Monte Carlo Methods with Financial Applications Assignment 1

Viktor Ågren

November 2022

Problem 1.

(a) Implement a pseudorandom generator of your choice and use it to generate 1000 uniformly distributed points in $[0, 1]$. In your report, reflect your opinion how random your generator is.

(b) Making use of part (a), generate 1000 values distributed according to a Gumbel distribution with cumulative distribution function $F(x) = e^{-e^{-x}}$. Represent the outcome in a plot.

Solution

(a)

For the first part we are asked to implement a pseudorandom generator of our choice, and with this method we are asked to produce a sequence of random variables $u_1, u_2, \ldots, u_{1000}$ with the property that

(i) each u_i is uniformly distributed between 0 and 1,

(ii) the u_i are mutually independent.

In general, a pseudorandom generator produces a finite sequence of numbers for a given unit interval, in this case $[0, 1]$. For this problem we decided on a *Linear Congruential Generator* (LCG) of the following form

$$x_{i+1} = (ax_i + c) \bmod m,$$
$$u_{i+1} = x_{i+1}/m,$$

given a seed x_0, where a is the multiplier, c is the increment and m is the modulus. Furthermore, we assume that the variables have to meet the requirements where $m > 0$, $0 < a < m$, $0 \leq c < m$, and $0 \leq x_0 < m$ [1]. In the case when $c \neq 0$ we say that the method is called a mixed congruential generator and when $c = 0$ the generator is called a multiplicative congruential generator. Further assumptions on the variables are made to ensure that the generator has full period.

Proposition 1 (Hull/Dobell 1962 [2]) *The linear congruential generator with generating function $x_{i+1} = (ax_i + c)$ mod m has period m iff the following three conditions hold*

(i) c is relatively prime to m,

(ii) every prime number that divides m divides $a - 1$,

(iii) $a - 1$ is divisible by 4 if m is.

There are plenty of parameters for linear congruential generators that produces pseudorandom numbers such that Proposition 1 holds, to name a few we can utilize $m = 2^{31}, a = 1103515245, c = 12345$ or $m = 2^{16} + 1, a = 75, c = 74$.

A statistical test for Uniformity will be applied, in this case the Kolmogorov-Smirnov test is which is defined as

$$H_0: \text{ The data follow a uniform distribution}$$
$$H_a: \text{The data do not follow the uniform distribution}$$

with the test statistic

$$D = \max_{1 \leq i \leq N} \left(F(U_i) - \frac{i-1}{N}, \frac{i}{N} - F(U_i) \right)$$

where F is the theoretical Uniform distribution for the simulated data U_i. In Python the Kolmogorov-Smirnov test can be performed by the scipy package as `scipy.stats.kstest`.

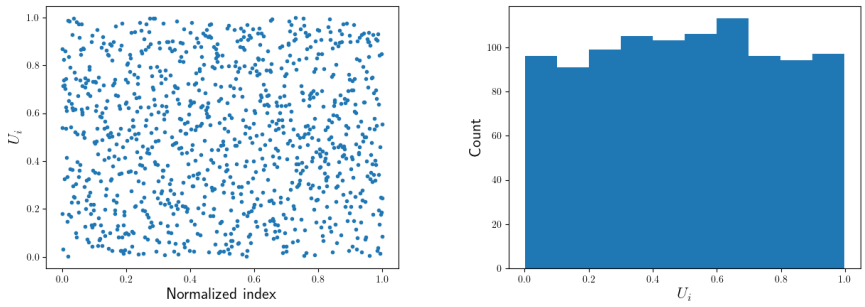
Numerical Results the LCG was implemented with values that satisfy Proposition 1 ($x_0 = 147, m = 2^{31}, a = 1103515245$, and $c = 12345$), which states that the resulting sequence of generated values will be uniformly distributed on the interval $[0, 1]$ if the values of x_0, m, a, and c are chosen correctly.

Figures 1a and 1b provide a visual representation of the generated values. The scatter plot in Figure 1a shows that the values do not appear to exhibit any dependence or structure, which is a desirable property of pseudorandom number generators. The histogram in Figure 1b shows a somewhat uniform distribution on the interval $[0, 1]$. However, it should be noted that the relatively small number of simulated iterations (1000) may result in some variation in the results.

The Kolmogorov-Smirnov test was applied to the generated sequence and yielded a p-value of 0.9 at a 95% confidence level. This indicates that we cannot reject the null hypothesis that the generated values are drawn from a uniform distribution, further supporting the claim that the LCG is providing sufficiently random results.

Additionally, we can calculate the mean and variance of the generated values, which are expected to be $\mu = 0.5$ and $\sigma = 1/12$ for a uniform distribution on $[0, 1]$. The calculated values of $\mu_U \approx 0.5014$ and $\sigma_U \approx 0.0801$ are close to the analytical answer, further supporting the validity of the LCG.

Overall, the scatter plot and histogram of the generated values, as well as the results of the Kolmogorov-Smirnov test and calculations of the mean and variance, all support the conclusion that the LCG provides sufficient and random results.



(a) Generated points (b) Histogram

Figure 1: LCG uniformly distributed

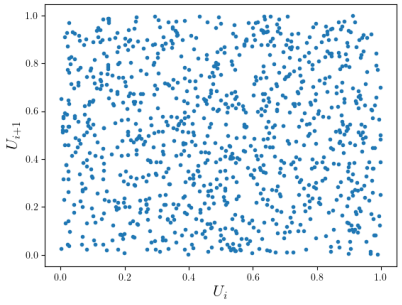


Figure 2: U_{i+1} as a function of U_i

```
Mean of U is 0.5014279581662268
Variance of U is 0.08006437100293842
Kolmogorov-Smirnov-test statistic 0.017821583081036818 and
p-value 0.9027925996219108 for 95% confidence level
```

Solution

(b) We want to sample from a cumulative distribution function F where

$$F(x) = e^{-e^{-x}}$$

is the standard Gumbel distribution. We want to generate the random variable X with property that $\mathbb{P}(X \leq x) = F(x)\ \forall x$. The inverse transform method will be applied to solve for

$$X = F^{-1}(U), \quad U \sim \text{Unif}[0,1]$$

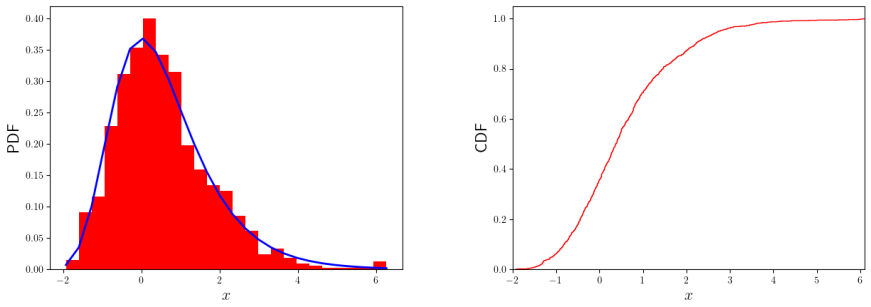
where F^{-1} is the inverse of F and Unif$[0, 1]$ is generated from part **(a)**. The inverse F of the Gumbel distribution is elementary and can be solved as

$$\begin{aligned} F(x) &= U \\ e^{-e^{-x}} &= U \\ -e^{-x} &= \ln U \\ -x &= \ln(-\ln U) \\ x &= -\ln(-\ln U). \end{aligned}$$

