White Paper Session

Topic: Monte Carlo Simulation

Name: Sourajyoti Das
Roll no.: 20CH30031

Introduction

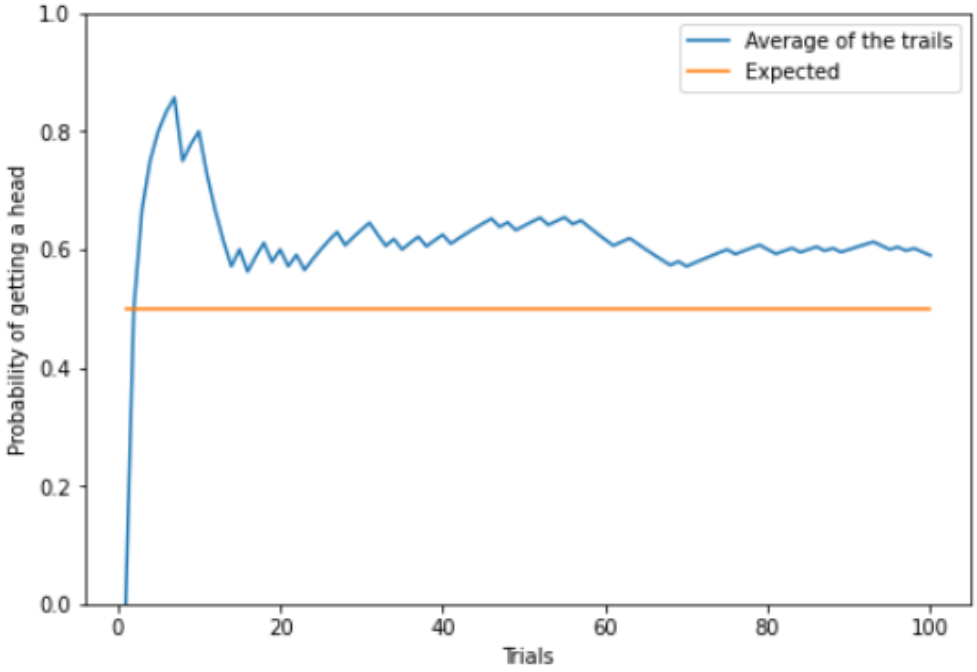
Monte Carlo methods, or experiments, are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. The concept is to make use of randomness in a system to get to a deterministic system and a fixed answer. Monte Carlo methods are used to simulate systems to predict their behaviour. For example, in physics, systems such as fluids, disordered materials, strongly coupled solids and cellular structures can be simulated using these methods. In mathematical areas, these methods can be used for problems of optimization, numerical integration and probability distribution. These methods also find place in financial areas which can predict failures, cash flow, cost overruns and risks with good accuracy. Monte Carlo methods find usage in various other fields as well where the exact outcome is unknown.

Now, let's come to a few things like computational algorithms, simple random sample and deterministic systems. Computational algorithms are a set of sequence of well-defined instructions, arithmetic or otherwise, used to solve a problem. In a simple random sample of a given size, each subset of a sample space has an equal probability of being selected. A deterministic system is a model in which from a given initial condition or state will always produce the same output after the requisite processes.

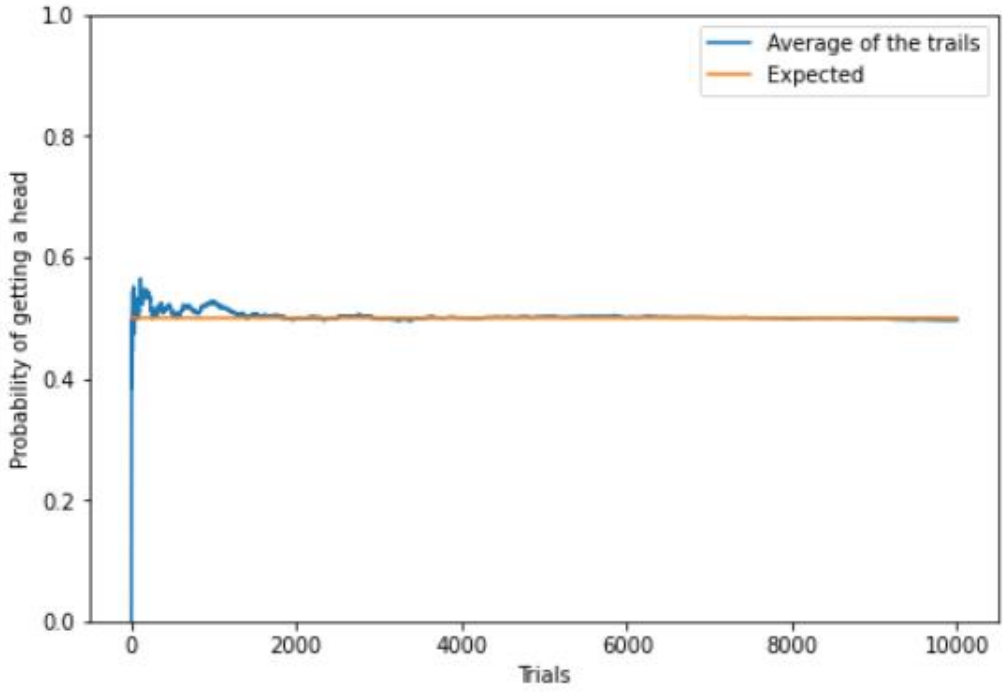
In principle, Monte Carlo methods can be used to solve any problem having a probabilistic interpretation, i.e, where each outcome is equally likely. This is so, due to the repetitive nature of the method which supports the theory using the law of large numbers. The law of large numbers is a theorem which simply states that the average of the results obtained from a large number of trails should be close to the expected value and as the number of trails keeps on increasing, the difference between the expected value and observed value tends to zero.

Here is a simple example to demonstrate the law of large numbers. Here, a simulation of a coin toss has been performed using Monte Carlo simulation. For a fair coin, the probability of the occurrence of a head should be equally likely to the occurrence of a tails. In the simulation, a toss is simulated to result in either a head or a tail. The occurrence of a head or a tail is randomly done. A graph is then plotted as the probability of getting a head to the number of trails. Ideally, this value is expected to be half. The algorithm performs one hundred, ten thousand and one million tosses respectively to make the plots. It is evident from the graphs that the average

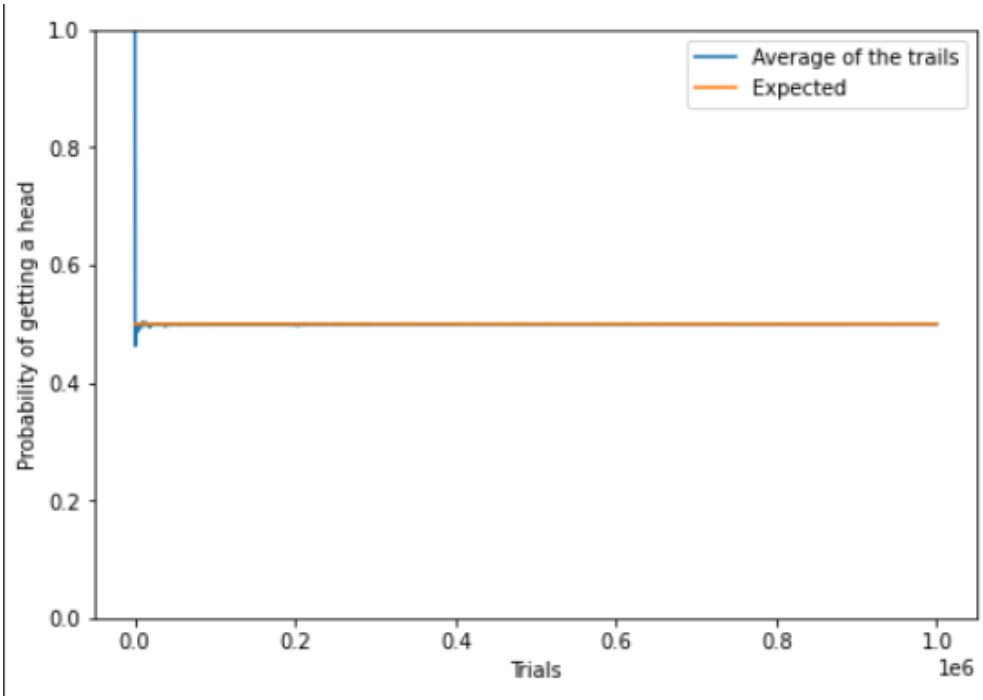
essentially becomes half as the number of trails is increased. Also, note that this is a problem with probabilistic interpretation.



