Monte Carlo Techniques and Analysis Using Python

Keegan W$^{m.}$ Karbach

February 8, 2020

Introduction

In this project, we explore Monte Carlo integration techniques (MCI). This powerful method of numerical integration uses Brownian Convergence of a large series of random numbers to approximate an N-dimensional volume. A series of random numbers $X : \{X \in \mathbb{R}^{\mathbb{N}} \sim U^N[0, 1)\}$ is mapped to a domain that encloses the desired volume to be integrated and the ratio of samples that are enclosed by the integrand to the number of samples total gives a numerical approximation of the integral itself.

The prototypical example is finding π by numerically integrating a circle inscribed in a square domain and solving for an estimate of π, and while this example may seem trivial, this concept can be extended into an amazingly wide range of applications. Monte Carlo techniques are used in situations where there are many unknown variables (as the technique can be extended to an arbitrary number of dimensions), multiple applications across STEM fields, as well as risk assessment in business and economic models.

Personally, I find it fascinating that one can find an unknown parameter purely by utilizing a large enough sample of random numbers applied to an integrable model without *a priori* knowledge of the range to which that unknown belongs.

1 Determining π through MCI

1.1 Write an MCI routine to determine the value of π. How might π be found from simple integration?

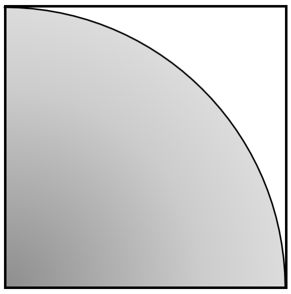
The MCI routine that I used to determine the value of π was designed for simplicity. Using a unit circle of area A and a domain, D, of a square with a side value of 1, we can inscribe one quarter of the unit circle into the domain. Analytically, the problem is thus:

Simple integration to find π over this domain lends:

$$A_{cir} = \frac{\pi r^2}{4} \longrightarrow \pi = \frac{4}{r^2}\int_0^r f(x)dx$$

Where $r = 1$ and $f(x) = \sqrt{1-x^2}$ which reduces to:

$$4\left[\frac{\sin^{-1}(x)}{2} + \frac{x}{2}\sqrt{1-x^2}\right]_0^1 = 4\left[\frac{\pi}{4}\right] = \pi$$



$$\begin{cases} A_D = 1\text{u}^2 \\ A_{cir} = \frac{\pi}{4}\text{u}^2 \end{cases}$$

Numerical integration using MCI will consist of taking a two-dimensional sample of random numbers over a uniform distribution $X \in \mathbb{R}^2 \sim U^2[0,1)$ an no mapping will be necessary for this domain. A sketch of the code is as follows:

```
for j=1,2,...,n
c=0
random(x,y) = i
check if i satisfies the condition
if true, add 1 to c
est=c/n
```

As each random number is chosen, the inequality $y \leq \sqrt{1-x^2}$ will be checked. An estimate of the area of the domain will be calculated as the ratio for samples for which this inequality holds over the total number of samples taken.

$$\frac{N_{cir}}{N_{tot}} = \frac{A_{cir}}{A_{tot}} = \frac{\pi}{4} \longrightarrow \pi \approx \frac{4c}{n}$$

Four algorithms were created using different Pythonic methods; all of these are listed in the code appendix, and their timing tests are included in Appendix C. Here is the algorithm used for this project – the first version creates an array of estimates which can be used to create a convergence plot while the second updates the estimate value at each iteration, eliminating the necessity of storing a large array of values.

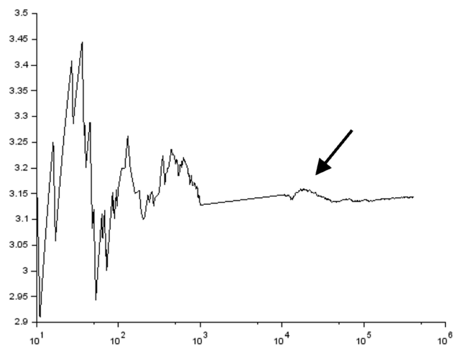
```python
@njit
def sim4(n, empty_array):
    c = 0
    for i in prange(n):
        x = rn.rand(2)
        if x[0]**2 + x[1]**2 <= 1:
            c = c + 1
        empty_array[i] = (4 * c / (i + 1))
    return empty_array
```

The `njit` decorator and `prange()` command are both from the Numba package used for optimization of Python code.

1.2 Legacy estimation π to 3 and 5 digits of accuracy

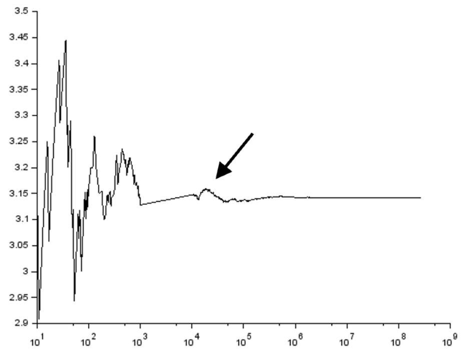
I could not figure out how to deprecate the use of the Mersenne Twister PRNG in multiple Python libraries – the only way I could seem to make this happen was to use an earlier version of Python. I decided to instead use SciLab which allows easy transition between generating engines. Using the legacy PRNG[1]in SciLab with a seed value set to 15215, two simulations were performed to required accuracy.

In Figure 1 we see a Brownian walk using a PRNG with a period of 2^{31}. I was curious if the artifact that is noted in both graphics below is a product of correlation in the random number generator.