Implementation The implementation of the inverse transform method with 1000 iterations of uniform distributed random variables produces the results shown in Figure 3. Figure 3a shows that the resulting probability density function, as represented by a histogram, follows the same structure as the analytical solution, as shown in blue. This indicates that the inverse transform method is providing accurate results.

Furthermore, Figure 3b shows the cumulative density function for the simulated result, which also aligns with the analytical solution. This suggests that the cumulative distribution of the generated values accurately matches the expected distribution, further supporting the validity of the inverse transform method.

Overall, the visualizations in Figure 3 indicate that the implementation of the inverse transform method is producing accurate and reliable results for generating random variables with a specified probability density function.



(a) PDF (b) CDF

Figure 3: Gumbel Distribution

Problem 2.

Let T denote the triangle with vertices $(0,0)$, $(2,0)$ and $(2,2)$. Let

$$f(x,y) = \begin{cases} \sin^2(\pi xy) & \text{if } (x,y) \in T, \\ 0 & \text{otherwise.} \end{cases}$$

(a) Generate 2000 uniformly distributed points in T and plot the outcome.

(b) Using the acceptance-rejection method generate 2000 points in T which are distributed according to the probability density function of the form f/C, where $C > 0$ is a normalizing constant, but without calculating the value of C. Plot the outcome.

Solution

(a)

Given a triangle T with vertices $(0,0)$, $(2,0)$, and $(2,2)$, the Barycentric coordinates of a point within the triangle can be defined as x, $y-x$, and $1-y$, where x and y are uniformly distributed points on the interval $[0,1]$. This allows us to compute the weighted average of the vertices of the triangle, with the weights determined by the Barycentric coordinates to get the coordinates of the points within the triangle. This is done by defining the coordinates of a point within the triangle as a linear combination of the vertices of the triangle, with weights determined by the generated x and y.

Implementation We implemented the Barycentric coordinates method for generating uniformly distributed points within a triangle. First, define T as a triangle with vertices $(0,0)$, $(2,0)$, and $(2,2)$. Then, generate uniformly distributed points on the interval $[0,1]$.

Next, using the uniformly distributed points to compute the Barycentric coordinates of a point within the triangle. This was done in the function `points_on_triangle` using matrix multiplication and the definition of the Barycentric coordinates as a linear combination of the triangle vertices. Finally, calculating the weighted average of the vertices given the Barycentric coordinates with x and y. Figure 4 provides a visual representation, showing the uniformly distributed points generated on the triangle.

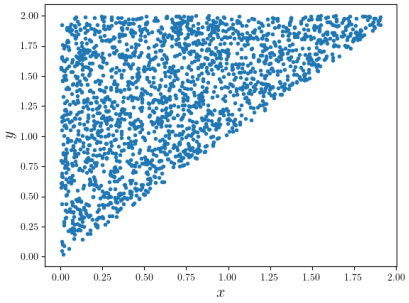


Figure 4: distributed points in T

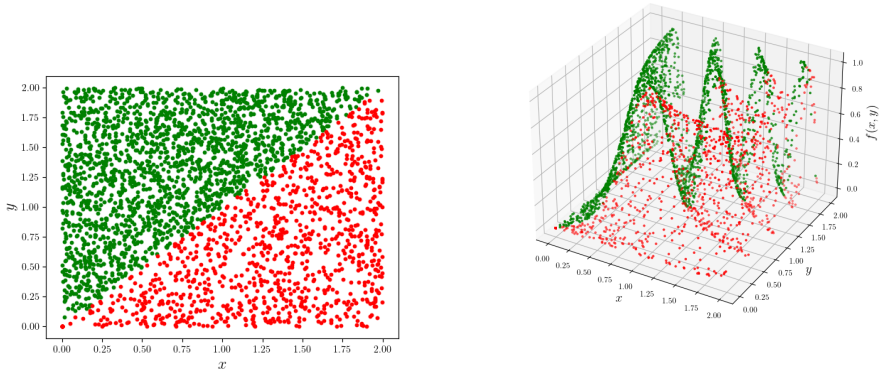
Solution

b

The acceptance-rejection method is a Monte Carlo technique used to generate random samples from a given probability distribution. The basic idea is to generate a sample from a known probability distribution, called the "proposal distribution", and then accept or reject the sample based on its similarity to the desired target distribution. If the sample is accepted, it is used as a sample from the target distribution; if it is rejected, a new sample is generated until an acceptable sample is found. This procedure can be iterated until a desired number of samples from the target distribution are obtained.

Based on the assumption that we can simply simulate on the entire domain of the square rather than the triangle. Then by the acceptance-rejection method we can use the samples generated on the triangle to produce the function $f(x, y)$.

Implementation The simulation is quite similar to part (a), but opposed to only simulating on the triangle we do so by the acceptance-rejection method. The result, shown in Figure 5a where the green scatter is the accepted scatter region and red is the rejected simulation region. Furthermore, we apply the points to $f(x, y) = \sin^2(\pi xy)$, where Figure 5b shows the 3d scatter for the same accepted and rejected simulation.



(a) Simulated points

(b) f(x,y)

Figure 5: Simulated points by the acceptance-rejection method

Problem 3.

Assume that $r = 2\%$ is the risk-free rate. Estimate the price of the Asian put option with strike price $K = 150$ expiring in six months, with a basket of three dividend-free stocks S_1, S_2, S_3 (one share of each) as the underlying instrument. The assumed volatilities of these stocks are, respectively, $\sigma_1 = 0.2, \sigma_2 = 0.3, \sigma_3 = 0.4$, and correlations between the Brownian motions W_1, W_2, W_3 driving S_1, S_2, S_3 are $\varrho_{12} = 0.2, \varrho_{23} = -0.1, \varrho_{13} = -0.2$.

Furthermore, $S_1(0) = 40, S_2(0) = 40, S_3(0) = 70$. Initially, assume that there are 10 uniformly spaced monitoring dates per month. Then investigate how change in the number of monitoring dates changes the price of this option.

