Monte Carlo Option Pricing

Kannan Singaravelu, CQF

1 Monte Carlo Approach

Monte Carlo methods are any process that consumes random numbers. These are part of computational algorithms which are based on random sampling to obtain numerical results. Monte Carlo methods are proved to be a very valuable and flexible computational tool in finance and is one of the most widely used methods for optimization and numerical integration problems.

These methods are widely used in high dimensional problems; pricing exotics and complex derivatives where closed form solutions are not directly available. Monte Carlo methods are not just adapted in pricing complex derivatives, It is also extensively used in estimating the portfolio risk such as Value-at-Risk and Expected Shortfall and used in the calculation of worst-case scenarios in stress testing. The downside to that is, it is very computational intensive and demanding.

1.1 Monte Carlo Simulation

A method of estimating the value of an unknown quantity using the principles of inferential statistics.

We take the **population** and then we **sample** it by drawing a proper subset. And then we make an inference about the population based upon some set of statistics we do on the sample.

And, the key fact that makes them work, that if we choose the sample at **random**, the sample will tend to exhibit the same properties as the population from which it is drawn.

1.2 Option Pricing Techniques

As with other option pricing techniques Monte Carlo methods are used to price options using what is essentially a three step process.

Step 1: Simulate potential price paths of the underlying asset.

Step 2: Calculate the option payoff for each of these price paths.

Step 3: Average the payoff and discount back to today to determine the option price.

2 Simulating Asset Prices

Next, we will simulate the asset price at maturity S_T. Following Black-Scholes-Merton where the underlying follows under risk neutrality, a geometric Brownian motion with a stochastic differential equation (SDE) is given as

$$dS_t = rS_t dt + \sigma S_t dW_t$$

where S_t is the price of the underlying at time t, σ is constant volatility, r is the constant risk-free interest rate and W is the brownian motion.

Applying Euler discretization of SDE, we get

$$S_{t+\delta t} = S_t * (1 + r\delta t + \sigma\sqrt{\delta} t w_t)$$

It is often more convenient to express in time stepping form

$$S_{t+\delta t} = S_t exp^{((r-\frac{1}{2}\sigma^2)\delta t + \sigma\sqrt{\delta} t w_t)}$$

The variable w is a standard normally distributed random variable, $0 < \delta t <$ T, time interval. It also holds $0 <$ t T with T the final time horizon.

2.1 Generate Price Paths

Simulating price paths plays an important role in the valuation of derivatives and it is always prudent to create a separate path function.

Import Required Libraries

[ ]:
```
# Importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Set max row to 300
pd.set_option('display.max_rows', 300)
```

[ ]:
```
class MonteCarloOptionPricing:
    ''' Monte Carlo Option Pricing Engine'''

    def __init__(self, S0:float, strike:float, rate:float, sigma:float, dte:int,␣
 ↪nsim:int, timesteps:int=252) -> float:

        self.S0 = S0                        # Initial stock price
        self.K = strike                     # Strike price
        self.r = rate                       # Risk-free interest rate
        self.sigma = sigma                  # Volatility
        self.T = dte                        # Time to expiration
```

```
        self.N = nsim                          # Number of simulations
        self.ts = timesteps                    # Time steps

    @property
    def psuedorandomnumber(self):
        ''' generate psuedo random numbers'''
        return np.random.standard_normal(self.N)

    @property
    def simulatepath(self):
        ''' simulate price path'''
        np.random.seed(2024)

        # define dt
        dt = self.T/self.ts

        # simulate path
        S = np.zeros((self.ts, self.N))
        S[0] = self.S0

        for i in range(0, self.ts-1):
            w = self.psuedorandomnumber
            S[i+1] = S[i] * (1+ self.r*dt + self.sigma * np.sqrt(dt)*w)

        return S

    @property
    def vanillaoption(self):
        ''' calculate vanilla option payoff'''
        S = self.simulatepath

        # calculate the discounted value of the expeced payoff
        vanilla_call = np.exp(-self.r*self.T) * np.mean(np.maximum(0, S[-1]-self.
↪K))
        vanilla_put = np.exp(-self.r*self.T) * np.mean(np.maximum(0, self.
↪K-S[-1]))

        return [vanilla_call, vanilla_put]

    @property
    def asianoption(self):
        ''' calculate asian option payoff'''
        S = self.simulatepath

        # average the price across days
        A = S.mean(axis=0)
```

```
        # calculate the discounted value of the expeced payoff
        asian_call = np.exp(-self.r*self.T) * np.mean(np.maximum(0, A-self.K))
        asian_put = np.exp(-self.r*self.T) * np.mean(np.maximum(0, self.K-A))

        return [asian_call, asian_put]


    def upandoutcall(self, barrier:int=150, rebate:int=0) -> float:
        ''' calculate up-and-out barrier option payoff'''
        S = self.simulatepath

        # Barrier shift - continuity correction for discrete monitoring
        barriershift = barrier*np.exp(0.5826 * self.sigma * np.sqrt(self.T/self.
↪ts))

        value=0
        # Calculate the discounted value of the expeced payoff
        for i in range(self.N):
            if S[:,i].max() < barriershift:
                value += np.maximum(0, S[-1,i]-self.K)
            else:
                value += rebate

        return [np.exp(-self.r*self.T) * value/self.N, barriershift]
```

