Exploring Monte Carlo Simulations in Python

Madaly Gregory
BPHYS450 Computational and Theoretical Modelling in Physics
March 19th, 2026

The goal of a Monte Carlo simulation is to estimate a "fair" guess of a system outcome based on random variables whose functions are not known. In complex systems, random and interconnected variables can be impossible to integrate directly without enormous computational cost [3]. To generate a prediction, the system under analysis is simulated over thousands of runs (or more) and the behavior of the random variables is accounted for by averaging the target value over all of the runs.

f(x) represents the Probability Density Function (PDF). This is the function of random variables whose individual functions are unknown. F(x) represents the Cumulative Density Function (CDF), which updates the average value over every run with new information about the random situation at that point [3]. Any information about the current situation can be used to contribute to the prediction value. For example, when calculating stock prices, the last stock price at some point will be necessary to determine the current stock price estimation. This is manually written into the Cumulative Density Function based on the desired value (stock price after a specific period of time in this case).

In the following analysis I will walk through the code for two different use cases of this Monte Carlo approach, first relating to particle physics and then relating to finance.

Case #1: Particle Detection in a Calorimeter

At the CERN Large Hadron Collider, the ATLAS calorimeter contains detector cells which stop particles in their tracks [1]. These cells detect absorbed energies to determine characteristic properties of fundamental particles. To replicate a typical experiment, I started by simulating particles arriving at a detector cell. The number of incoming particles (X) and the energy of these particles (E) and the random variable to analyze probabilistically.

The first step in a Monte Carlo simulation is to generate these random variables based off of the probability distribution. The amount of particles that enter a cell over 10 seconds (X) is given by

