Computational Finance Assignment 2

Misho Yanakiev, Max Meijer, Diogo Franquinho

March 8, 2021

1 Part I: Basic Option Valuation

We have written a Monte Carlo method program to find the option price. The computed option price is 4.774300376168909. The algorithm is explained below.
First, we evaluated various stock values at time $T = 1$ using the formula $S_T = S_0 e^{(r-0.5\sigma^2)T+\sigma\sqrt{T}Z}$ with $Z \sim \mathcal{N}(0,1)$. Then we computed the payoffs of the put option for these values of S_T. Taking these various payoffs we took the mean and discount it to get the price of the option at time $t = 0$. In our experiment we took the payoffs of 1 000 000 stock prices.
We also ran a similar program to the one from the last report to estimate the price of a European put option using the 50 step binomial tree estimate. The result obtained was of 4.781123675595926.
Furthermore we also calculated the theoretical value using the Black-Scholes formula which is 4.778969051891707.
In conclusion the binomial method for estimating the option price is actually closer to the theoretical value of the option using Black-Scholes even tough in the Monte Carlo method we use do many more simulations . To make a unbiased comparison of the two methods we plotted the error of both methods in relation to the number of simulations fig. 1.

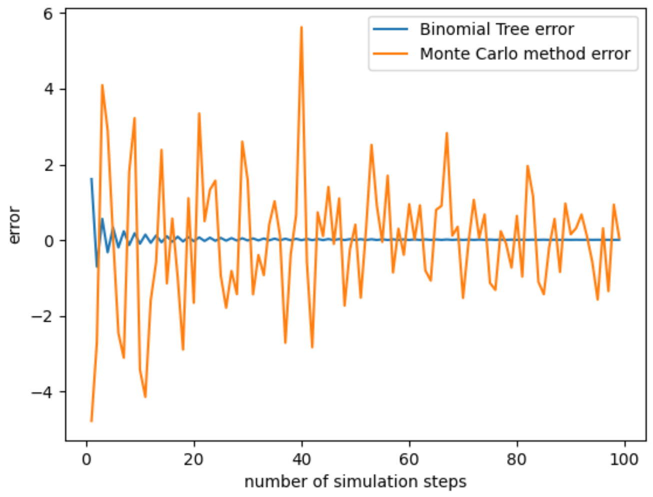

Figure 1: Binomial Tree and Monte Carlo method error

Standard error of our estimate

Furthermore we calculated the standard error of our estimate using the formula:

$$\frac{\sigma(payoffs)}{\sqrt{n}}$$

In our case note that $n = 10^6$ and $\sigma(payoffs)$ is going to be the standard deviation of the different payoffs produced by the Monte Carlo method. Our value for the standard error is 0.007968797353218282. This means that the 95% confidence interval for the estimate is $4.774300376168909 \pm 0.01561884281$, which means that in 95% of the runs the computed value is off by at most 0.01561884281 of the true value. (It should be noted that the confidence interval radius is itself an estimate based on the sample variance (instead of the true variance).) Indeed, we see that the Black-Scholes formula and the Monte Carlo estimate differ by 0.00466867572 which is less than 0.01561884281.
In fig. 2 we plotted a histogram of 10 000 Monte Carlo estimates each using 10 000 sample paths. This distribution when evaluated in it's limit should converge to a normal distribution with mean equal to the true value of the option. From the histogram we see some resemblance to the normal distribution starting to occur, but more estimates would be needed for this histogram to truly look like a normal distribution.

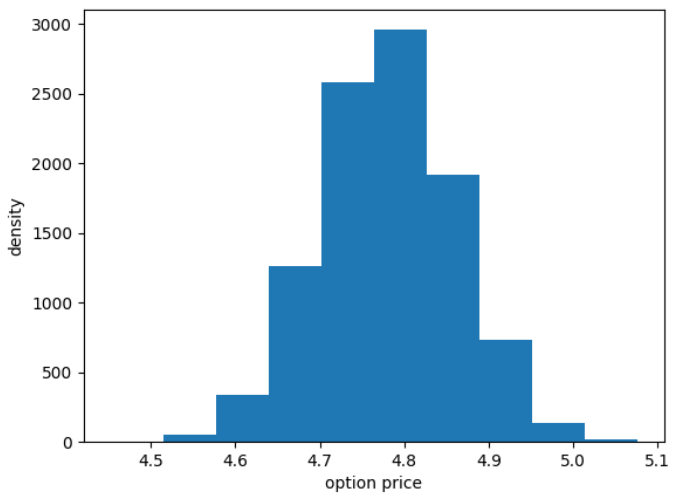

Figure 2: Distribution of the Monte Carlo estimates of the option price

Varying the number of Simulations

In fig. 3 we see how the estimate converges to the theoretical value given by the Black-Scholes formula as more and more simulations are run. In fig. 4 the absolute error has been computed as the absolute value of the difference between the estimated option price and the Black-Scholes option price. It can be seen that the error decreases as the number of simulations increases although the convergence is not monotone. This is likely to be caused by random fluctuations around the true value.

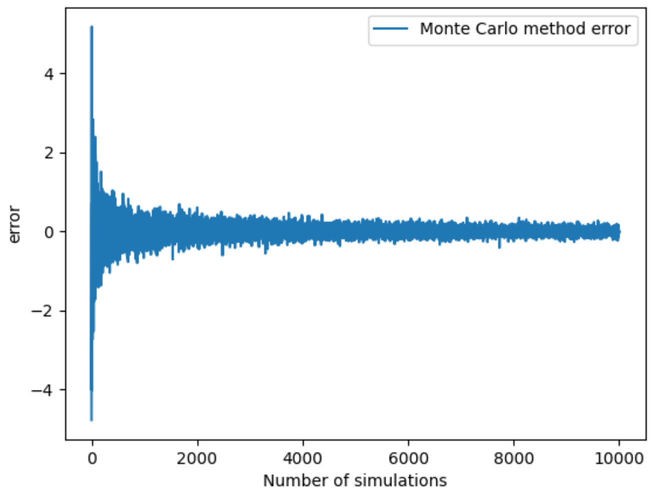

Figure 3: Convergence of option price estimation

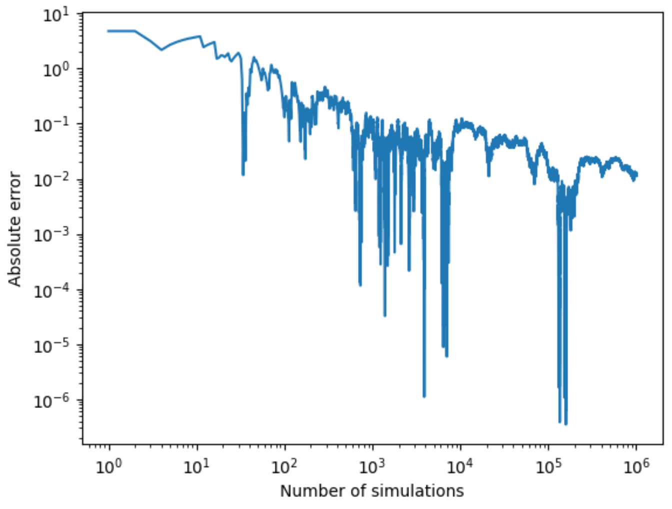

Figure 4: Convergence of option price estimation

Varying the Volatility

In fig. 5, we have plotted the simulated option price for different values of the volatility (with a different seed each time). To save computing resources, we only ran 10000 simulations for these estimations. The combination of the fact that we are using a different seed for each volatility and the relatively low number of simulations causes the graph to not be entirely monotone. The graph is very similar to the volatility graph of the call option that we generated in assignment 1.