Solution

The Asian basket option method is an option pricing model used to price options on a basket of underlying assets. The model takes into account the correlation between the assets in the basket, as well as the volatility of each underlying asset. The model is useful for pricing options on baskets of assets that have a high degree of correlation, such as stocks in the same sector or commodities. The model is also useful for pricing options when the underlying assets have different volatility levels. In the Asian basket option method, the option price is calculated by taking a weighted average of the option prices for each underlying asset, weighted by the correlation between the assets in the basket.

Asian options are path-dependent options where the payoff depends on the average level of the underlying asset(s) until time T. The Asian put (discrete monitoring) has payoff $\phi = (\bar{S} - K)^+$, where the strike price K is a constant and

$$\bar{S} = \frac{1}{N} \sum_{i=1}^{N} S(t_i)$$

is the average price of the underlying S over discrete monitoring dates $t_1, \ldots, T_N$. There are no true analytical solution to this problem as it is highly path-dependant, therefore we will use Monte Carlo methods to estimate the option price.In this problem we want to solve the Asian put option on three stocks S_j, $j = 1, 2, 3$ with correlated Brownian motions; We then assume that we have one share of each and denote the option payoff for the basket option as

$$\left(K - \sum_{j=1}^{3} w_j \bar{S}_j \right)^+$$

Simulating each underlying asset by a Geometric Brownian motion and Euler-Maruyama discretization

$$dS_i(t) = rS_i(t)dt + S_i(t) \sum_{j=1}^{3} \Sigma_{ij} dW_j(t).$$

Let $S_1(t), S_2(t), S_3(t)$ be the stock prices of three stocks at time t. Let ρ_{ij} be the correlation coefficients between $S_i(t)$ and $S_j(t)$ for $i, j = 1, 2, 3$.

Correlated Brownian Motion in Asian Basket Option with three stocks can be represented as follows:

$$dS_i(t) = \mu_i S_i(t)dt + \sigma_i S_i(t)dW_i(t)$$

where μ_i is the expected return rate of stock i, σ_i is the volatility of stock i, $dW_i(t)$ is the standard Wiener process.

Moreover, the correlations between stock prices $S_i(t)$ and $S_j(t)$ is given by:

$$dS_i(t)dS_j(t) = \rho_{ij}\sigma_i\sigma_j S_i(t)S_j(t)dt.$$

The variance reduction technique implemented is antithetic variates which generally works by introducing negative dependence between pairs of replications. In particular, since we are using independent standard normal variables, then implement the antithetic variates by pairing sequences of $Z_1, Z_2, \ldots$ with the sequence $-Z_1, -Z_2, \ldots$. Then $-Z_i$'s simulates the reflection of the path and the idea is that running a pair of simulations may result in lower variance.

The antithetic variates method, in the context of Monte Carlo option pricing, is a variance reduction technique, which can be described mathematically as follows.

Let X_1, X_2 be two i.i.d (independent, identically distributed) random variables, each with a cumulative probability distribution function $F_X(x)$. We use the antithetic variates method by constructing a new random variable Y, as follows:

$$Y = \begin{cases} X_1 & \text{if } U \sim U[0,1] \leq 0.5 \\ 1 - X_2 & \text{if } U \sim U[0,1] > 0.5 \end{cases}$$

The cumulative probability distribution function of Y can be written as:

$$F_Y(y) = 2 \cdot \min\{F_X(y), 1 - F_X(y)\}$$

Therefore, using the antithetic variates method, we can reduce the variance of the estimator of the option price.

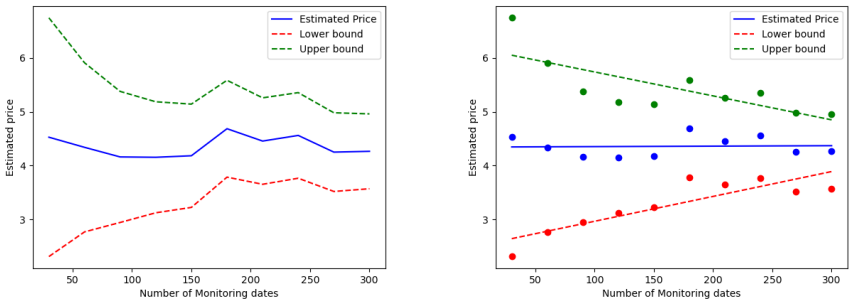
Implementation The `AsianBasket` function calculates the price of an Asian basket option using Monte Carlo simulation with correlated Brownian motion. The function takes in several parameters including the initial stock prices, the strike price, the risk-free interest rate, the time to maturity, the volatilities, the correlation matrix, and the number of time steps and simulations to run. The function outputs the mean price of the option, the 95% confidence interval, and the standard deviation of the price. The function begins by creating an empty price vector and then enters a simulation loop where it generates correlated Brownian motions and uses them to simulate the stock prices. The function then calculates the price of the option and saves it to the price vector. After

the simulation loop is complete, the function calculates the mean, standard deviation, and confidence interval of the price vector and returns these values as output.

The variables $S_0, K, T, N, r, \sigma, \rho_{12}, \rho_{23}, \rho_{13}$, and M are all defined. S_0 is an array of initial stock prices for three different stocks. K is the strike price of the option. T is the time to maturity of the option in years. N is the number of observations per year. r is the risk-free interest rate. sigma is an array of volatilities for the three stocks. ρ_{12}, ρ_{23}, and ρ_{13} are the correlation coefficients between the different stocks. $Corr$ is a correlation matrix derived from the correlation coefficients. M is the number of simulations to run.

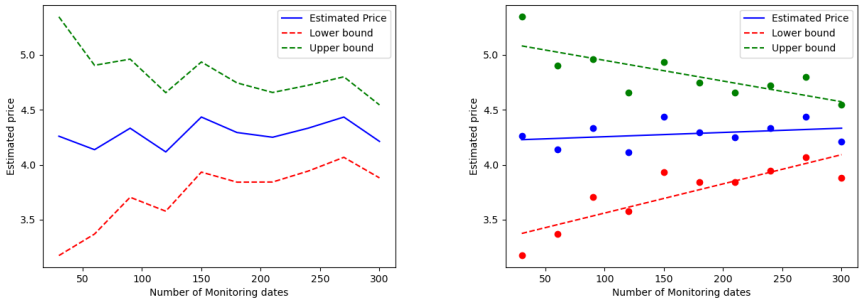
The code then creates a range of possible values for the number of monitoring dates, and uses this range to simulate a basket Asian put option. The resulting values are used to generate a plot of the estimated price, along with lower and upper bounds. The x-axis of the plot is the number of monitoring dates, and the y-axis is the estimated price. The resulting figure can be seen in Figure 6. Figure 6a shows the estimated price as a function of the number of monitoring days, and Figure 6b shows the linear regression of the previous figure. Unfortunately, due to the Basket Asian options not having an analytical solution we can not infer any convergence to a true value. This can be solved using packages that efficiently simulate an approximate to the true solution or utilize a large number of simulations to solve for an another better approximation. Since the problem does not ask for this we can assume the code is correct then as the number of observations per year increase, we expect the estimated price to be a better approximation to the analytical price. This is due to Asian options being strongly path dependant.

