PHY324 Computational Project

1 Project Description

This document gives you some theory, and then has some exercises. You must write your own Python code to answer the exercises. The report that you submit should include some text that you write which describes what you have done. It should also include one graph per exercise, except for Exercises 4 and 5 which require three graphs each. Make sure your graphs are labeled well and easy to read! **Your report should not include your Python code!**

This is a paper, not a homework assignment. Do not write "The answer to Exercise 1 is ..." Instead, describe what you are doing. For example, for Exercise 1 you might start with "I estimated the value of π by picking random points inside a square and then checking whether they are also inside a circle which was circumscribed by the original square." Then continue describing what you did in order to get the graph that you need to create. It would be even better if your paper had two parts: part 1 combines Exercises 1 through 4; part 2 combines Exercises 5 and 6.

This project has 6 exercises which should take about 6 hours to finish. If you are clever, you can reuse the code for some of the exercises to finish other exercises rather than starting from scratch with each new exercise. This time estimate does not include the time needed to write your report. If you take more than two hours to do an exercise you should stop and get assistance.

2 Monte Carlo Methods

2.1 What is a Monte Carlo Method?

Sometimes you cannot possibly calculate something exactly in a finite amount of time. In many such cases, the best that you can do is calculate a very tiny fraction of the real problem, hope that the rest of the calculation will be similar to what you did calculate, and then use that calculation as your answer. This happens all the time in thermal physics, where the number of particles can be huge (10^{20} is a small number), and the number of combinations of things might be ridiculous ($2^{10^{20}}$ can be a small number too). No computer can hope to make $2^{10^{20}}$ calculations in the lifetime of the universe, so we have to try something else. In such cases, Monte Carlo methods are often our best choice.

A calculation is called a Monte Carlo method if it uses random numbers to approximately calculate the correct answer. The requirement is that if you use more random numbers to do the calculation then your result should be statistically more accurate. Monte Carlo methods can only asymptotically approach the correct answer. Note that there are many different type of Monte Carlo methods. We will only look at two of the most basic types: integrations and simulations.

Here is an example program that flips a coin ten times.

```
from random import random
for i in range(10):
    if random() > 0.5:
        print("Heads")
    else:
        print("Tails")
```

Properly speaking, this is not a random process. Computers are completely deterministic. However, I cannot predict what the outcome of you running this program on your computer will be. In fact, I do have a 1 in 1024 chance of correctly predicting it since that is how many different results there are of flipping a coin 10 times. We should call this a pseudo-random process, but this distinction is not important unless you are doing cryptography (**never** use pseudo-random number generators for cryptography) or if you need trillions of random numbers (in which case the pseudo-random number generator risks generating statistical patterns which might affect your results). For this project, we do not care about these risks.

The *random* library has many functions. Of them, *random()* returns a random number between 0 and 1. The function we will use is *uniform(a,b)* which generates a random float x with value $a \leq x \leq b$. Note that *uniform(0,1)* is the same as *random()*.

2.2 Estimating Integrals

If you pick a random location in the box defined by $-1 \leq x \leq 1$ and $-1 \leq y \leq 1$, what are the chances that your point falls inside the circle of radius $r = 1$? This depends on how you choose the random location. If you are throwing darts, and you are good at darts, the odds might be very high. However, if you are so random that every location is as likely as every other location (which is called a uniform distribution), then the odds are the same as the ratio of the area of the circle to the area of the square, with value $\frac{\pi}{4}$. This gives us a (terrible) way of estimating pi.

> **Exercise 1** Write a Python program that uses *uniform(-1,1)* from the library *random* in order to generate N (x, y) data points, and then count how many are within the circle of radius 1. It should record its value after N data points. Then repeat this 100 times (with N data points each time). Find the mean value, and the uncertainty of the mean. (Reminder: the uncertainty of the mean is defined as $\sigma/\sqrt{M}$ where σ is the standard deviation of your 100 values you are averaging, and $M = 100$ is how many values you are averaging.) Plot your means with your estimated uncertainties of the means as a function of N, using the values of $N = 2^i$ for i of integer values 8, 9, 10, 11, 12, 13. Your graph should also include a horizontal line with the exact value $(\pi/4)$. A semi-log plot is probably your best bet.