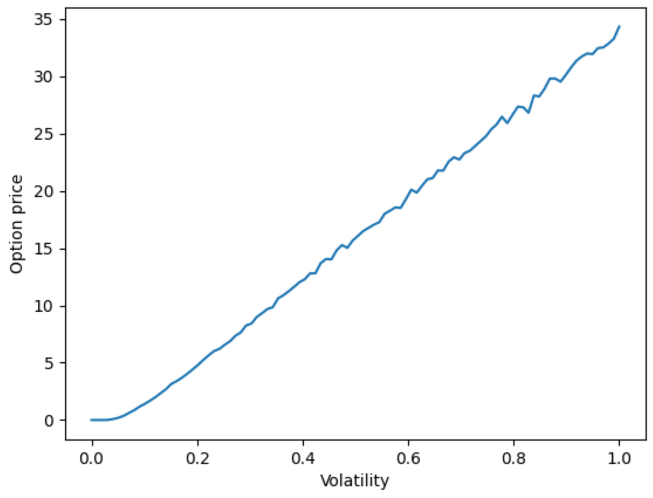

Figure 5: Effect of volatility on option price

Varying the Strike Price

In fig. 6, we have plotted the simulated option price for different values of the strike price (with a different seed each time). To save computing resources, we only ran 10000 simulations for these estimations. The graphs shows what we expected of a put option as a higher strike price will make the value of the option increase, as the buyer as the option to sell the asset for a higher price. The combination of the fact that we are using a different seed for each volatility and the relatively low number of simulations causes the graph to not be entirely monotone.

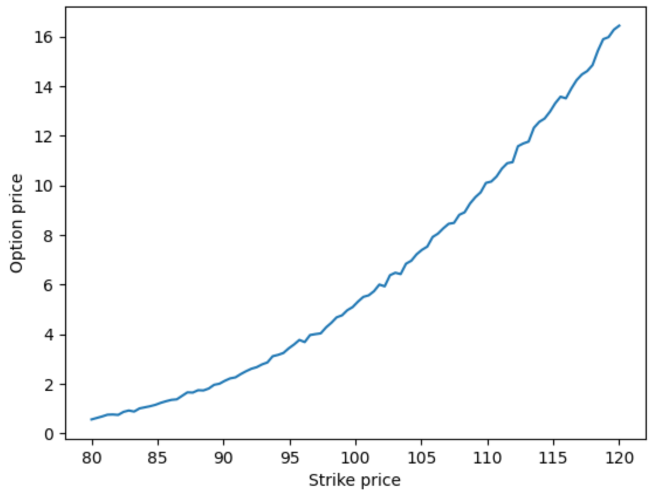

Figure 6: Effect of strike price on option price

2 Part II: Estimation of Sensitivities in MC

Part 1

In this experiment, we run a MC simulation to estimate the delta hedge for the put option. The simulation is based on the finite difference approximation applied to the definition of the delta hedge, namely:

$$\Delta(P_t) = \frac{\partial P_t}{\partial S_t} = \frac{P_t(S_0 + h) - P_t(S_0)}{h}$$

The step size h was chosen such that $h \propto n^{-\frac{1}{6}}$ where n is the number of simulations. The underlying stock price process evolves according to the integrated form, namely:

$$S_T = S_0 e^{r - 0.5\sigma^2 T + \sigma\sqrt{T}Z}$$

where $r = 0.06$ is the annual interest rate, $\sigma = 0.2$ is the volatility, $T = 1$ since it's a one-year contract and $Z \sim \mathcal{N}(0,1)$ sampled at each iteration step. In this case, we ran $n = 10^7$ simulations. We ran two separate MC simulations with the distinction being the random walk(s) that generate the stock price process. In the first simulation, bearing the subscript ss, we fix a random seed for the MC simulation so that we ensure that when the MC is performed on the two spot prices S_0 and $S_0 + h$ we have the same underlying random walk and that the expression for the delta hedge is consistent. In the latter experiment, we removed the random seed requirement and thus the MC simulation running on the S_0 and $S_0 + h$ was actually performed on two independent processes. The delta for this setup has the ds subscript. The effects of the different seed settings resulted in the delta values of $\Delta(P_t)_{ss} = -0.32443704910456683 \approx -0.3244$ and $\Delta(P_t)_{ds} = 5.782222159002082 \approx 5.7822$.

We can quickly see that the first experimental values Δ_{ss} conforms to the expectations set by the theory, $-1 \leqslant \Delta_{ss} \leqslant 0$ and the relationship it holds w.r.t. the put-call parity. We know from Assignment 1 that the delta value for the call option defined on the same process is given by $\Delta(C_t) \approx 0.6725$. Then, by the put-call parity, we can derive that:

$$\begin{aligned}\Delta_{ss}(P_t) = \frac{\partial P_t}{\partial S_t} &= \frac{\partial C_t}{\partial S_t} - \frac{\partial S_t}{\partial S_t} + 0 \\ &= \Delta(C_t) - 1 \\ &\approx 0.6752 - 1 = -0.3248\end{aligned}$$

We can observe that there is a great degree of agreement between the two values with the relative error being $\delta = \left(\frac{|0.3248 - 0.32444|}{|0.3248|}\right)100\% = 0.1\%$. Therefore, we can conclude that the MC simulation approximated the results of the Black-Scholes equation well.

On the other hand, the value $\Delta_{ds}(P_t) \approx 5.7822$ makes no sense as a delta hedge as it's both positive and greater than 1. Thus, it is not a delta hedge. The reason for this can be attributed to the lack of a random seed in the MC simulation. This generates two independent processes $W_1 \equiv C(S_0 + h)$ and $W_2 \equiv C(S_0)$. Since the Z generating the random walk do not share the same sequence $\{Z_i\}_{i=1}^{n}$, we cannot talk about a hedging computation since what we are doing is trying to compute the difference between two separate processes. Therefore, not setting a random seed renders the simulation useless.

Part 2

Bump Method

We calculated the Δ for the option with payoff

$$C(S_T) = \begin{cases} 1, & \text{if } S_T - K > 0 \\ 0, & \text{otherwise} \end{cases}$$

We note that this follows a call option scheme with a fixed payoff, thus the $0 < \Delta < 1$. We used the same step size as above and the same number of simulations. The computation was done using the one-step process $S_T = S_0 e^{(r-0.5\sigma^2)T+\sigma\sqrt{T}Z}$. The deltas we found are $\Delta_{ss} \approx 0.01654$ and $\Delta_{ds} = -0.03687$. As we can see, we again observe that the different seeds result in a delta that doesn't make sense as it's negative and cannot conform to the call option condition. On the other hand, the delta obtained with the same seed conforms to what we expect and due to the performance of the method on the know derivative in the previous experiment, we can conclude that it's sufficiently close to the true delta.

We now try to improve the algorithm with the advanced methods presented in the notes.

