Application of Monte Carlo Integrals

By Hani Al-Sharif

A Capstone paper submitted to Georgia College and State University in partial fulfillment of the requirements for the degree of Bachelor of Science in Mathematics

Milledgeville, Georgia
Advisor: Dr. Jebessa Mijena

Table of Contents

Acknowledgment

Thanks to my advisor, Dr. Mijena, who has been a great advisor, who was ever patient and guided me continuously during the entire process. Dr. Mijena made this a fun and exciting task and shared my enthusiasm with me, which i must say is contagious. This research that has sparked a new passion in a field of mathematics I have never delved into before. Once starting this research it became apparent how practical it's uses would be especially in other areas that I study, such as my physics research. I'd also like to extend my gratitude to Matthew Dallas a classmate and friend who attempted to keep me focused and continued to remind me of my responsibilities as a research student.

Abstract

Monte Carlo simulation (MCS) is a stochastic process that utilizes random sampling to provide numerical approximations, the use of which varies from financial firms trying to predict the stock market to engineers determining how they should design their systems in regard to society's erratic nature. This research demonstrates the use of MCS in the topic of integration transforming tedious and time-consuming integrands into a body of code that provides a very good approximation to integrals.

Introduction

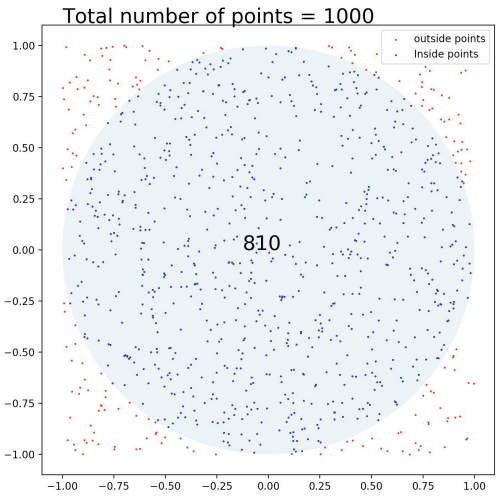
You're looking at an enigma of a problem, no solution presents itself using standard methods. This is where Monte Carlo Simulation comes to play. A tool whose specific purpose is to produce such good approximation that they can be taken as concrete solutions. The origins of Monte Carlo Simulation, harken back to the 1940s and 1950s during the development of Nuclear bombs in the Manhattan Project. After MCS's success in that region of Physics its use was spread further and further, into other regions of application such as Engineering and Economics. Today MCS has a strong presence in Risk Assessment and Stock Analysis as it is fundamentally based in Randomness for which both of these systems are thrive. Essentially Monte Carlo Simulation, takes a collection of random variables (usually of considerable size) and computes them through a process designed for the problem and produces either a specific result or a combination of paths that are likely to occur given the conditions. Our use of Monte Carlo is focused on Integral approximation, in order to assess typically unsolvable integrals. Of course, simple problems will be covered only to move to something that could only come from a comic book, literally.

Application

In order to apply Monte Carlo Simulation, it is necessary to use a programming language; in this research we used Python due to its versatility as a code and its large number of at ready libraries. With every example the code used to provide their respective estimates will be provided. A key point to observe is that a majority of the actual calculation is only a few lines whereas the rest is part of providing a visual for the user.

Estimating Pi

Estimating π - with a Limited Sample Size ($N = 1000$)



Finding π is a very common example used to demonstrate how effective Monte Carlo Simulation can be, this is because of the simplicity of the problem and its solution. Imagine having a square in a sand-pit, and then placing a hoolahoop inside of this square so that the circle is tangent with the four sides of the square. What we then do is take a thousand marbles and scatter them at random, after counting the ratio in the square vs in the circle we can determine the area of the circle. A visual can be see in above. Doing this

with a limited number of points will have an error that is based on the sample size, there for it is better to use a greater sample size; this will be further discussed later on. However, once we have our points counted we can use the ratio of areas to determine an approximation of π.