Plot 1: With one hundred tosses



Plot 2: With ten thousand tosses



Plot 3: With one million tosses

History and Definition

Monte Carlo simulation was initially invented to solve Buffon's needle problem to estimate the value of π, pi. The modern version of Monte Carlo Simulation was invented by Stanislaw Ulam, inventor of modern version of Markov Chain Monte Carlo, and John von Neumann, who programmed a special computer to perform Monte Carlo calculations.

Ulam's approach was based on a list of truly random numbers whereas von Neumann developed a method to calculate pseudorandom numbers. Von Neumann's approach was the faster one. The name "Monte Carlo" comes from the Monte Carlo Casino in Monaco. This theory has been since extensively studied and has now found application in various fields.

Shlomo S. Sawilowsky gave definitions distinguishing between a simulation, a Monte Carlo method, and a Monte Carlo simulation. Let us understand these using an experiment of finding the probability of getting a head in a coin toss.

- Simulation is a fictitious representation of reality. According to the experiment, we draw one pseudo-random uniform variable from the interval [0,1] and designate head if the value is greater than or equal to 0.5, otherwise it is assigned a tail.
- Monte Carlo method is a technique that can be used to solve a mathematical or statistical problem. This uses a large number of outcomes. According to the experiment, we can pour in a large number of coins and getting the ratio of the

occurrence of heads to the total number of coins. This is Monte Carlo method of determining the behavior of repeated coin tosses, but this is not a simulation.

- Monte Carlo simulation uses sampling to obtain the statistical properties of some phenomenon. According to the experiment, this is done drawing of large number of pseudo-random uniform variables from the interval [0,1] and assigning a head if the value is greater than or equal to 0.5, otherwise designating it a tail. This can be done by taking the whole lot at once or taking each toss one by one. The probability of getting heads is calculated accordingly. This is a Monte Carlo simulation of the behavior of repeatedly tossing a coin. The plot of probability of head versus number of tosses done is shown above.

The code used in the Monte Carlo simulation whose plots are shown above:

```
# imports needed
import random
import matplotlib.pyplot as plt

# number of trails in each run
o = [100, 10000, 1000000]

random.seed()

# each iteration of this loop runs one Monte Carlo simulation
for k in o:
    l = 0
    x = []
    y = []
    m = []

    # main processing of the Monte Carlo simulation
    for i in range(1, k + 1):

        # getting a random value uniformly in the range [0,1]
        s = random.uniform(0, 1)

        # assigning the value as head for being greater than or
equal to 0.5
        # otherwise it is made a tail
        if s >= 0.5:
            t = 1
        else:
            t = 0

        # calculating the cumulative average after each trail
        l = (t + l * (i - 1)) / i
        x.append(i)
        y.append(l)
        m.append(0.5)

    # plotting
```

```
ptl = plt.figure()
pt = ptl.add_axes([0, 0, 1, 1])
pt.plot(x, y, label="Average of the trails")
pt.plot(x, m, label="Expected")
plt.xlabel("Trials")
plt.ylabel("Probability of getting a head")
pt.set_ylim(0, 1)
pt.legend()
plt.show()
```

Random and pseudo-random numbers

Let us see a bit as to what are pseudo-random numbers. For this, let us see how random numbers are generated. The first method is based on physical process. Physical phenomenon which are expected to be random are taken as the source of randomness, for example radioactive decay, cosmic background radiation, atmospheric noise and more. The hardware consists of device that converts energy from these sources into electrical signals, followed by an amplification and conversion of the analog signal into digital number. Such numbers are said to be true random numbers.

Another way of generating apparently random numbers involves computational algorithms. These are called apparently random as the value of the number depends on the initial value, or the seed value or key. The key is then processed via an algorithm to generate a random number. These number sequence becomes repetitive with a period and are deterministic in nature. These numbers are called pseudo-random numbers. An example of algorithm used to generate such a number is the Mersenne Twister algorithm which is used in python.

Bias in results from Monte Carlo

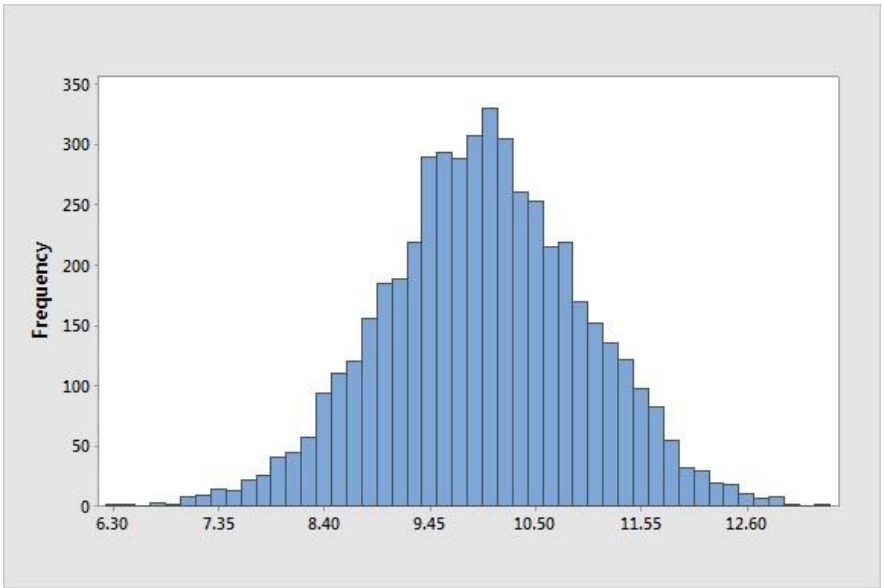
First let us know what an estimator is. An estimator is a rule for calculating an estimate of a given quantity based on observed data. The bias of an estimator is the difference between this estimator's expected value and the true value of the parameter estimated. If the bias is zero, or the estimator is unbiased, then we can say that if the sample size is high enough, the estimator will converge eventually to the value. How quick the convergence happens depends on the variance, i.e. lower the variance, quicker the convergence happens.

Monte Carlo uses the mean of all the outcomes from each individual simulation done. This is denoted by $\bar{X}_n$ or $\delta(X)$.

For an unbiased estimator, $\bar{X}_n \to \theta$ for $n \to \infty$.

And we have already discussed this property of Monte Carlo simulations as it uses the law of large numbers to converge to the expected estimate. So, we can say that Monte Carlo is an unbiased estimator.

For unbiased estimators, the frequencies of different outcomes generated forms a bell curve histogram with the expected estimate being near the middle of the curve. Monte Carlo simulations make such bell curves as well. Most of the situations gives the bell curve spike very close to the true value of the estimation. This type of distribution of the frequencies of any data is known as a normal distribution of the data.



A sample histogram with bell curve characteristics

Let us see an implementation of this with an example. Suppose that we want to find the value of pi, π. There are various methods to do this like Buffon's needle problem. Here, let us follow a much simpler approach to do the job.