Pathwise Approach

The straightforward appliance of the pathwise method doesn't work for this example since $C = 1_{S_T > K}$ since the derivative of the payoff is zero almost everywhere. That's why we need to apply smoothing to get this method to work. The smoothing function of choice here is the logistic sigmoid given by:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

This function has the property that it resembles the threshold function. Therefore, we can approximate the payoff function $1_{S \geqslant K}$ by $\sigma(\lambda(S-K))$ where λ is a tunable smoothing parameter. We discuss the value of this parameter in more detail below. Computing the derivative of the payoff is now straightforward as we know that $\partial_x \sigma(x) = \sigma(x)(1-\sigma(x))$ and as such we have that:

$$\begin{aligned}\partial_{S_0}\sigma(\lambda(S_T-K)) &= \sigma(\lambda(S_T-K))(1-\sigma(\lambda(S_T-K)))\frac{d}{dS_0}\lambda(S_T-K)\\ &= \sigma(\lambda(S_T-K))(1-\sigma(\lambda(S_T-K)))\frac{\lambda S_T}{S_0}\end{aligned}$$

As such the final formula for the Δ based on the pathwise approach is given by:

$$\Delta = e^{-rT}\sigma(\lambda(S_T-K))(1-\sigma(\lambda(S_T-K)))\frac{\lambda S_T}{S_0}$$

It can be seen that $\lim_{\lambda\to\infty}\sigma(\lambda(S-K)) = 1_{S\geqslant K}$ so it makes sense to choose a high value for λ. Still, there is a bias-variance trade-off here because higher values of λ will make it more and more unlikely that we will obtain values on the steep part of the curve (but if we do obtain values in that area they would be very high). In other words, choosing λ to be high increases the variance which will make the approach less accurate because higher variance means slower convergence (by the Central Limit Theorem). We have empirically investigated what the effect of various values of λ is on the calculated option price (when doing 10^5 simulations). This can be seen in fig. 7. In fig. 8 we have plotted the absolute error with respect to the theoretical value. From the graph, we deduce that the best choice of λ is 1 since this results in the smallest error. With higher values of λ it is also interesting to see that the error graph is no longer monotone due to the higher variance. This higher variance is even more visible in fig. 7 because the estimation is sometimes far too low and sometimes far too high for high values of λ. For values higher than 10^5, we see that the simulation consistently outputs zero. This is because the derivative is only nonzero where the final stock price is very close to the strike price and the probability of that happening is very small. However, this rare event has a very large effect on the expectation because the derivative there will be very high.

Setting $\lambda = 1$, we examined the convergence and looked the error of the estimate as we add more and more simulations. This is displayed in fig. 9

Likelihood Approach

We first derive the theoretical expression for the Δ according to the likelihood approach given in the paper on Sensitivies. We know that

$$\Delta = \frac{e^{-rT}}{S_0\sqrt{T}\sigma}\mathbb{E}[\mathbb{1}_{S_T>K}Z]$$

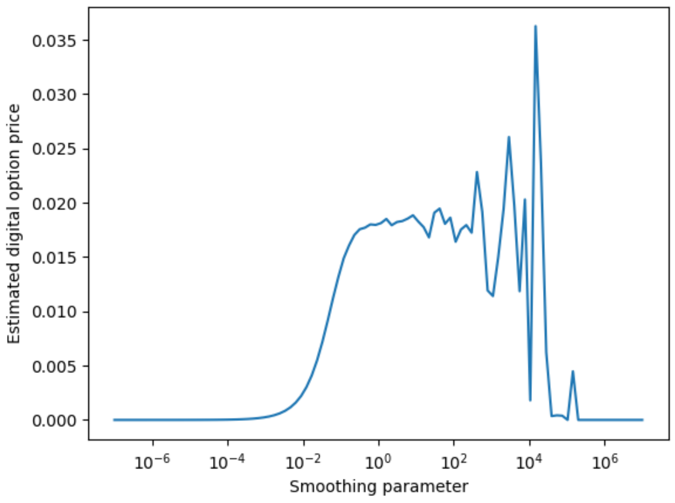

Figure 7: Results of the pathwise approach for various values of the smoothing parameter

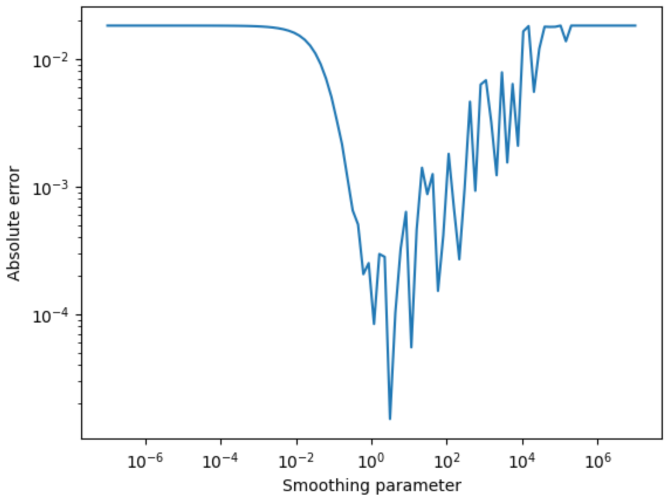

Figure 8: Results of the pathwise approach for various values of the smoothing parameter

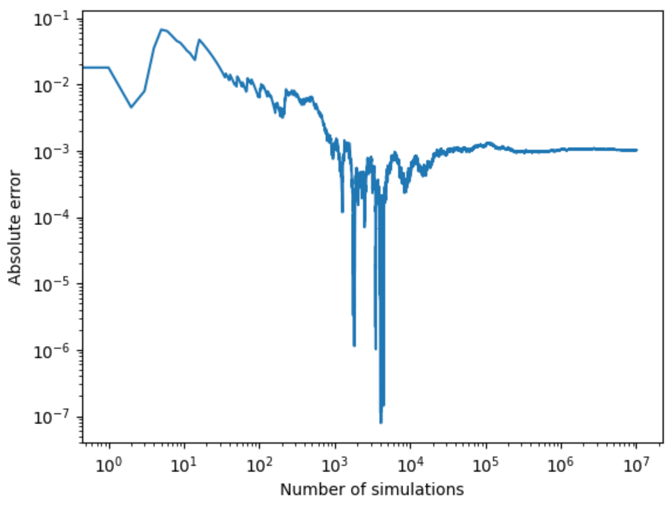

Figure 9: Results of the pathwise approach while adding more and more simulations

where $S_T = S_0 e^{(r-\frac{\sigma^2}{2})T+\sigma\sqrt{T}Z}$ and so the inequality in the indicator becomes:

$$\begin{aligned}
S_0 e^{(r-\frac{\sigma^2}{2})T+\sigma\sqrt{T}Z)} &> K \\
(r-\frac{\sigma^2}{2})T+\sigma\sqrt{T}Z) &> \ln(\frac{K}{S_0}) \\
\sigma\sqrt{T}Z &> \ln(\frac{K}{S_0}) - (r-\frac{\sigma^2}{2})T \\
Z &> \frac{1}{\sigma\sqrt{T}}\Big(\ln(\frac{K}{S_0}) - (r-\frac{\sigma^2}{2})T\Big) \\
Z &> \frac{1}{\sigma}\Big(\ln(\frac{K}{S_0}) - (r-\frac{\sigma^2}{2})\Big) \\
Z &> a
\end{aligned}$$

since $T = 1$ and where we have denoted $a = \frac{1}{\sigma}\Big(\ln(\frac{K}{S_0}) - (r-\frac{\sigma^2}{2})\Big) \approx -0.25025$. Then the Δ takes the form in terms of the Z measure:

$$\begin{aligned}
\Delta_{\text{theoretical}} &= \mathbb{E}_Z[\mathbb{1}_{Z>a}Z] \\
&= \int_a^\infty z\rho(z)dz \\
&= \int_a^\infty \frac{z}{\sqrt{2\pi}}e^{-\frac{1}{2}z^2}dz \\
&= 0.01820636977949049 \\
&\approx 0.018206
\end{aligned}$$

We now perform a simulation according to the likelihood method via the formula:

$$\Delta = \frac{e^{-rT}}{S_0\sqrt{T}\sigma}\mathbb{E}[\mathbb{1}_{S_T>K}Z]$$

with $T = 1$. The algorithm proceeded by generating $n = 10^7$ stock prices S_T and storing the corresponding $Z \sim \mathcal{N}(0,1)$ in memory so that then the Δ is computed as the averages of the product of the constants times the payoff option $C = \mathbb{1}_{S_T > K}$ evaluated at each price S_T. The simulation returned a value of $\Delta_{\text{sim}} \approx 0.018211$. This value is in great agreement with the theoretical as we get a relative error of only

$$\delta_{\text{rel}} = \Big(\frac{|0.018206 - 0.018211|}{0.018206}\Big)100\% \approx 0.0275\%$$

This accuracy is substantially higher than what we had in the bump method of 0.1% and is sufficiently high to say that the model predicts the Δ correctly.

