PHY407 Computational Lab 11
Monte Carlo Simulations

Computational and Physical Background

Monte Carlo Simulation for Statistical Mechanics

- Section 10.3.1 of the text discusses Monte Carlo simulations for statistical mechanics.
- We want to evaluate an expectation (or average) value for some quantity for a system in thermal equilibrium at temperature T. The system will pass through a succession of states such that at any particular moment, the probability of it occupying state i with energy E_i is given by the Boltzmann formula:

$$P(E_i) = \frac{\exp(-\beta E_i)}{Z}, \qquad Z = \sum_i \exp(-\beta E_i) \tag{1}$$

  where $\beta = 1/k_B T$, k_B is Boltzmann's constant, Z is called the 'partition function' and the sum is over all possible states at that temperature.

- The expectation value of a quantity X that takes the value X_i in the ith state is:

$$< X >= \sum_i X_i P(E_i) \tag{2}$$

  Essentially, this is a weighted average of the property X of each state over the sum of states.

- Normally we can't calculate the sum over all states because there are way too many. Instead, we use a Monte Carlo approach where we randomly sample terms in the sum to approximate it. So we approximate equation (2) with:

$$< X >\approx \frac{\sum_{k=1}^{N} X_i P(E_k)}{\sum_{k=1}^{N} P(E_k)} \tag{3}$$

  The denominator is there to ensure the average is normalized correctly.

- Importance sampling (which you saw last week for evaluating an integral) is used to evaluate the sum so as to choose more states that contribute significantly to the sum (since those with $E_i >> k_B T$ contribute very little to the sum). Using a weighted average, the sum can be written:

$$< X >\approx \left\langle \frac{X_i P(E_i)}{w_i} \right\rangle_w \sum_i w_i \tag{4}$$

  where the angular brackets on the right denote a weighted sum. This is identical to equation 10.38 that we used for the integral last week if you set $f(x) = X_i P(E_i)$ and do a sum instead of an integral.

- We evaluate this sum by selecting a set of N sample states randomly but non-uniformly, such that the probability of choosing state i is

$$p_i \approx \frac{w_i}{\sum_j w_j} \tag{5}$$

- Combining this with equation (4) results in:

$$< X > \approx \frac{1}{N} \sum_{k=1}^{N} \frac{X_k P(E_k)}{w_k} \sum_i w_i \tag{6}$$

  Note that the first sum is over only those states k that we sample, but the second is over all states i. This second sum is usually evaluated analytically.

- The goal is to choose the weights w_i so that most of the samples are in the region where $P(E_i)$ is big and such that we can analytically do the second sum. We choose $w_i = P(E_i)$ in which case $\sum_i w_i = \sum_i P(E_i) = 1$ and we are left with:

$$< X > \approx \frac{1}{N} \sum_{k=1}^{N} X_k \tag{7}$$

  So basically, we just choose N states in proportion to their Boltzmann probabilities and take the average of the quantity X over all of them.

The Markov Chain Method

- The problem we still have to deal with is how to calculate $P(E_i)$ from equation (1) (since we want to choose states with this probability). Notice P needs the partition function which is a sum over all states (and again, if we could do that, we wouldn't need a Monte Carlo simulation). We can find $P(E_i)$ without the partition function using the Markov Chain Method (see text for more details).

- The Metropolis algorithm allows us to choose values of the transition probabilities T_{ij}. The total Markov chain Monte Carlo simulation involves the following steps:

  1. Choose a random starting state.
  2. Choose a move uniformly at random from an allowed set of moves, such as changing a single molecule to a new state.
  3. Calculate the value of the acceptance probability P_a:

  $$P_a = \begin{cases} 1 & \text{if } E_j \leq E_i \\ \exp(-\beta(E_j - E_i)) & \text{if } E_j > E_i \end{cases} \tag{8}$$

  4. With probability P_a, accept the move, meaning the state of the system changes to the new state; otherwise reject it, meaning the system stays in its current state.
  5. Measure the value of the quantity of interest X in the current state and add it to a running sum of such measurements.
  6. Repeat from step 2.

Simulated Annealing

- This is a Monte Carlo method for finding GLOBAL maxima/minima of functions. Remember that in Chapter 6 we studied various methods for finding local maxima/minima, but sometimes we need the global value. Simulated Annealing can do this.

- For a physical system in equilibrium at temperature T, the probability that at any moment the system is in a state i is given by the Boltzmann probability (equation 1 above). Assume the system has a single unique ground state and choose the energy scale so that $E_i = 0$ in the ground state and $E_i > 0$ for all other states. If we cool down the system to $T = 0$, $\beta \to \infty \Rightarrow \exp(-\beta E_i) \to 0$ except for the ground state where $\exp(-\beta E_i) = 1$. Thus, in this limit, $Z = 1$ and
$$P_a = \begin{cases} 1 & \text{for } E_i = 0, \\ 0 & \text{for } E_i > 0 \end{cases} \tag{9}$$
This is just a way of saying that at absolute 0, the system will definitely be in the ground state.

- This suggests a computational strategy for finding the ground state: simulate the system at temperature T, using the Markov chain Monte Carlo method, then lower the temperature to 0 and the system should find the ground state. This approach can be used to find the minimum of ANY function f by treating the independent variables as defining a 'state' of the system and f as being the energy of that system.