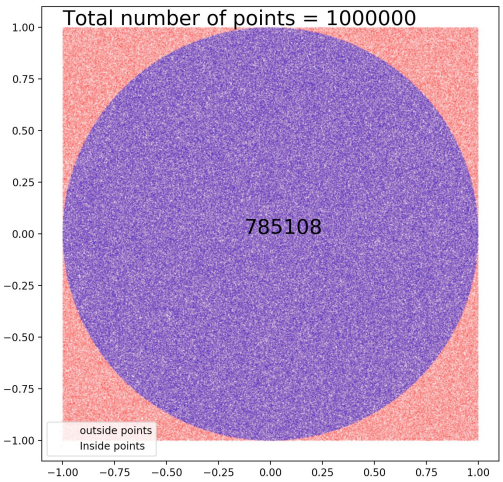
$$\text{Ratio of Areas} = \frac{\text{Area of Circle}}{\text{Area of Square}} = \frac{\pi r^2}{s^2} = \frac{\pi}{4}$$

$$\frac{\pi}{4} \approx 0.81 \Rightarrow \pi \approx 0.81 * 4 = 3.24$$

The approximation is close but not enough that it could be deemed a success, this can be remedied by increasing the sample size to a million or higher.

Estimating π - with a Greater Sample Size ($N = 10^6$)

Since we want greater accuracy and precision we'll run the same program with a greater sampling size N. Doing this we'll give an approximation with more decimal places and greater accuracy. Again this is due to the nature of Monte Carlo Simulation as a statistical process. On the figure to the right it is evident how a vastly larger sample size will produce a more accurate approximation. Our estimation for π with $N = 10^6$ is 3.140432, We could further increase the accuracy by increasing our value for N



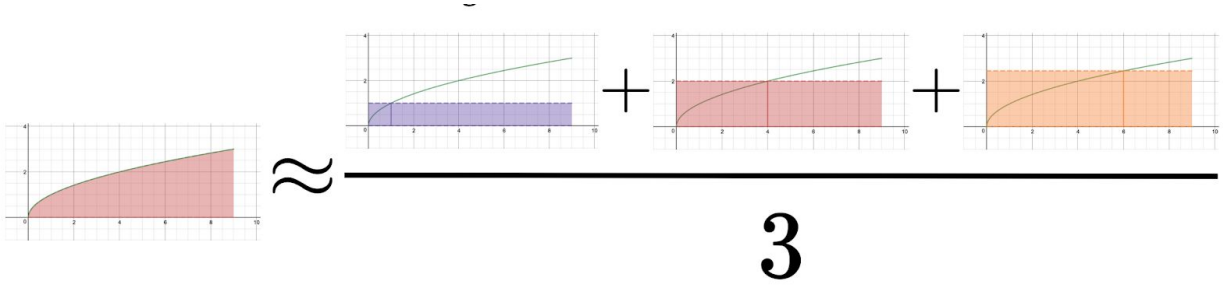
Code: Estimating π

```
import numpy as np
import matplotlib.pyplot as plt
plt.figure(figsize=(8, 8))  # set the figure size
# Calculate the radius of cirlce with radius
radius = 1
N = 1000000  # Use 10000 points
np.random.seed(69)
X = np.random.uniform(low=-radius, high=radius, size=N)  # Random numbers from -1 to 1
Y = np.random.uniform(low=-radius, high=radius, size=N)
plt.scatter(X, Y, c='red', s=0.9, alpha=0.9, label=u'outside points' )
# drawing a circle
crc = plt.Circle([0, 0], radius=radius, alpha=0.1)
fig = plt.gcf()
fig.gca().add_artist(crc)
# calculate the distance from the center
R = np.sqrt(X ** 2 + Y ** 2);
plt.scatter(X[R < radius], Y[R < radius], c='Blue', s=0.9, alpha=0.9, label=u'Inside points')
Abox = (1 * radius) ** 1  # This is the big box
Ninside = 0
Noutside = 0
for i in np.arange(0, N):
   if R[i] <= 1:
      Ninside = Ninside + 1
   else:
      Noutside = Noutside + 1
# Alternative and faster
Ninside = np.sum(R < radius)
Area = Abox * Ninside / N
plt.legend()
plt.text(-0.125,0, Ninside, fontsize=20, color ='black')
plt.text(-1, 1.11, 'Total number of points = {}'.format(N), fontsize =20)
# Area = Abox * Ninside/(Noutside+Ninside)
print ("Area=", Area, "pi=", 4*(Area / radius ** 2))
Picture = plt.show()
# calculate the distance from the center
R = np.sqrt(X ** 2 + Y ** 2);
```

```
plt.scatter(X[R < radius], Y[R < radius], c='Blue', s=0.9, alpha=0.9, label=u'Inside points')
Abox = (1 * radius) ** 1  # This is the big box
Ninside = 0
Noutside = 0
for i in np.arange(0, N):
    if R[i] <= 1:
        Ninside = Ninside + 1
    else:
        Noutside = Noutside + 1
# Alternative and faster
Ninside = np.sum(R < radius)
Area = Abox * Ninside / N
plt.legend()
plt.text(-0.125,0, Ninside, fontsize=20, color ='black')
plt.text(-1, 1.11, 'Total number of points = {}'.format(N), fontsize =20)
# Area = Abox * Ninside/(Noutside+Ninside)
print ("Area=", Area, "pi=", 4*(Area / radius ** 2))
Picture = plt.show()
```