With smoothing We also tried running the likelihood approach with smoothing by replacing the payoff with the sigmoid function as described in our pathwise approach. We ran 10^5 simulations and actually obtain a better value with the smoothing compared to without. With smoothing we get an error (compared to the theoretical value) of $4.9 \cdot 10^{-5}$ while without smoothing we get an error of 0.01806.

3 Part III: Variance Reduction

Part 1

We compute the geometric Asian by using the analytic Black-Scholes formula and by a Monte-Carlo simulation. The first method used the expression:

$$C_g^A(S_0, T) = e^{-rT}\Big(S_0 e^{\tilde{r}T}\Phi(\tilde{d}_1) - K\Phi(\tilde{d}_2)\Big)$$

where:

$$\begin{aligned}
\tilde{\sigma} &= \sigma\sqrt{\frac{2N+1}{6(N+1)}} \\
\tilde{r} &= \frac{(r - \frac{\sigma^2}{2}) + \tilde{\sigma}^2}{2} \\
\tilde{d}_1 &= \frac{\log(\frac{S_0}{K}) + (\tilde{r} + \frac{1}{2}\tilde{\sigma}^2)T}{\sqrt{T}\tilde{\sigma}} \\
\tilde{d}_2 &= \frac{\log(\frac{S_0}{K}) + (\tilde{r} - \frac{1}{2}\tilde{\sigma}^2)T}{\sqrt{T}\tilde{\sigma}}
\end{aligned}$$

for the given interest rate r and volatility σ. We chose the option to have been observed $N = 365$ times, that is once a day for the duration of $T = 1$ years. This gave us a theoretical result of $C_A(S_0, T) = 6.331828080598059 \approx 6.33183$. We then calculate the value of the same option using a Monte Carlo simulation with $n = 10^7$ iterations over the price S_T via the formula:

$$C_g^A(S_0, T) = e^{(\tilde{r}-r)T}C(S_0, T)$$

where $C(S_0, T) = \max\{S_T - K, 0\}$ is the regular call option price. Thus, we generated a set of call options, then discounted them by the factor of $e^{(\tilde{r}-r)T}$ and took the average to obtain the result of $C_g^A = 6.328277494435255 \approx 6.32828$. The relative error between the Monte-Carlo and theoretical model is:

$$\delta_{\text{rel}} = \frac{|6.33183 - 6.32828|}{6.33183}100\% = 0.0056\%$$

Therefore, due to the low relative error, we can say that the MC simulation approximates the theoretical result sufficiently well.

Part 2

The control variate strategy is a variance reduction technique used in order to reduce the variance of the sampling. By using the Asian call option based on geometric averages as a control variate we will use the information about the error in its Monte Carlo estimate to reduce the error of another option correlated with this one. For a variable to be used as a control variate it is vital that we know its mean. In the case of the Asian call option based on geometric averages, we already discussed in Part 1 how the mean can be computed analytically. A high correlation between the 2 options is a good indicator of how effective this method is.

When using a control variate C to improve the estimator B, the estimator is defined to be:

$$B + \beta(C - \mathbb{E}C)$$

where β is a parameter that can be tuned. The optimal value of this parameter is $\frac{\mathrm{Cov}(B,C)}{\mathrm{Var}(C)}$ and this can be approximated by using Monte Carlo estimators for the numerator and denominator.

In Part 3 we will use this option as a control variate to estimate the value of an Asian call option based on the arithmetic average. Another reason why we use the Asian Call option based of geometric averages is due to the fact that the value of these two options are highly correlated with a value of 0.99963032.

Part 3

First we begin by a simple Monte Carlo simulation to find the Asian arithmeic option price by doing $n = 10^6$ simulations with $N = 365$ observations and $K = 99$ generating 10^6 values of :

$$C_a^A = \max\left\{\frac{1}{N}\sum_{i=1}^{N} S_{t_i} - K, 0\right\}$$

and then taking the average. This yielded a result of $C_a^A = 4.196641374934199$ with standard deviation of 12.18462500504706. Then we ran the control-variate version of the price estimation according to:

$$\tilde{C}_a^A = \hat{C_a^A} - \frac{\mathrm{Cov}(\hat{C_a^A}, \hat{C_g^A})}{\mathrm{Var}(\hat{C_g^A})}(\hat{C_g^A} - C_g^A)$$

where:

$$\begin{aligned}
\tilde{C_a^A} &= \text{Control Variate estimate of Asian Call option based on arithmetic averages}\\
\hat{C_a^A} &= \text{Monte Carlo estimate of Asian Call option based on arithmetic averages}\\
\hat{C_g^A} &= \text{Monte Carlo estimate of Asian Call option based on geometric averages}\\
C_g^A &= \text{Theoretical Asian Call option based on geometric averages}
\end{aligned}$$

We first ran a simulation for $n = 10^6$ iterations with daily observations $N = 365$, strike $K = 99$ and $\tilde{r}, \tilde{\sigma}$ calculated according to the equations of Part 1. This returned a value of $C_a^A(S_0, K) = 6.580327470449082$ with standard deviation of 0.49295983016366024. Comparing this with the naive computation in the first equation of this part, we see that both the price estimate and the standard deviation have been improved. Thus, we can conclude that using the geometric option as a control variate has improved the computation of the price of the arithmetic option.

Following this, we explore the dependence of the price on variation of the number of simulations, number of observations and finally the strike price on the contract. We first give an explicit form of the arithmetic option as computed in the simulation:

$$\tilde{C}_a^A = \max\left\{\frac{1}{N}\sum_{i=1}^{N} S_{t_i} - K, 0\right\} - \frac{\mathrm{Cov}(\hat{C_a^A}, \hat{C_g^A})}{\mathrm{Var}(\hat{C_g^A})}\left(\max\left\{\Big(\prod_{i=1}^{N} S_{t_i}\Big)^{\frac{1}{N}} - K, 0\right\} - C_g^A\right)$$

This formula was iterated then n times depending on the size of the MC simulation and the average taken, thus yielding the final price.

Varying the number of Simulations

When varying the number of simulations, we can see how the contract value converges and the variance is reduced. This is due to the error progression in the MC method give by:

$$\delta_{\mathrm{MC}} \propto \frac{\sigma(\tilde{C}_a^A)}{\sqrt{n}}$$

Thus, for greater iteration numbers we observe better convergence and lower variance in the price of the arithmetic Asian option. This can be seen in fig. 10 below. The behavior observed is similar to that of fig. 3.

