Lab assignment #11: Monte Carlo Simulations

Instructor: Nicolas Grisouard (nicolas.grisouard@utoronto.ca)

Due Monday, December 7th 2020, 5 pm

First, try to sign up for room 1. If it is full, sign up for room 2 (and ask your partner to join you there if they are in room 1).

Office hours will take place in room 1 on Thursday.

Room 1 Mohamed Shaaban (Marker, m.shaaban@mail.utoronto.ca),
URL: `https://gather.town/xg1R1WVWH3wfgdCY/phy407-ms`
PWD: `phy407-2020-MS`

Room 2 Ahmed Rayyan (a.rayyan@mail.utoronto.ca),
URL: `https://gather.town/mIAKeWKElOnF4uL3/PHY407-AR`
PWD: `phy407ar`

General Advice

- **Work with a partner!**
- Read this document and do its suggested readings to help with the pre-labs and labs.
- Ask questions if you don't understand something in this background material: maybe we can explain things better, or maybe there are typos.
- Specific instructions regarding what to hand out are written for each question in the form:

  THIS IS WHAT IS REQUIRED IN THE QUESTION.

  Not all questions require a submission: some are only here to help you. When we do though, we are looking for "C^3" solutions, i.e., solutions that are **C**omplete, **C**lear and **C**oncise.
- An example of **C**larity: make sure to label all of your plot axes and include legends if you have more than one curve on a plot. Use fonts that are large enough. For example, when integrated into your report, the font size on your plots should visually be similar to, or larger than, the font size of the text in your report.
- 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 are as independent as possible form each other. That way, if anything goes wrong, you can test each piece independently. One way to practice modularity is to define external functions for repetitive tasks. An external function is a piece of code that looks like this:

```
def MyFunc(argument):
    """A header that  explains the function
    INPUT:
    argument [float] is the angle in rad
    OUTPUT:
    res [float] is twice the argument"""
    res = 2.*argument
    return res
```

Place these functions in a separate file called e.g. `functions_labNN.py`, and call and use them in your answer files with:

```
import functions_labNN as fl  # make sure file is in same folder
ZehValyou = 4.
ZehDubble = fl.MyFunc(ZehValyou)
```

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 (left, right, above, below and, if 3D, front and back; no diagonals). 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. We started the $d = 1$ case, which can be thought of as a linear horizontal lattice where each site only interacts with its left and right neighbour, in lecture. In one dimension, the solution admits no phase transition, namely, for any positive $\beta = 1/k_B T$, the system is disordered, which can be crudely described as the chain of spins having no particular patter. This was considered a failure of the model at the time, since it was well-known that ferromagnetic order happened for low temperatures, that at high temperatures, the system was disordered, and that the transition between the two was a second-order phase transition.[1]

In the $d \geq 2$ case however, the Ising model undergoes a phase transition between an ordered and a disordered phase, which was proven by Onsager in the 40's.

Protein folding: The material provided here is based on Section 12.1 of the textbook "Computational Physics" by Giordano and Nakanishi.

[1] which may be why Ising quit research afterwards.

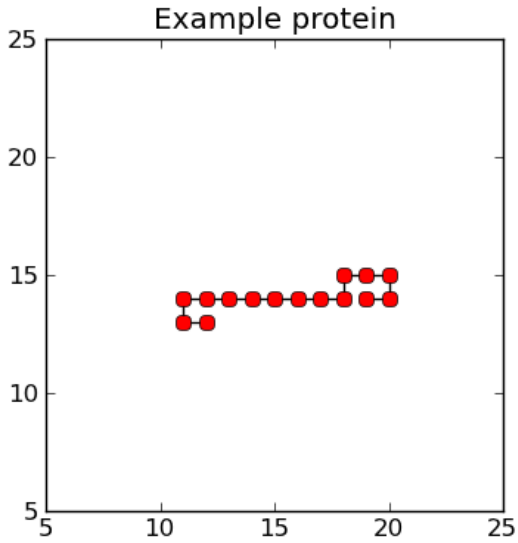

Figure 1: An example polymer of length $N = 15$ whose energy is given by $E = 3\epsilon$.

Proteins are chains or sequences of amino acids, which themselves are smaller molecules typically containing on the order of 10's 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.

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.

Computational background

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. In lecture, we will start (have started) solving the 1D problem, which I reproduce here.

The basic procedure is to reproduce a Boltzmann distribution by "picking" microstates that are more likely, based on said distribution. This amount to performing importance sampling, with the weights simply being the probability of a microstate to be realized. The final formulae look a lot like mean-value Monte Carlo (e.g., the expected value of X is $\langle X\rangle \approx \sum_{i=1}^{N} X_i/N$, with N the number of samples chosen), which hides the fact that the samples are drawn according to a probability density function. This probability density function is actually impossible to calculate because of its normalization factor. Instead, a Markov Chain Monte Carlo procedure "explores" the possible microstates by changing the system, one element at a time, according to transition probabilities that are compatible with, and conducive to, the Boltzmann distribution. In practise, the steps to take are as follows.