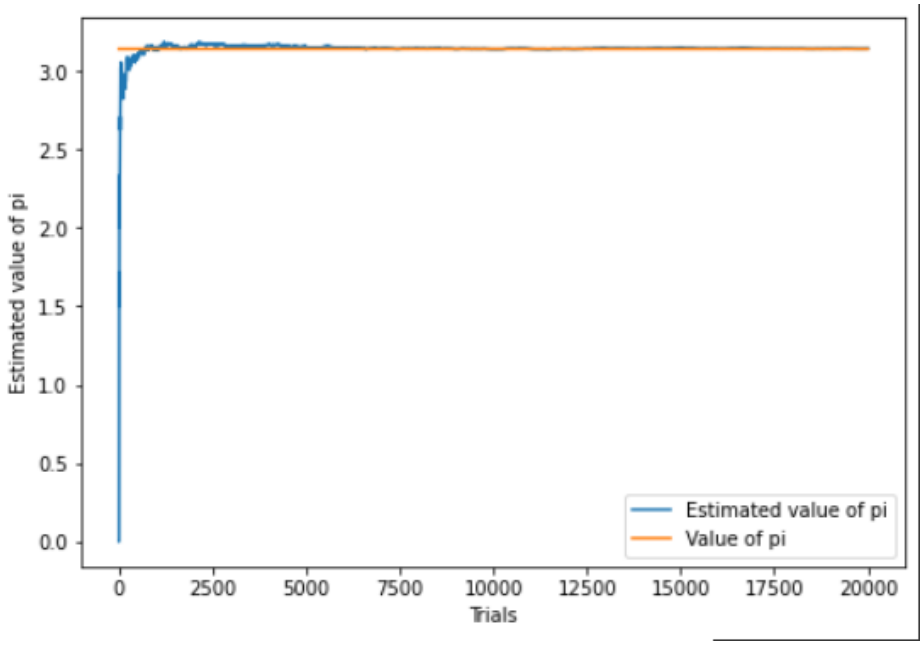
Let us take a square of side length as unity. In this, take an inscribed quarter of a circle of radius unity and center as one of the corners of the square. In this, we know that the area of the square is unity and the area of the quarter is $\pi/4$ units. Therefore, four times the ratio of the area of the quarter to the area of the square gives the value of π.

To make this into a Monte Carlo simulation, we generate random points inside the square and see if the point falls inside the quarter or not. Basically, the motive is to add up all such generated points to get the whole area of the square covered and then take the ratio of the number of points inside the quarter to the total number of points (as all the points fall inside the square) and multiply it with 4 to get the value of π. In practice, it is not possible to generate all the points as the number of points are infinite and the simulation will run forever. But taking an adequate number of randomly generated points, say 20,000 points can give a close approximation nearly up to second digit as well at times.

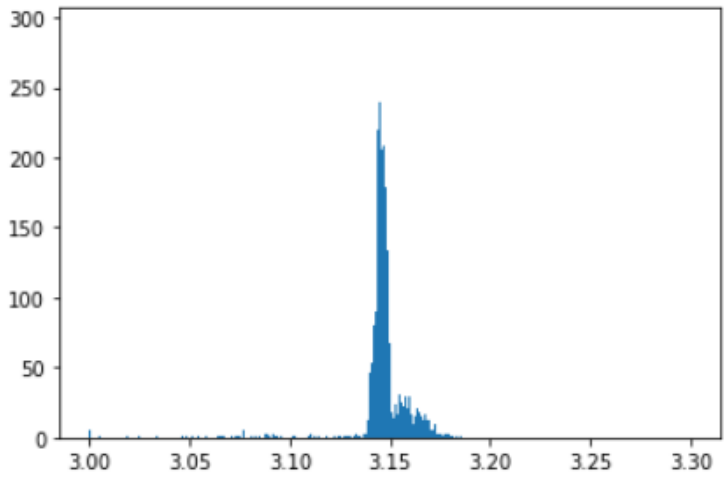
Here is a simple python code to do the above Monte Carlo simulation. The plot is done between the estimated value of π to the number of trails to do so. A histogram has been plotted for the values of π obtained after each try. Also, the points which were taken has been shown in another graph.

```
# imports needed
import matplotlib.pyplot as plt
import numpy as np
import random

# number of trails
k = 20000

l = 0
x = []
y = []
m = []
x1 = []
x2 = []
y1 = []
y2 = []

random.seed()

# main processing of the Monte Carlo simulation
for i in range(1, k + 1):

    # generating a random uniform point in the square space of
one unit area
    xx = random.uniform(0, 1)
    yy = random.uniform(0, 1)

    # checking if the generated point falls inside (or on) the
quarter
    if ((xx ** 2) + (yy ** 2)) ** 0.5 <= 1:
        l += 1
        x1.append(xx)
        y1.append(yy)
    else:
        x2.append(xx)
        y2.append(yy)

    # calculaitng the estimated pi
    y.append(4 * l / i)
    x.append(i)
    m.append(3.14159265359)

# plotting
a = np.arange(3, 3.3, 0.0001)
ptl = plt.figure()
pt = ptl.add_axes([0, 0, 1, 1])
```

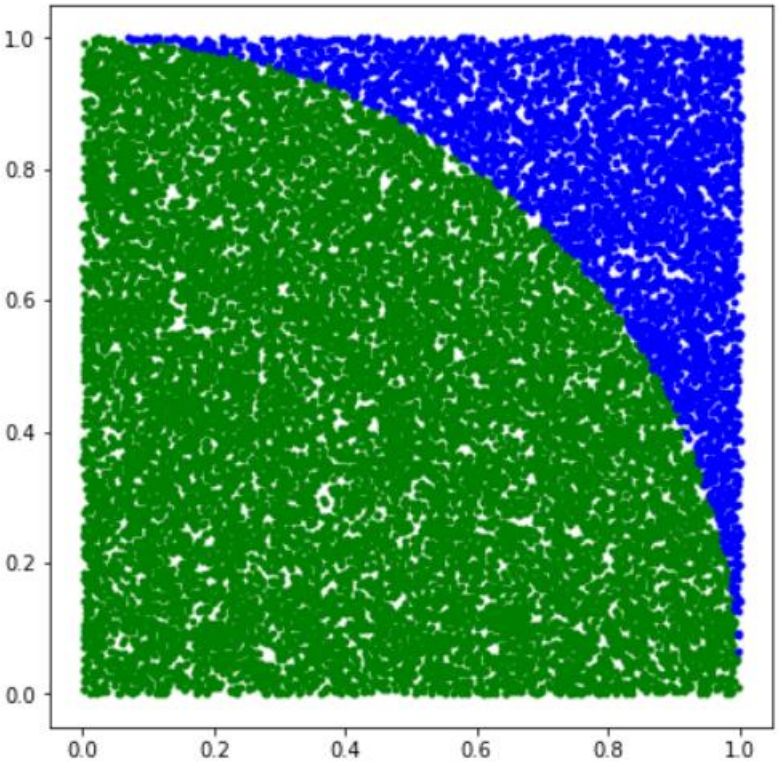
```
pt.plot(x, y, label="Estimated value of pi")
pt.plot(x, m, label="Value of pi")
plt.xlabel("Trials")
plt.ylabel("Estimated value of pi")
plt.legend()
fig, ax = plt.subplots()
ax.hist(y, bins=a)
plt.show()
ptl = plt.figure()
pt = ptl.add_axes([0, 0, 1, 1])
pt.plot(x1, y1, ".", color="green")
pt.plot(x2, y2, ".", color="blue")
ptl.set_figheight(5)
ptl.set_figwidth(5)
plt.show()

# value of pi that is estimated after the run of the Monte
Carlo simulation
print(f"Value of pi estimated is : {4*l/k}")
```



The plot for the Monte Carlo simulation



The histogram for the Monte Carlo simulation



The randomly taken points

The estimated value of π is 3.1448. Actual value of π is 3.14159265359.

In the histogram, the simulation results in a very rough bell curve kind of plot as can be seen. It is roughly a normal distribution. Also, it can be said from the line plot that the value of the estimation approaches the real value as the number of trails increases.

Number of trails for a Monte Carlo simulation

Till now, we have seen the example of use of Monte Carlo using the tossing a coin where the outcome of the experiments is binary. But, to be useful in various other

real-life problems, a Monte Carlo simulation first requires the development of a mathematical model which can represent the real-world situation of the scenario to be dealt with as accurately as possible. This includes the representation of various sources of random variability by appropriate probability models and the estimation of the parameters to be entered into the model. This model is then constructed into a computer program which has a domain for the samples, an algorithm to perform and boundaries. Then simulations are run by the computer to represent randomly selected real-world trails. So, this brings us to understand how many trails are needed for solving a problem.

