PHY324 Computational Project: Monte Carlo Methods

Juan Pablo Alfonzo
1003915132

January 31 2021

1 Introduction: What is a Monte Carlo Method?

A calculation is called a Monte Carlo method if it uses random numbers in order to approximate the correct solution to a problem. The Monte Carlo methods follow the trend that if you use more random numbers to do the calculation then your result should be statistically more accurate[1]. "Monte Carlo methods can only asymptotically approach the correct answer"[1]. Monte Carlo methods are often used in problems where it is impossible to exactly calculate something given a finite amount of time (for example when dealing with a 10^{20} particle system in which $2^{10^{20}}$ combinations may be possible). In such problems we often calculate a tiny fraction of the the real problem and hope that we can extrapolate the result to the rest of the problem. If the Monte Carlo method is successful than the answer we get for this tiny fraction of the problem is also the answer to the complete problem.

This paper will focus on two of the most basic uses of the Monte Carlo method: integrations and simulations. Section 2 of this paper will explore the use of Monte Carlo methods to solve integrals which cannot be done analytically, and also integrals which can be done analytically in order to gauge how good the method is. Section 3 of this paper will explore the simulation application of the Monte Carlo method by exploring a random walk of a particle, a key component of Brownian motion.

Note that throughout this paper I will use the term "random" when I really mean "pseudo random" as computers are fully deterministic in nature and are unable to generate truly random numbers. For the purposes of this lab however this will not be a major issue, and we can proceed to do Monte Carlo methods despite this. For more information on this consult section 2.1 of the cited project handout[1].

2 Estimating Integrals

2.1 Points Inside a Circle

If we were to pick a random point in the x, y plane such that it must lie in the box defined by $-1 \leq x \leq 1$ and $-1 \leq y \leq 1$,what is the probability that it lies in a circle of radius 1 centered at the origin? (See Fig 1. below for a visual). Of course this depends by what we mean by "random location", in the case of Monte Carlo methods we will assume by random we mean that every point in this square is equally likely to be selected. If this is the case then the odds of the point being in the circle are the same as the ratio of the area of the box to the area of the circle ($\frac{\pi}{4}$). With this then we have a very crude and rudimentary way to estimate the value of π (we simply take the value of our ratio and multiply by 4).

Now we can go about estimating this ratio by using a Python script that utilizes the *uniform()* function from the `random` library. The *uniform()* function takes two integer inputs a and b as such: *uniform(a,b)*. It returns a random float with value x such that $a \leq x \leq b$. With this in mind we write a Python script that generates N random (x,y) points inside the box. We then see how many are in the circle (i.e. how many satisfy $x^2 + y^2 \leq 1$) and take the ratio of those inside the square to the total number of points generated (N).

We can then repeat this 100 times for each value of N to get a mean value for the ratio and an uncertainty defined by:

$$\frac{\sigma}{\sqrt{M}} \tag{1}$$

Where σ is the standard deviation of the 100 values and M is the number of trials (in our case $M = 100$)

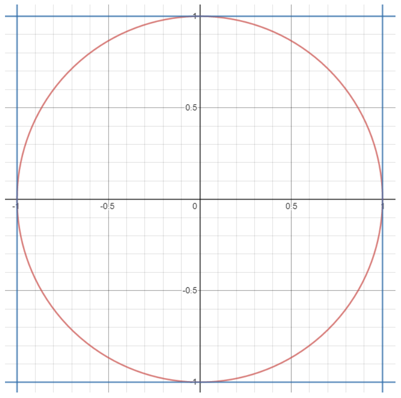

Figure 1: Circle of radius 1 Centered at Origin (Red), and concentric 2 by 2 box (blue)

We can now then do these 100 trials with values of $N = 2^n$ for $n = 8, 9, 10, 11, 12, 13$. The results are then plotted below in Fig. 2 where the ratios are on the y-axis and N is on the x-axis. The error bars are given by the uncertainty calculated by Eqn. 1 and then normalized (divided by N). The theoretical value of $\frac{\pi}{4}$ is also plotted for comparison purposes.