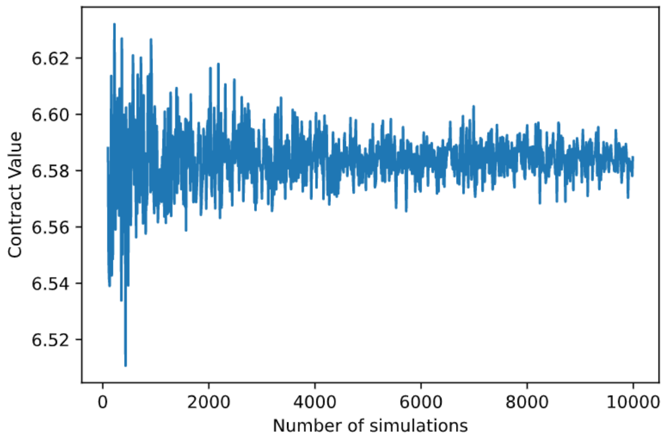

Figure 10: Effect of number of Monte Carlo simulations on option price

Variation in the number of Observations

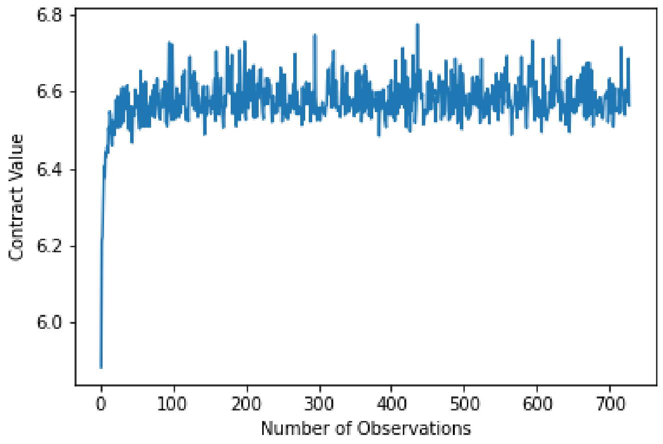

Figure 11: Effect of number of path observations on option price

In fig. 11 we plotted the value of the option in relation to the number of observations in the path of the stocks values, we looked at values from $N = 1$, one observation per year, (note that this is just a regular call option) to $N = 730$ which corresponds to 2 observations per day. For a low number of observations we can see that the value of the option will decrease as one takes more risk, having a low number of points to average. As the number of observations increases we can see that the value of the option will fluctuate randomly around the true value of the option (no convergence seems to happen). This is due to the random nature of the stock values at each observation point.

Varying the Strike Price

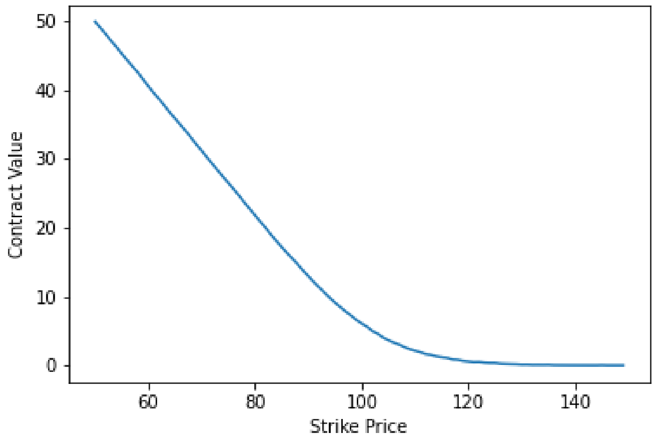

Figure 12: Effect of the strike price on option price

In fig. 12, we have plotted the simulated option price for different values of the strike price (with a different seed each time). To save computing resources, we only ran 1000 simulations for these estimations, with number of observation equal to 365 (daily). The graphs shows what we expected of a call option as a lower strike price will make the value of the option increase.