As Monte Carlo simulation involves random values, so naturally, the results of a simulation are subject to statistical fluctuations. Hence, it is clear that Monte Carlo estimate will not be exact in real life, but will always have an associated error band. Assuming the algorithm in the model is highly accurate to the actual processes, with larger number of trails one can get smaller statistical error leading to a more precise final answer. The cost of performing a trail increases with the complexity of the model. Also, more the number of trails, more is the cost. Thus, one would want to conduct the minimum number of runs on the simulation to achieve the desired degree of precision.

One common way of terminating the simulation is to running it until the observed fluctuation of the estimate gets a so-called stability to the required degree of precision. This method, though might probably give the required result, has no way by which one can know the number of trails required beforehand and might also end up taking more runs than necessary.

Another way to find the number of trials required to get the true average, μ, of the required estimate is to use the formula,

$$n = \left[\frac{z_{(1+\gamma)/2}\sigma'}{E}\right]^2$$

Where, n is the required number of trails. E is the maximum allowable error in estimating the true mean. γ is the desired probability or confidence level that the estimated mean $\bar{X}$ does not differ from μ by more than $\pm E$. γ is mostly taken either as 95% or 99%. σ' is an initial estimate of the standard deviation of the data. $z_{(1+\gamma)/2}$ is the $100[(1+\gamma)/2]$ percent point of a standard normal distribution. The adequacy of this expression depends on the closeness of the estimate σ' to the true standard deviation σ. If a one-sided error band is required on μ rather than a two-sided one, z_γ is used in the formula in place of $z_{(1+\gamma)/2}$. The table below gives the values of z_γ and $z_{(1+\gamma)/2}$ commonly used.

γ	$z_{(1+\gamma)/2}$	z_γ
95%	1.96	1.65
99%	2.58	2.33

Another method can be with respect to the range of error that the user puts as permissible on the value obtained. This will not per say be the error from the actual value. This implementation follows a running squared mean error of the last few (say 20) estimates. A limit on this will be set upon reaching which, the simulations will stop to produce the final result. Here by reaching the limit refers to the error being less than the given limit. This will bring about how stable the estimate gets. Also note that this error is not the difference between the actual value and the estimated value. This only checks the stability of the prediction of the estimate.

Let us try this method to solve an integral using Monte Carlo simulations.

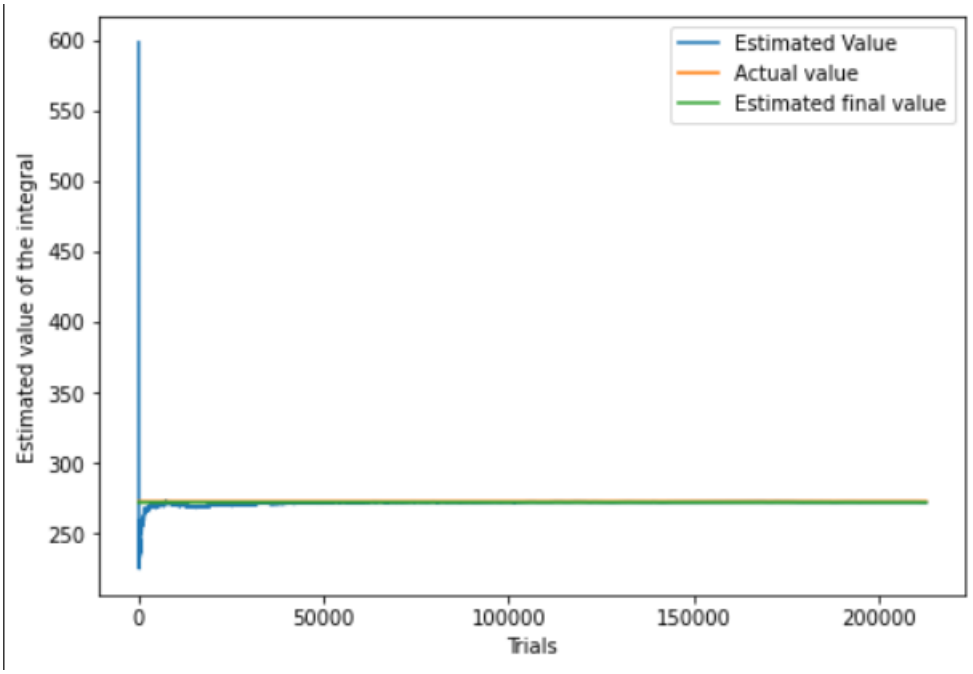
We will be evaluating the integral $\int_{0.2}^{7.7} \left(\frac{\sin x}{x} + x^{3.2} + e^{-5.4x}\right)^{0.73} dx$

```
# imports needed
import matplotlib.pyplot as plt
import math
import random

# inputs
error = float(input("Enter the error limit in the decimals: "))
value = 272.745090535
starting = 0.2
ending = 7.7

l = 0
i = 0
x = []
y = []
final = []
m = []
streak = 0
err = []
domm = ending - starting

# formula
def formula (a):
  b = ((math.sin(a))/a + a**3.2 + math.exp(-5.4*a))**0.73
  return b

# main processing of the monte carlo
while True:
  i = i + 1
  xx = random.uniform(starting, ending)
  yy = formula(xx)
  l = (l*(i-1) + yy*domm)/(i)
  x.append(i)
  y.append(l)
  s = 0
  if (i>20):
    for j in range(i-22, i):
      s = s + (y[j]-y[j-1])**2
```

```
    s = s/20
    s = s**0.5
    if s<error:
      streak = streak + 1
    else:
      streak = 0
  err.append(s)
  if streak==20:
    break

for j in range(i):
  final.append(l)
  m.append(value)

# plots
print()
ptl = plt.figure()
pt = ptl.add_axes([0,0,1,1])
pt.plot(x,y,label='Estimated Value')
pt.plot(x,m, label='Actual value')
pt.plot(x,final, label='Estimated final value')
pt.legend()
plt.xlabel('Trials')
plt.ylabel('Estimated value of the integral')
plt.show()
print()

# prints
print(f'Estimated Value = {l}')
print(f'Actual value= {value}')
if l>value:
  s = l - value
else:
  s = value - l
print(f'The error% in calculation = {(1 -(l/value))*100}')
print(f'The number of iterations needed = {i}')
```

The error limit was set as 0.001.

The estimated value was 272.0575. The actual value is 272.7451. 212923 iterations were needed by the simulation.



The plot of Estimated value of the integral versus the number of trials

Monte Carlo simulation in finance

Monte Carlo simulation is a stochastic method to account for the inherent uncertainty in our financial models. It has the benefit of forcing all engaged parties to recognize this uncertainty and think about probabilities rather than simple views.

An example of use of Monte Carlo simulation can be for portfolio management. A portfolio is a collection of financial investments like stocks, bonds, commodities, cash, cash equivalents and their fund counterparts that a person owns. An analyst factors into a distribution of reinvestment rates, inflation rates, asset class returns, tax rates and possible lifespan of the client to determine the size of the portfolio that will be needed as per the client's demands. The analyst can use Monte Carlo simulation to determine the expected value and distribution of portfolio after a certain date. This is enabled by taking in multi-period view of the portfolio. Using various asset allocations with varying degrees of risks, different corelation between assets and distribution of a large number of factors into account like savings and expenditure, the probability of running out of funds is estimated. The client's risk and return profile is an important factor influencing portfolio management.

Another example can be the use of Monte Carlo simulation in modelling components of cash flow. Cash flow refers to the net amount of cash and cash equivalents being transferred in and out of a company. To maximize the free cash flow (FCF), which is the cash generated by a company form its normal business operations after subtracting any money spent on capital expenditure (CapEx), is the motive of any company. The net present value (NPV) of a company, which is the difference between the present value of cash inflows and the present value of cash outflows over a period

of time is generated in large numbers by considering various associated factors using Monte Carlo methods. This shows an investor the estimated probability that NPV will be greater than zero which will help the investor decide on investing on the company.