UNI out to 3 decimals:
Estimate: 3.143060 +/- 0.005000
n:
413882

Figure 1: Legacy PRNG to 3 digits of accuracy



UNI out to 5 decimals:
Estimate: 3.141603 +/- 0.000195
n:
273594760

Figure 2: Legacy PRNG to 5 digits of accuracy

Note: using a different program for the legacy generator precluded a full statistical analysis of the error in the estimate. Here, I'm going off of the assumption that $\epsilon \sim \mathcal{O}(1/\sqrt{n})$ which is where I get the error bounds given in the above figures.

1.3 Estimation of π to 3 and 5 digits of accuracy

In Figure 3 we see a visual representation of the problem - the black dots are inside the desired integral while the blue are not. This represents a run of n =1e5 samples where π_{est} = 3.15184

[1] "Urand, A Universal Random Number Generator" by Michael A. Malcolm, Cleve B. Moler, Stan-Cs-73-334, January 1973, Computer Science Department, School Of Humanities And Sciences, Stanford University

and the error is of the order $\mathcal{O}(1/\sqrt{n}) \sim 1\text{e} - 3$.

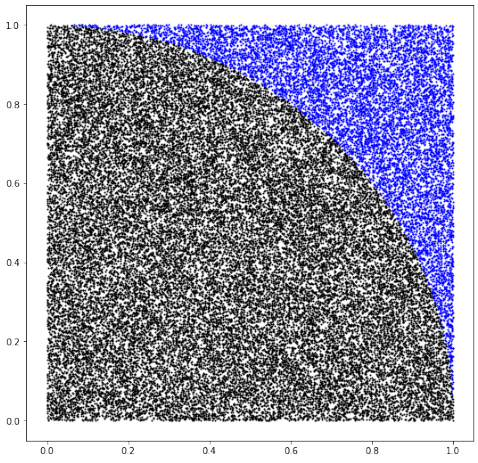


Figure 3

In Figure 4 we see the convergence of a run in Python using the Mersenne Twister PRNG to 7.8×10^6 samples with the corresponding estimate, the sample standard deviation for the final 1×10^5 samples, and the absolute error between the the estimate and the theoretical value.

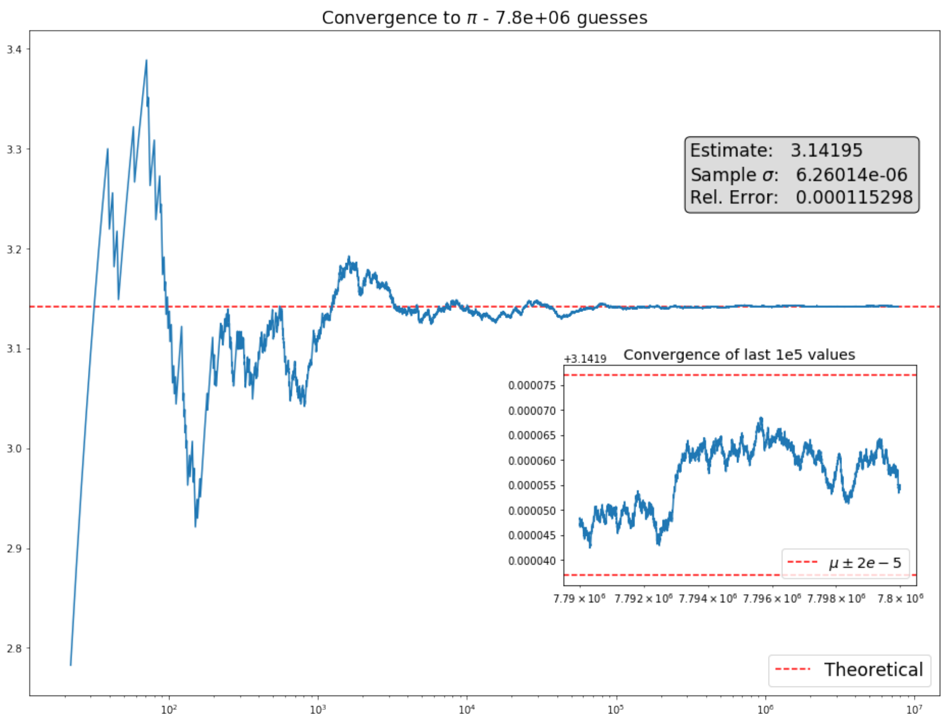


Figure 4

Using the standard deviation has value in understanding the variance of the samples as the simulation progresses, but cannot be used to determine variation between the estimation and the theoretical value as this process Brownian and requires different statistics.
The accuracy of the estimate goes as $1/\sqrt{n}$, so for the number of correct decimals to increase by two, we need four orders of magnitude more data points. For accuracy to three decimals we want an error bound $\epsilon = \pm 0.0005$ which means that we need $n = (5\text{e} - 4)^{-2}$ points. This relation is illustrated in Figure 5. Using this calculation, to have five digits of guaranteed accuracy within the indicated bounds, a simulation of *40 billion* points is required. While I have done this – these results are posted at the end of the document – it becomes very computationally expensive. These runs take a little over 85 minutes on this machine and I haven't been able to produce a convergence plot of this magnitude. Perhaps there is a better way.

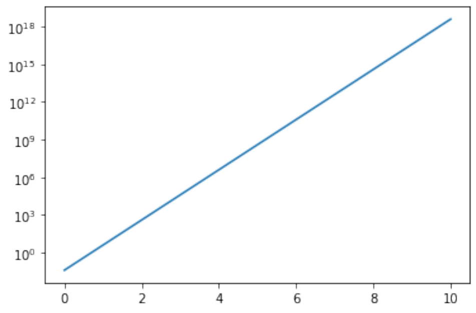


Figure 5: Semi-log plot of number of data points as a function of desired accuracy

While standard statistics can not be used for a single MCI run, the estimation produced by each run can be viewed as a random value. Hence, by taking a large enough sample of these values, a histogram can be produced that will tend towards the normal distribution for increasing sample size.