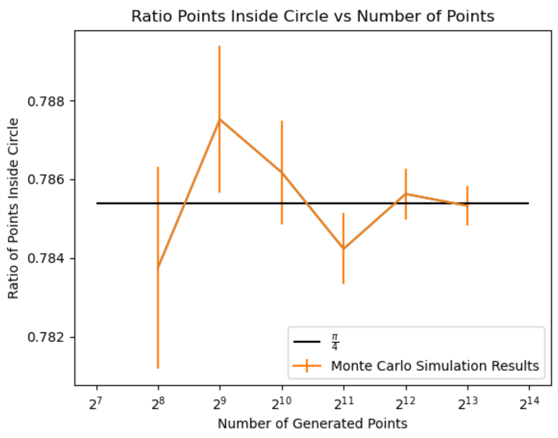

Figure 2: Ratio of points generated within the square that were inside the circle

As we can see from Fig. 2 the larger the value of N the closer we get to the theoretical value. This is expected as we know Monte Carlo methods get more accurate the more random numbers you give it to do the calculation. This is also reflected in the fact that the error bars are proportional to $\frac{1}{\sqrt{N}}$, i.e. smaller error for larger N.

2.2 Solving $\int_0^1 sin^2(\frac{1}{x})dx$ Using Monte Carlo Method

This integral cannot be solved analytically, so instead we turn to a Monte Carlo method to estimate it. We can use essentially the same method outlined in the previous subsection, but we simply change the Python script to generate points in a box defined $0 \leq x \leq 1$ due to integration bounds and $0 \leq y \leq 1$ since we know $0 \leq sin^2(\frac{1}{x}) \leq 1$ for all x. Additionally, we change the condition in the previous script that checked whether the point was inside the circle to one which ensures the point is under the curve (i.e. instead of checking that $x^2 + y^2 \leq 1$ is satisfied, check instead that $y \leq sin^2(\frac{1}{x})$ is satisfied). We repeat the rest of the exercise the same way with 100 trials of N points, with the same N values as before. Note that here the script should just return the value of the integral we are interested in since the area of the box is 1 and not 4 like before. The results of this script are plotted below in Fig. 3 in a similar fashion to how Fig. 2 was plotted.

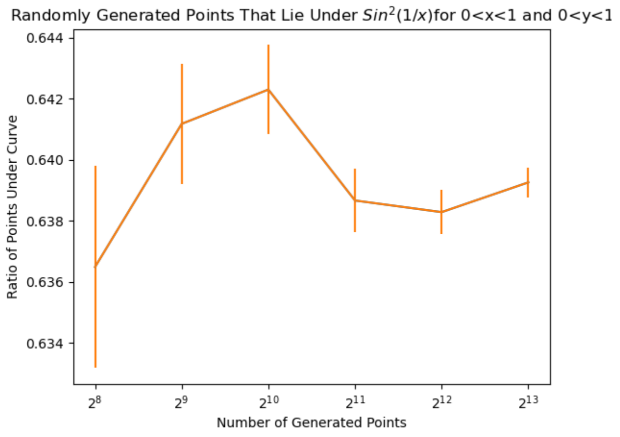

Figure 3: Ratio of points generated within the square that were under the curve

Note that Fig. 3 follows the same trend as Fig. 2 with regard to larger N and their uncertainties. In Fig. 3 no theoretical value is plotted since we are to assume it is unknown (this is the whole point of doing this Monte Carlo method). However, if we look at the plot we see it seems to be centering around 0.639, this is far from what more advanced methods tell us the value is (Mathematica gives the value 0.673). This is to be expected however as Monte Carlo methods are only crude estimates and even $N = 2^{13}$ is a relatively small number of points. This method of approximating integrals is called the acceptance-rejection method, because we accept some points and reject others, and ratio tells us about the area[1].