$$f(x) = \frac{e^{-\lambda}\lambda^{x}}{x!}$$

The energy of each particle (E) is given by

$$g(x) = F_1 e^{-\sqrt{x/E_1}} + F_2 e^{-\sqrt{x/E_2}}$$

Where

$\lambda = 4$

$F_1 = 1.3\ GeV\ x\ 10^{-1}$

$F_2 = 1.4\ GeV\ x\ 10^{-1}$

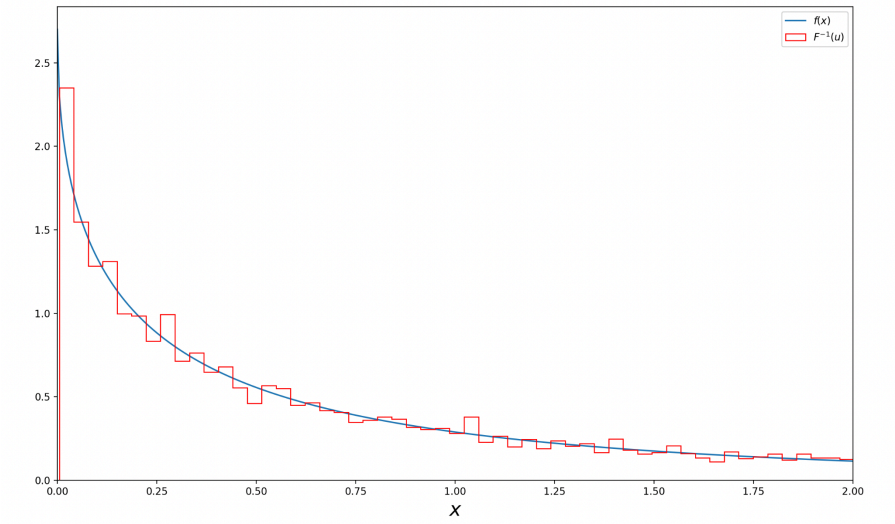
$E_1 = 0.1\ GeV$

$E_2 = 0.2\ GeV$

According to the Inverse Transform Sampling theorem), for a random variable X with a CDF of F(x), the random variable defined by Y = $F^{-1}(U)$ (where U is a random uniform variable between 0-1) follows the F(x) distribution. However, directly calculating the inverse of the CDF when random variables are involved is usually challenging if not impossible. To solve for X, I'm relying on the numpy searchsorted function which looks through the sorted list of CDF values (these are the candidate energy values from 0 to 5 GeV that a particle might have) to find exactly where U fits and finds the corresponding energy value from the x array. First the searchsorted function is plotted against the original function f(x) to verify that searchsorted produces a similar distribution:

```
Us = np.random.rand(10000)

F_inv_Us = -np.log(1-Us)/2

# Solve for the inverse CDF
x, y, F1, F2, E1, E2 = smp.symbols('x y F_1 F_2 E_1 E_2', real=True, positive=True)
fs = F1*smp.exp(-smp.sqrt(x/E1)) + F2*smp.exp(-smp.sqrt(x/E2))
fs

Fs = smp.integrate(fs, (x,0,y)).doit()
Fs

Fn = smp.lambdify((y, E1, E2, F1, F2), Fs)
fn = smp.lambdify((x, E1, E2, F1, F2), fs)

# ------ Test the inverse CDF method by simulating 10,000 energies from the distribution defined by f(x) and F(x) ------
E1 = E2 = 0.2
F1 = 1.3
F2 = 1.4
x = np.linspace(0,5,1000) # Create a "look-up table" by calculating the CDF ($F$) for a thousand specific energy values
f = fn(x, E1, E2, F1, F2)
F = Fn(x, E1, E2, F1, F2)

F_inv_Us = x[np.searchsorted(F[:-1], Us)] # Looks through the sorted list of CDF values for a random number U and finds
# the corresponding energy value in x

plt.figure(figsize=(8,3))
plt.plot(x, f, label=r'$f(x)$')
plt.hist(F_inv_Us, histtype='step', color='red', density='norm', bins=100, label='$F^{-1}(u)$')
plt.legend()
plt.xlabel('$x$', fontsize=20)
plt.legend()
plt.xlim(0,2)
plt.show()

# ------
```

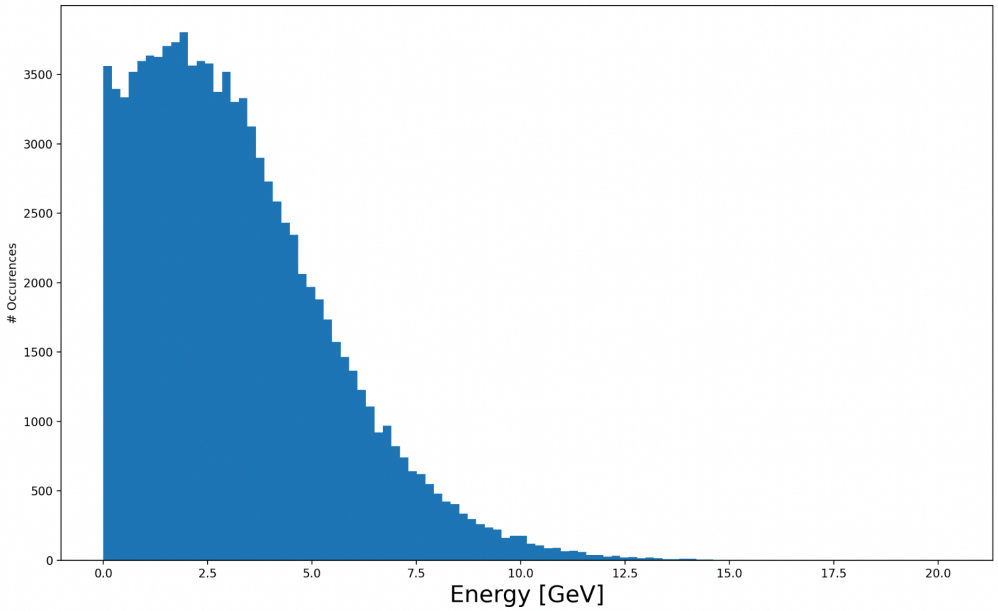
After plotting 10,000 cases, the searchsort method appears to align closely with the original function:



After verifying that the searchsorted method of finding the inverse CDF closely matches the original function fs, searchsorted is called on 100,000 experiments to determine the amount of energy to be deposited in the detector within that 10-second timeframe:

```
# ------ Simulating 100,000 10-second experiments as 100,000 samples from a Poisson distribution ------
N = 100000
X = np.random.poisson(lam=4, size=N)

x = np.linspace(0,5,1000)
F = Fn(x, E1, E2, F1, F2)
Us = np.random.rand(X.sum())
E = x[np.searchsorted(F[:-1], Us)]

# Sum of how many particles are detected total after all N experiments
idx = np.insert(X.cumsum(), 0, 0)[:-1]
idx[0:10]

# Sum the energies of the particles detected in each experiment
E_10s_sum = np.add.reduceat(E, idx)

# Plot the distribution of energies detected in all of the 10-second experiments
plt.figure(figsize=(5,3))
plt.hist(E_10s_sum, bins=100)
plt.xlabel('Energy [GeV]', fontsize=20)
plt.ylabel('# Occurences')
plt.show()

# ------
```

The following results are plotted:



z

Out of 100,000 experiments, most result in 0 to 4 GeV of energy being released into the detector. This energy range should be expected to deposit in the detector most of the time. the 10–second interval. Although the chances are relatively low, there is still a 0.2% chance that a stream of particles might leave 10 GeV of energy in the detector. In this case, ATLAS might be able to detect heavier particles or unusual physics behavior if the calorimeter is set up to handle a surge in energy this high.

Case #2: Estimating the Value of a Stock over a Year

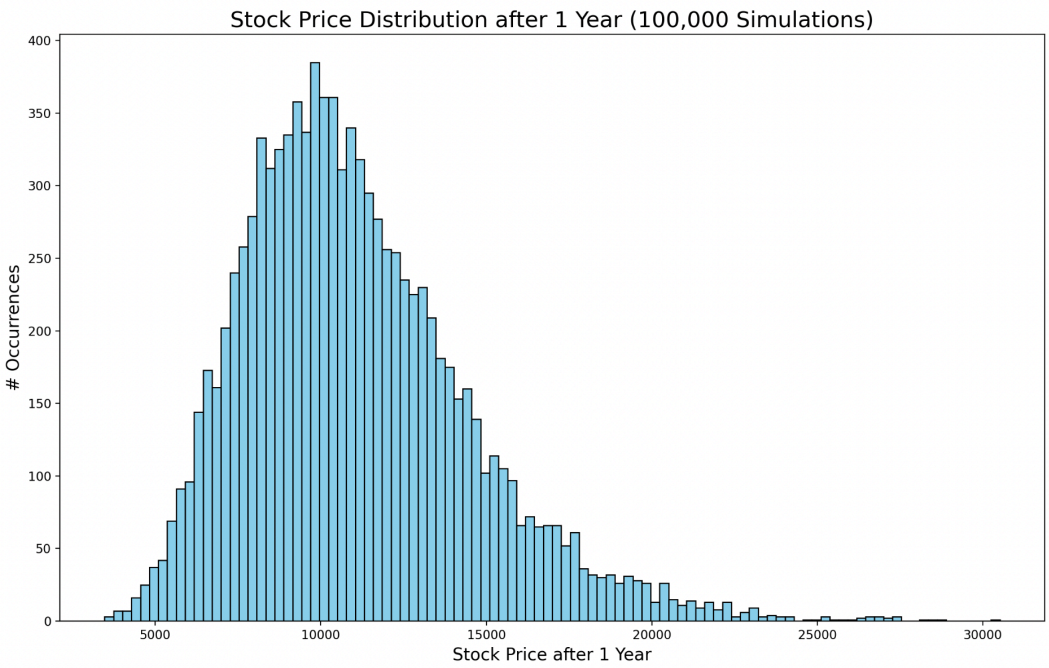
I decided to use this Monte Carlo simulation to predict the price of an individual stock over a given period of time. To begin with, I estimated the price of a stock bought at $100 over one year.
I first initialized a few of the main variables related to stock prices. One of the common methods to predict the change in a stochastic process (like stock price over time) is by modelling its growth via Geometric Brownian Motion [2]. The drift in price is reflected by:

$$\mu - \sigma^2/2$$

Where μ is the drift parameter and σ is the volatility parameter. The drift and volatility are recalculated every day of the simulation. Similarly to how random variables U and E were calculated in case #1, array r generates the difference in price for every day over the simulated year. The Inverse Transform Sampling step is included in numpy.random.normal on line 92.

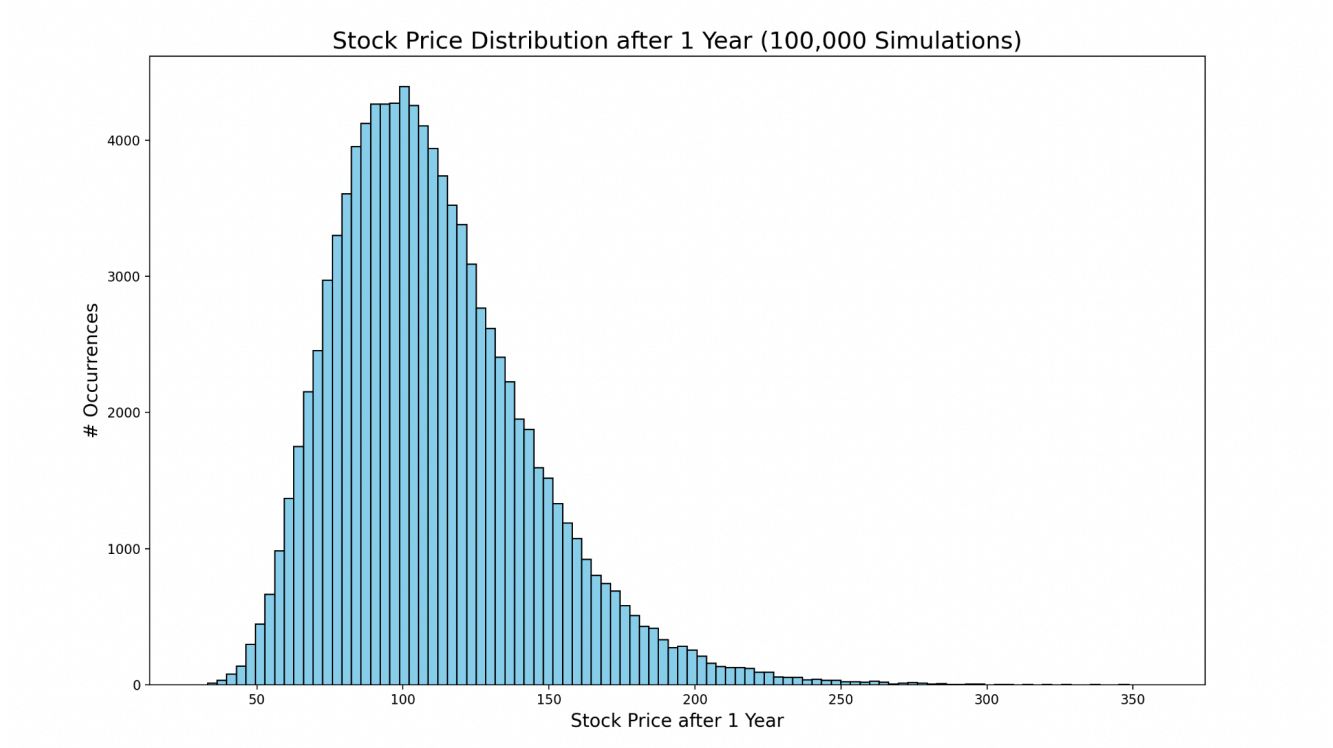
```python
# ------ Simulating a stock value over one year ------

# Parameters
S0 = 10000          # Starting price
mu = 0.1          # Annual expected return (10%)
sigma = 0.3       # Annual volatility (30%)
n_years = 100000   # Number of simulated year trials

# 2. Generating the Random Variables (Log-Returns)
# Total number of "events" (trading days across all simulated year trials)
total_days = n_years * 365

# Calculate daily drift and volatility based on Geometric Brownian Motion
dt = 1 / 365
daily_mu = (mu - 0.5 * sigma**2) * dt #Ito correction for drift
daily_sigma = sigma * np.sqrt(dt)

# Generate all random daily returns at once (similar to U and E)
# Daily returns follow a normal distribution, mean = daily_mu, std = daily_sigma, need total_days of them
r = np.random.normal(loc=daily_mu, scale=daily_sigma, size=total_days)


indices = np.arange(0, total_days, 365)
yearly_log_returns = np.add.reduceat(r, indices)

# Calculate ending prices: S_T = S_0 * e^(sum of log returns)
ending_prices = S0 * np.exp(yearly_log_returns)

# 4. Visualization (Matching your notebook's style)
plt.figure(figsize=(8,4))
plt.hist(ending_prices, bins=100, color='skyblue', edgecolor='black')
plt.xlabel('Stock Price after 1 Year', fontsize=14)
plt.ylabel('# Occurrences', fontsize=14)
plt.title('Stock Price Distribution after 1 Year (100,000 Simulations)', fontsize=18)
plt.show()

prob_profit = np.sum(ending_prices > S0) / len(ending_prices)
expected_val = np.mean(ending_prices)

print("Probability of making a profit: " + str(prob_profit*100))
print("Expected stock price: " + str(expected_val))

# ------
```