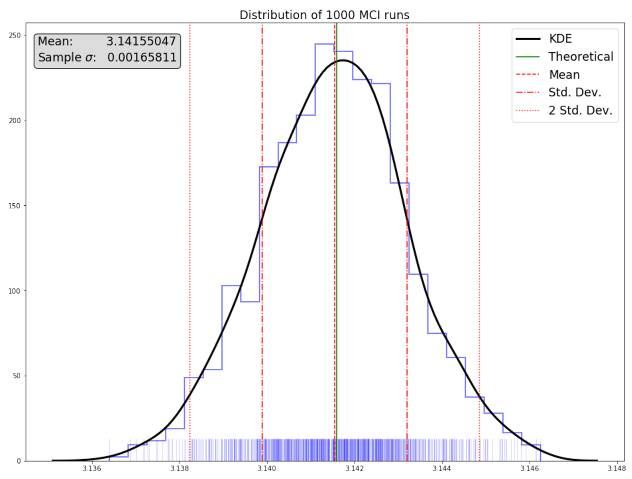


Figure 6

The results of this analysis are shown in Figure 6 which illustrates a distribution plot of the estimates of 1000 MCI runs of 1e6 samples each. The standard deviation of this plot represents the variation of the estimation over each MCI simulation. Note that, in order to decrease this value, we must take more samples in each MCI run.

Now we perform a full analysis of π to five digits of accuracy. As previously stated, this would require 4e10 samples, but by using the statistical analysis illustrated above we can compile that many data samples into a kernel density function. First, in Figure 7, we can see a convergence plot to 2e9 samples. The computation of this, while expensive, wasn't exorbitant; it took about 6 minutes. Problems did arise when attempting to plot this data. Plotting packages initialize a grid of $\mathcal{O}(n^d)$ for n samples and d plotting dimensions.

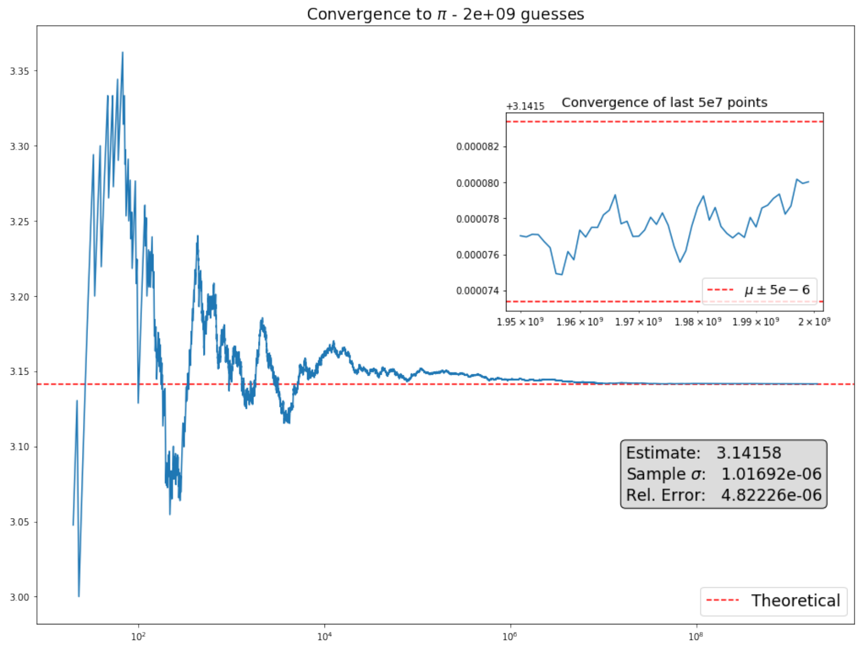


Figure 7

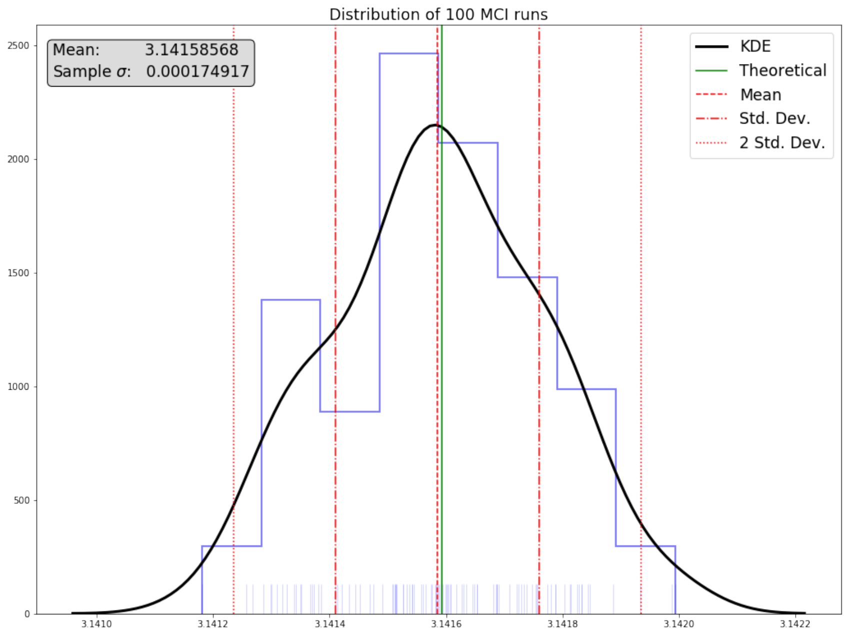


Figure 8

This would have required around 200 GiB of memory. Ergo, I reduced the arrays via scaling after running the MCI simulation. I increased the granularity towards the end of the arrays to better illustrate convergence towards the tail end of the plot. Again, we perform a series of simulations to get a distribution of the estimates. In Figure 8, we have run 100 simulations of 1e8 values, an aggregate of 1e10. Notice that the standard deviation of this distribution has only been reduced by one order of magnitude, not two. The theoretical particulars informing this are beyond the scope of this project, but can be found in Kendall et al. [2007].