2.3 The Average of a Function over an Interval

Consider the acceptance-rejection method discussed in the previous subsection. If we tried applying it to $\int_0^{2\pi} sin(x)dx$ we will not get zero as we know is the answer. We then turn to a different way to solve integrals using Monte Carlo methods. Recall that we can express an integral of a function $f(x)$ over an interval $a \leq x \leq b$ as follows:

$$\int_a^b f(x)dx = (b-a)\langle f(x)\rangle \tag{2}$$

Where $\langle f(x)\rangle$ is defined as the average value of $f(x)$ over an interval $a \leq x \leq b$

Using this fact we can also calculate integrals by obtaining the average of a function using Monte Carlo methods and multiplying it by the difference of our end points.

By sampling this interval randomly we can easily find the value of the integrand. We will consider solving $\int_0^1 \sqrt{1-x^2}$ to illustrate this method. We will use the exact same method as outlined in the previous subsection to create random points in the same box, but now the condition the point must satisfy is $y = \sqrt{1-x^2}$. The Python script generates Fig. 4 below:

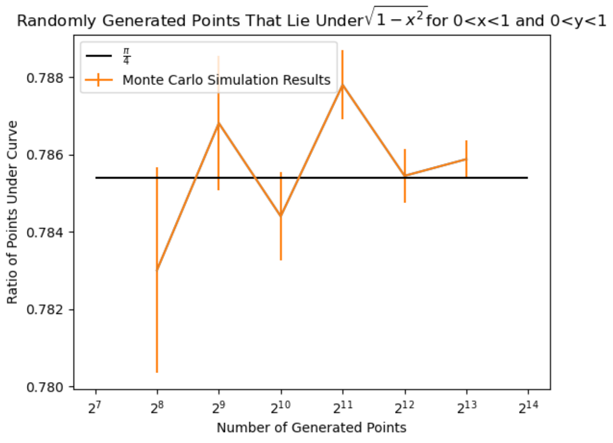

Figure 4: Ratio of points generated in box that fell under curve

As we expect this plot is almost identical Fig. 2 as we are sampling in the same way (Monte Carlo method) and both of them approach theoretical values of $\frac{\pi}{4}$. We see also the same trend of the results getting closer to the theoretical value for larger N values, and the uncertainties being proportionate to $\frac{1}{\sqrt{N}}$ as with Fig. 3 as well. The difference between this and what was discussed in the first subsection is we are only using a quarter of the circle, but since we are only using a quarter of the original box as well the ratio is preserved. For a visual of this look at Fig.2 and convince yourself that we are doing here is basically only dealing with quadrant 1 in that figure.

2.4 Average of a 2-D Function

Despite the Monte Carlo method being fairly accurate in the previous subsection it is always a bad idea to do so for 1-D integrals[1]. The generalization of Eqn. 2 to higher dimensions is given by:

$$\int f(\vec{x})dV = V\langle f(\vec{x})\rangle \tag{3}$$

We can then consider the moment of inertia of a disk as follows:

$$\frac{1}{\pi}\int_{x^2+y^2\leq 1}(x^2+y^2)dxdy = \frac{1}{\pi}\pi\langle x^2+y^2\rangle = \frac{1}{2} \tag{4}$$

In other words we can use Monte Carlo integration to show that $\langle x^2+y^2\rangle = \frac{1}{2}$, using a similar methodology as we saw in the previous subsection. This time we set $N = 10^4$ and still do 100 trials in order to find the mean number of points inside the circle, and the uncertainty. We will test three methods, with N=1000, of generating random points within the circle to see which of the three gives us the most even sampling of the circle.

1. r=uniform(0,1)
   θ=uniform(0,2π)
   x=rcos(θ)
   y=rsin(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 $x^2 + y^2 > 1$

At first glance all three ways seem like they equally sample the whole circle, but let us plot 1000 points that each model generates:

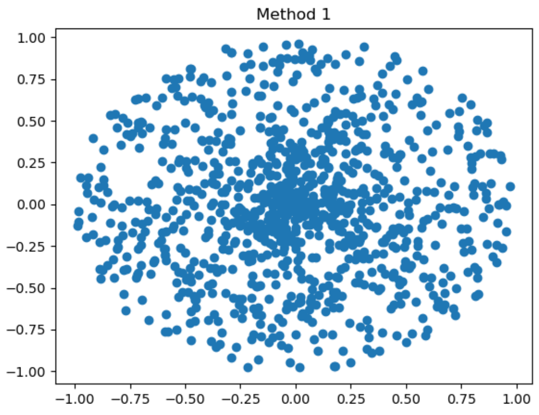

Figure 5: Plot of points generated by method 1

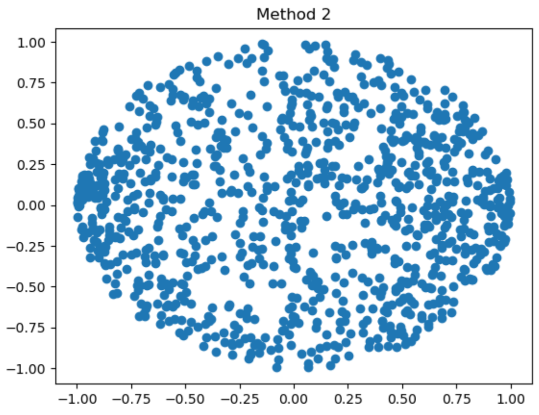

Figure 6: Plot of points generated by method 2

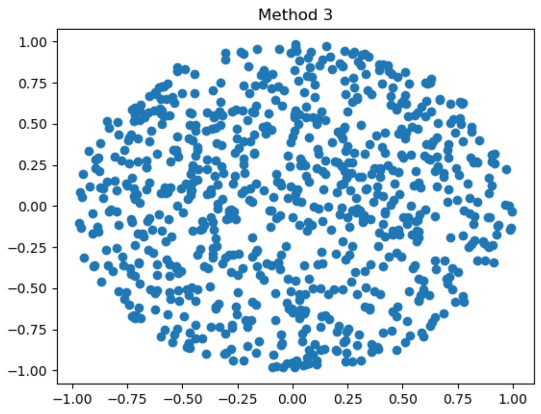

Figure 7: Plot of points generated by method 3

We can now clearly see that method 1 over samples the center of the circle, while method 2 over samples the edges around $x = \pm 1$. Method 3 however appears to sample the circle pretty evenly. We can verify this observation quantitatively by taking the average of $x^2 + y^2$ for all points generated by each method. Note that this is the moment of inertia, which is basically a measure of how the mass is distributed (close/far from the origin). The theoretical result we expect for this for an even distribution is 0.5. For method 1 the mean was found to be 0.33 which is verifies that most of the points are clustering around the center or that this method is over sampling the center of the circle. For method 2 the mean is around 0.55 which would indicate the points are more clustered towards the edges of the circle. Finally, for method 3 the mean is found to be 0.49 which makes it the most even distribution recalling that 0.5 would be a perfectly even distribution.

Why is the case though? Why is it that method 1 and 2 both fail despite looking like good methods on the surface? We can find the answer by closely examining method 1 and 2. The first method chooses as many locations (on average) from $0 \le r \le 0.1$ as from $0.9 \le r \le 1$ (it's uniform in r), but those two rings have very different areas hence very different densities. This leads to a higher density of points for lower values of r as it has the same odds as a larger r of having a point generated within it, but has a lower area than a ring with a larger r. Method 1 effectively has a $\frac{1}{r}$ mass density, and if you include that factor in the relevant integrals you get that $\frac{1}{3}$ is the correct value for its moment of inertia. Method 2 has a similar issue with the density function of points in the x-direction, as it over samples the edges of the circle for a similar reason that method 1 over samples the inner part of the circle. This being that the x components of the points have equal probability of being generated throughout the circle, but as we get closer to the edges of the circle ($x = \pm 1$) we see the value of $\pm\sqrt{1 - x^2}$ start to get closer together. This means the range that a random y is chosen from near the edge is smaller and we are bound to more points bunching up in this area (a higher density) than anywhere else in the circle. So a similar idea to why method 1 fails, but instead of being rings from the origin of different densities its regions around the edges along the x axis with different densities.

