Computational Lab 11 : Monte Carlo integration II

Due November 29th, 2019 (@ 17:00)

- Read this document and do its suggested readings to help with the pre-labs and labs.
- This lab's topics revolve around computing solutions using random numbers and Monte Carlo techniques.
- Ask questions if you don't understand something in this background material: maybe we can explain things better, or maybe there are typos.
- Whether or not you are asked to hand in pseudocode, you **need** to strategize and pseudocode **before** you start coding. Writing code should be your last step, not your first step.
- Test your code as you go, **not** when it is finished. The easiest way to test code is with `print('')` statements. Print out values that you set or calculate to make sure they are what you think they are.
- Practice modularity. It is the concept of breaking up your code into pieces that as independent as possible form each other. That way, if anything goes wrong, you can test each piece independently.

Physics Background

Ising model Most of this blurb is taken from Wikipedia (https://en.wikipedia.org/wiki/Ising_model).
The Ising model is a mathematical model of ferromagnetism in statistical mechanics. The model consists of discrete variables that represent magnetic dipole moments of atomic spins that can be in one of two states (+1 or -1). Each spin interacts with its neighbours on a lattice. The model allows the identification of phase transitions, as a simplified model of reality. The two–dimensional square-lattice Ising model is one of the simplest statistical models to show a phase transition.
A popular case of the Ising model is the translation-invariant ferromagnetic zero-field model on a d—dimensional lattice, which we study in this question. In one dimension, the solution admits no phase transition. Namely, for any positive $\beta = \frac{1}{k_B T}$, the system is disordered, which can be crudely described as the chain of spins having no particular pattern.
In the $d \geq 2$ case, the Ising model undergoes a phase transition between an ordered and a disordered phase. Namely, the system is disordered for small β (high temperature), whereas for large β (low temperature), the system exhibits ferromagnetic order.

Protein folding: The material provided here is based on Section 12.1 of Computational Physics by Giordano and Nakanishi. It is a very brief introduction to the most fundamental concepts of protein folding, and you should refer to Giordano and Nakanishi if you would like a (slightly) more in depth discussion.
Proteins are chains or sequences of amino acids, which themselves are smaller molecules typically containing on the order of 10s of atoms. There are 20 different amino acids found in nature, but these can be combined in an enormous number of different ways to form proteins, which typically consist of something between 50 to several thousand amino acids. Furthermore, given the particular sequence of amino acids that make up a protein, there are many different ways in which the amino acids can be oriented with respect to each other (in terms of the angles of the bonds between each pair of amino acids) and each of these orientations will lead to a different energy state for the protein. In practice, the proteins will "fold" themselves into their lowest energy state. In this lab, we will use Monte Carlo simulations to explore the physics of this behaviour. Note that the amino acids and proteins are sometimes referred to as monomers and polymers, respectively.

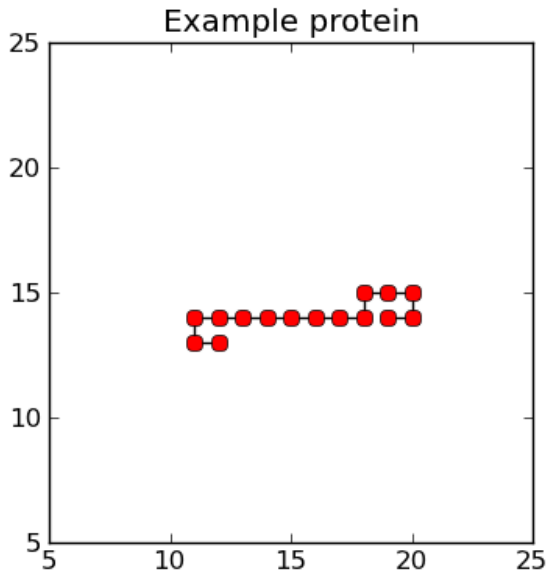

Figure 1: An example polymer of length N=15 whose energy is given by $E = 3\epsilon$.