2 3D Integration

2.1 Mass of an ellipsoid

For the equation $3x^2 + 2y^2 + z^2 = 25$, we can convert to the standard form of equation for an ellipsoid:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1$$

where $a^2 = 25/3$, $b^2 = 25/2$, and $c^2 = 25$.

We can analytically find the mass M over this domain D by taking the triple integral:

$$\begin{aligned} M &= \iiint_D \rho \, dV \\ &= \int_{\pm\gamma} \int_{\pm\beta} \int_{\pm\alpha} \rho \, dydxdz \end{aligned}$$

Where

$$\begin{cases} \alpha = b\sqrt{1 - \frac{x^2}{a^2} - \frac{z^2}{c^2}} \\ \beta = a\sqrt{1 - \frac{z^2}{c^2}} \\ \gamma = c \\ \rho = 1 \end{cases}$$

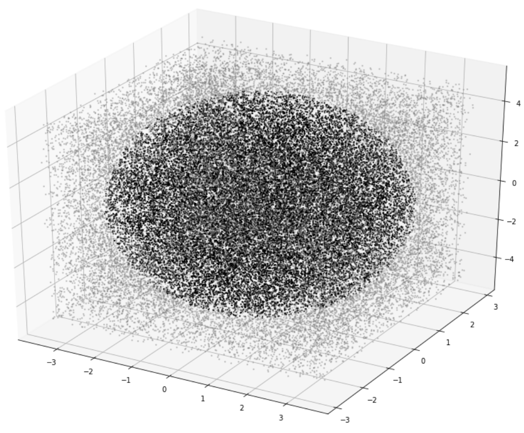
Integrating gives us the mass:

$$\begin{aligned} M =& 8\rho \int_0^\gamma \int_0^\beta \int_0^\alpha dydxdz \\ =& 8\rho b \int^\gamma \underbrace{\int^\beta \sqrt{u^2 - x^2} dx}\, dz \Bigg| \, u = \sqrt{a^2 - \frac{a^2 z^2}{c^2}} = \beta \\ & = \int^\beta \frac{1}{a}\sqrt{u^2 - x^2} \\ & = \frac{1}{2a}\left[x\sqrt{u^2 - x^2} + u^2 \tan^{-1}\left(\frac{x}{\sqrt{u^2 - x^2}}\right)\right]_0^u \\ & = \frac{1}{2a}\left[\left(u^2 \frac{\pi}{2}\right) - (0)\right] \quad \because \quad \lim_{x \to u}\left(\frac{x}{\sqrt{u^2 - x^2}}\right) = \frac{\pi}{2} \\ & = \frac{\pi}{4a} u^2 \\ =& \frac{2\pi b}{a}\rho \int^\gamma u^2 dz \longrightarrow \frac{2\pi b}{a}\rho \int^\gamma a^2\left(1 - \frac{z^2}{c^2}\right) dz \end{aligned}$$

$$=\frac{2\pi b}{a}\rho\left[\int^{\gamma} a^2dz - \int^{\gamma}\frac{a^2z^2}{c^2}dz\right]$$
$$=\frac{2\pi b}{a}\rho\left[a^2c - \frac{a^2}{c^2}\frac{c^3}{3}\right]$$
$$=2\pi ab\rho\left[c - \frac{c}{3}\right] = \boxed{\rho\left(\frac{4}{3}\pi abc\right)}$$

Which gives us the density times the volume formula for an ellipse, as expected. Plugging in the values for a, b, and c gives us a volume of $V \approx 213.758\text{m}^3$. Multiplying this result by the mass density $\rho = 1.000\text{kg/m}^3$ gives us a mass of 213.758kg – we can perform these operations in this case as the density function is a constant.

Figure 9: MCI of full ellipse



Numerically simulating this problem is a matter of choosing a domain, and much like I did with the estimate of π from the first part of the question, I initially simplified the domain to the first quadrant, then scaled my guess to the correct quantity by multiplying the ratio of samples inside the domain to total number of samples by a factor of $8abc$ – the factor of $4\pi/3$ is built into this ratio. This code gives me the mass out to two decimals with a 95% confidence interval. The density cloud pictured in Figure 9 is over the entire domain, as this is the method I used for the next two extensions to the problem.

Despite the difficulty in the analytical solution of this ellipse, the modification to the Monte Carlo algorithm is quite simple. Due to the nature of the optimization routines that I used, I could not call a command for a uniform distribution outside of $U[0, 1)$ and therefore needed an additional three lines of code to modify the random samples for each iteration. Despite the addition of these arithmetic operations, the code still ran quite fast.

```
@njit
def sim_ellipse(n, empty_array):
    m = 0
    for i in prange(n):
        w = rn.rand(3)
        x = w[0] * (2 * a) - a
        y = w[1] * (2 * b) - b
        z = w[2] * (2 * c) - c
        if 2*x**2 + 3*y**2 + z**2 <= 25:
            m = m + 1
        empty_array[i] = (8 * a * b * c * m / (i + 1))
    return empty_array
```

In Figure 10, we can see the convergence plot of the modified algorithm to 1e6 decimals. Again, a statistical analysis was performed based on this size simulation and will be compared to the large-scale runs that are included at the end of this document.