Appendix

```
#%%
from math import exp, log, sqrt
import numpy as np
import matplotlib.pyplot as plt
import statsmodels as sts
from statsmodels.graphics.gofplots import qqplot as qq
import scipy.stats as stats
from scipy.stats import norm, gmean
#%%
#PROBLEM 1 ALL CODE
def computeUD(vol, dt):
  u = exp(vol * sqrt(dt))
  d = exp(-vol * sqrt(dt))
  return u, d

def buildTree(S, vol, T, N):
  dt = T / N
  matrix = np.zeros((N+1, N+1))

  # up and down move multiplier
```

```
    u, d = computeUD(vol, dt)

    matrix[0][0] = S

    # Iterate over the lower triangle
    for i in np.arange(1, N+1): # i is the time
        for j in np.arange(i): # j is the number of up moves
            # matrix[i,j] is the same thing as at the previous time (i-1),
            # but with an additional down-move
            # (Because the number of up-moves stays the same)
            # This means that we multiply by d.
            matrix[i, j] = matrix[i-1,j] * d
        # In case we have i up-moves within i time steps,
        # we have the same thing as i-1 up-moves within i-1 time steps
        # but with an additional up-move
        # This means that we multiply by u.
        matrix[i,i] = matrix[i-1,i-1] * u

    return matrix

#%%

# Compute the risk-neutral probability p
def computeP(r, dt, u, d):
    # Use formula (3) in A.2
    return (exp(r*dt)-d) / (u-d)

def valueOptionMatrix(tree, T, r, K, vol, N):
    dt = T / N
    u, d = computeUD(vol, dt)
    p = computeP(r, dt, u, d)

    columns = tree.shape[1]
    rows = tree.shape[0]

    # Walk backward, we start in the last row of the matrix

    # Add the payoff function in the last row
    for c in np.arange(columns):
        S = tree[rows - 1, c] # value of the stock
        tree[rows - 1, c] = max(0, K - S)

    # For all other rows, we need to combine from previous rows
    # We walk backwards, from the last row to the first row

    for i in np.arange(rows - 1)[::-1]:
        for j in np.arange(i + 1):
            current = tree[i,j]
            down = tree[i+1, j]
            up = tree[i+1, j+1]
            # Use equation (4) from A.2.3
            tree[i, j] = exp(-r*dt) * (p*up + (1-p)*down)

    return tree
```

```
def computeEuropeanPutOptionPrice(S, sigma, T, N, r, K):
  tree = buildTree(S, sigma, T, N)
  matrix = valueOptionMatrix(tree, T, r, K, sigma, N)
  return matrix[0][0]


# In[3]:


sigma = 0.20 # 20% volatility
S = 100. # stock price in euros
T = 1. # 1 year maturity
N = 50 # Number of steps
K = 99. # Strike price
r = 0.06 # 6% interest rate

# Question 1
print("Binomial␣Tree␣Put␣price:", computeEuropeanPutOptionPrice(S, sigma, T, N
    ↪ , r,K))


# Euler Method for the Stcok Price

# In[4]:


# def Stock_Value_Path_Euler(S_0, N, r, sigma, T):
#      dt = T/N
#      S = np.zeros(N)
#      S[0] = S_0
#      Zs = np.random.normal(size=N)
#      for i in np.arange(1,N):
#           z = Zs[i]
#           S[i]=S[i-1]+r*S[i-1]*dt+sigma*S[i-1]*np.sqrt(dt)*z
#      return S[-1]


# In[5]:


# Stock_Value_Path_Euler(100, 50, 0.06, 0.2, 1)


# In[6]:


def Step_1_Stock_Value(S_0, r, sigma, T):
     Z=np.random.normal()
     S_T = S_0*np.exp((r-0.5*(sigma)**2)*T+sigma*np.sqrt(T)*Z)
     return S_T


# In[7]:
```

```
Step_1_Stock_Value(100, 0.06, 0.2, 1)


# In[8]:


def Price_PutOption_Path(S_0, r, sigma, T, K):
    Discounted_Payoff = np.exp(-r*T)*max(0, K - Step_1_Stock_Value(S_0, r,
      ↪ sigma, T))
    return Discounted_Payoff


# In[9]:


Price_PutOption_Path(100, 0.06, 0.2, 1, 99)


# In[10]:


def Monte_Carlo_Method(S_0, r, sigma, T, K, N): #N is the number of paths
    P = np.zeros(N)
    for i in np.arange(0, N):
        P[i] = Price_PutOption_Path(S_0, r, sigma, T, K)
    return np.mean(P), P


# In[11]:


Monte_Carlo_prices = Monte_Carlo_Method(100, 0.06, 0.2, 1, 99, 1000000) #Monte
    ↪ Carlo 1 million Paths
print('Option␣Price␣Monte␣Carlo', Monte_Carlo_prices[0])


# Standard Error of our Estimate: $$\frac{\sigma(payoffs)}{\sqrt{N}}$$

# In[12]:


N = 1000000
Standard_Error=np.std(Monte_Carlo_prices[1])/np.sqrt(N)
Standard_Error


# In[13]:


def Multi_Monte_Carlo(S_0, r, sigma, T, K, N, sims): #sims is the times we run
    ↪ the Monte Carlo Mathod
    results = np.zeros(sims)
    for i in range(sims):
        results[i] = Monte_Carlo_Method(S_0, r, sigma, T, K, N)[0]
    return results


# In[36]:


results_x_times = Multi_Monte_Carlo(100, 0.06, 0.2, 1, 99, 10000, 10000)
plt.hist(results_x_times)
plt.xlabel('option_price')
plt.ylabel('density')
plt.savefig('mcm-histogram.png')
plt.close()


# In[16]:


def BlackScholes_put(S_0, K, r,T, vol):
  d1 = (log(S_0 / K) + (r + vol * vol / 2) * T)/vol / sqrt(T)
  d2 = d1 - vol * sqrt(T)
  return exp(-r*T)*K*norm.cdf(-d2) - S_0 * norm.cdf(-d1)


# In[17]:


print('BS',BlackScholes_put(100, 99, 0.06, 1, 0.2))


# Convergence Number of Steps Comparison

# In[18]:


# Changing number of steps (convergence)
Ns = np.arange(1,100)

def computePriceByNTree(N):
  return computeEuropeanPutOptionPrice(S, sigma, T, N, r, K)


# In[35]:


# Changing number of steps (convergence)
S_0 = 100
Ns = np.arange(1,100)

def computePriceByNTree(N):
  return computeEuropeanPutOptionPrice(S, sigma, T, N, r, K)
results1 = list(map(computePriceByNTree, Ns))
```

```
def computePriceByNMonteCarlo(N):
    return Monte_Carlo_Method(S_0, r, sigma, T, K, N)[0]
results2 = list(map(computePriceByNMonteCarlo, Ns))

theoretical = list(map(lambda _ : BlackScholes_put(100, 99, 0.06, 1, 0.2), Ns)
    ↪ )
plt.plot(Ns, np.array(results1) - np.array(theoretical), label='Binomial Tree 
    ↪ error')
plt.plot(Ns, np.array(results2) - np.array(theoretical), label='Monte Carlo 
    ↪ method error')
plt.xlabel('number of simulation steps')
plt.ylabel('error')
plt.legend()
plt.savefig('Error-Binomial-MC.png')
plt.close()

Ns = np.logspace(0,6, 75, dtype='int')
# def computePriceByNTree(N):
#  return computeEuropeanPutOptionPrice(S, sigma, T, N, r, K)
# results1 = list(map(computePriceByNTree, Ns))


# In[33]:


# Changing number of steps (convergence)
S_0 = 100
Ns = np.arange(1,10000)

# def computePriceByNTree(N):
#    return computeEuropeanPutOptionPrice(S, sigma, T, N, r, K)
# results1 = list(map(computePriceByNTree, Ns))

def computePriceByNMonteCarlo(N):
    return Monte_Carlo_Method(S_0, r, sigma, T, K, N)[0]
results2 = list(map(computePriceByNMonteCarlo, Ns))

theoretical = list(map(lambda _ : BlackScholes_put(100, 99, 0.06, 1, 0.2), Ns)
    ↪ )
# plt.plot(Ns, np.array(results1) - np.array(theoretical), label='Binomial
    ↪ Tree error')
plt.plot(Ns, np.array(results2) - np.array(theoretical), label='Monte Carlo 
    ↪ method error')
plt.xlabel('Number of simulations')
plt.ylabel('error')
plt.legend()
plt.savefig('Error-MC.png')
plt.close()

# Ns = np.logspace(0,6, 75, dtype='int')
# def computePriceByNTree(N):
#  return computeEuropeanPutOptionPrice(S, sigma, T, N, r, K)
# results1 = list(map(computePriceByNTree, Ns))
```

```
#%%
#PROBLEM 2 ALL CODE
def payoff(K,S):
    return max(K-S, 0)

def payoff_call(S,K):