Monte Carlo variance reduction is a technique used to improve the accuracy of a Monte Carlo simulation, One way to achieve variance reduction is to use antithetic variates. Figure 7 visualizes this simulation. The similarity is not much different to the original Monte Carlo method but the variance reduction technique results in a standard deviation of 3.09 where the original produces 6.21. In conclusion the antithetic variate technique results in a lower variance and therefore the preferred result.



(a) Estimated price (b) Linear Regression

Figure 6: Estimated price for a basket Asian put option



(a) Estimated price (b) Linear Regression

Figure 7: Estimated price for a basket Asian put option with antithetic variates

Problem 4.

Suppose that the risk-free interest rate is $r = 2.5\%$. Consider a stock with the price process $S(t)$ modeled by a geometric Brownian motion with volatility $\sigma = 0.3$. The stock is assumed to be worth $S(0) = 380$ at present. Suppose that $C(t)$ denotes the price at time t of the European call option written on this stock expiring in one year, with strike price $L = 400$. Estimate the price of the compound call-on-call option with strike price $K = 36$ and maturing in 3 months, written over that call option. By considering several different values of L below and above 400, investigate how L influences the price of the compound option.

Solution

Monte Carlo Compound Call-on-Call Option is a type of option contract that allows the holder to purchase a call option at a predetermined strike price on a predetermined underlying security at the expiration of a preceding call option.

The option is structured such that the buyer purchases two call options, a compound call and a preceding call. The preceding call has a shorter expiration date than the compound call, and both options are based on the same underlying asset. The strike price of the preceding call is typically lower than the strike price of the compound call. In other words, the difference between the two strikes is a long call spread.

At the expiration of the preceding call, the holder has the right to exercise the compound call, which will give them the right to purchase the underlying asset at the compound call's strike price. If the underlying asset's price is above the compound call's strike price, the holder will make a profit. Otherwise, they will make a loss.

The steps for Monte Carlo simulation follow the same structure as Problem 3. The difference is that we want to price a European call-on-call option. The compound option consists of two strikes L and K with expiration dates T_1 and T_2, respectively. By intuition at T_2 the holder has the right to buy the underlying call option. Therefore we can model the payoff as

$$C(x) = e^{-r(T1-T2)} E\left[(S(T_1) - K_1)^+ | S(T_2) = x\right].$$

The second step is to value the outer option at $t = 0$

$$V_{\text{CoC}} = e^{-rT_2} E[(C(S(T_2)) - K_2)^+].$$

In the simulation using Monte Carlo, we first simulate N values of the stock at time T_2

$$S_1(T_2), S_2(T_2), \ldots, S_N(T_2)$$

and using those stock simulations we simulate N values of the stock at time T_1

$$S_{11}(T_1), \ldots, S_{NN}(T_1).$$

For each of the simulated values at time T_2 we estimate the price of the inner option

$$\hat{C}_i(S_i(T_2)) = \frac{1}{N} e^{-r(T_1-T_2)} \sum_{j=1}^{N} (S_{ij}(T_1) - K_1)^+.$$

lastly, we utilize the estimate to simulate the price of the compound option

$$\hat{C} = \frac{1}{N} e^{-rT_2} \sum_{i=1}^{N} \left(\hat{C}_i(S_i(T_2)) - K_2 \right)^+ \Big).$$

The control variate technique is a tool for reducing the variance of estimates in Monte Carlo simulation. In the context of derivative pricing, the absence of arbitrage is equivalent to the requirement that appropriately discounted asset prices be martingales. This property can be used to construct a control variate estimator for the option price, by combining the estimated option price with the known value of the underlying asset's discounted price using the optimal weight.

To be more specific, suppose we are pricing a European option on an underlying asset with price $S(t)$. If the interest rate is a constant r, then the discounted asset price $e^{-rt}S(t)$ is a martingale, and its expected value at any time T is equal to its initial value $S(0)$. We can use this property to construct a control variate estimator for the option price.

From independent replications of the underlying asset's path over the time interval $[0, T]$, we can form the estimator:

$$\hat{V} = \frac{1}{N} \sum_{i=1}^{n} Y_i - \hat{b}(S_i(T) - e^{rT} S(0))$$

where Y_i is the discounted payoff of the option, $S_i(T)$ is the value of the underlying asset at time T in the i'th replication, and $\hat{b}$ is a weight that minimizes the variance of the estimator. This weight can be calculated using the following formula:

$$\hat{b} = \frac{\sum_{i=1}^{N} (S_i(T) - \bar{S(T)})(Y_i - \bar{Y})}{\sum_{i=1}^{N} (S_i(T) - \bar{S(T)})^2} \tag{1}$$

By combining the estimated option price with the known value of the underlying asset's discounted price using the optimal weight, the variance of the estimate can be reduced, resulting in more accurate option pricing.

Implementation The `call_on_call_option_price` function calculates the price of a call option on a call option using Monte Carlo simulation. To calculate the price of a call-on-call option using Monte Carlo simulation, the function first simulates the stock price for a period of time equal to the time to maturity of the underlying call option (denoted as T_2). This generates a sample of M simulated stock price paths, which we will denote as $\{S_1, \ldots, S_M\}$. For each simulated stock price path, the function calculates the price of the underlying call option

at the expiration of the double call option (denoted as T_1) using $\{S_1, \ldots, S_M\}$ as initial stock prices for the second stock simulation $\{S_{11}, \ldots, S_{MM}\}$ with strike price L and expiration date $T_1 - T_2$. This generates a sample of M prices for the underlying call option, which we will denote as $\{C_1, \ldots, C_M\}$.

Next, the function calculates the payoffs for each of these underlying call option prices. The payoff for a price C_i is defined as the maximum of 0 and the difference between the price of the underlying call option and the strike price of the double call option, i.e.

$$V_i = \max(C_i - K, 0)$$

where K is the strike price of the double call option. Finally, the function returns the expected value of these payoffs, discounted back to the current time, i.e.

$$\hat{C} = \mathbb{E}[V] \cdot \exp(-rT_2)$$

where $\mathbb{E}[F]$ is the expected value of the payoffs and r is the risk-free interest rate. This expected value is approximated using the sample mean of the payoffs, i.e.

$$\hat{C} = \frac{1}{M} \sum_{i=1}^{M} V_i \cdot \exp(-rT_2)$$

and this is the first value that the `call_on_call_option_price` function returns.