[ ]:
```
# instantiate
mc = MonteCarloOptionPricing(100,100,0.05,0.2,1,100000)

# Verify the generated price paths
pd.DataFrame(mc.simulatepath).head(2)
```

Histogram of Psuedo Random Numbers

[ ]:
```
# Plot the histogram
plt.hist(mc.psuedorandomnumber, bins=100);
```

Histogram of Simulated Paths

[ ]:
```
# Plot the histogram of the simulated price path at maturity
plt.hist(mc.simulatepath[-1], bins=100);
```

Visualization of Simulated Paths

[ ]:
```
# Plot initial 100 simulated path using matplotlib
plt.plot(mc.simulatepath[:,:100])
plt.xlabel('time steps')
plt.xlim(0,252)
plt.ylabel('index levels')
plt.title('Monte Carlo Simulated Asset Prices');
```

3 Risk-Neutral Valuation

A call option gives the holder of the option the right to buy the asset at a pre-defined price. A call buyer makes money if the price of the asset at maturity, denoted by S_T, is above the strike price K, otherwise it's worth nothing.

$$C_T = max(0, S_T - K)$$

The price of an option using a Monte Carlo simulation is the expected value of its future payoff. So at any date before maturity, denoted by t, the option's value is the present value of the expectation of its payoff at maturity, T.

$$C = PV(E[max(0, S_T - K)])$$

Under the risk-neutral framework, we assume the asset is going to earn, on average, the risk-free interest rate. Hence, the option value at time t would simply be the discounted value of the expected payoff.

$$C = e^{r(Tt)}(E[max(0, S_T - K)])$$

European Option To price an option, we generate many possible price paths that the asset might take at maturity and then calculate option payoffs for each of those generated prices, average them to get the expected payoff and then discount it at risk free to arrive at the final value.

Given that Monte Carlo algorithms are computationaly heavy, it is necessary to implement efficiently. We'll use vectorization with NumPy for effective algorithm as NumPy syntax are more compact and are faster.

[ ]:
```
# Get option vaues
print(f"European Call Option Value is {mc.vanillaoption[0]:0.4f}")
print(f"European Put Option Value is {mc.vanillaoption[1]:0.4f}")
```

Plot Payoff

[ ]:
```
# range of spot prices
sT= np.linspace(50,150,100)

# visualize call and put price for range of spot prices
figure, axes = plt.subplots(1,2, figsize=(20,6), sharey=True)
```

```
title, payoff, color, label = ['Call Payoff', 'Put Payoff'], [np.maximum(sT-mc.
↪K, 0), np.maximum(mc.K-sT, 0)], ['green', 'red'], ['Call', 'Put']

# plot payoff
for i in range(2):
    axes[i].plot(sT, payoff[i], color=color[i], label=label[i])
    axes[i].set_title(title[i])
    axes[i].legend()

figure.suptitle('Option Payoff at Maturity');
```

Asian Call Option An Asian option is an option where the payoff depends on the average price of the underlying asset over a certain period of time. Averaging can be either be Arithmetic or Geometric. There are two types of Asian options: **fixed strike**, where averaging price is used in place of underlying price; and **fixed price**, where averaging price is used in place of strike.