    return max(S-K,0)

def opt_2_payoff_likelihood_smooth(K, S):
    return sigmoid(S-K)

#Euler Method

def stock_price(S_0, r, sigma, N, T):

    z = np.random.normal()
    dt = T/N
    S = np.zeros(N)
    S[0] = S_0 + S_0*(r * dt + sigma*(z)*np.sqrt(dt))

    for i in np.arange(1,N):

        k = np.random.normal()
        S[i] = S[i-1] + S[i-1]*(r * dt + sigma * k * np.sqrt(dt))

    return S


#Part 2 problem 2.1

#simulations with the same random seed
def monte_carlo_same(num_sims, S_0, r, sigma, T):
    np.random.seed(5)
    prices = np.zeros(num_sims)

    for n in np.arange(num_sims):
        z = np.random.normal()
        prices[n] = S_0*np.exp((r-0.5*sigma**2)*T + sigma*np.sqrt(T)*z)

    def payoff1(S):
        return payoff(99, S)

    pay = np.average(list(map(payoff1, prices)))*np.exp(-r)

    return pay

#simulations with different random seed
def monte_carlo(num_sims, S_0, r, sigma, T):
    prices = np.zeros(num_sims)
    for n in np.arange(num_sims):
        prices[n] = S_0*np.exp((r-0.5*sigma**2)*T + sigma*np.sqrt(T)*np.random
            ↪ .normal())

    def payoff1(S):
        return payoff_call(K=99, S=S)*np.exp(-r)

    pay = np.average(list(map(payoff1, prices)))

    return pay

#Bumped vs Unbumped comparison function
def bumped_vs_unbumped(num_sims, S_0, r, sigma, T, h, s):

    if s == 'Same':
        b_opt = monte_carlo_same(num_sims, S_0 + h, r, sigma, T)
        nb_opt = monte_carlo_same(num_sims, S_0 , r, sigma, T)

    else:
        b_opt = monte_carlo(num_sims, S_0 + h, r, sigma, T)
        nb_opt = monte_carlo(num_sims, S_0, r, sigma, T)

    delta_bvnb = (1.0/(h))*(b_opt - nb_opt)
    return delta_bvnb


#Part 2 problem 2

def opt_2_payoff(K, S):

    if S > K:
        return 1
    else:
        return 0

def opt_2_payoff_likelihood(K, S):

    if S >= K:
        return 1
    else:
        return 0

##Using the methods of problem 1

#simulations with the same random seed
def monte_carlo_same2(num_sims, S_0, r, sigma, T):
    np.random.seed(123)
    prices = np.zeros(num_sims)

    for n in np.arange(num_sims):
        z = np.random.normal()
        prices[n] = S_0*np.exp((r-0.5*sigma**2)*T + sigma*np.sqrt(T)*z)

    def payoff1(S):
        return opt_2_payoff(99, S)*np.exp(-r)

    pay = np.average(list(map(payoff1, prices)))*np.exp(-r)

    return pay

#simulations with different random seed
def monte_carlo2(num_sims, S_0, r, sigma, T):
    prices = np.zeros(num_sims)

    for n in np.arange(num_sims):
        z = np.random.normal()
        prices[n] = S_0*np.exp((r-0.5*sigma**2)*T + sigma*np.sqrt(T)*z)

    def payoff1(S):
        return opt_2_payoff(99, S)*np.exp(-r)

    pay = np.average(list(map(payoff1, prices)))

    return pay

#Bumped vs Unbumped for 2.2
def bumped_vs_unbumped_2(num_sims, S_0, r, sigma, T, h, s):

    if s == 'Same':

        b_opt = monte_carlo_same2(num_sims, S_0 + h, r, sigma, T)
        nb_opt = monte_carlo_same2(num_sims, S_0 - h, r, sigma, T)

    else:

        b_opt = monte_carlo2(num_sims, S_0 + h, r, sigma, T)
        nb_opt = monte_carlo2(num_sims, S_0 - h, r, sigma, T)

    delta_bvnb = (1/(2*h))*(b_opt - nb_opt)
    return delta_bvnb

#Simulations using the sigmoid smoothed pathwise method

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def der_sigmoid(x):
    return sigmoid(x=x) * (1-sigmoid(x=x))

def payoff_sigmoid(S,K, smoothing):
    return der_sigmoid((S-K) * smoothing) * smoothing

def delta_computation(num_sims, S, K, S_0, smoothing): #NOT DISCOUNTED, WE
    ↪ APPLY 1 DISCOUNT ON THE AVERAGE!

    delta = np.zeros(num_sims)
    for n in np.arange(num_sims):
        delta[n] = (S[n] / S_0) * payoff_sigmoid(S[n], K, smoothing)
    return delta

def mc_delta_pathwise(num_sims,S_0, K, r, sigma, T, smoothing=1):
    prices = np.zeros(num_sims)

    for n in np.arange(num_sims):
        z = np.random.normal()
        prices[n] = S_0*np.exp((r-0.5*sigma**2)*T + sigma*np.sqrt(T)*z)

    delta = delta_computation(num_sims = num_sims, S = prices, K = K , S_0 =
        ↪ S_0, smoothing=smoothing)
    return np.mean(delta)*np.exp(-r) #Here is the single discount

#Simulation using the likelihood ratio model

def mc_delta_likelihood(num_sims,S_0, r, sigma, T):

    prices = np.zeros(num_sims)
    rw = np.zeros(num_sims)

    for n in np.arange(num_sims):
        z = np.random.normal()
        rw[n] = z #stores the random walk to apply at the end of the
            ↪ computation of the delta
        prices[n] = S_0*np.exp((r-0.5*sigma**2)*T + sigma*np.sqrt(T)*z)

    def non_z_term(S):
        return opt_2_payoff_likelihood(K = 99, S = S)*np.exp(-r*T)/ (S_0 *
            ↪ sigma * np.sqrt(T))
        #return payoff_call(S, 99)*np.exp(-r*T)/ (S_0 * sigma * np.sqrt(T))
    mod_pay = list(map(non_z_term, prices))

    def times_random_normal(x,w):
        deltas = np.zeros(len(x))

        for i in np.arange(len(x)):
            deltas[i] = x[i] * w[i]

        return deltas
    deltas = times_random_normal(mod_pay,rw)
    delta = np.average(times_random_normal(mod_pay,rw))

    return delta, prices, deltas

#Smoothing the Likelihood method

def mc_delta_likelihood_smooth(num_sims,S_0, r, sigma, T):

    prices = np.zeros(num_sims)
    rw = np.zeros(num_sims)

    for n in np.arange(num_sims):
        z = np.random.normal()
        rw[n] = z #stores the random walk to apply at the end of the
            ↪ computation of the delta
        prices[n] = S_0*np.exp((r-0.5*sigma**2)*T + sigma*np.sqrt(T)*z)

    def non_z_term(S):
        return opt_2_payoff_likelihood_smooth(K = 99, S = S)*np.exp(-r*T)/ (
            ↪ S_0 * sigma * np.sqrt(T))
        #return payoff_call(S, 99)*np.exp(-r*T)/ (S_0 * sigma * np.sqrt(T))
    mod_pay = list(map(non_z_term, prices))

    def times_random_normal(x,w):
        deltas = np.zeros(len(x))

        for i in np.arange(len(x)):
            deltas[i] = x[i] * w[i]

        return deltas
    deltas = times_random_normal(mod_pay,rw)
    delta = np.average(times_random_normal(mod_pay,rw))

    return delta, prices, deltas

#Problem 3
#Using geometric averages

def calculate_s_hat(sigma, N):
    return sigma * np.sqrt((2*N+1)/(6*(N+1)))

def calculate_r_hat(r, sigma, s_hat, N):
    return 0.5*(r- 0.5*sigma**2 + s_hat**2)

def Step_1_Stock_Value(S_0, r_hat, s_hat, T):
    Z=np.random.normal()
    S_T = S_0*np.exp((r_hat -0.5*(s_hat)**2)*T+s_hat*np.sqrt(T)*Z)
    return S_T

def Price_CallOption_Path(S_0, r, sigma, T, K, N): #N is used for r_hat, s_hat
    s_hat = calculate_s_hat(sigma, N)
    r_hat = calculate_r_hat(r, sigma, s_hat, N)
    Discounted_Payoff = np.exp(-r_hat*T)*max(0, Step_1_Stock_Value(S_0, r_hat,
        ↪ s_hat, T) - K)
    return Discounted_Payoff

def Price_Asian_Option(S_0, r, sigma, T, K, N):
    s_hat = calculate_s_hat(sigma, N)
    r_hat = calculate_r_hat(r, sigma, s_hat, N)
    return  np.exp((r_hat-r)*T)*Price_CallOption_Path(S_0, r, sigma, T, K, N)


def Monte_Carlo_Method_Asian(S_0, r, sigma, T, K, N, M):
    P = np.zeros(M)
    for i in np.arange(0, M):
