# AtomicScaleSimulations_HW6

Created by Yu, Xiongjie, last modified by Yang, Paul on Oct 29, 2016

Questions, clarifications, comments, etc. about this homework can be posted on the course forum on Piazza.

**In this assignment we will explore the Ising model and simulate it in three different ways: conventional Metropolis Monte Carlo, the heat bath algorithm and a cluster algorithm.**
**Example skeleton code is available for your benefit. Although you do not have to use it. If you chose to design your own code, you may skip the system setup section, which is rather specific to the example code.**

Throughout this homework, our system is the classical 2D Ising model with no magnetic field with periodic boundary conditions (PBC). You may picture a lattice of spins, each of which can either point up or down. In the absence of an external magnetic field, the only interaction in the system occurs between nearest-neighbor spins; the Hamiltonian is:

$$H = -J \sum_{\langle i,j \rangle} s_i \cdot s_j$$

where the summation is taken over all nearest neighbor spin pairs in the system. $J$ is the Ising interaction parameter. $s_i$ and $s_j$ are total spin on site i and j, respectively. For classical spins of magnitude s=1, $s_i$ is either 1 for "spin up" or -1 for "spin down".

To observe the evolution of the system, we will measure the total magnetization per spin squared of the system:

$$M^2 = \left\{ \frac{1}{N} \sum_{i=1}^{N} s_i \right\}^2 .$$

This quantity measures the amount of magnetic order in the system in a size-independent manner. If all spins are pointing up, then $M^2$=1. If every spin points up or down with equal probability, then $M^2$=0.

Ideally we would like to study $M^2$ as a function of the spin-coupling strength from βJ=0 to βJ=1.0. If your code is fast enough, this is what you should do. Unfortunately, because Python is slow and it is sometimes hard to write fast code (and some computers are slow) we will only require 3 values (for each algorithm). These values will be:

1. βJ=0.3
2. βJ=0.44069
3. βJ=0.8

For each of these algorithms, you should specify the value after 1000 sweeps of

1. $M^2$, the magnetization per spin squared and
2. the autocorrelation time of $M^2$.

In order to calculate the error, we typically take the standard deviation and divide by square root of the number of effective points. For everything but the conventional metropolis at 0.44096 you should use this method. For this one point though (because the correlation time might be deceivingly large) we will use a different method to calculate the error of your simulation: Run your algorithm 10 times, then calculate the standard deviation of these ten values. This should give you a more accurate assessment of your error and automatically deal with the autocorrelation time.

Let's define a sweep as 1 step of the cluster algorithm or N steps of the conventional Metropolis or the heat bath algorithm, where N is the total number of spins in the system.

To help you validate and debug your code, you should check against the following result

| βJ | M^2 (remember this is per spin) |
|---|---|
| 0.25 | 0.0107(1) |

recall "comparison of datasets" section of HW1 when comparing statistical numbers.

## System Setup

To simulate an Ising lattice *in silico*, let us first initialize a data structure to hold the lattice along with convenient methods to manipulate it. i.e. store a lattice of spins, and create a method flip_spin() to change them. If you are using the example code, open a new file and use cubic_ising_lattice as follows:

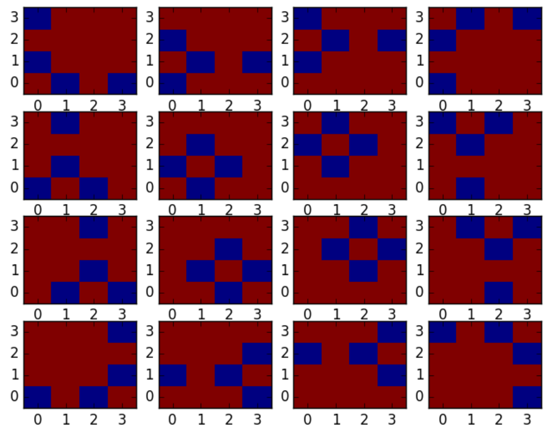
```
import cubic_ising_lattice

spins_per_side=20
isingJ = 1.0

lat = cubic_ising_lattice.CubicIsingLattice(spins_per_side,spin_mag=isingJ)

#lat.flip_spin(0)

# visualize the lattice
import matplotlib.pyplot as plt
fig = plt.figure()
ax  = fig.add_subplot(111)

lat.visualize(ax)
plt.show()
```

The default is to have all spins point up, which does not look very exciting. You can sprinkle some flip_spin(ispin) in between initialization and visualization to make the picture more interesting. Be sure to read the comments at the top of cubic_ising_lattice.py. Pay special attention to how the spins are indexed, also inspect the flip_spin() method. You may notice that flip_spin() wants to return the change in total magnetization due to the spin flip. You will implement mag_change() momentarily.