Monte Carlo Integration

Essentially we want to estimate: $I = \int_a^b f(x)dx$



To do this we have to randomly sample values x_1, x_2, ... , x_n from $[a, b]$, and then evaluate those values using $(b-a) \cdot f(x_i)$ for $i = 1, \dots, n$ which is the area of a rectangle with height $f(x_i)$ and a width $(b-a)$ and then find the average of these areas such that,

$I_n = \frac{(b-a)\sum_{i=1}^{n} f(x_i)}{n} = (b-a) \cdot \frac{\sum_{i=1}^{n} f(x_i)}{n} = (b-a) \cdot Average\,(f)$, which will be our approximation.

And so Monte Carlo Integration has specific properties that allow for the approximations to be used consistently. These being that,

- Monte Carlo Estimation is consistent

$$P[\lim_{n\to\infty} I_n = I] = 1$$

- Monte Carlo Estimates are unbiased

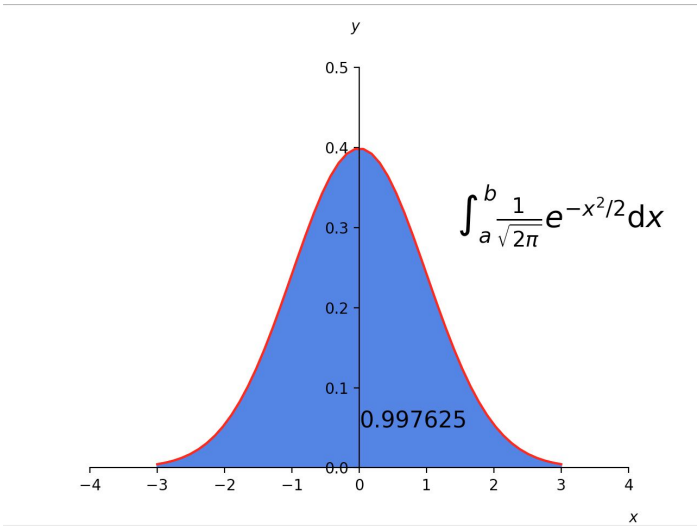
$$\text{Expected Value of } I_n : \; E[I_n] = I$$

- The rate of convergence is proportional to $\frac{1}{\sqrt{n}}$. That is,

$$Std(I_n) \sim \frac{1}{\sqrt{n}}$$

Examples

Example 1: Normal Distribution Curve



The estimate of the probability of standard normal distribution between -3 and 3 is estimated to be 0.997625 with $N = 10^6$.