Monte Carlo simulation in stock market

Now let us see an example of using the concept of Monte Carlo simulation in stock markets. We will use a simple Monte Carlo simulation to predict the future price of a stock.

Here we will use one of the most common models in finance, the geometric Brownian motion or GBM as the prices at times do resemble a random walk. Geometric Brownian motion is a Markov chain concept. So, following this, we say that the next step from the current step is independent of all the steps taken before. Simply put in this context that the price of the stock in the next step from the current one will not be influenced by any of the prices before the current price. GBM is also consistent with the weak form of the efficient market hypothesis. This states that all the past prices of a stock are reflected in current stock price and the next movement in price is not totally dependent on this price.

The formula used for this simulation will be as follows

$$\Delta S = S \times (\mu \Delta t + \sigma \varepsilon \sqrt{\Delta t})$$

Here, ΔS is the change in stock price. S is the stock price. μ is the expected return. σ is the standard deviation of returns. ε is a random variable. Δt is the elapsed time period. Here, the term $\mu \Delta t$ is called drift and $\sigma \varepsilon \sqrt{\Delta t}$ is known as shock. So, for each time period, this model assumes that the price will drift up by the expected return then the drift will be shocked by a random shock. The shock can be positive as well as negative. The shock is a simple way of scaling the standard deviation.

The code in here tries to give an estimate of the closing price at the end of a month from current date, given the historical data of the previous month from the current date. This is also known as the historical approach. The data is taken for the stocks of Reliance Industries Limited from date 10th November, 2021 to 10th December, 2021.

The code used is given as follows:

```
# function to change the returns into new set of returns
def return_returns(rows, returns, add):
    temp_returns = []
    if len(returns) == (rows - 1):
        temp_returns.append(returns[0])
    for i in range(1, len(returns)):
        temp_returns.append(returns[i])
    temp_returns.append(add)
    return temp_returns


# function to calculate mean and standard deviation
```

```
def mean_and_sigma(rows, returns):
    limit = len(returns)
    mu = 0
    for i in range(limit):
        mu += returns[i]
    mu /= rows
    sigma = 0
    for i in range(limit):
        sigma += (returns[i] - mu) ** 2
    sigma /= rows
    sigma = sigma ** 0.5
    return mu, sigma


# function for next stock price and final return
def values(returns, rows, so, timestep):
    mu, sigma = mean_and_sigma(rows, returns)
    e = random.uniform(-2, 2)
    return_ = mu / timestep + sigma * e * (timestep ** -0.5)
    returns_ = return_returns(rows, returns, return_)
    s = so * (return_ + 1)
    return returns_, s


# plots
def plotting(opens, closes, paths, rows, iters, dates,
prev_dates):

    # plotting historical data
    f = plt.figure()
    f.set_figwidth(20)
    f.set_figheight(10)
    plt.plot(prev_dates, closes, label="Closes")
    plt.plot(prev_dates, opens, label="Opens")
    plt.title("Historical data", fontweight="bold")
    plt.xlabel("Timesteps")
    plt.ylabel("Stock closing price")
    plt.legend()
    plt.show()

    # plotting the paths generated and making a list of the
price at the end of the time period
    # also making the list for an average plot
    finals = []
    avg_path = []
    f = plt.figure()
    f.set_figwidth(20)
    f.set_figheight(10)
    for i in range(rows + 1):
        avg_path.append(0)
    for y in paths:
```

```
            plt.plot(dates, y)
            finals.append(y[rows])
            for i in range(rows + 1):
                avg_path[i] += y[i]
        for i in range(rows + 1):
            avg_path[i] /= iters
        plt.title("Generated paths", fontweight="bold")
        plt.xlabel("Timesteps")
        plt.ylabel("Stock closing price")
        plt.show()

        #plotting the cumulative average of the final stock prices
        cumm = 0
        xx = []
        yy = []
        for i in range(len(finals)):
          cumm = (cumm*i + finals[i])/(i+1)
          xx.append(i)
          yy.append(cumm)
        f = plt.figure()
        f.set_figwidth(10)
        f.set_figheight(5)
        plt.plot(xx,yy)
        plt.title('Cumulative average of final stock price',
fontweight = 'bold')
        plt.xlabel('Trails')
        plt.ylabel('Cumulative average of final stock price')
        plt.show()

        # plotting the average path
        f = plt.figure()
        f.set_figwidth(20)
        f.set_figheight(10)
        plt.plot(dates, avg_path)
        plt.title("Estimated path", fontweight="bold")
        plt.xlabel("Timesteps")
        plt.ylabel("Stock closing price")
        plt.show()

        # finding the maximum and minimum stock price
        maximum = max(finals)
        minimum = min(finals)

        # histogram plot for final stock prices
        a = numpy.arange(round(minimum) - 1, round(maximum))
        fig, ax = plt.subplots()
        ax.hist(finals, bins=a)
        plt.title("Histogram of the final stock prices",
fontweight="bold")
        plt.show()
```

```
    # returns
    return avg_path[rows], minimum, maximum, avg_path


# number of trading days in a month is being taken as 22
month = 22

# monte carlo body
def montecarlo(sheetnum, iters, timestep):

    # opening the sheet needed to open
    sheet = wb[sheets[sheetnum]]
    # the number of rows is the number of entries for a month +
the heading
    # the columns have contents in the order Dates, Series,
Open, High, Low, Prev. Close, LTP, Close, VWAP, 52W H, 52W L,
Volume, Value, No. of Trades
    rows = month * timestep + 1

    # getting the next trading dates to happen for a month
    dates = []
    counter = 0
    date_temp = sheet.cell(rows, 1).value
    daychange = datetime.timedelta(days=1 / timestep)
    dates.append(date_temp)
    while True:
        date_temp = date_temp + daychange
        if date_temp.isoweekday() < 6:
            dates.append(date_temp)
            counter += 1
        if counter == month * timestep:
            break

    # getting open prices, close prices and dates
    opens = []
    closes = []
    prev_dates = []
    for i in range(2, rows + 1):
        opens.append(sheet.cell(i, 3).value)
        closes.append(sheet.cell(i, 8).value)
        prev_dates.append(sheet.cell(i, 1).value)

    # monte carlo simulation
    paths = []
    returns_ = []
    for i in range(rows - 2):
        returns.append((closes[i + 1] / closes[i]) - 1)
    for i in range(iters):
        path = []
        random.seed()
        returns = returns_
```

```
        path.append(closes[rows - 2])
        for j in range(rows - 1):
            returns, s = values(returns, rows - 1, path[j],
timestep)
            path.append(s)
        paths.append(path)

    # calling the plot function
    estimate, min, max, avg_path = plotting(opens, closes,
paths, rows - 1, iters)

    # final print statements
    print("The estimated price of stock at the end of month
is", end = " ")
    print("%.2f"%estimate + " INR.")
    print(f"The maximum estimated price of the stock at the end
of month is", end = " ")
    print("%.2f"%max + " INR.")
    print(f"The minimum estimated price of the stock at the end
of month is", end = " ")
    print("%.2f"%min + " INR.")

    # return statement
    paths.append(avg_path)
    paths.append(dates)
    return paths


# required imports
import openpyxl as op
import datetime
import random
import matplotlib.pyplot as plt
import numpy

# opening workbook
wb = op.load_workbook(input("Enter path of the file: "))
sheets = wb.sheetnames
wb_report = op.Workbook()

# conditions
iters = int(input("Enter number of trails to be done: "))
timestep = int(
    input("Enter number of times the note of stock price is
taken in a day: ")
)

# Monte Carlo simulation
paths = montecarlo(0, iters, timestep)

# saving in a new spreadsheet
```

```
name = " Report"
sheet_name = sheets[0] + name
wb_report["Sheet"].title = sheet_name
sheet = wb_report.active
headings = []
c = "Trail "
for i in range(iters, 0, -1):
    x = c + str(i)
    headings.append(x)
headings.append("Average path")
headings.append("Dates")
entries = 22 * timestep + 1
counter = 0
for i in range(iters + 1, -1, -1):
    counter += 1
    sheet.cell(row=1, column=counter).value = headings[i]
    for j in range(entries):
        sheet.cell(row=j + 2, column=counter).value =
paths[i][j]
name = input("Enter the path to store the data: ")
wb_report.save(name)
```