What have we done? Basically, we have estimated the area of a circle by picking random locations and seeing how many of those locations are inside the area we care about. This can clearly be used to integrate functions.

Exercise 2 Estimate $\int_0^1 \sin^2(1/x)\,\mathrm{d}x$. This cannot be solved analytically. Looking at the function, we know the graph lies inside the box defined by $0 \le x \le 1$ and $0 \le y \le 1$. Thus you can use the uniform function to pick random points in that box, and see if they fall under the curve. That ratio will be the ratio of the areas (the area of the box is 1). You should do this the same way as in Exercise 1: generate N data points (where $N = 2^i$ with i integer values from 8 to 13), do this 100 times, and find the mean and the uncertainty of the mean. Plot a graph which is similar to the graph which you produced in Exercise 1. Hint: You can almost completely reuse your code from Exercise 1, you just have to change a couple of lines.

Note the similarity of your graphs from Exercises 1 and 2. While the values your data points converge on are different, they should display the same behaviour. The error bars should get smaller, roughly as $1/\sqrt{N}$. Since the uncertainties are getting smaller, the values should get closer to the correct value. The fact that the uncertainties get smaller like $1/\sqrt{N}$ is a generic feature of Monte Carlo integrations (and of many other random processes). This is a terrible scaling: to improve your answer by 1 digit requires 100 times as many calculations! This is worse than other numerical integration techniques. However, there is one benefit to Monte Carlo integrations. No matter what the dimension of your integral (1-D, 2-D, 100-D) your uncertainty always falls like $1/\sqrt{N}$. Other (non-random) methods do not perform so well for 100-D integrals. If you need to find the minimum energy configuration of a protein, and the result requires evaluating a 1000-dimensional integral, Monte Carlo methods are your only option.

This method of approximating integrals is called the acceptance-rejection method, because we accept some random points and reject others, and the ratio tells about the area. It's not a great method, but it is easy to understand and it works well enough in many cases. One improvement would be to not sample the integral uniformly. Instead, we could pick more points where the function is large (with more area) and fewer points where the function is small. We will not go into detail, but the basic idea is to try to cleverly reduce the number of rejected data points you produce (which, after all, are not contributing to finding the answer). As a trivial example, in Exercise 1 if you pick random points in a box which is 100 times larger than your circle, then you will be throwing out 99% of your random values, which means your program will take 100 times longer to get the same accuracy.

What if we want to find the answer to $\int_0^{2\pi} \sin(x)\ dx$? The correct answer is zero, but the acceptance-rejection method as defined won't work properly. We can use a different idea about integrals to answer this question:

$$\int_a^b f(x)\ dx = (b-a)\langle f(x)\rangle$$

where $\langle f(x)\rangle$ is the average value of f(x) on the interval $a \le x \le b$. By sampling the region randomly we can easily find the average value of the integrand.

Exercise 3 Using the same technique from Exercise 1 (plot values of $N = 2^i$ with error bars) evaluate $\int_0^1 \sqrt{1-x^2}\ dx$ by finding the average value of $\sqrt{1-x^2}$ (averaged over N trials) 100 times for each value of N.

Monte Carlo integration is almost always a bad idea for 1-D integrals. You usually want it for multi-dimensional integrals. The generalization of our earlier result becomes

$$\int f(\vec{x})\ dV = V\ \langle f(\vec{x})\rangle$$

Let's find the moment of inertia of a disk using Monte Carlo integration. That is, we want to solve the following integral:

$$\frac{1}{\pi}\int_{x^2+y^2\leq 1}(x^2+y^2)\ dx\ dy = \frac{1}{\pi}\pi\langle x^2+y^2\rangle = \frac{1}{2}$$

Exercise 4 Evaluate the integral by picking $N = 10^4$ random locations inside a radius $r = 1$ circle, finding the average value of $x^2 + y^2$ ($\langle f(\vec{x})\rangle$). You must multiply the result by the area of the integrand ($V = \pi$) and the constant in front of the integral ($\frac{1}{\pi}$) to get the (approximate) correct results of $\frac{1}{2}$. You can ignore the last two steps and just find the average value since they result in a multiplication of $\frac{\pi}{\pi} = 1$. Do this 100 times so you can find the uncertainty (as was done in Exercise 1).

