Monte Carlo Methods: additional Example

This repository contains a Python implementation of the Monte Carlo method to price a European basket call option under the Black-Scholes framework. The project showcases various techniques to simulate correlated asset prices, compute option prices, and apply variance reduction strategies.

Case Scenario

Let us suppose that the price of an asset at time t, S_t^i, evolves, under the risk-neutral measure $\mathbb{Q}$, according to the stochastic differential equation (SDE):

$$dS_t^i \ = \ r\, S_t^i\, dt \ + \ \sigma_i\, S_t^i\, dB_t.$$

With:

- r: risk-free rate,
- σ_i: volatility of the i-th asset,
- dB_t: increment of a Brownian motion under the risk-neutral measure.

Now the European basket call option payoff at maturity T with strike K is given by

$$f_T = max\left(\sum_{i=1}^{d} S_t^i - K, 0\right)$$

Since no closed-form solution exists, we estimate the option price using Monte Carlo simulation.

Project Highlights

1. Euler Discretization

In this project, we aim to generate a vector $(S_T^1, \dots, S_T^d)$ representing the d asset prices at maturity T, given the volatility vector $\sigma = (\sigma_1, \dots, \sigma_d)$ and other parameters. To achieve this, we will utilize the Euler discretization method for the stochastic differential equation (SDE) provided. This involves partitioning the interval $[0, T]$ into time steps $\varDelta t = \frac{T}{M}$ with $\{t_m = m\varDelta t : m = 0,1, \dots, M\}$.

The approach follows the same principles applied in our previous project, where we employed numerical methods to simulate stochastic processes and compute derivative prices. This ensures consistency in methodology while extending its application to the pricing of basket options.

Note 1: The SDE is driven by a single (common) Brownian motion B_t for all $i \ = 1, \dots, d$. That is, the assets prices are perfectly correlated as $\left\langle \ln S_.^i\,, \ln S_.^j \right\rangle_t = \sigma_i\, \sigma_j\, \langle B_.^i\,, B_.^j \rangle = \sigma_i\, \sigma_j\, t$

2. Monte Carlo Estimation

The code implemented in Figure 2 computes the price of a European Basket call using Monte-Carlo simulation using the vector of simulated asset terminal prices with the function from part a. It is being done in a similar fashion to our previous project.

3. Control Variate

We reduce the variance by introducing a variate whose payoff is

$$g_T = \sum_{j=1}^{d} (S_t^j - K_j)^+$$

Where K_j is proportionally derived from K.

$$K_j = \frac{K S_0^j}{\sum_{l=1}^{d} S_0^l}$$

Our code will calculate the optimal c that we will be using in our control variate method.

Key Parameters

- **Number of assets (d)**: 4
- **Initial asset prices (S_0^i):** [100, 100, 100, 100]
- **Volatilities (σ)**: [0.18, 0.22, 0.28, 0.36]
- **Risk-free rate (r)**: 0.02
- **Strike price (K)**: 425
- **Maturity (T)**: 2
- **Time steps (M)**: 10^4
- **Paths (N)**: 10^2, 10^3, 10^4, 10^5

Code

1. PART 1

```
import numpy as np
from scipy.stats import norm  # for part c
import pandas as pd # for better visualization of the results


# we will initialize a list to store results for the efficiency part in a dat
aframe
```

```
results = []

np.random.seed(420)

# Function to simulate asset prices using Euler discretization
def simulate_asset_prices(d, T, M, S0, r, sigma):
    delta_t = T / M
    S = np.zeros((M + 1, d))  # Matrix to store asset prices at each step
    S[0] = S0  # Initial prices

    # We will generate Brownian increments for each asset
    # Note that np.random.randn(M,d) gives :matrix of independent standard no
rmal random variables N(0,1) with M rows (time steps) and d columns (assets).
    dW = np.random.randn(M, d) * np.sqrt(delta_t)

    #  We will apply the discritization
    for t in range(M):
        S[t + 1] = S[t] + r * S[t] * delta_t + sigma * S[t] * dW[t]

    return S[-1]  # We only need the final prices at time T
```