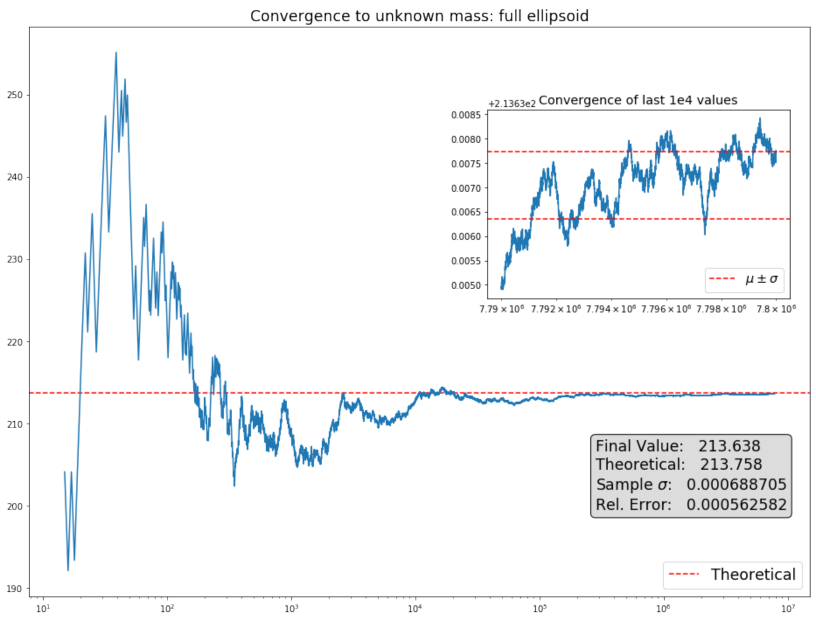


Figure 10

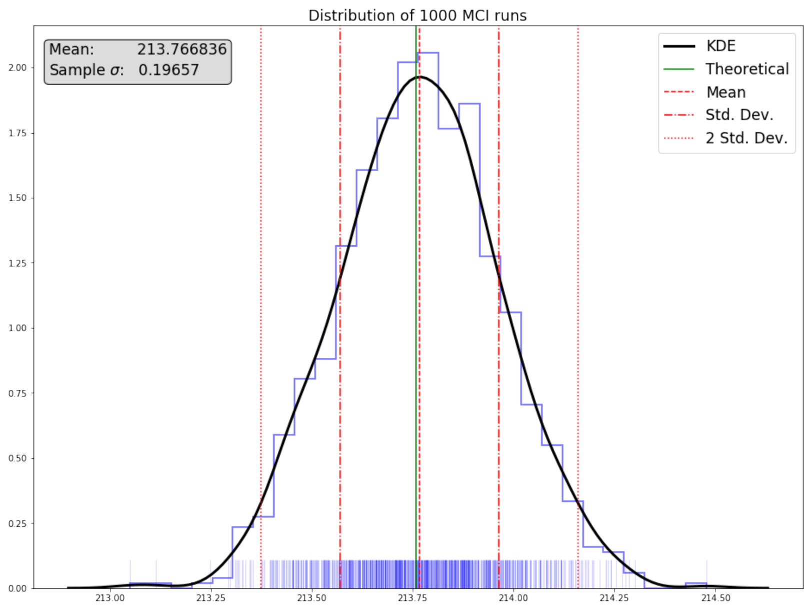


Figure 11: Full ellipsoid

Using this analysis, we arrive at a mass of 213.767 kg, which deviates from the theoretical value by approximately 9 grams.

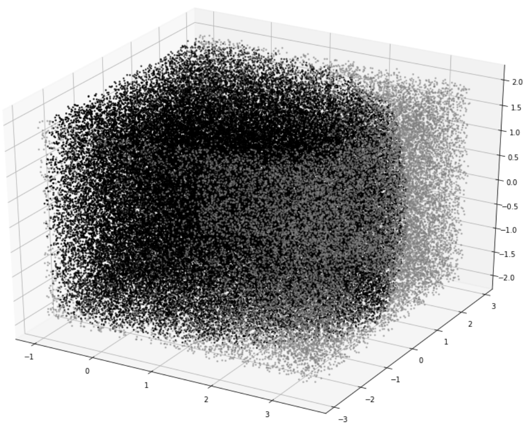
2.2 Mass of a truncated ellipsoid

This truncation, upon first examination, seems to be an issue of simply changing the limits of integration from the full ellipsoid:

$$M = \rho \int_\gamma \int_\beta \int_\alpha dydxdz \begin{cases} \alpha = \pm b\sqrt{1 - \frac{x^2}{a^2} - \frac{z^2}{c^2}} \\ \beta = \begin{cases} a\sqrt{1 - \frac{z^2}{c^2}} \\ -1 \end{cases} \\ \gamma = \pm 2 \\ \rho = 1 \end{cases}$$

Appendix B.1 contains the attempted analytical solution to this integral and B.3 contains a Mathematica notebook with the numerical integrations used to get the theoretical values. The analytical solution is clearly incorrect, deviating by 49 grams (59.33% error). The Monte Carlo value agrees with the black-box numerical integration to an error of 3.680×10^{-4}. Changing the error bounds removes the ability to use symmetry arguments to simplify the integral and makes the analytical solution much more difficult.