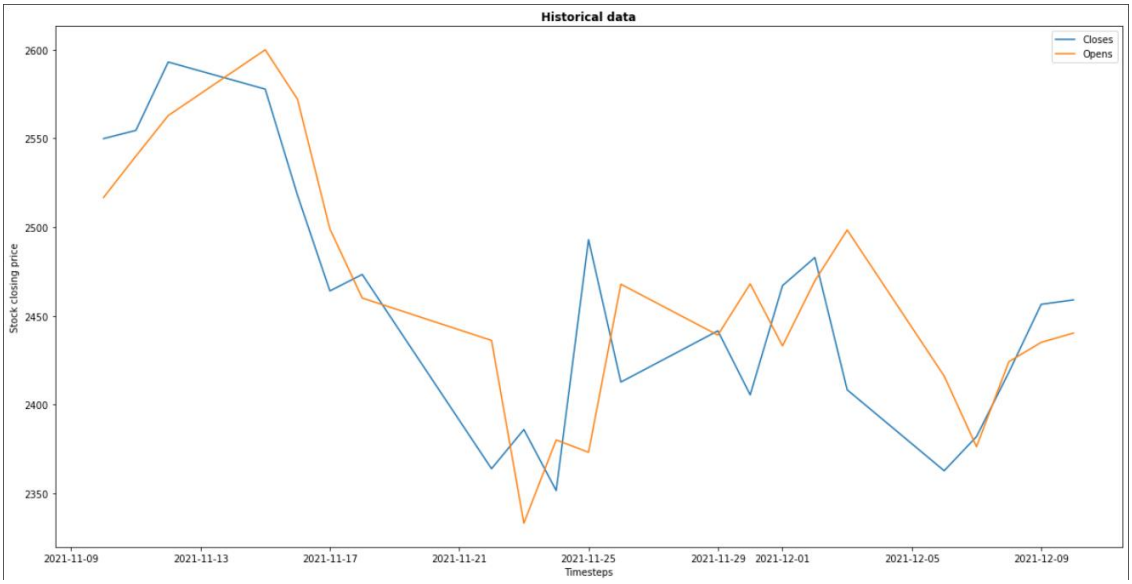
The number of trails used for this simulation was 10,000.
The timestep was taken as one day.

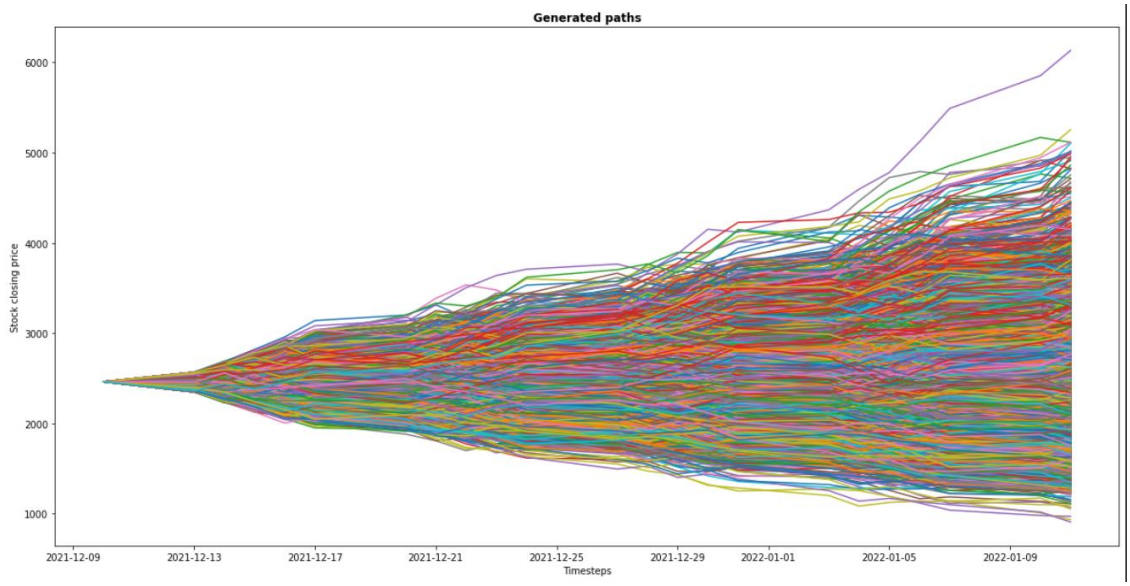
Dates	Series	Open	High	Low	Prev. Close	LTP	Close	VWAP	52W H	52W L	Volume	Value
10-Nov-21	EQ	2,516.70	2,558.00	2,503.65	2,521.70	**2,546.45**	2,549.90	2,538.63	2,751.35	1,830.00	47,31,975	12,01,27,42,203.00
11-Nov-21	EQ	2,540.05	2,570.50	2,535.55	2,549.90	**2,556.00**	2,554.55	2,551.89	2,751.35	1,830.00	49,48,422	12,62,78,28,736.25
12-Nov-21	EQ	2,562.90	2,598.75	2,557.00	2,554.55	**2,588.00**	2,593.10	2,582.61	2,751.35	1,830.00	40,20,744	10,38,40,09,844.25
15-Nov-21	EQ	2,600.00	2,602.20	2,570.00	2,593.10	**2,574.50**	2,577.80	2,581.57	2,751.35	1,830.00	22,16,708	5,72,25,94,256.70
16-Nov-21	EQ	2,572.05	2,575.00	2,495.00	2,577.80	**2,497.25**	2,517.90	2,536.01	2,751.35	1,830.00	52,32,292	13,26,91,27,406.10
17-Nov-21	EQ	2,498.95	2,498.95	2,461.00	2,517.90	**2,462.80**	2,464.00	2,479.92	2,751.35	1,830.00	47,08,235	11,67,60,42,764.70
18-Nov-21	EQ	2,460.00	2,489.00	2,450.05	2,464.00	**2,473.00**	2,473.30	2,467.65	2,751.35	1,830.00	39,25,345	9,68,63,86,834.65
22-Nov-21	EQ	2,436.10	2,449.00	2,351.00	2,473.30	**2,365.65**	2,363.75	2,380.24	2,751.35	1,830.00	1,11,33,364	26,50,01,12,857.50
23-Nov-21	EQ	2,333.05	2,401.25	2,309.00	2,363.75	**2,389.95**	2,385.85	2,362.47	2,751.35	1,830.00	1,16,88,406	27,61,35,01,135.60
24-Nov-21	EQ	2,380.00	2,409.90	2,343.55	2,385.85	**2,343.55**	2,351.40	2,376.12	2,751.35	1,830.00	77,62,564	18,44,47,68,823.80
25-Nov-21	EQ	2,373.00	2,502.00	2,357.15	2,351.40	**2,501.00**	2,492.95	2,452.67	2,751.35	1,830.00	1,95,68,487	47,99,49,98,310.55
26-Nov-21	EQ	2,467.80	2,477.60	2,401.50	2,492.95	**2,405.10**	2,412.60	2,433.36	2,751.35	1,830.00	72,74,686	17,70,18,96,486.35
29-Nov-21	EQ	2,439.10	2,500.00	2,399.10	2,412.60	**2,437.70**	2,441.50	2,464.20	2,751.35	1,830.00	1,12,26,147	27,66,34,93,598.00
30-Nov-21	EQ	2,468.00	2,475.90	2,388.85	2,441.50	**2,401.20**	2,405.40	2,424.91	2,751.35	1,830.00	1,48,47,511	36,00,38,21,297.30
01-Dec-21	EQ	2,433.00	2,474.00	2,425.10	2,405.40	**2,467.75**	2,467.00	2,449.60	2,751.35	1,830.00	46,63,276	11,42,31,47,561.65
02-Dec-21	EQ	2,469.70	2,496.20	2,461.60	2,467.00	**2,481.95**	2,482.85	2,482.51	2,751.35	1,830.00	58,70,468	14,57,35,15,872.05
03-Dec-21	EQ	2,498.40	2,498.50	2,400.00	2,482.85	**2,413.15**	2,408.25	2,436.10	2,751.35	1,830.00	88,71,172	21,61,10,45,769.40
06-Dec-21	EQ	2,416.00	2,425.00	2,357.15	2,408.25	**2,365.95**	2,362.60	2,388.06	2,751.35	1,830.00	47,68,334	11,38,70,71,064.70
07-Dec-21	EQ	2,376.15	2,404.00	2,360.00	2,362.60	**2,382.00**	2,381.85	2,387.87	2,751.35	1,830.00	51,85,007	12,38,11,07,849.30
08-Dec-21	EQ	2,424.10	2,431.80	2,406.30	2,381.85	**2,420.95**	2,418.10	2,418.81	2,751.35	1,830.00	40,35,437	9,76,09,53,632.95
09-Dec-21	EQ	2,435.00	2,474.90	2,425.00	2,418.10	**2,453.00**	2,456.45	2,453.15	2,751.35	1,830.00	60,98,050	14,95,94,57,280.85
10-Dec-21	EQ	2,440.25	2,466.00	2,430.35	2,456.45	**2,457.25**	2,458.95	2,443.62	2,751.35	1,830.00	38,60,176	9,43,28,10,632.75