Currently the lattice of spins are oblivious of each other's existence. Introduce them to their neighbours by constructing a neighbour list i.e. for each spin, a list of linear indices of its neighbours. You may achieve this in the example code by implementing the neighbours(self,ispin) method. This method is only called during the construction of the lattice object and not invoked during MC. Therefore speed does not matter, only correctness does (i.e. do your worst!). To check your code, invoke the python interpreter on the cubic_ising_lattice.py file directly. This will run the unit tests in the "__main__" block and plot all neighbours of all spins on a 4x4 lattice. Ignore errors regarding magnetization methods, we will implement them next.

```
python cubic_ising_lattice.py
```

Correct neighbours should look like this:



Now, implement two functions to calculate and update the total magnetization of the system. In the example code, implement magnetization() and mag_change(). The first method will calculate the total magnetization $NM$ from scratch, where $N$ is the total number of spins and $M$ is the total magnetization per spin. The second method will calculate the change in total magnetization due to a single spin flip. You should check mag_change() against magnetization(), which is easier to get right. It is expected that you will calculate $M^2$ in some other parts of the code, because it is easier for the lattice to keep track of $M$ rather than $M^2$. Run the unit tests in cubic_ising_lattice.py again and make sure they all pass.

Now that we have a working lattice, head over to ising_hamiltonian.py and implement the compute_energy() and compute_spin_energy() methods. Again, the first method should compute the total energy of the lattice of spins from scratch, while the second method should compute the energy change due to a single spin flip. Test the second method against the first method. Take advantage of the neighbour list you constructed (nb_list()). You may also want to use linear_idx() and multi_idx() to figure out where the spins are. Note: lattice.spins() only holds the states of the spins (up or down), while their magnitudes are stored in lattice.spin_mag(). When you are done, try pass the unit tests in ising hamiltonian.py with

```
python ising_hamiltonian.py
```

## Conventional Metropolis

The first method of simulating the Ising model will be conventional Metropolis. Recall the basics steps of a Monte Carlo algorithm:

1. Choose a random spin to flip. i.e. propose a move with probability T(x→x')
2. Calculate the change in energy.
3. Accept/reject with an appropriate A(x→x') to sample the canonical ensemble.

1. Write down the probability you are going to change from your current lattice configuration x, to a new configuration x' with one spin flipped i.e T(x→x') .
2. Write down the probability you would make the "old" move given that you started in the "new" move i.e. T(x'→x)
3. Write down the exp(-δV) where δV = V_new - V_old.
4. Finally write down A(x→x')

Now, write the main Monte Carlo loops. Remember, you need to keep a trace $M^2$. A bad way of doing this would be to compute $M^2$ from scratch each time. A good way of doing this would be to update $M^2$ each time you flip a spin.

You should now compute $M^2$ and its autocorrelation for our three temperatures!

## Heat Bath Algorithm

We are now going to modify your code to work in a more refined fashion. Remember our previous algorithm was:

1. Choose a random spin to flip.
2. Calculate the change in energy.
3. Accept or reject.

Our new algorithm (which is very similar) will be

1. Choose a random spin to change.
2. Sample a new spin for this lattice location using the heat-bath technique.
3. Accept or reject.

> This algorithm does not care what the current spin is.

Again:
1. Write down the probability to make the "new" move T(x→x') .
2. Write down the reverse probability T(x'→x)
3. Write down the exp(-δV) where δV = V_new - V_old.
4. Finally write down A(x→x')

Run your simulation and report the same information as conventional Metropolis.
Is the autocorrelation time larger or smaller? Why is this?

(this paragraph for your edification only) Of course, one might be interested in getting even a better speedup. One natural way of doing this is instead of flipping one spin at once, we could flip a whole block of spins at once. Imagine we take a 3x3 block of spins. Then we can use the heat-bath technique and write down all the probabilities for all the different 2^9 spin configurations and select one with probability proportional to their likelihood.

## Cluster moves

In this problem you will write a cluster move that flips a large collection of spins at once. To begin with, we are going to write a function to build clusters

```
def build_cluster(lattice):
    #returns list of cluster
# end def build_cluster
```

The steps to do this will be the following:
To begin with, start your list of "spins you are checking" with a randomly selected spin.

Then,

1. Choose a spin off the list of spins you are checking.
2. For each neighbor which has the same spin and is not already in the cluster (hint: if you don't want your code to run sloth-like, find a fast way to check that a spin does not belong to a cluster already):
   a. add it to the cluster with probability p=1-exp(-K), find the form of K such that your acceptance rate for the cluster move is always 1.
   b. add it to the list of things whose neighbors you will check
3. Repeat until there are no new sites to check.

Now, we are going to write a function

```
def flip_spins(lattice,cluster):
    # returns how much the magnetization has changed.
# end def flip_spins
```

that takes your cluster and flips all the spins in this cluster. It should also return how much the magnetization has changed due to your spin flipping.