Figure 12: MCI of truncated ellipse



This is, however, a great example of a difficult analytical problem that lends itself quite readily to Monte Carlo techniques. It was a simple matter of changing the domain of the random sample and following the same procedure as the full ellipsoid. I couldn't use the same shortcut of keeping the numbers in the first quadrant, as I didn't have the symmetry to do so, so I modified the sample domain and test clause in the algorithm and the simulation worked as intended. Included below is the modified algorithm as well as the convergence plot and distribution of results.

```
@njit
def sim_tellipse(n, empty_array):
    m = 0
    for i in prange(n):
        w = rn.rand(3)
        x = w[0] * (a + 1) - 1
        y = w[1] * (2 * b) - b
        z = w[2] * (4) - 2
        if 2*x**2 + 3*y**2 + z**2 <= 25:
            m = m + 1
        empty_array[i] = (2 * (a + 1) * b * 4 * m / (i + 1))
    return empty_array
```

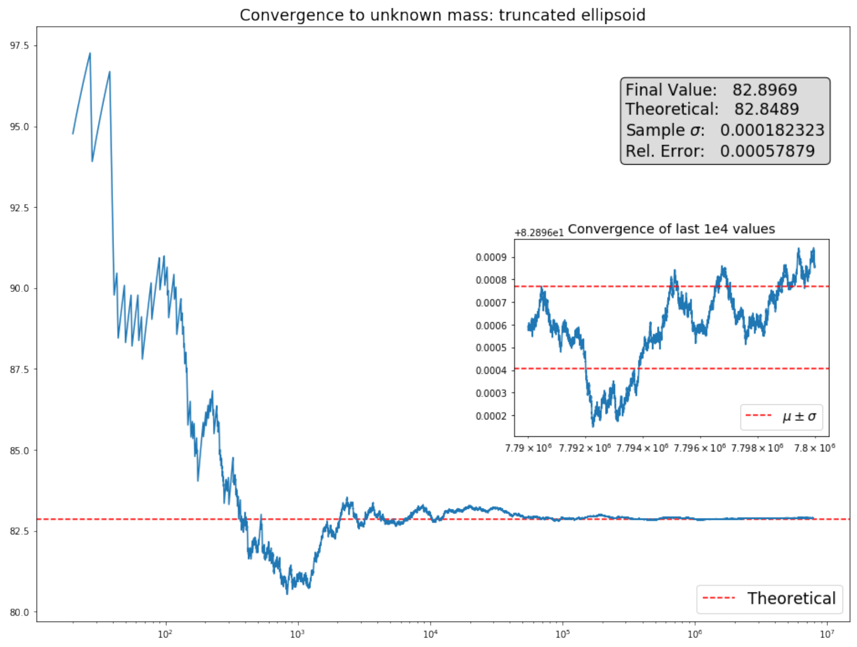


Figure 13

A point of illustration in these two figures: the inset in Figure 13 shows that the solution is still quite variant towards the end. The fact that the solution still varies outside of one standard deviation indicates that a longer simulation should be run. We can see this in comparing the estimated value with the theoretical – there is deviation of 48 grams. However, when multiple simulations are run as in Figure 14, we can see that the deviation drops to a tenth of a gram (an improvement of two orders of magnitude).

2.3 Mass of a truncated ellipsoid with variant density

Analytically, this problem takes the model designed for the truncation and appends the density function:

$$M = \rho \int_\gamma \int_\beta \int_\alpha dydxdz \begin{cases} \alpha = \pm b\sqrt{1 - \frac{x^2}{a^2} - \frac{z^2}{c^2}} \\ \beta = \begin{cases} a\sqrt{1 - \frac{z^2}{c^2}} \\ -1 \end{cases} \\ \gamma = \pm 2 \\ \rho = x^2 \end{cases}$$

Appendix B.2 contains the attempted analytical solution to this integral which deviates from the numerical solution by 194 grams (*103.49%*) while the Monte Carlo value agrees with the black-box numerical integration to an error of 3.091×10^{-4}. The addition of a density term and the asymmetric limits of integration makes the analytical solution much more difficult.