2. PART 2

```
# Problem Parameters

volatility = [0.18, 0.22, 0.28, 0.36]  # Volatilities for 4 assets
T = 2  # Time to maturity (1 year)
num_assets = 4  # Number of assets
initial_prices = [100, 100, 100, 100]  # Initial prices of the assets
K = 425
risk_free_rate = 0.02  # Risk-free rate (2%)
Ns = [10**2, 10**3, 10**4, 10**5]
p = 0.1
num_steps = 104

# Function for the Monte Carlo estimator of the European basket call option
```

```python
def monte_carlo_basket_call(S0, K, r, sigma, T, M, d, N):

    temp = 0  # Sum of payoffs
    temp2 = 0  # Sum of squared payoffs

    for _ in range(N):
        final_prices = simulate_asset_prices(d, T, M, S0, r, sigma)  # Final prices of all assets
        basket_payoff = max(np.sum(final_prices) - K, 0)  # Basket call payoff
        discounted_payoff = np.exp(-r * T) * basket_payoff  # Discounted payoff

        temp += discounted_payoff
        temp2 += discounted_payoff**2

    # Monte Carlo estimates
    C0_CR = temp / N  # Mean payoff
    S2_CR = (1 / (N - 1)) * temp2 - (N / (N - 1)) * C0_CR**2
    Var_CR = S2_CR / N  # Adjusted variance
    CI_CR = [C0_CR - 1.96 * np.sqrt(Var_CR), C0_CR + 1.96 * np.sqrt(Var_CR)]  # 95% Confidence Interval

    return C0_CR, CI_CR, Var_CR
```