To calculate the price of a double call option using Monte Carlo simulation with control variates, the function first simulates the stock price for a period of time equal to the time to maturity of the underlying call option (denoted as T_2). This generates a sample of M simulated stock price paths $\{S_1, \ldots, S_M\}$. For each simulated stock price path, the function calculates the price of the underlying call option at the expiration of the double call option (denoted as T_1) using the `blsprice_mc_cv` function. This function uses the `BetaEstimation` function to calculate the weight that minimizes the variance of the estimator from eq. 1 between the simulated stock prices and the corresponding payoffs of the underlying call options.

Next, the function calculates the payoffs for each of these underlying call option prices. The payoff for a price C_i

$$V_i = \max(C_i - K, 0)$$

where K is the strike price of the call-on-call option. Finally, the function returns the expected value of these payoffs, discounted back to the current time, i.e.

$$\hat{C}_{\text{cv}} = \mathbb{E}[V] \cdot \exp(-rT_2)$$

where $\mathbb{E}[V]$ is the expected value of the payoffs and r is the risk-free interest rate. This expected value is approximated using the sample mean of the payoffs and this is the value that the `call_on_call_option_price_cv` function returns.

To answer the first question regarding how L influences the price of the compound option we simulate the call-on-call price using both the standard simulation and control variate simulation. The resulting estimates are shown in Figure 8 .The figure shows the estimated price as a function of $L = 300, 310, \ldots, 500$, and the resulting plot mimics a exponential decay when the strike price L increases. By intuition this is because when L becomes too large, the simulation for the call-on-call option gets fever and smaller inputs as the initial stock price for the second stock simulation.

The second investigation is if the variance reduction technique, control variates, reduces the variance. For this we compared the standard deviation of both methods shown in Figure 9. The result shows that the control variate manages to marginally decrease the standard deviation from 0.284 to 0.225, in the figure this is represented as a concentrated density. By the experiments made, this was not always the case and sometimes the result showed option price estimations that varied quite a lot for the two methods without any lower variance. Finally, the CI and estimate for the standard method is $\text{CI} = [13.2, 13.36]$, $\hat{C} = 13.30$ and for the control varites method is $\text{CI}_{\text{cv}} = [13.93, 14.01]$, $\hat{C}_{\text{cv}} = 13.97$.

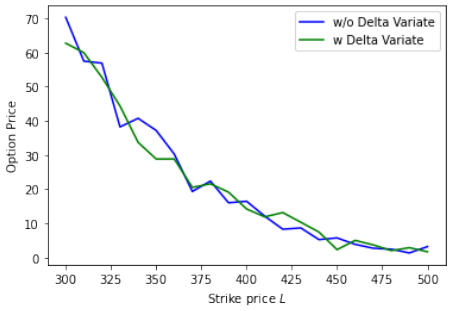


Figure 8: The simulated price as a function of L

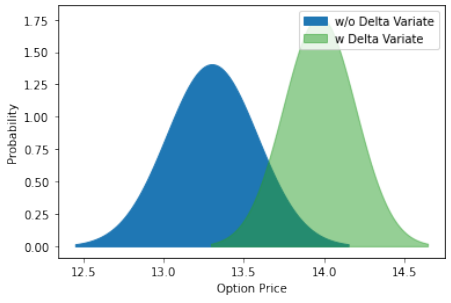


Figure 9: The probability as a function of the option price

References

[1] P. Glasserman, *Monte Carlo Methods in Financial Engineering*. Applications of mathematics : stochastic modelling and applied probability, Springer, 2004.

[2] T. E. Hull and A. R. Dobell, "Random number generators," *SIAM Review*, vol. 4, no. 3, pp. 230–254, 1962.

A Code

A.1 Question 1

```
# Initialize packages
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats
plt.rcParams['text.usetex'] = True

# (a)
def generate_lcg(iter):
    """
    Linear congruential generator to generate pseudorandom
    uniformly distributied points in [0,1]
    :param iter: number of simulated variables (int)
    :return u: Vector of uniformly distributied points in [0,1]
    """
    # Inital variables
    x = 147               # Seed
    a = 1103515245        # Multiplier
    c = 12345             # Increment
    m = (2 ** 31)         # modulus
    u = []                # Output vector

    # Loop for number of iterations
    for num in range(iter):
        # LCG formula
        x = (a * x + c) % m

        # Solve uniformly distributied points in [0,1]
        u.append(x/m)

    return u

def NormalizeData(data):
    """
    Normalize data in [0,1]
    :param data: defined size
    :return out: Vector of Normalize data in [0,1]
    """
    return (data - np.min(data)) / (np.max(data) - np.min(data))

iter = 1000 # Number of points
U = generate_lcg(iter) # Simulate uniformly distributed point
```

```
mu = np.mean(U) # Mean of uniformly distributed points
sigma = np.var(U) # Variance of uniformly distributed point

print(f'Mean of U is {mu}')
print(f'Variance of U is {sigma}')

# (Figure 1) Generated U_i's as a function of the normalized index
plt.figure(1)
plt.scatter(NormalizeData(range(len(U))),U,marker='.')
plt.ylabel('$U_i$',fontsize=16)
plt.xlabel('Normalized index',fontsize=16)
plt.show()

# (Figure 2) Count as a function of the generated U_i's
plt.figure(2)
plt.hist(U,bins=10)
plt.ylabel('Count',fontsize=16)
plt.xlabel('$U_i$',fontsize=16)
plt.show()

# Vectors for Figure 3
U1=U[1:iter-1]
U2=U[2:iter]

# (Figure 3) generated U_{i+1}'s as a function of U_i
plt.figure(3)
plt.scatter(U1,U2,marker='.')
plt.xlabel('$U_i$',fontsize=16)
plt.ylabel('$U_{i+1}$',fontsize=16)
plt.show()

# Kolmogorov-Smirnov test
stat, p = stats.kstest(U, stats.uniform(loc=0.0, scale=1).cdf)
print(f'Kolmogorov-Smirnov-test statistic {stat} and p-value {p}
    for 95% confidence level')

# (b)
def gumbel_icdf(U):
    """
    Inverse transform method applied to Gumbel distribution
    :param U: Uniform distributed data
    :return: Gumbel distributed variables
    """
    return -np.log(-np.log(U))

# Generate Gumbel distribution (iter=1000)
```

```
x = gumbel_icdf(U)

# Gumbel distribution analytical solution
x_analytical = np.random.gumbel(loc=0.0, scale=1.0, size=100000)

# (Figure 4) PDF plot for Gumbel distribution with analytical PDF
plt.figure(4)
count, bins, ignored = plt.hist(x, 25, color='red',
                        density=True, linewidth=2)
plt.plot(bins, np.exp(-bins)*
        np.exp(-np.exp(-bins)),linewidth=2, color='b')
plt.xlabel('$x$',fontsize=16)
plt.ylabel('PDF',fontsize=16)
plt.show()

# (Figure 5) CDF plot for Gumbel distribution
plt.figure(5)
plt.hist(x, bins=1000, density=True, cumulative=True, label='CDF',
         histtype='step', color='r')
plt.xlim(-2,6.1)
plt.xlabel('$x$',fontsize=16)
plt.ylabel('CDF',fontsize=16)
plt.show()
```