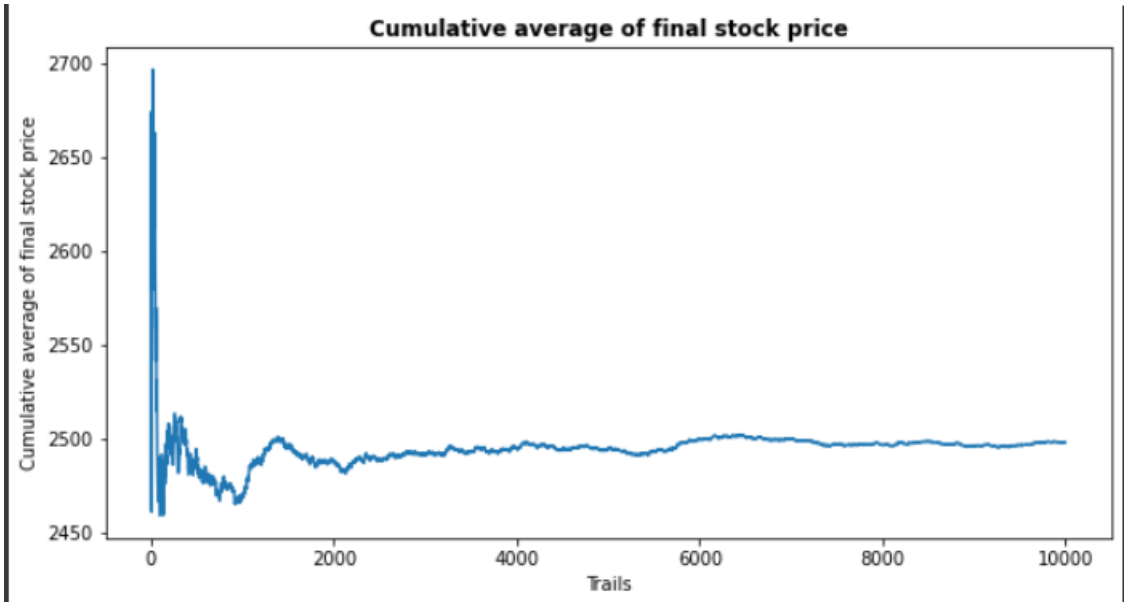
Historical Data

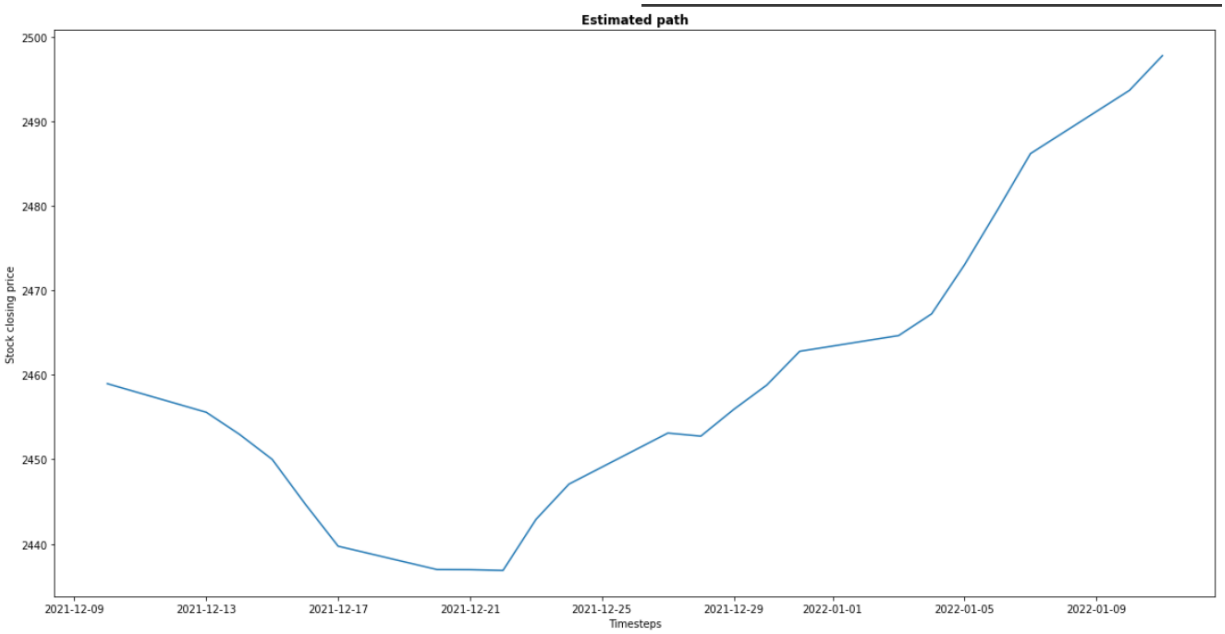
** The last column containing the total number of trades could not be fit in this page

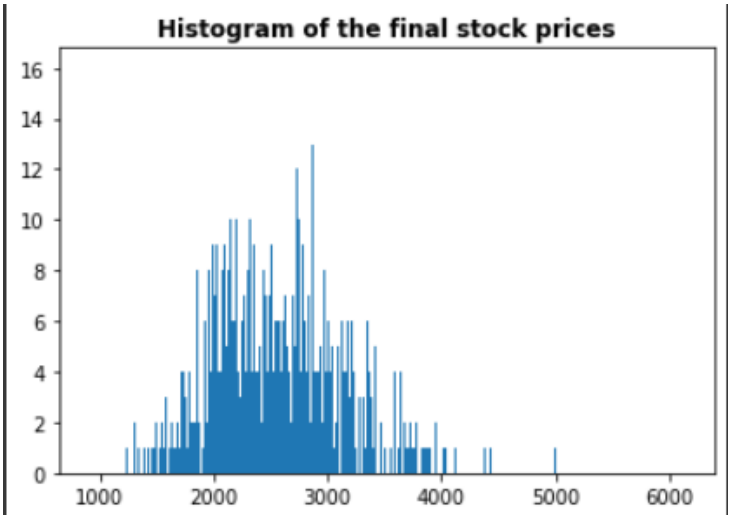
Outputs:






Cumulative average of final stock price
Cumulative average of final stock price
Trails
2700
2650
2600
2550
2500
2450
0
2000
4000
6000
8000
10000


Estimated path
Stock closing price
Timesteps
2500
2490
2480
2470
2460
2450
2440
2021-12-09
2021-12-13
2021-12-17
2021-12-21
2021-12-25
2021-12-29
2022-01-01
2022-01-05
2022-01-09


Histogram of the final stock prices
16
14
12
10
8
6
4
2
0
1000
2000
3000
4000
5000
6000

The estimated price of stock at the end of month is 2497.78 INR.
The maximum estimated price of the stock at the end of month is 6133.20 INR.
The minimum estimated price of the stock at the end of month is 903.97 INR.