3. PART 3

```python
# Function to calculate K adj

def adjusted_K(K, S0):
    K_adjusted = K * np.array(S0) / np.sum(S0)
    return K_adjusted



# Function to be used as E[g(Ui)]
```

```python
def get_Analytical_Value(d, T, S0, r, sigma, K):
    # We will adjust the strike prices
    K_adj = adjusted_K(K, S0)

    # We will transform into NumPy arrays and not a list
    S0 = np.array(S0)
    sigma = np.array(sigma)

    # We will compute d1 and d2 using vectorized operations
    d1_values = (np.log(S0 / K_adj) + (r + 0.5 * sigma**2) * T) / (sigma * np
.sqrt(T))
    d2_values = d1_values - sigma * np.sqrt(T)

    # We will compute the Black-Scholes prices using vectorized operations
    BS_prices = S0 * norm.cdf(d1_values) - K_adj * np.exp(-r * T) * norm.cdf(
d2_values)

    # Return the sum of all option prices
    return np.sum(BS_prices)



# Function for the Monte Carlo estimator of the European basket call option w
ith control variates
def monte_carlo_basket_call_control(S0, K, r, sigma, T, M, d, N):

    # We will compute the adjusted strikes for the control variates
    m = int(p * N)
    M_CV = N - m
    K_adjusted = adjusted_K(K, S0)

    temp_muB = 0
    temp_muG = 0

    temp_s2G = 0
```

```
    disc_AG = 0
    disc_B_gT = 0

    for _ in range(m):
        final_prices = simulate_asset_prices(d, T, M, S0, r, sigma)
        basket_payoff = max(np.sum(final_prices) - K, 0)
        gT_payoff = np.sum(np.maximum(final_prices - K_adjusted, 0))
        discounted_payoff = np.exp(-r * T) * basket_payoff
        discounted_gT_payoff = np.exp(-r * T) * gT_payoff



        disc_B_gT += discounted_payoff * discounted_gT_payoff
        temp_muB += discounted_payoff
        temp_muG += discounted_gT_payoff
        temp_s2G += discounted_gT_payoff ** 2

    muB = temp_muB / m
    muG = temp_muG / m
    s2G = (temp_s2G / (m - 1)) - (m / (m - 1)) * muG ** 2
    chat = (disc_B_gT - m * muB * muG) / ((m - 1) * s2G)   #optimal chat

    # Main CV Estimator
    temp_muCV = 0
    temp_s2CV = 0
    c0_CV_est = get_Analytical_Value(d, T, S0, r, sigma, K)  #true value of t
he control variate

    for _ in range(M_CV):
        final_prices = simulate_asset_prices(d, T, M, S0, r, sigma)
        basket_payoff = max(np.sum(final_prices) - K, 0)
        gT_payoff = np.sum(np.maximum(final_prices - K_adjusted, 0))
        discounted_payoff = np.exp(-r * T) * basket_payoff
        discounted_gT_payoff = np.exp(-r * T) * gT_payoff
```

```
        temp_CV = discounted_payoff - chat * (discounted_gT_payoff - c0_CV_es
t )

        temp_muCV += temp_CV

        temp_s2CV += temp_CV ** 2


    muCV = temp_muCV / M_CV

    s2CV = (temp_s2CV / (M_CV - 1)) - (M_CV / (M_CV - 1)) * muCV ** 2

    sCV = np.sqrt(s2CV)

    MSE = s2CV / M_CV

    ci_error = 1.96 * sCV / np.sqrt(M_CV)

    ci_lower = muCV - ci_error

    ci_upper = muCV + ci_error

    CI = [ muCV - 1.96 * sCV / np.sqrt(M_CV) , muCV + 1.96 * sCV / np.sqrt(M_
CV)]  # 95% Confidence Interval


    return  muCV, CI , MSE





# We will compute and store results for each N

for N in Ns:

    # Crude Monte Carlo

    price_CR, CI_CR, mse_crude = monte_carlo_basket_call(initial_prices, K, r
isk_free_rate, volatility, T, num_steps, num_assets, N)


    # Control Variate Monte Carlo

    price_CV, CI_CV, mse_cv = monte_carlo_basket_call_control(initial_prices,
K, risk_free_rate, volatility, T, num_steps, num_assets, N)


    # We will append the result to the list

    results.append({'N': N, 'MSE Crude': mse_crude, 'MSE CV': mse_cv})

    print(f"N = {N}: Estimated Crude Price = {price_CR:.4f}, 95% CI = {CI_CR}
, MSE = {mse_crude:.6f}")

    print(f"N = {N}: Estimated Control Variate Price = {price_CV:.4f}, 95% CI
= {CI_CV}, MSE = {mse_cv:.6f}")
```

```
## N = 100: Estimated Crude Price = 52.1566, 95% CI = [29.82257632736575, 74.
49059373182165], MSE = 129.843801
```

```
## N = 100: Estimated Control Variate Price = 54.7707, 95% CI = [54.706522971081476, 54.83483639108933], MSE = 0.001071

## N = 1000: Estimated Crude Price = 59.2102, 95% CI = [52.21101230220023, 66.20942874710268], MSE = 12.752217

## N = 1000: Estimated Control Variate Price = 54.7932, 95% CI = [54.780608231979464, 54.80584267900016], MSE = 0.000041

## N = 10000: Estimated Crude Price = 53.7395, 95% CI = [51.6535486083318, 55.82552981876027], MSE = 1.132694

## N = 10000: Estimated Control Variate Price = 54.7823, 95% CI = [54.77771931182483, 54.78680571394059], MSE = 0.000005

## N = 100000: Estimated Crude Price = 54.5511, 95% CI = [53.89648795168591, 55.20572064910513], MSE = 0.111548

## N = 100000: Estimated Control Variate Price = 54.7822, 95% CI = [54.780749778314664, 54.783681268182015], MSE = 0.000001
```

```python
# we will create a dataframe to showcase the MSE

df_results = pd.DataFrame(results)



# Ensure all columns of the dataframe are displayed

pd.set_option('display.max_columns', None)



# We will add the efficiency component

df_results['Efficiency Crude'] = 1 / df_results['MSE Crude']

df_results['Efficiency CV'] = 1 / df_results['MSE CV']

df_results['Efficiency Improvement'] = df_results['Efficiency CV'] / df_results['Efficiency Crude'] - 1


# Now we display the end result

print("\nUpdated Dataframe with Efficiency Columns:")
```

```
##

## Updated Dataframe with Efficiency Columns:
```

```python
print(df_results)
```

```
##          N   MSE Crude        MSE CV  Efficiency Crude  Efficiency CV  \

## 0      100  129.843801  1.071450e-03          0.007702   9.333144e+02

## 1     1000   12.752217  4.143959e-05          0.078418   2.413151e+04

## 2    10000    1.132694  5.372937e-06          0.882851   1.861179e+05
```

```
## 3   100000    0.111548  5.592483e-07          8.964755   1.788115e+06
##
##    Efficiency Improvement
## 0           121184.090672
## 1           307729.281695
## 2           210813.647552
## 3           199459.494465
```