A.2 Question 2

```
# 2 (a)
def points_on_triangle(iter):
    """
    Give iter random points uniformly on a triangle where the vertices are given
    by (0,0), (0,2), and (2,2).

    :param iter: number of points
    :out: random points uniformly on a triangle
    """
    # Vertices
    v = np.array([(0, 0), (0, 2), (2, 2)])
    # 2 dimensional standard normal random points
    x = np.sort(np.random.rand(2, iter), axis=0)
    return np.column_stack([x[0], x[1]-x[0], 1-x[1]]) @ v
def points_on_triangle_inv(iter):
    """
    Give iter random points uniformly on a triangle where the vertices are given
    by (0,0), (0,2), and (2,2).

    :param iter: number of points
```

```
    :out: random points uniformly on a triangle
    """
    v = np.array([(0, 0), (0, 2), (2, 2)])
    x = np.sort(np.random.rand(2, iter), axis=0)
    return np.column_stack([1-x[0], -x[1]+x[0], x[1]]) @ v

# Number of iterations
iter = 2000
# Simulate points on triangle
points = points_on_triangle(iter)
# Aggregate into tuple
x, y = zip(*points)

# (Figure 6) Plot of uniformly generated poitns on triangle
plt.figure(6)
plt.scatter(x, y, s=10)
plt.xlabel('$x$', fontsize=16)
plt.ylabel('$y$', fontsize=16)
plt.show()

# 2 (b)
def f(x,y):
    return np.sin(np.pi*x*y)**2
iter = 2000

i = 0
x_accept = np.zeros(iter)
y_accept = np.zeros(iter)
x_reject = np.zeros(iter)
y_reject = np.zeros(iter)
while i < iter:
    x = np.random.random()*2
    y = np.random.random()*2
    if y >= x:
        x_accept[i] = x
        y_accept[i] = y
        i = i + 1
    else:
        x_reject[i] = x
        y_reject[i] = y

plt.scatter(x_accept,y_accept,s=10,color='green')
plt.scatter(x_reject,y_reject,s=10,color='red')
plt.xlabel('$x$', fontsize=16)
plt.ylabel('$y$', fontsize=16)
```

```
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(projection='3d')

ax.scatter(x_accept, y_accept, f(x_accept,y_accept), s=5, color='green')
ax.scatter(x_reject, y_reject, f(x_reject,y_reject), s=5, color='red')
ax.set_xlabel('$x$', fontsize=16)
ax.set_ylabel('$y$', fontsize=16)
ax.set_zlabel('$f(x,y)$', fontsize=16);
plt.show()
```