The order of monomers that make up the chain is referred to as the primary structure of the protein. The particular orientation or folding pattern that the protein takes is known as its tertiary structure. To capture the essential physics of protein folding in the simplest possible way, we will assume that the amino acids are arranged on a rectangular 2D lattice. The protein will have N individual monomers, and we will assume there is only one type of amino acid (or monomer). An example of such a simplified protein is given in Figure 1. The energy of the particular orientation of a protein sequence is defined by the adjacent monomers that are not directly connected to each other. Each not directly connected pair of adjacent monomers is assumed to have an interaction energy of ϵ, which we will always take to be negative here. Thus, the polymer in Figure 1 has an energy of $E = 3\epsilon$ since there are three pairs of adjacent monomers that are not directly connected.
Given this setup, the Monte Carlo procedure goes as follows. A random monomer on the chain is chosen. Then, one of the four possible diagonal nearest neighbour positions next to that monomer is chosen. It is then checked if it would be possible to move the given monomer to that position without "stretching" the chain (i.e. ensuring that the same distance between connected monomers is always maintained). If a given move is possible, the difference in energy, ΔE, between the original and new states is calculated. If the difference is negative and the new state would be at a lower energy, the move is made. If the difference is positive, then the move is made only if the Boltzmann factor $\exp\left(-\frac{\Delta E}{T}\right)$ is greater than a random number between 0 and 1. Many such moves are made and then averages over many steps can be computed to find the typical energy of a given protein at some temperature. The code provided with this lab, `Lab11-Qprotein-start.py`, implements the Monte Carlo algorithm described above. The initial condition for the protein is taken to be a flat (unfolded) horizontal line. The most important parameters in the code that you can adjust are N, the length of the chain, T, the temperature, ϵ, the interaction energy, and n, the number of Monte Carlo steps that the simulation will take.

Computational Background

Monte Carlo Simulation and the Markov Chain/Metropolis method for Statistical Mechanics (general information relevant for the lab as a whole): 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:

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

$$Z = \sum_i \exp\left(-\beta E_i\right), \tag{2}$$

where $\beta = \frac{1}{k_B T}$, 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

$$\langle X \rangle = \sum_i X_i P(E_i) \tag{3}$$

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 3 with

$$\langle X \rangle \approx \frac{\sum_{k=1}^{N} X_i P(E_k)}{\sum_{k=1}^{N} P(E_k)}. \tag{4}$$

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 $E_i \gg k_B T$ contribute very little to the sum). Using a weighted average, the sum can be written

$$\langle X \rangle \approx \langle \frac{X_i P(E_i)}{w_i} \rangle_w \sum_i w_i, \tag{5}$$

where the angular brackets on the right denote a weighted sum. This is identical to equation 10.38 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

$$p_i \approx \frac{w_i}{\sum_j w_j}. \tag{6}$$

Combining this with equation 5 results in:

$$\langle X \rangle \approx \frac{1}{N} \sum_{k=1}^{N} \frac{X_k P(E_k)}{E_k} \sum_i w_i. \tag{7}$$

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:

$$\langle X \rangle \approx \frac{1}{N} \sum_{k=1}^{N} X_k. \tag{8}$$

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 problem we still have to deal with is how to calculate $P(E_i)$ from equation 3 (since we want to choose states with this probability). Notice P needs the partition function which is a sum over all states (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:

$$P_a = \begin{cases} 1, & E_j \leq E_i \\ \exp\left(-\beta(E_j - E_i)\right), & E_j > E_i \end{cases} \tag{9}$$

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 Chapter 6 of Newman discusses 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

$$P(E_i) = \frac{\exp\left(-\beta E_i\right)}{Z}, \tag{10}$$

$$Z = \sum_i \exp\left(-\beta E_i\right), \tag{11}$$

and $\beta = \frac{1}{k_B T}$. 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. As the system cools down, $T \to 0$, $\beta \to \infty$,$\exp(-\beta E_i) \to 0$ except for the ground state where $\exp(-\beta E_i) = 1$. Thus in this limit $Z = 1$ and

$$P_a = \begin{cases} 0, & E_i = 0 \\ 1, & E_i > 0 \end{cases} \tag{12}$$

This is just a way of saying that at absolute zero, 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 $\beta(E_j - E_i) \ll 1$ 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:

$$T = T_0 \exp\left(-\frac{t}{\tau}\right) \tag{13}$$

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.

Monte Carlo Simulation and the Markov Chain/Metropolis method for Statistical Mechanics See section 10.3.1 of the textbook.

One-dimensional Ising model Exercise 10.9 on page 487-489 discusses the Ising model of magnetization, well known in the field of statistical mechanics. The following outlines the solution for the 1D problem.

1. Consider N dipoles each with a spin state s_i that can equal either +1 or -1. Create an array to hold the spin state for all N dipoles. Initialize the array so that all spins are initially set to +1. Write a function to calculate the total energy of the system which is given by:

$$E = -J \sum_{\langle ij \rangle} s_i s_j \tag{14}$$

   where J is an exchange energy constant which you can set = 1.0 and $\langle i\, j \rangle$ means that the sum is over pairs i, j that are adjacent on the lattice. For example, if $N = 5$, then:

$$E = -J(s_0 s_1 + s_1 s_2 + s_2 s_3 + s_3 s_4) \tag{15}$$

   Notice that you don't double-count pairs (i.e. if you have $s_1 s_2$ then you don't need $s_2 s_1$). You will want to use array math for the sum rather than a for loop over all the possible states, so think about how to implement that (the functions `dot` or `sum` from `numpy` will work well for this).