After plotting the simulated stock prices, the resulting distribution is as follows:



After a year has passed, the value of the stock hovers around $90-$110. The value of the stock is most likely to be within $90-$110 by the end of the year. In finance, the Value at Risk (VaR) describes the most an investor would lose in the worst case scenario under a certain confidence level. In this case, the most an investor would lose at the end of the year (according to the bottom 5th percentile) is $48. There is a low chance (< 0.25%) of this occurring. The number n_years represents the number of “trial” years to simulate, similarly to how 100,000 experiments were simulated in case #1. To visualize the effect of

increasing the number of trials, I plotted 100,000 simulated trials:



Over a 10x larger sample size, the distribution clearly smooths out and becomes more centered around the peak at roughly $110. The most an investor would lose at the end of the year (according to the bottom 5th percentile) is $50. There is a low chance (< 0.5%) of this occurring.

Summary

The Monte Carlo method turns a stochastic problem (one consisting of random variables) into a probabilistic problem. Although these first simulations are relatively simple, they can provide useful rough estimations of values that would otherwise be impossible to calculate by hand. I can continue to expand these models to improve their predictive accuracy. I plan to experiment with different functions for Inverse Transform Sampling (inverting the CDF) besides searchsorted and np.random.normal. I also plan to explore problems that will benefit from implementing a Markov Chain into the Monte Carlo algorithm to estimate future values based only on the current state.

.

References:

[1] CERN (2026). Calorimeter. Atlas Experiment at CERN.
https://atlas.cern/Discover/Detector/Calorimeter

[2] Siegrist, K. (2022). 18.4: Geometric Brownian Motion. LibreTexts Statistics.
https://stats.libretexts.org/Bookshelves/Probability_Theory/Probability_Mathematical_Statistics_and_Stochastic_Processes_(Siegrist)/18%3A_Brownian_Motion/18.04%3A_Geometric_Brownian_Motion

[3] Polson, K. (2021) Monte Carlo Tutorial. Github Repository.
https://github.com/lukepolson/youtube_channel/blob/main/Python%20Tutorial%20Series/montecarlo1.ipynb