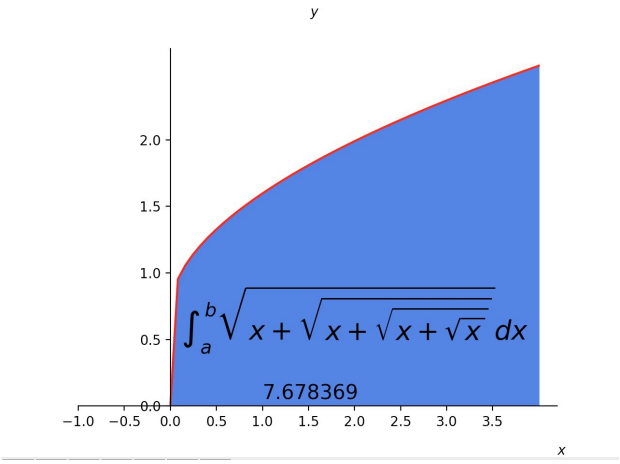
This estimation if very strong when compared to the standard used values of the Normal Distribution Curve from -3 to 3.

Code: Normal Distribution Curve

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon

def f(x):
```

```
    return (1/np.sqrt((2*np.pi)))*np.exp(-((x**2)/2))
aString = input("Enter first bound: ")
a = float(aString)
bString = input("Enter second bound: ")
b = float(bString)

# use N draws
N= 1000000
X = np.random.uniform(low=a, high=b, size=N) # N values uniformly drawn from a to b
Y =f(X)   # CALCULATE THE f(x)
Value= (b-a) * np.sum(Y)/ N;

x = np.linspace(a,b)
y =f(x)

fig, ax = plt.subplots()
plt.plot(x, y, 'Red', linewidth=1.5)
plt.ylim(ymin=0)
print ("The MC Estimation is ", Value)
# Make the shaded region
ix = np.linspace(a, b)
iy = f(ix)
verts = [(a, 0)] + list(zip(ix, iy)) + [(b, 0)]
poly = Polygon(verts, facecolor='0.5', edgecolor='0.5', color='RoyalBlue')

ax.add_patch(poly)
plt.text(3, 0.3, r"$\int_a^b \frac{1}{\sqrt{2\pi}}e^{-x^{2}/2}\mathrm{d}x$",
        horizontalalignment='center', fontsize=20)
plt.text((0.4*(a+b)),0.05, round(Value,6), fontsize=15)
plt.figtext(0.9, 0.005, '$x$')
plt.figtext(0.5, 0.95, '$y$')

ax.spines['left'].set_position(('data', 0.0))
ax.spines['bottom'].set_position(('data', 0.0))
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')

ax.set_xticks([-4,-3,-2,-1,0,1,2,3,4])
ax.set_yticks([0,0.1,0.2,0.3,0.4,0.5])

plt.show()
```

Example 2: Integral of Difficulty



In Mathematics it is common to come across many integrals that are near impossible to evaluate by hand due to the complexity and lack of techniques. If we take the example of $\int_a^b \sqrt{x+\sqrt{x+\sqrt{x+\sqrt{x}}}}\,dx$ we can evaluate it with high accuracy very quickly. The estimation comes out to 7.678369 with $N = 10^6$, a = 0, b = 4, and only takes a matter of seconds to run even when using a million points. Keeping this in mind, the idea of using Monte Carlo in many applications becomes quite common.

Code: Integral of Difficulty

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon
def f(x):
    return np.sqrt(x + np.sqrt(x + np.sqrt(x + np.sqrt(x))))
aString = input("Enter first bound: ")
a = float(aString)
bString = input("Enter second bound: ")
b = float(bString)

# use N draws
N= 1000000

X = np.random.uniform(low=a, high=b, size=N) # N values uniformly drawn from a to b
Y =f(X)   # CALCULATE THE f(x)
```

```
Value= (b-a) * np.sum(Y)/ N;

x = np.linspace(a,b)
y =f(x)

fig, ax = plt.subplots()
plt.plot(x, y, 'Red', linewidth=1.5)
plt.ylim(ymin=0)
print ("The MC Estimation is ", Value)
# Make the shaded region
ix = np.linspace(a, b)
iy = f(ix)
verts = [(a, 0)] + list(zip(ix, iy)) + [(b, 0)]
poly = Polygon(verts, facecolor='0.5', edgecolor='0.5', color='RoyalBlue')

ax.add_patch(poly)
plt.text(2, 0.5, r"$\int_{a}^{b} \sqrt{x +\sqrt{x +\sqrt{x + \sqrt{x}}}}$",
        horizontalalignment='center', fontsize=20)
plt.text(1,0.05, round(Value,6), fontsize=15)
plt.figtext(0.9, 0.005, '$x$')
plt.figtext(0.5, 0.95, '$y$')

ax.spines['left'].set_position(('data', 0.0))
ax.spines['bottom'].set_position(('data', 0.0))
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')

ax.set_xticks([-1,-0.5,0,0.5,1,1.5,2,2.5,3,3.5])
ax.set_yticks([0,0.5,1,1.5,2,])


plt.show()
```

