interval Frequency')
19     plt.ylabel('No. of occurrences')
20 plt.tight_layout()
21 plt.savefig('images/histograms.png')
22 plt.show()

```