        P[i] = Price_Asian_Option(S_0, r, sigma, T, K, N)
    return np.mean(P), P
```

```
def asian_geometric_theoretical(S_0, K, N, T, r, sigma):

    sigma_hat = sigma * np.sqrt( (2*N+1) / (6*(N+1)) )
    r_hat = 0.5*(r- 0.5*sigma**2 + sigma_hat**2)
    d1 = (np.log(S_0/ K) + (r_hat + 0.5 * sigma_hat**2) *T) / (np.sqrt(T) *
        ↪ sigma_hat)
    d2 = (np.log(S_0/ K) + (r_hat - 0.5 * sigma_hat**2) *T) / (np.sqrt(T) *
        ↪ sigma_hat)

    return np.exp(-r * T)*(S_0*np.exp(r_hat * T)*stats.norm.cdf(d1) - K *
        ↪ stats.norm.cdf(d2))

#Using the geometric as a covariate to compute the arithmetic based on MC-
    ↪ Euler
#simulations of the stock price process
def find_A(num_sims, obs, K, T):
    a = np.zeros(num_sims)
    x = np.zeros(num_sims)
    y = np.zeros(num_sims)

    payoff = asian_geometric_theoretical(S_0 = 100, K = K , N = obs, T = T, r
        ↪ = 0.06, sigma = 0.2)

    for i in np.arange(num_sims):
        stock_data = stock_price(S_0 = 100 , r = 0.06 , sigma = 0.2, N = obs,
            ↪ T = T)
        x[i] = np.mean(stock_data)
        y[i] = stats.mstats.gmean(stock_data)

    for i in np.arange(num_sims):
        a[i] = max(x[i]-K,0) - ((np.cov(x,y)[0,1]/np.var(x))) * (max(y[i]-K,0)
            ↪  - payoff)

    return np.mean(a)

#Create a simulation keeping everything fixed but the number of iterations:
def path_sim(paths):
    return find_A(paths, obs = 365, K =99, T = 1)

#Simulation keeping everything fixed but the number of observations:
def obs_sim(obs):
    return find_A(100, obs, K = 99, T = 1)

#Simulation keeping everything fixed but the strike price
def strike_sim(strikes):
    return find_A(100, obs = 365, K = strikes , T = 1)
# %%
delta1 = bumped_vs_unbumped(1000000, 100, 0.06, 0.2, 1, 1/(100000**(1/6)), '
    ↪ Same')
#%%
delta2 = bumped_vs_unbumped(1000000, 100, 0.06, 0.2, 1, 0.001, 'Not')
# %%
```

```
delta_digi_1 = bumped_vs_unbumped_2(10000000, 100, 0.06, 0.2, 1, 0.001, 'Same'
    ↪ )
delta_digi_2 = bumped_vs_unbumped_2(10000000, 100, 0.06, 0.2, 1, 0.001, 'not')
# %%
delta_digi_pathwise = mc_delta_pathwise(100000000,100, 0.06, 0.2, 1)
#%%
print(delta_digi_pathwise)
# %%
delta_likelihood, stocks, deltas = mc_delta_likelihood(10000, 100, 0.06, 0.2,
    ↪ 1)
#%%
print(delta_likelihood)
# %%
plt.hist(deltas, density = True)
plt.show()
# %%
fig = qq(deltas, stats.expon())
plt.show()
#%%
a_opt = asian_geometric_theoretical(S_0 = 100, K = 99, N = 365, T = 1, r =
    ↪ 0.06, sigma = 0.2)
# %%
sims = np.arange(1000, 10000, step = 1000)
obs = np.arange(1,2*364,1)
strike = np.arange(50, 150, 1)
#%%
simulated_contracts = list(map(path_sim, sims))
#%%
simulated_obs = list(map(obs_sim,obs))
#%%
simulated_strikes = list(map(strike_sim, strike))
# %%
plt.plot(sims,simulated_contracts)
plt.xlabel('Number of simulations')
plt.ylabel('Contract Value')
plt.show()
# %%
plt.plot(obs,simulated_obs)
plt.xlabel('Number of Observations')
plt.ylabel('Contract Value')
plt.savefig('ObsProgression.png')
plt.show()
# %%
plt.plot(strike, simulated_strikes)
plt.xlabel('Strike Price')
plt.ylabel('Contract Value')
plt.savefig('StrikeProgression.png')
plt.show()
# %%
simulated_contracts
# %%
single_sim_val = find_A(num_sims = 10000, obs = 365, K = 99, T = 1)
# %%
single_sim_val
```

```
# %%
def asian_opt(sims, K):
    c = np.zeros(sims)
    x = np.zeros(sims)

    for i in np.arange(sims):
        stock_data = stock_price(S_0 = 100 , r = 0.06, sigma = 0.2, N = 365, T
            ↪ = 1)
        c[i] = np.max(np.mean(stock_data) - K, 0)

    return np.std(c), np.mean(c)
# %%
asn_std, asn_price = asian_opt(10000, 99)
# %%
print(asn_std)
print(asn_price)
# %%
def find_A(num_sims, obs, K, T):
    a = np.zeros(num_sims)
    x = np.zeros(num_sims)
    y = np.zeros(num_sims)
    geo_prices = np.zeros(num_sims)
    ar_prices = np.zeros(num_sims)
    payoff = asian_geometric_theoretical(S_0 = 100, K = K , N = obs, T = T, r
        ↪ = 0.06, sigma = 0.2)

    for i in np.arange(num_sims):
        stock_data = stock_price(S_0 = 100 , r = 0.06 , sigma = 0.2, N = obs,
            ↪ T = T)
        x[i] = np.mean(stock_data)
        y[i] = stats.mstats.gmean(stock_data)

    for i in np.arange(num_sims):
        a[i] = max(x[i]-K,0) - ((np.cov(x,y)[0,1]/np.var(x))) * (max(y[i]-K,0)
            ↪ - payoff)
        ar_prices[i] = np.max(x[i] - K, 0)
        geo_prices[i] = np.max(y[i] - K, 0)

    ar_mean = np.mean(ar_prices)
    ar_std = np.std(ar_prices)
    asian_std = np.std(a)
    var_ratio = asian_std / ar_std

    return np.mean(a), asian_std, ar_mean, ar_std, np.abs(var_ratio - (1- np.
        ↪ sqrt(np.cov(x,y)[0,1]))**2)
# %%
covariate_asian, asia_std, ar_price, ar_std, ratio_difference = find_A(10000,
    ↪ 365, K = 99, T=1)
# %%
print(covariate_asian)
print(asia_std)
# %%
print(ar_price)
print(ar_std)
```

```
# %%
print(ratio_difference)
# %%
def delta_analytic(S_0, K, r, sigma, T):
  d_2 = (np.log(S_0 / K) + (r-sigma**2/2)*T)/(sigma* np.sqrt(T))
  return np.exp(-r*T)*stats.norm.pdf(d_2)/(sigma*S_0*np.sqrt(T))
theoretical = delta_analytic(100, 99, 0.06, 0.2, 1)
delta_likelihood_smooth, stocks, deltas = mc_delta_likelihood_smooth(int(1e7),
    ↪  100, 0.06, 0.2, 1)
delta_likelihood, stocks, deltas = mc_delta_likelihood(int(1e7), 100, 0.06,
    ↪ 0.2, 1)
print(delta_likelihood_smooth - theoretical)
print(delta_likelihood - theoretical)
#%%
smoothings = np.logspace(-7, 7, 100)
def mc_delta_pathwise_by_smoothing(smoothing):
  print(smoothing)
  return mc_delta_pathwise(int(1e5), 100, 99, 0.06, 0.2, 1, smoothing)
def delta_analytic(S_0, K, r, sigma, T):
  d_2 = (np.log(S_0 / K) + (r-sigma**2/2)*T)/(sigma* np.sqrt(T))
  return np.exp(-r*T)*stats.norm.pdf(d_2)/(sigma*S_0*np.sqrt(T))
theoretical = delta_analytic(100, 99, 0.06, 0.2, 1)
results = list(map(mc_delta_pathwise_by_smoothing, smoothings))
plt.plot(smoothings, results)
plt.xscale('log')
plt.xlabel('Smoothing parameter')
plt.ylabel('Estimated digital option price')
plt.savefig('a2.2.pathwise.smoothing.png')
plt.close()
plt.plot(smoothings, np.abs(results - theoretical))
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Smoothing parameter')
plt.ylabel('Absolute error')
plt.savefig('a2.2.pathwise.smoothing.error.png')
plt.close()
print(results - theoretical)

def mc_delta_pathwise_graph(num_sims,S_0, K, r, sigma, T, smoothing=1):
    prices = np.zeros(num_sims)

    for n in np.arange(num_sims):
        z = np.random.normal()
        prices[n] = S_0*np.exp((r-0.5*sigma**2)*T + sigma*np.sqrt(T)*z)

    delta = delta_computation(num_sims = num_sims, S = prices, K = K , S_0 =
        ↪ S_0, smoothing=smoothing)
    plt.plot(np.arange(num_sims), np.abs(np.array(delta).cumsum() / np.arange
        ↪ (1,num_sims+1) -theoretical))
    return np.mean(delta)*np.exp(-r) #Here is the single discount

mc_delta_pathwise_graph(int(1e7), 100, 99, 0.06, 0.2, 1, 1)

plt.xscale('log')
plt.yscale('log')
plt.xlabel('Number of simulations')
plt.ylabel('Absolute error')
plt.savefig('a2.2.pathwise.convergence.png')
```