Example 3: Triple Integral of Difficulty

Taking the difficulty up another level by evaluating an integral to the third dimension. Again this integral is hard due to its complexity and the fact that its to the third dimension. The

method to estimate this integral is similar to that of a single dimension integral however we utilize an array in the code. In the example, We assess the integral $\int_0^2 \int_0^2 \int_0^2 (x+y)e^{(x+y+z^2)}\,dxdydz$ with an $N = 10^6$

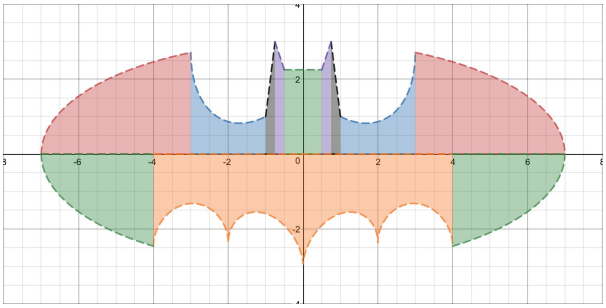
Code: Triple Integral of Difficulty

```
ımport numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon

def f(x,y,z):
   return (x+y)*np.e**(x+y+(z**2))

# use N draws
N= 1000000
sum = 0
np.random.seed(5)
X = np.random.uniform(low=0, high=2, size=N) # N values uniformly drawn from a to b
Y = np.random.uniform(low=0, high=2, size=N) # N values uniformly drawn from c to d
Z = np.random.uniform(low=0, high=2, size=N) # N values uniformly drawn from 0 to 2Pi
#  CALCULATE THE f(x)

A = [X,Y,Z]
for i in range(0,N):
   k = f(A[0][i],A[1][i],A[2][i])
   sum = sum +k

Value = (sum*8)/N

print(sum)
print("A =", A)
print(Value)
```

The estimate value of the triple integral is 1763.9459 with a random sample of size $N = 10^6$.

Example 4: Batman Curve

To culminate we have our comic book hero assessed by mathematics and so, the area of the Batman symbol was found using Monte Carlo Estimation. The result was that a piecewise whose integral would take a lot of time and some serious effort can be reduced to a runtime of 1.986 seconds. Looking at the piecewise it is clear how hard it would be to take the integral of such a function:



$$y = 3|x| + 0.75 \quad \{-0.75 < x < -0.5\} \cup \{0.5 < x < 0.75\}$$

$$y = 2.25 \quad \{-0.5 < x < 0.5\}$$

$$y = 9 - 8|x| \quad \{-1 < x < -0.75\} \cup \{0.75 < x < 1\}$$

$$y = \left(\frac{6\sqrt{10}}{7} + (1.5 - 0.5|x|)\right) - \left(\frac{6\sqrt{10}}{14}\right)\sqrt{4 - (|x| - 1)^2} \quad \{-3 < x - 1\} \cup \{1 < x < 3\}$$

$$y = \sqrt{9\left(1 - \left(\frac{x}{7}\right)^2\right)} \quad \{-7 < x < -3\} \cup \{3 < x < 7\}$$

$$y = \left(\left|\frac{x}{2}\right| - \left(\frac{3\sqrt{33} - 7}{112}\right)x^2 - 3\right) + \sqrt{1 - (||x| - 2| - 1)^2} \quad \{-4 < x < 4\}$$

When using Monte Carlo Estimation on a piecewise, it is as simple as adding the estimations of each individual part into a whole.