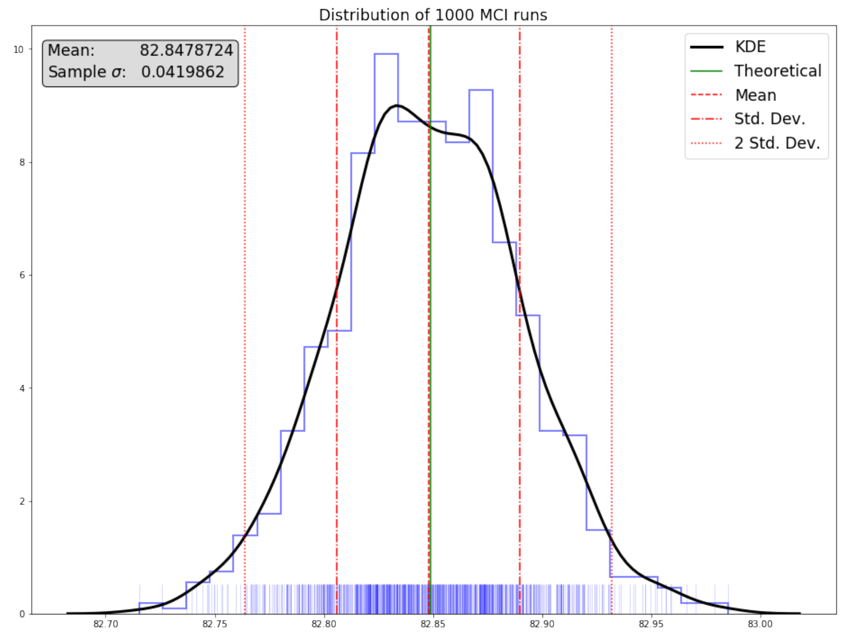


Figure 14: Truncated ellipsoid

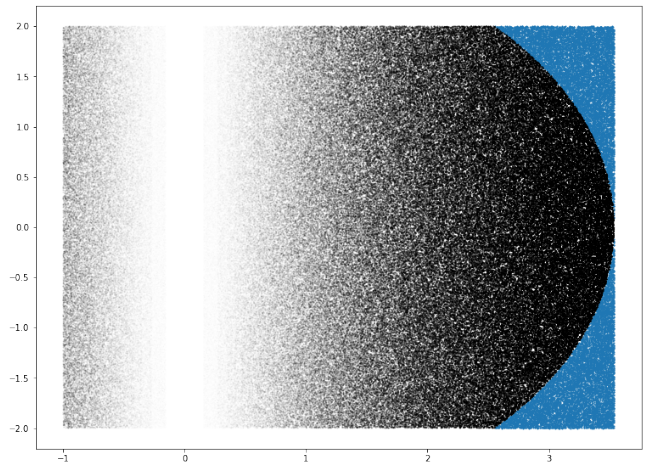


Figure 15: Cross-section of XZ-plane

Here we can see the graphical set-up for this problem. After a random point $X \in \mathbb{R}^3 \sim U^3[0,1)$ is chosen, it is verified whether or not it is contained in the volume of interest. If it is, the value corresponding to the x-coordinate is assigned a density value which is added to the plot on a continuous color spectrum. Similarly, in the algorithm, a line of code is added to track the cumulative sum of the density of the points. This is divided by the number of samples within the volume of interest to give the average density of each point. This is multiplied by the estimate of the volume to give the estimated mass of the truncated ellipsoid.

Included below is the modified algorithm as well as the convergence plot and distribution of results.

```
@njit
def sim_tvellipse(n, empty_array):
    m = 0
    d = 0
    for i in prange(n):
        w = rn.rand(3)
        x = w[0] * (a + 1) - 1
        y = w[1] * (2 * b) - b
        z = w[2] * (4) - 2
        if 2*x**2 + 3*y**2 + z**2 <= 25:
            m = m + 1
            d = d + x**2
        empty_array[i] = (d / m) * (2 * (a + 1) * b * 4 * m / (i + 1))
    return empty_array
```

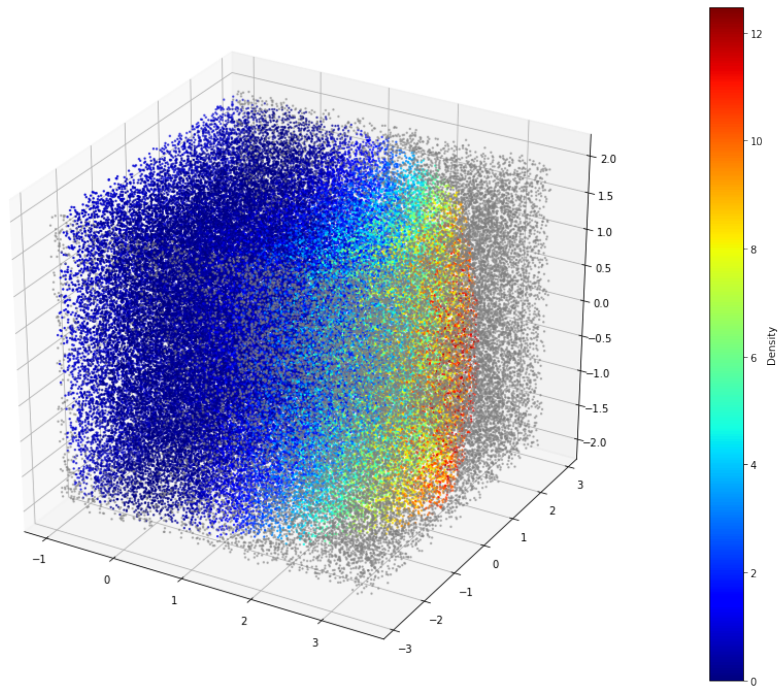


Figure 16: MCI of truncated ellipse - varying density

Again, the inset in Figure 17 shows that the solution is still quite variant towards the end of the simulation. The fact that the solution still varies outside of one standard deviation indicates that a longer simulation should be run. We can see this in comparing the estimated value with the theoretical – there is deviation of 121 grams. However, when multiple simulations are run as in Figure 18, we can see that the deviation drops to a 14 grams (an improvement by an order of magnitude).