We'll now price a fixed strike arthmetic average option using Monte Carlo simulation.

The payoff of the options is given by

$$C_T = max(0, \frac{1}{T}\sum_{i=1}^{T} S_i - K)$$

$$C_T = max(0, S_{Avg} - K)$$

where S_{Avg} is the average price of the underlying asset over the life of the option. To price an option using a Monte Carlo simulation we use a risk-neutral valuation, where the fair value for a derivative is the expected value of its future payoff. So at any date before maturity, denoted by t , the option's value is the present value of the expectation of its payoff at maturity, T.

$$C = PV(E[max(0, S_{Avg} - K)])$$

Under the risk-neutral framework, we assume the asset is going to earn, on average, the risk-free interest rate. Hence, the option value at time t would simply be the discounted value of the expected payoff.

$$C = e^{r(Tt)}(E[max(0, S_{Avg} - K)])$$

[ ]:
```
# Get option values
print(f"Asian Call Option Value is {mc.asianoption[0]:0.4f}")
print(f"Asian Put Option Value is {mc.asianoption[1]:0.4f}")
```

Up-and-out Barrier Call Option Barrier Options are path dependent exotic options whose payoff depends on whether the price of the underlying asset crosses a pre specified level (called the 'barrier') before the expiration. The four main types of barrier options are:

- Up-and-out
- Down-and-out
- Up-and-in
- Down-and-in

Refer Paul Wilmott on Quantitative Finance Chapter 23 — Barrier Options and Chapter 77 — Finite Difference Methods for One-factor Models for further details on barriers.

Next, we will price a Up-Out-Call barrier with and without rebate using Monte Carlo simulation. Barrier options can be priced using analytical solutions if we assume continuous monitoring of the barrier. However, in reality many barrier contracts specify discrete monitoring.

In a paper titled *A Continuity Correction for Discrete Barrier Option*, Mark Broadie, Paul Glasserman and Steven Kou have shown us that the discrete barrier options can be priced using continuous barrier formulas by applying a simple continuity correction to the barrier. The correction shifts the barrier away from the underlying by a factor of

$$exp^{(\beta\sigma\sqrt{\Delta t})}$$

where $\beta \approx 0.5826$ and σ is the underlying volatility, and Δt is the time between monitoring instants. We will apply this continuity correction in our pricing method as well.

[ ]:
```
# Get barrier option values for B=150 and rebate=0
print(f"Up-and-Out Barrier Call Option Value is {mc.upandoutcall()[0]:0.4f}")
```

Plot Payoff

[ ]:
```
figure, axes = plt.subplots(1,3, figsize=(20,6), constrained_layout=True)
title = ['Visualising the Barrier Condition', 'Spot Touched Barrier', 'Spot␣
 ↪Below Barrier']

# Get simulated path
S = mc.simulatepath
B_shift = mc.upandoutcall()[1]

axes[0].plot(S[:,:200])
for i in range(200):
    axes[1].plot(S[:,i]) if S[:,i].max() > B_shift else axes[2].plot(S[:,i])

for i in range(3):
    axes[i].set_title(title[i])
    axes[i].hlines(B_shift, 0, 252, colors='k', linestyles='dashed')

figure.supxlabel('time steps')
figure.supylabel('index levels')

plt.show()
```

4 References

- Paul Glasserman (2004), Monte Carlo Methods in Financial Engineering
- Paul Wilmott (2007), Paul Wilmott introduces Quantitative Finance
- Python Resources
- Understanding Options

Python Labs by Kannan Singaravelu.