Results

N (Paths)	Method	Estimated Price	MSE	95% Confidence Interval	Efficiency Improvement
100	CMC	59.2102	12.752	[52.211, 66.209]	-
	CV	54.7707	0.0011	[54.707, 54.835]	121,184
1,000	CMC	56.8423	4.236	[55.212, 58.472]	-
	CV	54.7932	0.000041	[54.781, 54.806]	307,729
10,000	CMC	53.7395	0.752	[51.654, 55.826]	-
	CV	54.781	3.9E-06	[54.779, 54.783]	210,814
100,000	CMC	54.5511	0.1115	[53.896, 55.206]	-
	CV	54.7822	0.000001	[54.781, 54.784]	199,459

The crude Monte Carlo method yields reasonable estimates of the option price as NNN increases, but the results demonstrate significant variance for small values of NNN. For instance:

- For $N = 100$, the estimated price is 59.2102 with a wide 95% confidence interval $[52.211, 66.209]$ and a large MSE of 12.752. This indicates high uncertainty in the estimate.
- Even for $N = 10000$, the crude Monte Carlo method yields an estimated price of 53.7395 with a still-wide confidence [51.654, 55.826], reflecting persistent variance.
- As $N = 100{,}000$, the crude Monte Carlo estimate stabilizes with a price of 54.5511 , narrower confidence intervals [53.896, 55.206] , and reduced MSE (0.1115), showing improved accuracy but still lagging behind the control variate approach.

Control Variate Results

The control variate method dramatically enhances the precision of the estimates:

- For $N = 100$, the estimated price is 54.7707 with a narrow 95% confidence interval [[54.707, 54.835] , and a very low MSE of 0.0011.

- For $N = 1000$, the control variate estimate improves further, yielding 54.7932, with a highly precise confidence interval [54.781, 54.806] and a significantly smaller MSE of 0.000041.
- Even at $N = 100{,}000$, the control variate consistently produces precise estimates. The price converges 54.7822 with extremely tight confidence intervals [54.781, 54.784] and a negligible MSE of 0.000001.

Efficiency Comparison

The efficiency of the control variate method compared to the crude Monte Carlo estimator is evident from the metrics:

- For $N = 100$ the efficiency improvement is over $12M\%$, reflecting the drastic reduction in variance.
- For $N = 1000$, the efficiency improvement grows to 30M% demonstrating the method's reliability even with moderate sample sizes.
- At $N = 10{,}000$ and $N = 100{,}000$, the efficiency improvements are over 21M% and , 19M% respectively, highlighting the control variate method's scalability and robustness.

The control variate's superiority stems from leveraging the high correlation between the option payoff and the chosen control variate, which substantially reduces the variance.

Conclusion

The control variate method proves to be a highly effective variance reduction technique, when applicable and the use of it comprehensible, yielding accurate and precise option price estimates even for smaller sample sizes. While crude Monte Carlo estimates converge with larger N, the control variate achieves similar or better results with far fewer paths, underscoring its practical advantage in computational finance.