   Also write a function to calculate the total magnetization of the system which is given by

$$M = \sum_{i=1}^{N} s_i \tag{16}$$

   Set $N = 100$ and call your function. Print out the total energy and magnetization.

2. Write a function that implements the Metropolis algorithm to flip a spin state randomly, calculate the new total energy and decide whether to accept the flip. Work in units such that k_B and T are both unity. Here is what this function needs to do:

   - Randomly select an element of the spin array and flip its spin.
   - Call your energy function from Q1 to calculate the energy of this new spin state (E_{new}).
   - Calculate the Boltzmann factor for this change in energy:

$$p = \exp[-\frac{(E_{\text{new}} - E_{\text{old}})}{k_B T}] \tag{17}$$

     where E_{old} is the energy before the flip.
   - Keep this new state if one of the following conditions is met:
     (a) if $E_{\text{new}} - E_{\text{old}} \leq 0$
     (b) if $E_{\text{new}} - E_{\text{old}} > 0$ and $p >$ random().
   - If neither of these is true, keep the old state.

   Run your program to calculate the new energy and new magnetization for one flip. Print out the values.

3. Now implement a loop in your program to run your Metropolis algorithm for 10000 flips, calculating the energy and magnetization after each flip. Plot the energy and magnetization as a function of the number of flips. You have now implemented the Markov Chain process.

Questions

1. **[30%] Examples of simulated annealing optimization**

   (a) Example 10.4 in the book shows you how to implement simulated annealing optimization in the travelling salesman problem. In this exercise,you will test the sensitivity of the method to the cooling schedule time constant τ. To do this carefully, you need to pick a particular single set of points and find optimal paths for that set of points. To pick the same set of points each time, you should `seed` the random number generator with a known value, for example:

   ```
   from random import seed
   seed(10)
   ...
   ```

   before the first set of calls to `random`. If you don't like the set of points generated, just change the seed number.

   Now the program will run exactly the same way each time on your computer. To get the annealing procedure to take a different optimization path, you can change the seed number by inserting a second `seed(n)` call with a different `n` before the `while` loop.

   With this background, do the following. First, choose a set of points that you like with an initial seed. Then carry out simulated annealing optimization on this set of points a few times (by varying the second seed before the while loop each time), and tell us how much the final value of the distance `D` tends to vary as a result of different paths taken. Next, vary the scheduling time constant τ by making it shorter and then longer than the default value. Vary the seed each time to increase the number of paths taken. What is the impact on D as you allow the system to cool more quickly or more slowly?

   SUBMIT A PLOT SHOWING A FEW DIFFERENT PATHS TAKEN, AND YOUR WRITTEN ANSWERS

   (b) (see Newman Exercise 10.10) Consider the function

   $$f(x,y) = x^2 - \cos 4\pi x + (y-1)^2. \tag{18}$$

   The profile of f at $y = 1$, can be found plotted on p.497. The global minimum of this function is at $(x,y) = (0,1)$. Write a program to confirm this fact using simulated annealing starting at, say, $(x,y) = (2,2)$, with Monte Carlo moves of the form $(x,y) \to (x+\delta x, y+\delta y)$ where δx and δy are random numbers drawn from a Gaussian distribution with mean zero and standard deviation one. (See Section 10.1.6 for a reminder of how to generate Gaussian random numbers.) Use an exponential cooling schedule and adjust the start and end temperatures, as well as the exponential constant, until you find values that give good answers in reasonable time. Have your program make a plot of the values of (x,y) as a function of time during the run and have it print out the final value of (x,y) at the end. You will find the plot easier to interpret if you make it using dots rather than lines.

   Now adapt your program to find the minimum of the more complicated function

   $$f(x,y) = \cos x + \cos\sqrt{2x} + \cos\sqrt{3x} + (y-1)^2 \tag{19}$$

   in the range $0 < x < 50$, $-20 < y < 20$ (This means you should reject (x,y) values outside these ranges.) The correct answer is around $x \approx 16$ and $y = 1$, but there are competing minima for $y = 1$ and $x \approx 2$ and $x \approx 42$, so if the program settles on these other solutions it is not necessarily wrong.

   SUBMIT YOUR CODE, PLOTS, AND WRITTEN ANSWERS

2. **[35%] Ising model**

   Do Exercise 10.9 in the book.

   SUBMIT CODE, PLOTS AND EXPLANATORY NOTES.