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 or −1, randomly or not. Write a function to calculate the total energy of the system which is given by:

$$E = -J \sum_{\langle ij \rangle} s_i s_j \tag{1}$$

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

$$E = -J(s_0 s_1 + s_1 s_2 + s_2 s_3 + s_3 s_4) \tag{2}$$

   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

$$M = \sum_{i=1}^{N} s_i \tag{3}$$

   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:

$$p = \exp[-(E_{new} - E_{old})/k_B T] \tag{4}$$

     where E_{old} is the energy before the flip.

- Keep this new state if one of the following conditions is met:

  (a) `if E_new - E_old <= 0`

  (b) `if E_new-E_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.

Monte Carlo Simulation and the Markov Chain/Metropolis method for Protein folding Given the setup presented in the Physics Background, the procedure goes as follows.

1. Choose a random monomer on the chain.
2. Then, choose one of the four possible diagonal nearest neighbour positions next to that monomer.
3. Check 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).
4. If the move is possible, calculate the difference in energy, ΔE, between the original and new states.
   - If the difference is negative and the new state would be at a lower energy, make the move.
   - If the difference is positive, then make the move only if the Boltzmann factor $\exp\left(-\frac{\Delta E}{T}\right)$ is greater than a random number between 0 and 1.
5. Repeat.

After many moves, you can compute averages over many steps to find the typical energy of a given protein at some temperature.

The code provided with this lab, `L11-protein-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.

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

$$P(E_i) = \frac{\exp\left(-\beta E_i\right)}{Z}, \tag{5}$$

$$Z = \sum_i \exp\left(-\beta E_i\right), \tag{6}$$

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

$$P_a = \begin{cases} 0, & E_i = 0 \\ 1, & E_i > 0 \end{cases} \tag{7}$$

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, such as

$$T = T_0 \exp\left(-\frac{t}{\tau}\right) \tag{8}$$

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.

Questions

1. **[35%] 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) (Adapted from Newman Exercise 10.10, p. 497)

i. Consider the function

$$f(x,y) = x^2 - \cos(4\pi x) + (y-1)^2. \tag{9}$$

The profile of f at $y = 1$, can be found 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

$$(x,y) \to (x + \delta x, y + \delta y) \tag{10}$$

where δx and δy are random numbers drawn from a Gaussian distribution with mean zero and standard deviation one (write your own code to draw from a Gaussian distribution). 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.

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

$$f(x,y) = \cos x + \cos(\sqrt{2}x) + \cos(\sqrt{3}x) + (y-1)^2 \tag{11}$$

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.

Note: See Section 10.1.6 for a reminder of how to generate Gaussian random numbers.

SUBMIT YOUR CODE (FOR EITHER OR BOTH PARTS), PLOTS, AND WRITTEN ANSWERS

2. **[30%] Ising model**

Do Exercise 10.9 in the book. You may use `L11-Ising1D-start.py` to get started.

SUBMIT CODE, PLOTS AND EXPLANATORY NOTES.

Notes:

- *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 `L11-Qprotein-start.py` using the default parameters, i.e. $N = 30$, $T = 1.5$, $\epsilon = -5$, $n = 10^5$. 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 SIGNIFICANTLY (I.E., MORE THAN JUST CHANGED THE PARAMETERS).

(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.

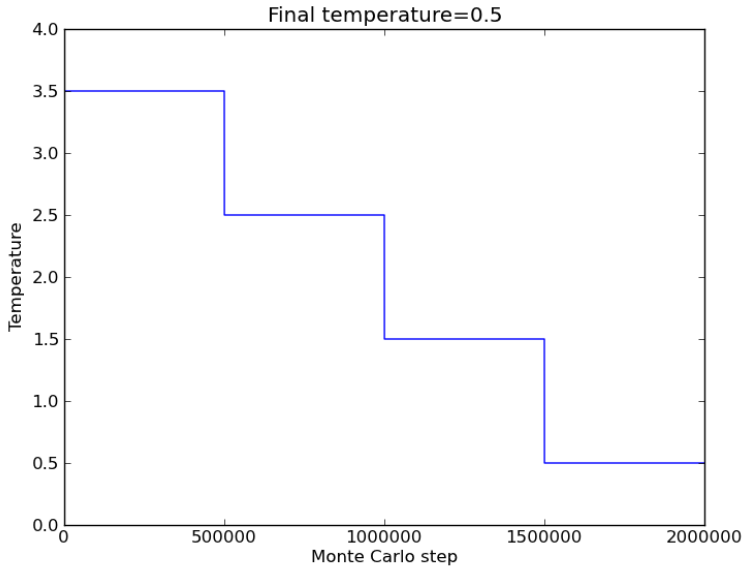

Figure 2: Temperature steps.

```
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 range(T_steps):
    T_array[step*n//T_steps:(step+1)*n//T_steps] = \
        (T_i-T_f)*(1-step/(T_steps-1)) + T_f
```

Now, for the i^{th} Monte Carlo step, use the i^{th} 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 $n = 2 \times 10^6$.

Implement this change to the code, and run it with $T_f = 0.5$, $T_{\text{steps}} = 4$ and $n = 2 \times 10^6$. 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 3b 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. 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 (look into `matplotlib.pyplot.errorbar`). 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)?

Note: 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.

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