So we see that method 3 is the only one that properly samples the circle. Knowing this we select this method to do our 100 trials with setting $N = 10^4$. The result of the 100 trials is a mean value of 0.50006 and an uncertainty of 0.00099.

3 Random Walk

Imagine trying to track one specific particle in a fluid. In theory, if you knew the exact position and velocity of every molecule in the fluid then you could predict the motion of this specific particle perfectly. In practice however, this is impossible. We can however assume the particle moves in a straight line (ignoring gravity) until it hits another molecule. If we know how far, on average, the molecule moves before its next collision then we can model its motion as a random walk. This is the basis of Brownian motion[1].

3.1 1-D Random Walk

Let us model such a random walk in 1-D using a Python script. We will start the particle at the origin and have it move either to the left (-1) or the right (+1) randomly. This script will be similar to the ones used above but we want to pick up either -1 or 1, and not any random real number in between. For this we can use something like `np.random.choice([-1,1],1)` which takes 1 random element from the array [-1,1] (in other words chooses either 1 or -1). We can then repeat this process 1000 times, adding every new entry to the sum of all the past entries, to model a random walk in 1-D. Below is a plot of one such random walk that the script generated:

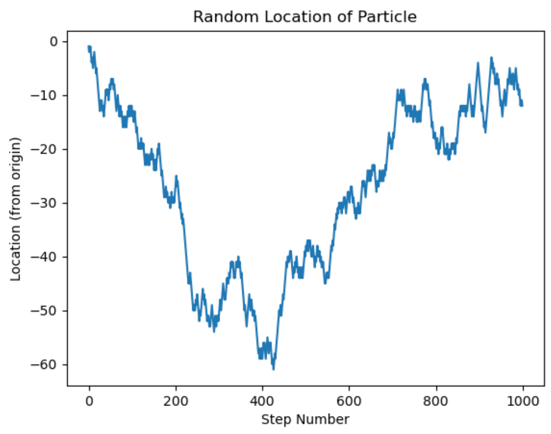

Figure 8: Random walk of a particle in 1D that always moves 1 unit length

Note that since this generating a random step every time the graph can vastly differ every time you run the same script as it is random.

3.2 Mean Position of Random Walk

Consider now doing this walk a much larger of number of times, and after each time recording the final position you end up from with respect to the origin. We can then do something similar to what we did throughout section 2 of this paper and apply Monte Carlo techniques to this process as well. In this case however we will have 100 random walks each one consisting of N steps, where like in section 2 of this paper we will take $N = 2^n$ for $n = 8, 9, 10, 11, 12, 13$. We can as before then plot the results as shown on the next page.

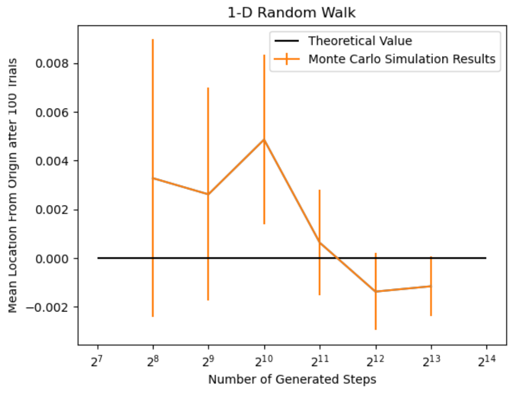

Figure 9: Mean final position of 100 random walks with N steps

Fig. 9 shows us what we have come to expect thus far in this paper from Monte Carlo methods, that the larger the N the closer we get to the theoretical value and our uncertainties get smaller. Note, the theoretical value here is 0 because we would expect over a large enough number of trials we would end up at the origin as we have equal odds of going right (getting +1) or going left (-1).

3.3 1-D RMS

We can now think of a more interesting number when it comes to random walks, the RMS (root-mean-square) of the final locations we found in the previous subsection. We expect the RMS to follow:

$$\sqrt{\langle d^2 \rangle} = L\sqrt{N} \tag{5}$$

Where the LHS of the equation is the RMS, L is the average distance traveled by the particle between collisions, and N is the number of steps in the walk.