Do this in the following three ways:

1. r = uniform(0,1)
   θ = uniform(0,2π)
   x = r cos(θ)
   y = r sin(θ)
2. x = uniform(-1,1)
   y = uniform(-$\sqrt{1-x^2}$,$\sqrt{1-x^2}$)
3. x = uniform (-1,1)
   y = uniform (-1,1)
   Throw out (x,y) if $x^2 + y^2 > 1$, don't include this in your average.

Note that only one of these three methods gets the correct answer. The other two methods do not sample the circle evenly, they over-sample certain areas and under-sample other areas. Explain why two of these methods fail. It will help if you plot an (x,y) scatter plot of all $N = 10^4$ random locations of one trial, it should be visually obvious which distribution is properly uniform. Do not plot $100N = 10^6$ random locations or you will just have a solid blot.

Note: this exercise requires 3 scatter plots of your random samplings, but only numerical values (with uncertainties) for the integrals.

2.3 Modeling A Physical Situation: A Random Walk

Imagine that you have coloured one air molecule so that you can easily see how it moves in the air. Hypothetically, if you knew the exact position and velocity of every air molecule in the room then you could predict the motion of your molecule perfectly. In practice, this

is impossible. However, we can assume that the molecule moves in a straight line (if we ignore gravity) until it hits another molecule, and that the result of the collision is that it now travels in a randomly chosen direction. If we know how far, on average, the molecule moves before its next collision then we can easily model its motion as a random walk. This is the basis of Brownian motion (named after the biologist Robert Brown who was the first to document and study this type of motion).

A statistical mechanics course which deals with this problem can easily find that the net displacement is on average zero, but that very few particles will actually be at the origin after N random steps. A more interesting thing to look at is the root-mean-square (RMS) of the final locations of the particle. That same course on statistical mechanics would find that $\sqrt{\langle d^2 \rangle} = L\sqrt{N}$ where L is the average distance traveled between collisions (the mean free path). Since you will use $L = 1$, you should get that your graph goes like $\sqrt{N}$. In two dimensions (or any number of dimensions), it should always be $\sqrt{\langle d^2 \rangle} = L\sqrt{N}$.

Exercise 5 Write a Python program that models a one-dimensional random walk. Start your particle at position zero. Then move it a distance 1 (in arbitrary units) to the left or right with equal probability (50% chance it goes either way). Plot the particle's location for the first 1000 steps.

Next, do something similar to the first few exercises. Make your program go for N steps, and have it do this 100 times. Find the mean location, and uncertainty of the mean, after N steps. Plot this for $N = 2^i$ for i of integer values from 8 to 13. Plot this on a graph (semi-log). Note: as mentioned above, the mean should be zero, however it should not be exactly zero because this is a random process.

Finally, plot the RMS of the final locations after N steps. You should probably calculate this at the same time that you do the calculations for the previous graph. Plot this on a graph (semi-log); include a graph of the theoretical values mentioned before this exercise ($\sqrt{N}$). Note: this exercise needs 3 graphs.

Exercise 6 Repeat the last graph of Exercise 5 (just the RMS graph, do not repeat the first two graphs) but do it in 2 dimensions. Assume that the direction the particle moves after a collision is uniformly distributed over all possible angles. I recommend that you use the following as your template code, because the math library defines pi for you (avoiding the possibility of you making a typo).

```
from math import pi
from random import uniform
theta=uniform(0,2*pi)
```

Note: for this exercise, by location we mean distance from the origin, *i.e.* $\sqrt{x^2 + y^2}$.

Note: if you find this section interesting, there is a lab experiment on Thermal Motion where you can experimentally verify these results.