- There is one issue that needs to be dealt with: If the system finds itself in a local minimum of the energy, then all proposed Monte Carlo moves will be to states with higher energy and if we then set $T = 0$ the acceptance probability becomes 0 for every move so the system will never escape the local minimum. To get around this, we need to cool the system slowly by gradually lowering the temperature rather than setting it directly to 0.

- To implement simulated annealing: Perform a Monte Carlo simulation of the system and slowly lower the temperature until the state stops changing. The final state that the system comes to rest in is our estimate of the global minimum.

- For efficiency, pick the initial temperature such that $\beta(E_j - E_i) << 1$ meaning that most moves will be accepted and the state of the system will be rapidly randomized no matter what the starting state. Then choose a cooling rate (typically exponential):
$$T = T_0 \exp(-t/\tau) \tag{10}$$
where T_0 is the initial temperature and τ is a time constant.

- Some trial and error is needed in picking τ. The larger the value, the better the results (because of slower cooling) but also the longer it takes the system to reach the ground state.

Lab Instructions

Recommended reading: Sections 10.3-10.4 of Newman

For grading purposes, you only need to hand in solutions to the following parts:

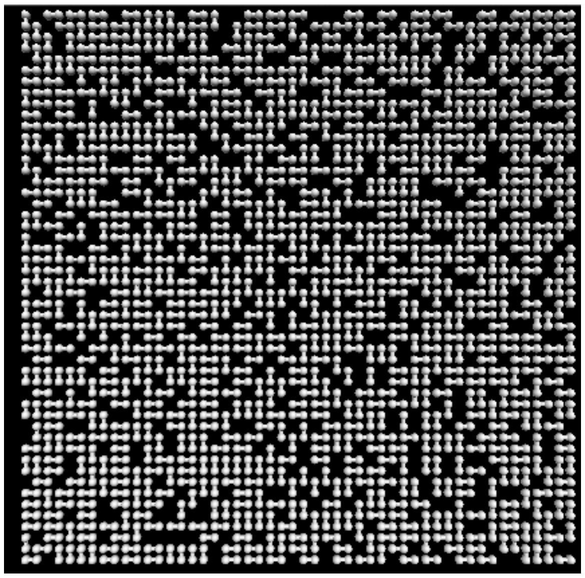
- Q1: Hand in your code and figures for the magnetization and energy.
- Q2: Hand in your code and a snapshot of your final magnetization visualizations at T=1.0, T=2.0, T=3.0, and a brief discussion to answer part (d).
- Q3: Hand in your code and a snapshot of your final dimer coverage visualizations for tau=1e3, 1e4 and 1e5.

The Ising Model for Magnetism

1. Do Exercise 10.9 parts (a)-(c). Start from the 1D version of the program that we developed in the lecture. Hint: import "sum" from numpy to sum over all elements in an array (even 2D arrays, unlike the regular sum function which you will end up using if you don't import it from numpy).
2. Do Exercise 10.9 parts (d) and (e).

The Dimer Covering Problem

3. Do Exercise 10.11 parts (a) and (b). See the appendix for info on how to make visualizations of the dimers like in the textbook and also on how to use Dictionaries to store the dimers. Here is an example of what my dimers looked like:



Appendix A: Visualization of Dimers

- Here is how I recommend making a visualization of the molecules and dimers for question 3 using vpython.
- Make an array of spheres to represent the molecules on each grid point using:

  ```
  site = empty([L,L],sphere)
  for i in range(L):
      for j in range(L):
          site[i,j] = sphere(pos=[i,j],radius=2*R)
          site[i,j].visible = False
  ```

  This last line makes all the spheres invisible for now.
- While running your metropolis algorithm, if a dimer gets added to the system, change those specific spheres such that they are visible with a command like:

  ```
  site[i1,j1].visible = True
  site[i2,j2].visible = True
  ```

  where i1,j1 and i2,j2 are grid points next to each other that hold the molecules that make up the dimer.
- Anytime you remove a dimer, set the visible property of both spheres to false again.
- You will also want to represent the connection between the molecules making up the dimer. I recommend making these using the cylinder object. For example, a cylinder joining the 2 molecules above can be made using the command:

  ```
  dimer = cylinder(pos=[i1,j1], axis = [i2-i1,j2-j1], radius = R)
  ```
- Again, you can make the cylinders visible or invisible by setting their visible property to true or false.
- See appendix B for a method of storing the location of these dimers in a dictionary.

Appendix B: Python Dictionaries

- Dictionary is a data type in python. Dictionaries can be thought of as a set of *key:value* pairs.
- The main operations on a dictionary are storing a value with some key and extracting the value given the key.
- See http://www.tutorialspoint.com/python/python_dictionary.htm for further info and examples.

- For question 3, you can make a dictionary to store dimer locations. Initialize the dimer dictionary using:

  ```
  dimer = dict()
  ```

  Then every time you want to create a dimer, make the 'key' an index and the 'value' a cylinder to join the molecules. For example, here is how you would add a new dimer:

  ```
  d=0
  if (some condition that means you want to add a dimer):
       d +=1
       dimer[d] = cylinder(pos=[i1,j1], axis = [i2-i1,j2-j1], radius = R)
  ```

  The above commands will store a cylinder with key 'd' in the dictionary.

- You might also need to delete dimers in your dictionary. You can do this using:

  ```
  del dimer[ind]
  ```

  where 'ind' is the specific key for the dimer you want to delete.