In this paper we will always consider the particle to move by 1 unit and so we can fix $L = 1$, and are left with the RMS is equal to $\sqrt{N}$. Note this is true regardless of the number of dimensions the particle is moving in. So now let us calculate the RMS for the 100 walks we simulated in the previous subsection. This can be done by generating N random steps, summing them all together to get a final displacement with respect to the origin, then squaring all 100 of these values and taking the mean. A Python script was developed to do this and generates the following plot:

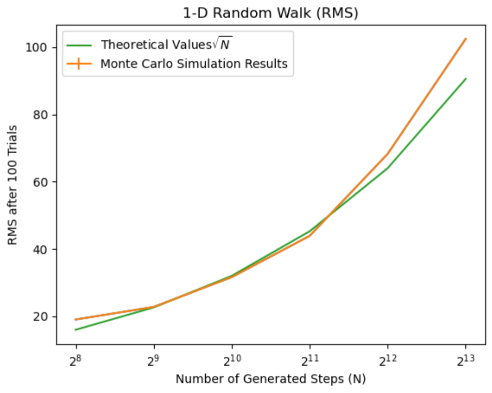

Figure 10: RMS in 1D

We see from Fig. 10 that the RMS simulated using Monte Carlo methods is very close to the theoretical value $\sqrt{N}$.

3.4 2-D RMS

Now we can do the RMS but in 2-D to see how well the Monte Carlo method fairs for 2-D random walks. We will still the particle can only move 1 unit length between collisions and we still assume that that the direction the particle moves after a collision is uniformly distributed over all possible angles. This can be done thinking in polar coordinates in which we randomly generate a θ in the interval $0 \leq \theta \leq 2\pi$, and then setting $x = cos\theta$ and $y = sin\theta$ (since we assume $r = 1$). From here we sum and square all the generated steps in the x direction and y direction independently and then add them together in quadruple. In other words like such:

$$d^2 = \sqrt{(\Sigma_1^N x_i)^2 + (\Sigma_1^N y_i)^2} \tag{6}$$

Where we add all the x steps together first, and then square the result of the sum. Likewise for y. Then sum those terms together and then take the square root. We take the result of Eqn. 6 and then take its mean value over 100 trials and square root it to find the RMS in 2-D for a given set of 100 trials with N steps. The results from the Python script written to do this are shown in the plot below:

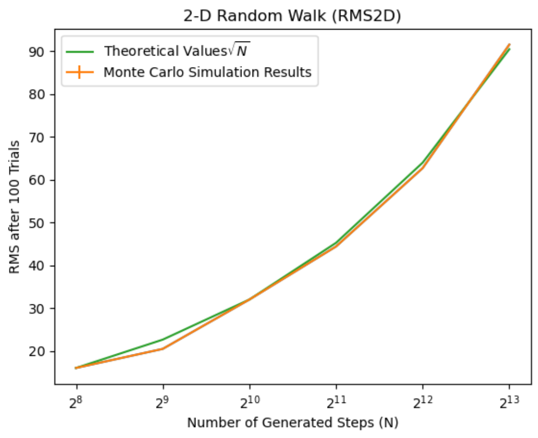

Figure 11: RMS in 2D

Just as we saw for RMS in 1-D we see that the trend of RMS being equal to $\sqrt{N}$ holds true in 2-D, as is expected.

4 Conclusion

In closing we have used Monte Carlo methods to solve integrals and for random walk simulations. We found that for finding integrals it is a rather crude method (highlighted in section 2.2), but the number returned is reasonable considering the low computing time that was needed. For the random walk final displacement from origin simulation (section 3.2) the Monte Carlo method is about as accurate as it was for the integrals. For the RMS calculations it was also pretty reasonably close to the theoretical values. Overall, the Monte Carlo method is successful in allowing us to calculate a small part of a bigger problem and to extrapolate the data to the whole problem. It has a good trade off between accuracy and computation time, especially the more computationally heavy the problem gets.

References

[1] *PHY324 Computational Project.*