   Notes:

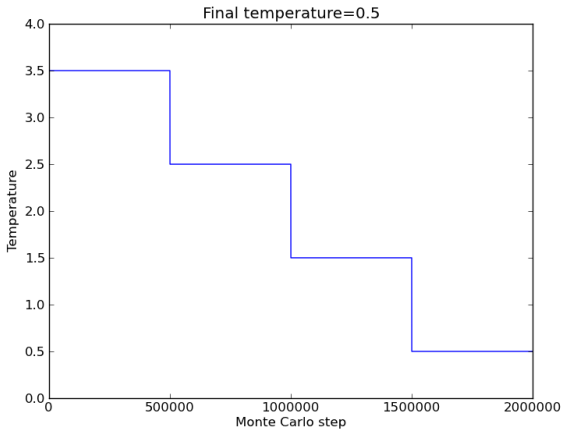

Figure 2: .

- *In part (e), do not use the* `visual` *package. A simple*

  ```
  matplotlib.pyplot.imshow(s)
  ```

  assuming `s` *is the array containing the spins, will suffice. Also, only run 100,000 steps and do not show an animation every step, or it will take too long.*

3. **[35%]** Protein folding

   (a) Run the script `Lab11-Qprotein-start.py` using the default parameters, i.e. $N = 30$, $T = 1.5$, $\epsilon = -5$, $n = 100,000$. Note this make take up to 20–30 seconds to run, depending on your computer. The script will create two figures, one which shows the final structure of the protein, and the other which shows the energy as a function of the Monte Carlo step. Briefly describe what you see in the energy plot. Now change the temperature to $T = 0.5$ and $T = 5$ and describe what you see for each of those cases.

   SUBMIT THE PLOT AND WRITTEN ANSWERS

   (b) Now, just for the $T = 0.5$ and $T = 1.5$ cases, run the simulation for $n = 1,000,000$ steps. This may take up to a couple minutes for each simulation. (Feel free to look for ways to speed up the code.) In which case is the typical energy of the protein (over say, the second half of the simulation) lower? Does this agree with what you would expect? Looking at the final structure of the protein, and knowing that the initial condition for the protein is just a straight horizontal line, what can you say about what is happening in the $T = 0.5$ case?

   SUBMIT THE PLOT AND WRITTEN ANSWERS. SUBMIT THE CODE IF YOU HAVE MODIFIED IT.

   (c) As you should have found in the previous parts, when using a low temperature like $T = 0.5$, the protein does not change substantially from its initial conditions. To get around this, we can start with a higher temperature and steadily decrease the temperature over the course of the simulation in order to more quickly reach equilibrium for the given final temperature. Implement this as follows. Create an array of temperatures of length n. Then, given the final temperature T_f, have the temperature decrease by 1 over T_{steps} until it reaches T_f . Explicitly, this can be done as follows:

```
import numpy as np
T_f=0.5
T_steps=4
T_i=T_f+T_steps-1
T_array=np.zeros(n)
for step in np.range(T_steps):
        T_array[step*n/T_steps:(step+1)*n/T_steps]=
                                    (T_i-T_f)*(1-float(step)/(T_steps-1))+T_f
```

Now, for the ith Monte Carlo step, use the ith temperature from T_{array}. See Figure 2 for a plot of the temperature as a function of Monte Carlo step for $T_f = 0.5$, $T_{\text{steps}} = 4$ and $n = 2{,}000{,}000$.

Implement this change to the code, and run it with $T_f = 0.5$, $T_{\text{steps}} = 4$ and $n = 2{,}000{,}000$. What is the approximate energy the protein has over the last quarter of the simulation (i.e. when $T = 0.5$)? How does this compare to the $T = 0.5$ simulation from part b where there was no simulated annealing?

SUBMIT ANY CHANGED CODE AND WRITTEN ANSWERS TO THE QUESTIONS.

(d) For the last part of this question, we will more quantitatively explore the temperature dependence of the energy of a particular protein. Simulate the protein folding starting at a temperature of $T = 10$, stepping by $\delta T = 0.5$ until you reach $T = 0.5$. At each temperature simulate 500,000 steps and calculate the mean energy and standard deviation at that temperature.[1] At the end of the simulation you should have values of temperature, mean energy, and standard deviation in energy. Make a plot of temperature versus energy. What do you find? Do you think there is evidence for a phase transition (i.e. a sharp jump in energies over a small temperature range)?

SUBMIT CODE, PLOTS, AND WRITTEN ANSWERS TO THE QUESTIONS.

Question:	1	2	3	Total
Points:	30	35	35	100

[1] Running the simulation for this many steps will take a while (up to 30 minutes). You should test the code with a smaller n. There are also likely ways to speed up the code.