Code: Batman Curve

```
import numpy as np
import matplotlib.pyplot as plt
import sympy
from sympy import symbols
```

```
#The Number of Random Points
import timeit

start = timeit.default_timer()

N = 10000000;
np.random.seed(69)
def a(x):
  return 2.25
aa=0
ab=0.5
Xa = np.random.uniform(low=aa, high=ab, size=N)
Ya = a(Xa)
ValueA = (ab - aa) * 2.25;

def b(x):
  return 9 - 8*abs(x)
ba=0.75
bb=1
Xb = np.random.uniform(low=ba, high=bb, size=N)
Yb = b(Xb)
ValueB = (bb - ba) * np.sum(Yb)/ N;

def c(x):
  return 3*abs(x) + 0.75
ca=0.5
cb=0.75
Xc = np.random.uniform(low=ca, high=cb, size=N)
Yc = c(Xc)
ValueC = (cb - ca) * np.sum(Yc)/ N;

def d(x):
  return np.sqrt(9*(1-(x/7)**2))
da_upper=3
db_upper=7
Xd_upper = np.random.uniform(low=da_upper, high=db_upper, size=N)
Yd_upper = d(Xd_upper)
ValueD_upper = (db_upper - da_upper) * np.sum(Yd_upper)/ N;

da_lower=4
db_lower=7
Xd_lower = np.random.uniform(low=da_lower, high=db_lower, size=N)
Yd_lower = d(Xd_lower)
ValueD_lower = (db_lower - da_lower) * np.sum(Yd_lower)/ N;

def e(x):
  return (abs(x/2)-((3*np.sqrt(33)-7)/112) * x**2 - 3) + np.sqrt(1-((abs(abs(x)-2))-1)**2)
```

```
ea=0
eb=4
Xe = np.random.uniform(low=ea, high=eb, size=N)
Ye = e(Xe)
ValueE = ((eb - ea) * np.sum(Ye)/ N)*-1;

def f(x):
  return ((6*np.sqrt(10)/7) + (1.5 - 0.5*abs(x))) - (6*np.sqrt(10)/14) * np.sqrt(4-(abs(x)-1)**2)
fa=1
fb=3
Xf = np.random.uniform(low=fa, high=fb, size=N)
Yf = f(Xf)
ValueF = (fb - fa) * np.sum(Yf)/ N;

BatmanArea = 2*(ValueA + ValueB + ValueC + ValueD_lower + ValueD_upper + ValueE + ValueF)
print(BatmanArea)
stop = timeit.default_timer()

print(ValueA)
print(ValueB)
print(ValueC)
print(ValueD_upper)
print(ValueD_lower)
print(ValueE)
print(ValueF)

print('Time: ', stop - start)
```

We have an approximate area of Batman curve to be 48.42398 square units with a random sample of size 10^7.

Conclusion

The ability to make sense of randomness is quite something, and that ability is demonstrated time and time again with Monte Carlo Simulations. Functionality of such a process is endless and clearly of importance since a lot of the worlds problems contain endless and random variables. From Economics to Engineering it is understandable as to

why Monte Carlo is a popular tool. Further from those fields, it can even be seen as a base for the starting of AI systems that could use Monte Carlo Markov Chains to determine how to act as if it were a brain. Societies erratic nature could be 'predicted' even in such a way to guide people the best and most efficient behavior in many areas. All the possibilities are exciting and are due to the ability to take something random and give it meaning.

References

- Weisstein, Eric W. "Batman Curve." From *MathWorld*--A Wolfram Web Resource. http://mathworld.wolfram.com/BatmanCurve.html
- Pavlos Protopapas. "Advanced Scientific Computing: Stochastic Optimization Methods. Monte Carlo Methods for Inference and Data Analysis" From Harvard University Online Course. http://iacs-courses.seas.harvard.edu/courses/am207/index.html
- Unknow. "Monte Carlo, and Other Kinds of Stochastic Simulation" http://bactra.org/notebooks/monte-carlo.html
- Harrison, Robert L. "Introduction To Monte Carlo Simulation" *AIP conference proceedings* vol. 1204 (2010): 17-21.