Here it can be seen that most of the random paths taken do lead to various final results. The range of the random variable can be changed to see more possible ups and downs in the prices. The cumulative average of the final stock prices show that the price fluctuation becomes less as the number of trails increase. The final stock prices at the end of the period roughly shows a bell curve plot nature.

Another simple model for doing the simulation can have the formula:

$$S_t = S_{t-1} \cdot e^{\left(\mu - \frac{1}{2}\sigma^2\right) + \sigma\varepsilon}$$

Here, S_t is the stock price to be calculated. S_{t-1} is the previous stock price. $e^{\left(\mu - \frac{1}{2}\sigma^2\right) + \sigma\varepsilon}$ is the return obtained. μ is the mean of the returns and σ is the standard deviation of the returns. ε is a random number taken uniformly from [0,1]. $\left(\mu - \frac{1}{2}\sigma^2\right)$ is the drift and $\sigma\varepsilon$ represents the volatility. This is from Black-Scholes model (B-S-model).

With both the models give various paths will be formed and the average of all the paths will be taken. These models do not consider the market sentiments and sudden news which might cause sudden changes into the stock prices. Hence, these averages might give a very near to the actual values that will then come, but, it is very much possible for this to not happen as well.

Quasi Monte Carlo Simulations

As the name suggests, this method is not totally a Monte Carlo simulation method. Here, the samples taken are not totally random. They are pseudo-random. This reduces the number of trails needed for the simulation to run on. The samples used in the simulation are well-distributed in the domain to give enough accuracy.

Such sample spaces are created by various methods. The generated sample spaces are generally referred to as Low-Discrepancy Sequences or Quasi-random sequences. These sequences also give better results, as in closer to the actual results, as compared with that from Monte Carlo simulations. Some of the most common ways to generate such a sequence is the Van der Corput method (the subsequent sequence is called Van der Corput sequence) and Halton Sampling (the subsequent sequence is called Halton sequence).

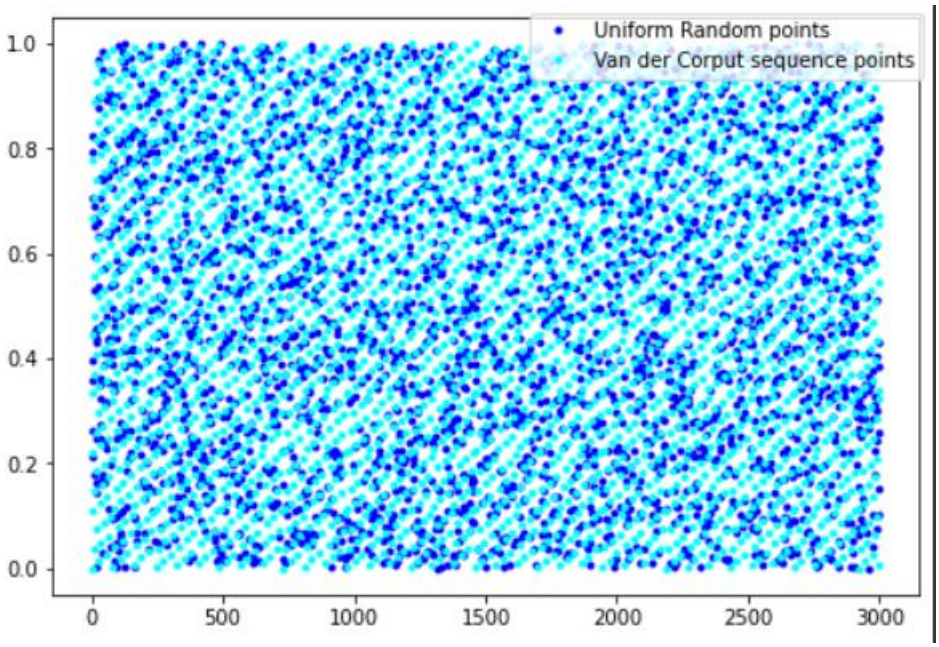
Let's discuss the way to make the Van der Corput sequence for a given base B. Then the i-th element of the sequence is given by the following steps. First, the number in the i-th element is converted into the base B. Then the converted number is reflected about the decimal point. This new number is converted back to decimal from the base B. This generates a sequence of non-repeating set of decimal numbers in the range

[0,1]. This sequence can then be manipulated to fit into the required domain to get a well-distributed sample space.
Let's say we take I as 7 as the i-th element and base B as 5. Then, the value of decimal 7 in base 5 is 12. This when reflected about the decimal point is 0.21. This is taken as to be in base 5. Now this 0.21 in base 5 is converted to decimal as 0.44. So, 0.44 becomes the i-th element of the Van der Corput series.

Let's see a quick code to make a Van der Corput sequence.

```
# imports needed
import matplotlib.pyplot as plt
import random

# inputs
n = 3000
base = 3
Van_der_Corput_sequence = []
t = 1
x = []
random_ = []

# function to make one number in the sequence
def Van_der_Corput(c,base):
  convert_1 = []
  i = float(c)
  if i<0:
    t = '-'
    i*=-1
  elif i == 0:
    return 0
  else:
    t = '+'
  while (i):
    d = i%base
    convert_1.append(d)
    i//=base
  i=0
  h=1
  for k in convert_1:
    i = i + k/float(base**h)
    h+=1
  if t=='-':
    i*=-1
  return i

# making of the sequence
for i in range(n):
  Van_der_Corput_sequence.append(Van_der_Corput(i,base))
  x.append(t)
  t+=1
```

```
for i in range(n):
  random_.append(random.uniform(0,1))

# plots
ptl = plt.figure()
pt = ptl.add_axes([0,0,1,1])
pt.plot(x,random_,'.',color='blue',label='Uniform Random
points')
pt.plot(x,Van_der_Corput_sequence,'.', color='cyan', label='Van
der Corput sequence points')
ptl.legend()
plt.show()
```



3000 points distributed in [0,1]

Here, the points in cyan show the points of the Van der Corput sequence generated from integer elements from [0,3000) with base as 3. The blue points are randomly generated uniform points in [0,1] by python's random.uniform function. It is pretty evident that the cyan points are well-distributed in the area whereas the blue points are not.

Now suppose this Van der Corput sequence is to be used to form a sample space from a domain which has a range [a,b]. For this, then the sequence will be modified as,

$$J_i = (b - a) \times V_i + a$$

Where, J is the new sequence and V is the generated Van der Corput sequence.

The prime advantage of using a Low Discrepancy Sequence for the simulations is that it gives the surety of a better covering of the whole domain. This makes the estimate thus obtained much better.

Conclusion

Monte Carlo simulation is a very powerful tool when it comes to dealing with systems with random behavior. If the correct boundaries are recognized, the simulation can survey the parameter space of problem.

The time taken to complete a Monte Carlo simulation depends on the number of variables and complexity of the model. If both are high, the time taken is quite significant. In other words, the computational efficiency decreases.

The input file used in the Monte Carlo simulation for stocks and all the codes can be found here:
https://colab.research.google.com/drive/1UVGFNeRftuaf62gdOOkTXOHbvZjnBi_b?usp=sharing

References

https://corporatefinanceinstitute.com/

https://www.freecodecamp.org/

https://www.geeksforgeeks.org/

https://ieeexplore.ieee.org/

https://www.investopedia.com/

https://www.mathbootcamps.com/

https://medium.com/

https://www.nseindia.com/

https://www.scratchapixel.com/

https://towardsdatascience.com/

https://en.wikipedia.org/