A.3 Question 3

```
def MonteCarlo(S0, r, T, sigma, W, N):
    """
    Monte Carlo simulation of a stock price
    :param S0: initial stock price
    :param r: risk-free interest rate
    :param T: time to maturity
    :param sigma: volatility
    :param W: Brownian motion
    :param N: number of time steps
    :return: simulated stock price
    """
    # Time steps
    dt = T / N
    # Empty stock vector
    S = np.zeros(N)
    # Assign initial stock price
    S[0] = S0
    # Time loop
    for i in range(1, N):
        # Simulate stock movement
        S[i] = S[i-1] * np.exp((r - sigma ** 2 / 2) * dt + sigma * W[i]
               * np.sqrt(dt))
    return S

def AsianBasket(S0, K, r, T, sigma, Corr, M, N):
    """
    Calculates the price of an Asian basket option using Monte Carlo
    simulation with correlated Brownian motion.

    :param S0: Vector of initial stock prices.
    :param K: Strike price
    :param r: Risk-free interest rate
    :param T: Time to maturity
    :param sigma: Vector of volatilities
```

```
    :param Corr: Correlation matrix
    :param N: Number of time steps
    :Param M: Number of simulations
    :out mean_price: Mean price of the option
    :out CI: 95% confidence interval
    :out std_price: Standard deviation of the price
    """
    price = np.zeros(M) # Empty price vector
    for j in range(M): # Simulation loop
        S = np.zeros([3,N]) # Empty stock matrix

        # Generate correlated Brownian motions
        Z1 = np.random.normal(size=(N, 1))
        Z2 = Corr[0][1] * Z1 + np.sqrt(1 - Corr[0][1] ** 2)
             * np.random.normal(size=(N, 1))
        Z3 = Corr[0][2] * Z1 + Corr[1][2] * Z2 + np.sqrt(1 -
             Corr[0][2] ** 2 - Corr[1][2] ** 2 - 2 *  Corr[0][1]
             * Corr[0][2] * Corr[1][2])* np.random.normal(size=(N, 1))

        # Simulate stock prices
        S[0,:] = MonteCarlo(S0[0], r, T, sigma[0], Z1, N)
        S[1,:] = MonteCarlo(S0[1], r, T, sigma[1], Z2, N)
        S[2,:] = MonteCarlo(S0[2], r, T, sigma[2], Z3, N)

        # Calculate the basket stock price
        Shat = (S[0,:] + S[1,:] + S[2,:])

        # Calculate the price of the option
        price[j] = np.maximum(K-np.mean(Shat),0)*np.exp(-r*T)

    mean_price = np.mean(price) # Mean price
    std_price = np.std(price) # Standard deviation
    # Confidence interval
    CI = [mean_price - 1.96*std_price/np.sqrt(N), mean_price
        + 1.96*std_price/np.sqrt(N)]
    return mean_price, CI, std_price

S0 = np.array([40,40,70]) # Inital stock prices for S_1, S_2, and S_3
K = 150 # Strike price
T = 0.5 # Expiration in 6 months
N = 60 # Number of observations per year
r = 0.02 # Interest rate
sigma = np.array([0.2,0.3,0.4]) # Initial volatility
rho12, rho23, rho13 = 0.2, -0.1, -0.2 # Correlation coefficients
Corr = np.array([(1, rho12, rho13), (rho12, 1, rho23), (rho13, rho23, 1)])
       # Correlation matrix
```

```
M = 1000  #Number of simulations

N = np.linspace(30,300,10, dtype=int) # Vector for number of monitoring dates
mean_price = np.zeros([M,len(N)]) # Empty matrix for F0
CIs = np.zeros([len(N),2]) # Empty matrix for confidence interval
std_price = np.zeros([len(N),1]) # Empty vector for standard deviation
for j in range(len(N)): # loop for monitoring dates
    # Simulate basket Asian put
    mean_price[:,j], CIs[j], std_price[j] = AsianBasket(S0, K, r, T,
                                            sigma, Corr, M, N[j])
print(f'Standard Deviation {np.mean(std_price)}')

plt.figure(9)
line1, = plt.plot(N,np.mean(mean_price,axis=0),'b', label='Estimated Price')
line2, = plt.plot(N,CIs[:,0], 'r--', label='Lower bound')
line3, = plt.plot(N,CIs[:,1], 'g--', label='Upper bound')
plt.xlabel('Number of Monitoring dates')
plt.ylabel('Estimated price')
plt.legend(handles=[line1, line2, line3])
plt.show()

# Linear regression on estimated price
plt.figure(10)
plt.scatter(N,np.mean(mean_price,axis=0),color='b')
plt.scatter(N,CIs[:,0],color='r')
plt.scatter(N,CIs[:,1],color='g')
N = N.reshape((-1, 1))
m1 = LinearRegression().fit(N, np.mean(mean_price,axis=0))
y1 = m1.predict(N)
m2 = LinearRegression().fit(N, CIs[:,0])
y2 = m2.predict(N)
m3 = LinearRegression().fit(N, CIs[:,1])
y3 = m3.predict(N)
line1, = plt.plot(N,y1,'b', label='Estimated Price')
line2, = plt.plot(N,y2, 'r--', label='Lower bound')
line3, = plt.plot(N,y3, 'g--', label='Upper bound')
plt.xlabel('Number of Monitoring dates')
plt.ylabel('Estimated price')
plt.legend(handles=[line1, line2, line3])
plt.show()

# Antithetic Variates
def MonteCarlo_av(S0, r, T, sigma, W, N):
    """
    Computes the Monte Carlo simulation of the price of a stock with antithetic variates
    :param S0: initial price of the stock
```

```
    :param r: risk-free interest rate
    :param T: maturity
    :param sigma: volatility
    :param W: Brownian motion
    :param N: number of steps
    :return: the price of the stock at maturity
    """
    # Time steps
    dt = T / N
    # Empty stock vector
    S1, S2 = np.zeros(N), np.zeros(N)
    # Initial stock price
    S1[0], S2[0] = S0, S0
    # timestep loop
    for i in range(1, N):
        # Simulate stocks
        S1[i] = S1[i-1] * np.exp((r - sigma ** 2 / 2) * dt + sigma * W[i] * np.sqrt(dt))
        S2[i] = S2[i-1] * np.exp((r - sigma ** 2 / 2) * dt + sigma * (-W[i]) * np.sqrt(dt))
    return S1,S2

def AsianBasket_av(S0, K, r, T, sigma, Corr, M, N):
    """
    Calculates the price of an Asian basket option using Monte Carlo simulation
    with correlated Brownian motion and antithetic variates

    :param S0: Vector of initial stock prices.
    :param K: Strike price
    :param r: Risk-free interest rate
    :param T: Time to maturity
    :param sigma: Vector of volatilities
    :param Corr: Correlation matrix
    :param N: Number of time steps
    :Param M: Number of simulations
    :out mean_price: Mean price of the option
    :out CI: 95% confidence interval
    :out std_price: Standard deviation of the price
    """
    # empty price vector
    price = np.zeros(M)
    for j in range(M):
        S1,S2 = np.zeros([3,N]),np.zeros([3,N]) # Empty stock matrix
        # Generate correlated Brownian motions
        Z1 = np.random.normal(size=(N, 1))
        Z2 = Corr[0][1] * Z1 + np.sqrt(1 - Corr[0][1] ** 2)
             * np.random.normal(size=(N, 1))
        Z3 = Corr[0][2] * Z1 + Corr[1][2] * Z2 + np.sqrt(1
```

```
            - Corr[0][2] ** 2 - Corr[1][2] ** 2 - 2 * Corr[0][1] *
            Corr[0][2] * Corr[1][2])* np.random.normal(size=(N, 1))

        # Simulate stock prices
        S1[0,:], S2[0,:]  = MonteCarlo_av(S0[0], r, T, sigma[0], Z1, N)
        S1[1,:], S2[1,:]  = MonteCarlo_av(S0[1], r, T, sigma[1], Z2, N)
        S1[2,:], S2[2,:]  = MonteCarlo_av(S0[2], r, T, sigma[2], Z3, N)

        # Calculate the basket stock price
        Shat1 = S1[0,:] + S1[1,:] + S1[2,:]
        Shat2 = S2[0,:] + S2[1,:] + S2[2,:]

        # Calculate the price of the option
        price[j] = (np.maximum(K-np.mean(Shat1),0)+np.maximum(K-
        np.mean(Shat2),0))/2*np.exp(-r*T)

    mean_price = np.mean(price) # Mean payoff
    std_price = np.std(price) # Standard deviation
    CI = [mean_price - 1.96*std_price/np.sqrt(N), mean_price
          + 1.96*std_price/np.sqrt(N)]
    return mean_price, CI, std_price

N = np.linspace(30,300,10, dtype=int) # Vector for number of monitoring dates
mean_price = np.zeros([int(M),len(N)]) # Empty matrix for F0
CIs = np.zeros([len(N),2]) # Empty matrix for confidence interval
std_price = np.zeros([len(N),1]) # Empty vector for standard deviation
for j in range(len(N)): # loop for monitoring dates
    # Simulate basket Asian put with antithetic variates
    mean_price[:,j], CIs[j], std_price[j] = AsianBasket_av(S0, K, r,
                                            T, sigma, Corr, M, N[j])
print(f'Standard Deviation {np.mean(std_price)}')

plt.figure(11)
line1, = plt.plot(N,np.mean(mean_price,axis=0),'b', label='Estimated Price')
line2, = plt.plot(N,CIs[:,0], 'r--', label='Lower bound')
line3, = plt.plot(N,CIs[:,1], 'g--', label='Upper bound')
plt.xlabel('Number of Monitoring dates')
plt.ylabel('Estimated price')
plt.legend(handles=[line1, line2, line3])
plt.show()

# Linear regression on estimated price
plt.figure(12)
plt.scatter(N,np.mean(mean_price,axis=0),color='b')
plt.scatter(N,CIs[:,0],color='r')
plt.scatter(N,CIs[:,1],color='g')
```

```
N = N.reshape((-1, 1))
m1 = LinearRegression().fit(N, np.mean(mean_price,axis=0))
y1 = m1.predict(N)
m2 = LinearRegression().fit(N, CIs[:,0])
y2 = m2.predict(N)
m3 = LinearRegression().fit(N, CIs[:,1])
y3 = m3.predict(N)
line1, = plt.plot(N,y1,'b', label='Estimated Price')
line2, = plt.plot(N,y2, 'r--', label='Lower bound')
line3, = plt.plot(N,y3, 'g--', label='Upper bound')
plt.xlabel('Number of Monitoring dates')
plt.ylabel('Estimated price')
plt.legend(handles=[line1, line2, line3])
plt.show()
```