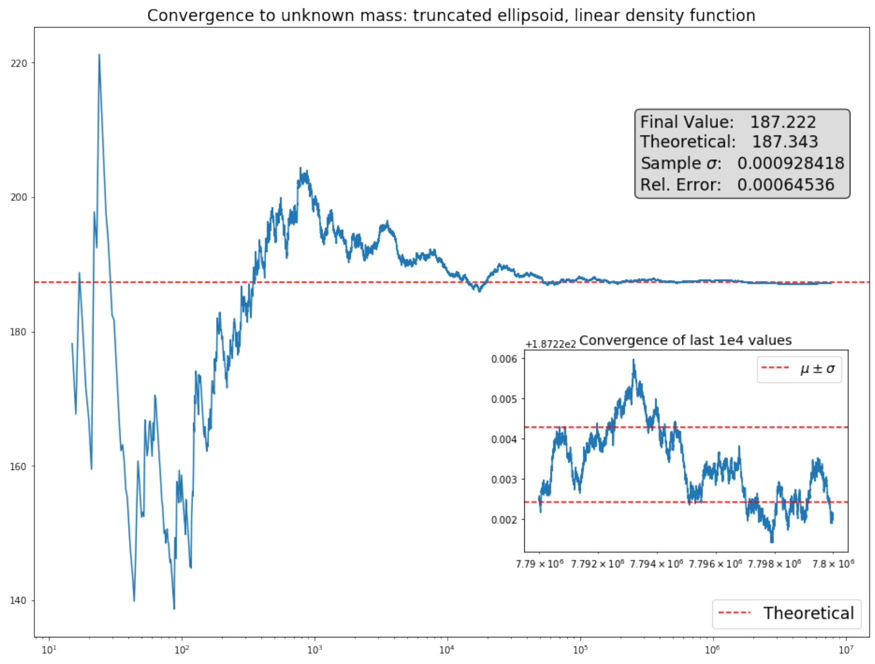


Figure 17

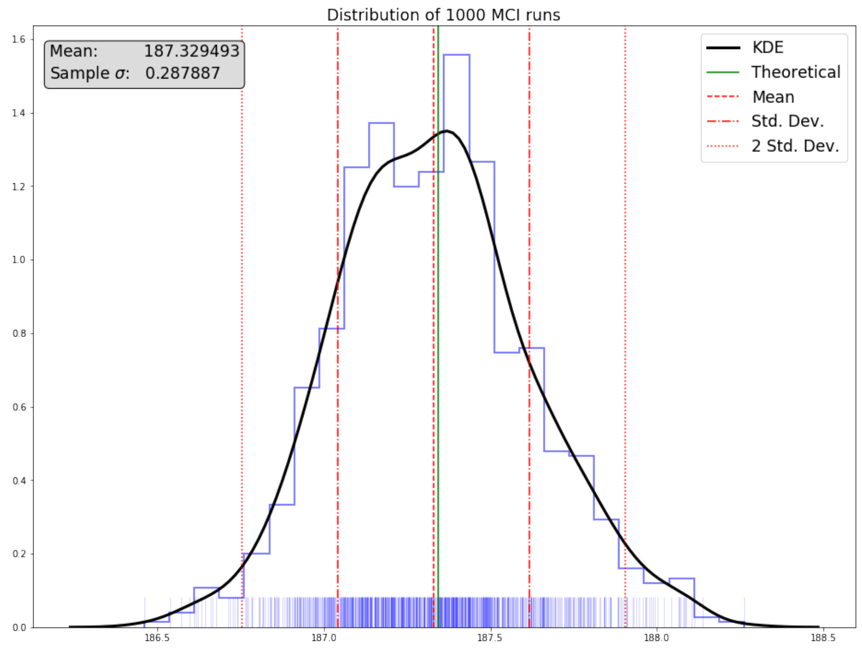


Figure 18: Truncated ellipsoid: variant density

2.4 Examples of the type of problem which works well with MCI but may be difficult or impossible to solve analytically.

The two truncated ellipses above are great examples of these kind of problems, but they aren't impossible to solve analytically, just much more difficult than solving them using numerical techniques. From an engineering standpoint, trying to find the center of mass of any complex shape could well be impossible to solve using integral calculus, but as long as one can define a domain from which to pull random numbers, the problem can be solved by taking volume and density integrals using Monte Carlo techniques. From a business statistics standpoint, MC techniques are used to determine risk analysis and timing of purchases from money markets to decrease the volatility of assets. In the statistics realm, MC techniques can give reasonable estimates to systems with many inputs with large variation.

3 Long-run simulations

In order to get to 5 digits of accuracy, we need on the order of 4e10 data points. This is computationally impractical for the purposes of plotting, so I created an algorithm that would update the estimate at each iteration as opposed to storing all of the estimates. These algorithms were tested for both efficiency and accuracy at the beginning of Appendix A. Here is the basic algorithm:

```
@njit
def sim2(n):
    c = 0
    for i in prange(n):
        x = rn.rand(2)
        if x[0]**2 + x[1]**2 <= 1:
            c = c + 1
    return 4 * c / n
```

This allowed me to take simulations out to an arbitrary sample size. Here are the results for the above four physical systems each taken to 4e10 samples along with the time that they each needed to complete.

Long-run estimates			
Variable	*Estimate*	*Time*	*Percent Error*
π	3.141599611	5145.5114	0.000221 %
Ellipsoid	213.7593151155921	5386.5485	0.000615 %
Trunc. Ellip.	82.84904035310528	5616.1020	0.000169 %
Var. Trunc. Ellip.	187.34382912909197	5437.6499	0.000442 %

Conclusion:

I learned the basics of Monte Carlo integration and found out how powerful a tool it can be after a long struggle with the analytic solution to the truncated ellipsoid. I also went down a bit of a rabbit hole on the general subject of statistics and error analysis as related to Monte Carlo techniques. The statistics that go into the error analysis of Monte Carlo walks are quite complicated, however, the Monte Carlo algorithms are very straight forward given a certain knowledge of coding and computer architecture. I always enjoy projects such as this one which introduce me to wide ranging topics that give me a ton of new information to explore, and I look forward to using these techniques in the future.

That being said, there were issues. As you can see in Appendix D - Old Code, I had to re-do almost all of my runs when I determined that the error conditions were incorrect. I also explored how to make the code more efficient; originally, I was trying to run simulations out to 10^9 while logging every data point into my final arrays. This proved to be very computationally expensive and some of my larger loops would freeze on my (relatively powerful for a laptop) machine. To alleviate this, I would run the sim a number of times (related to how large the arrays were) before logging the data into the arrays, which showed a significant increase (30% or so) on the time taken to run the larger sims. In my second attempt, I'd better learned how to optimize my code, and had found some Python-specific tricks that really helped.

The code included in Appendix A is more recent and reflects a bit more knowledge of efficient coding. In the appendices, I've included both the original notebook and the new notebook.

4 Appendices

A: Code

B: Analytical Solutions

- B.1 — Truncated Ellipsoid
- B.2 — Truncated Ellipsoid - Variant Density
- B.3 — Numerical Solutions

C: Algorithm Timing

D: Old Code

References

Yen-Chi Chen. Monte carlo simulation, 2017. URL `http://faculty.washington.edu/yenchic/17Sp_403/Lec2_MonteCarlo.pdf`.

Wilfrid S Kendall, J-M Marin, and Christian P Robert. Confidence bands for brownian motion and applications to monte carlo simulation. *Statistics and Computing*, 17(1):1–10, 2007.

Jan Palczewski and Andrzej Palczewski. Monte carlo simulation. URL `https://www.mimuw.edu.pl/~apalczew/CFP_lecture4.pdf`.