Now, we need to write the main loop that does the Monte Carlo. In this loop you want to

1. build a cluster
2. flip the cluster
3. update the current magnetization

Again:
1. Write down the probability to make the "new" move $T(x \rightarrow x')$ .
2. Write down the reverse probability $T(x' \rightarrow x)$
3. Write down the $\exp(-\delta V)$ where $\delta V = V\_new - V\_old$.
4. Finally write down $A(x \rightarrow x')$, this should be 1, always.

Again, measure the magnetization and autocorrelation time for our three temperatures.

## Dénouement

Compare your results from the three algorithms. Do they all give the same answer? (If not, shouldn't you worry?) Which algorithm has the least autocorrelation time for 1000 sweeps?

## Worm algorithm (Optional)

You may wish to read the following section for your edification.
However do NOT work on this section unless your project is almost done!

In this problem, we will again use a different technique for simulating the ising model. In the first two methods, we used the natural representation of the Ising model (i.e. spins on a lattice that were either up or down). Unfortunately, as we learned in the lecture (and you have been able to see from this homework) this representation either requires that you:

1. use local moves but deal with critical slowing down issues or
2. come up with sophisticated global moves.

In this problem, we will work in a different representation of the ising model. This representation will have the advantage that we will be able to use local moves and avoid the critical slowing down issues.

This representation will also involve a lattice. This time, though, instead of spins on lattice sites, their will be bonds that exist between nearest neighbor lattice sites and these bonds will either be "there" or "not there."

We know that monte carlo performs integrals of the form

$$\frac{1}{S}\int f(x)O(x)$$

where

$$S = \int f(x).$$

In the previous sections, to get the magnetization we were calculating

$$\frac{1}{Z}\int \exp\left(-\beta J\sum_{\langle i,j\rangle} s_i s_j\right)\left(\sum_i s_i\right)^2$$

where

$$Z = \int \exp\left(-\beta J\sum_{\langle i,j\rangle} s_i s_j\right)$$

In this section, we will do Monte Carlo and calculate a term proportional to the magnetization squared by doing the integral

$$A = \frac{1}{S}\int (\text{Prob of Worm Configs})(\text{Magnetization of worm Configs})$$

Unfortunately,

$$\int (\text{Prob of Worm Configs})$$

(i.e. S) does not equal Z. So we will also need to simultaneously calculate Z by calculating

$$B = \frac{1}{S}\int (\text{Prob of Worm Configs})(\text{Partition function contribution of worm Configs})$$

Then the quantity A/B will end up being M^2/Z as desired.

To do this, we will start with a new code. In this code, our data structure will be a 3d array. The first two dimensions of this array will represent the lattice sites of our 2d ising model. The third dimension of our array will then represent whether the bonds of our four potential nearest neighbors are respectively (a) **on** or (b) **off**. In other words, "myLattice[5,2,1]" will be referencing lattice site at position 5 × 2 and will indicate whether it's 1st (i.e. top) neighbor is on or off. (our convention will be to order the neighbors so they go left, top, right, bottom).

Also, we need to keep track of a head and a tail in our system.
Therefore, we will have a variable head and tail that will both be an array of 2 numbers indicating where we are.

We will also need to store our value of "Z" and our values of "MSquaredTimesZ". Each configuration that has only closed loops will contribute to "Z" (so every time we see one of these configurations we increment "Z"). Each configuration that has a worm and closed loops (which is all of the one's we are simulating) will contribute to "M2TimesZ" (so every time we see one of these configurations (i.e. every time) we increment "M2").

To begin with this assignment, let's write a function

```
def FlipNeighbor(myLattice,hd):
    # pick one of the neighbors (call it a) of the head
    # if the bond between hd and a is on, then turn it off
    # if the bond between hd and a is off, then with probability tanh(beta*J)
turn it on.
    # returns newHead
```

Notice that in the way we are representing the data, there are two spots you have to change if you are changing whether a bond is on or off.

Again write down the probability you are going to make your "new" move (i.e. T(old→new) ).

Also, write down the probability you would make the "old" move given that you started in the "new" move (i.e. T(new→old) )
Write down the exp(-δV) where δV = V_new - V_old.
Write down p_accept.

Now we want to write our Monte Carlo code. Remember our algorithm should be as follows:

1. Call your flipNeighbor function
2. if the head and tail are at the same spot,
3. increase "Z" by 1.
4. move the head and tail to (the same) new random spot
5. increment "M2TimesZ" by 1

At the end, the total magnetization squared will be "M2TimesZ/Z".

Use this algorithm to calculate the magnetization and autocorrelation time at the same three temperatures.

To calculate the magnetization you should calculate ⟨"M2TimesZ"⟩/⟨"Z"⟩. To calculate the autocorrelation time you should output after every sweep the value of ⟨"M2"⟩/⟨"Z"⟩ for that sweep and calculate the autocorrelation time of this quantity.

No labels