A.4 Question 4

```
def simulate_stock_price(S0, r, T, sigma, N):
    """
    Simulate a stock price using the geometric Brownian motion model
    :param S0: initial stock price
    :param r: risk-free interest rate
    :param T: time to maturity
    :param sigma: volatility
    :return: simulated stock price at time T
    """
    dt = T / N
    S = np.zeros(N)
    S[0] = S0
    for i in range(1, N):
        W = np.random.randn()
        S[i] = S[i-1] * np.exp((r - sigma ** 2 / 2) * dt + sigma * W * np.sqrt(dt))
    return S[-1]
def blsprice_mc(S0, K, T, r, sigma, N, M):
    """
    Calculate the price of a European call option using Monte Carlo simulation
    """
    S = np.zeros((M))
    for j in range(M):
        S[j] = simulate_stock_price(S0, r, T, sigma, N)
    C = np.maximum(S-K,0)*np.exp(-r*T)
    return np.mean(C)
def call_on_call_option_price(S0, L, K, T1, T2, r, sigma, N, M):
    """
    Calculate the price of a call option on a call option using Monte Carlo simulation
```

```
    :param S0: initial stock price
    :param L: strike price of the underlying call option
    :param K: strike price of the call option on the call option
    :param T1: time to maturity of the call option on the call option
    :param T2: time to maturity of the underlying call option
    :param r: risk-free interest rate
    :param sigma: volatility
    :param N: number of time steps for the Monte Carlo simulation
    :param M: number of Monte Carlo simulations
    :return: price of the call option on the call option
    :return: standard deviation of the call option on the call option
    """
    F0 = np.zeros(M)
    Payoff = np.zeros(M)
    for j in range(M):
        # Generate stock price path
        S = simulate_stock_price(S0, r, T2, sigma, N)
        # Calculate price of underlying call option
        Payoff[j] = blsprice_mc(S, L, T1 - T2, r, sigma, N, M)
        # Calculate payoff of call option on call option
        F0[j] = np.maximum(Payoff[j] - K, 0) * np.exp(-r * T2)
    # Return average of all payoffs
    return np.mean(F0), np.std(F0)
def BetaEstimation(X,Y):
    num = 0
    denom = 0
    e1,e2 =  np.mean(X), np.mean(Y)
    for i,x in enumerate(X):
        num += (X[i] - e1) * (Y[i] - e2)
        denom += (X[i] - e1) ** 2
    return num/denom
def blsprice_mc_cv(S0, K, T, r, sigma, N, M):
    """
    Calculate the price of a European call option using Monte Carlo simulation
    """
    S = np.zeros((M))
    for j in range(M):
        S[j] = simulate_stock_price(S0, r, T, sigma, N)
    F0 = np.maximum(S-K,0)*np.exp(-r*T)
    C = F0 - BetaEstimation(S,F0) * (S-S0*np.exp(r*T))
    return np.mean(C)
def call_on_call_option_price_cv(S0, L, K, T1, T2, r, sigma, N, M):
    """
    Calculate the price of a call option on a call option using Monte Carlo simulation
    :param S0: initial stock price
    :param L: strike price of the underlying call option
```

```
    :param K: strike price of the call option on the call option
    :param T1: time to maturity of the call option on the call option
    :param T2: time to maturity of the underlying call option
    :param r: risk-free interest rate
    :param sigma: volatility
    :param N: number of time steps for the Monte Carlo simulation
    :param M: number of Monte Carlo simulations
    :return: price of the call option on the call option
    :return: standard deviation of the call option on the call option
    """
    F0 = np.zeros(M)
    Payoff = np.zeros(M)
    for j in range(M):
        # Generate stock price path
        S = simulate_stock_price(S0, r, T2, sigma, N)
        # Calculate price of underlying call option
        Payoff[j] = blsprice_mc(S, L, T1 - T2, r, sigma, N, M)
        # Calculate payoff of call option on call option
        F0[j] = np.maximum(Payoff[j] - K, 0) * np.exp(-r * T2)
    # Return average of all payoffs
    return np.mean(F0), np.std(F0)
S0 = 380 # Inital stock price
L = [300,310,320,330,340,350,360,370,380,390,400,410,420,
     430,440,450,460,470,480,490,500] # Strike price
K = 36 # Strike price CC
T2 = 3/12 # Expiration in 3 months
T1 = 12/12 # Expiration in 12 months
r = 0.025 # Interest rate
sigma = 0.3 # Volatility
M = 100  # Number of simulations
N = 100 # Number of time steps

price = np.zeros(len(L))
std = np.zeros(len(L))
price_cv = np.zeros(len(L))
std_cv = np.zeros(len(L))
for i in range(len(L)):
    price[i], std[i] = call_on_call_option_price(S0, L[i], K, T1, T2, r, sigma, N, M)
    price_cv[i], std_cv[i] = call_on_call_option_price_cv(S0, L[i],
                             K, T1, T2, r, sigma, N, M)
plt.plot(L,price,color='blue' , label='w/o Delta Variate')
plt.plot(L,price_cv,color='green', label='w Delta Variate')
plt.xlabel("Strike price $L$")
plt.ylabel("Option Price")
plt.legend()
plt.show()
```

```
L = 400
M = 100
price, std = call_on_call_option_price(S0, L, K, T1, T2, r, sigma, N, M)
price_cv, std_cv = call_on_call_option_price_cv(S0, L, K, T1, T2, r, sigma, N, M)
std = std/100
std_cv = std_cv/100
x = np.linspace(price-3*std, price+3*std, 100)
xw = np.linspace(price_cv-3*std_cv, price_cv+3*std_cv, 100)
s = stats.norm.pdf(x, price, std)
sw = stats.norm.pdf(xw, price_cv, std_cv)
plt.fill_between(x, s, color='tab:blue', label='w/o Delta Variate')
plt.fill_between(xw, sw, color='tab:green',alpha=0.5, label='w Delta Variate')
plt.ylabel("Probability")
plt.xlabel("Option Price")
plt.legend()
plt.show()
```