Quantitative Finance Project Report

Max Malinov, SNR 2078507
Rory Kolster, SNR 2070539
Eugen Mojsović, SNR 2067563
Rūdolfs Jansons, SNR 2080485

20 December 2023

1 Question 1

1.1 Dataset choice

With respect to the dataset we've chosen to work on, we found data on the AEX-index pricing for the past 2 years, starting in December 2021. The website we got it from is linked here. The set contains daily records, from which we extract the 'close' price as the price we use to base our calculations on. We use the natural logarithm of daily returns, estimate the drift and volatility of the index, and finally annualize the estimates.

1.2 Item A

Given that the exercise assumes the index price follows a Geometric Brownian Motion, we know that the respective SDE has the funcional form:

$$dS_t = \mu S_t dt + \sigma S_t dW_t^P, \tag{1}$$

where μ is the drift term, σ is volatility and W^P is a standard Brownian motion.

Throughout the course, we've already established that applying Girsanov's theorem, we can find a probability measure Q, where Q is equivalent to P, such that a stochastic process W^Q is a standard Brownian motion under Q. Thus, we obtain

$$dS_t = rS_t dt + \sigma S_t dW_t^Q. \tag{2}$$

Finally, using Ito's lemma allows us to derive:

$$log(S_t/S_0) = (\mu - \frac{1}{2}\sigma^2)t + \sigma W_t^P. \tag{3}$$

The derivation above depicts that if we want to estimate μ and σ for the index price, we can calculate the standard deviation of log returns, which yields the σ approximation. Then, we take the mean of log returns to get the drift parameter of $log(S_t/S_0)$, which is equal to $\mu + \frac{1}{2}\sigma^2$. Having determined the drift of log returns and the index volatility estimate, we simply solve

$$\mu = \mu_{logreturns} + \frac{1}{2}\sigma^2 \tag{4}$$

However, the values we obtain above are daily based. Therefore, the drift and volatility we obtain aren't suitable for the purpose of the exercise yet. As a final step, to achieve the parameters of interest we multiply μ by 252 and σ by the root of 252. This adjustment ensures that the values we get capture the annual performance of the index, given that by convention we assume that generally there are 252 tradings day a year, which is what our research points at. Here are our respective results:

μ estimate	0.0046199
σ estimate	0.1765598

Table 1: Parameter Annual Estimates

As portrayed, and much expected, the drift is small over 12 months, because our estimation showed a very small drift for the daily computation, so we assumed the annual one would also be low. The sigma is around 17.7%, which captures the fluctuation in the index price.

1.3 Item B

Here, our goal is to generate a proxy for the option price and obtain a confidence interval for that price. First, we construct the sample paths for the stock price over 5 time points (in the exercise, we are asked to monitor the price across the 5 years maturity). Therefore, the time step we define, dt, is equal to 1. For each Monte Carlo simulation i=1,...,n, we estimate the index price for each of the 5 time points. At each time point $time$, we compute the index price as

$$S[i, time] = S[i, time - dt] \times e^{(r-0.5\sigma^2)dt+\sigma\sqrt{dt}Z} \tag{5}$$

where Z is a random standard normal variable (we draw a different Z for each time point). For S_0 we use the latest entry in our dataset.
Having constructed the index price paths, we take the average of S_1,S_2,S_3,S_4 and S_5, and subtract it from the strike price K. The option price we estimate is given by

$$C_T = \max(0, K - \sum_{i=1}^{5} S_i) \tag{6}$$

Since C_T is a vector, the proxy for the option price in our case is the average of the entries in the vector. Applying the Central Limit theorem, we obtain the 95% confidence interval by adding/subtracting to the proxy price the desired quantile, 1.96, of the standard normal, times the standard deviation of the C_T vector divided by $\sqrt{n}$. The number of simulations we choose to conduct is 100,000. This count is sufficiently large to produce cohesive results, whilst it isn't as computationally expensive as larger figures would be. Our results are displayed below: Given that

price	45.61582
95% CI	(44.06022, 47.17142)

Table 2: Option Price Estimates

across the dataset, especially towards the last records, the index price is significantly higher than the strike, and the drift and volatility estimated above indicate that it would be less likely for a major

enough fluctuation to occur so that the index price falls below the strike, we expected many zeros to occur in the option price vector. This turned out to be the case, hence our proxy for the price is quite low - 45.6 euros. However, it stands to reason given that the monitored index performance indicates steady price increase over time. Thus, we deem our results plausible. The confidence interval isn't too narrow, but we prescribe this to the option fluctuations and the fact that every index trajectory is driven by a random standard normal variable, which could give rise to wider gaps between simulation outputs.

1.4 Item C

Here, prior to constructing the common and non-common random numbers estimates, we generate functions for the determination of the index price trajectories to improve readability and help us see where our errors stem from in case anything counterintuitive arises. With respect to index price simulation, we develop three functions: *stock_price*, *stock_price_bump* and *stock_price_both*. The three carry out similar operations: they construct trajectories for the index price over 5 time steps (the 5 years investigated) for n MC simulations. However, they do it slightly differently. The first one is exactly as used in item B, in equation (5). The second function isn't substantially different, for the only difference is that we add a 'bump',η, to the initial index price value S_0; the rest of the procedure is identical to the one in *stock_price*. These two functions will be used for the non-common random numbers estimation, as they use different random standard normal variables in the construction of the paths for the index with and without the bump. Contrary, the *stock_price_both* function merges the previous two. It generates the paths with and without bump based on the same randomly drawn Z for each time point in each MC simulation, hence the name 'common random numbers estimation.' The function returns two arrays-one for the bumped index price and one for the ordinary S_0.
The final prerequisite function we define is $optionPrice$, which takes in an array of stock trajectories, strike K, interest r and maturity K and feeds back the discounted payoffs of the option based on the C_T formula.
Before initiating the estimation procedure, we choose our bump size, η. We set it to equal $n^{-\frac{1}{4}}$, because it is optimal for the bias-variance trade-off with regards to the estimator. For the common numbers case, we use the *stock_price_both* function to generate the index trajectories and then call $optionPrice$ to obtain the discounted payoffs for both. Taking the mean of each of them, we can calculate the delta proxy as

$$deltaProxy = \frac{mean(discountedPayoffBump) - mean(discountedPayoff)}{\eta} \tag{7}$$

To get a confidence interval, we also needed to take the standard deviation of the difference between the discounted payoffs with and without bump, and then divide it by η. The results we get are as follows:

Delta	-0.262093
95% CI	(-0.268961, -0.255225)

Table 3: Common Numbers Delta Estimate

We're happy with the output. Across different iterations, with common random numbers the delta was always into the negatives, with aligns well with our interpretation-the index price is steadily higher than the strike as previously discussed, so major fluctuations are less likely. If the price keeps

rising, the probability of exercising the option grows smaller and smaller, which disvalues the option itself. Therefore, as price increases, option price goes down, depicted by the negative delta.

In the non-common random numbers setup, we use the *stock_price* and *stock_price_bump* function to construct the index trajectories. In this scenario, the two are driven by different Brownian motions at each time step (unlike with common numbers). Having produced the prices, we again use *optionPrice* to get the discounted payoffs for each case (with and without bump), take the mean of the retrieved vectors and get a delta proxy as in equation (7). Computing the standard deviation as well, we construct a 95% confidence interval. The output is provided below:

Delta	-8.367657
95% CI	(-26.804932, 10.069618)

Table 4: Non-common Numbers Delta Estimate

As portrayed by the output, this method appears to be significantly more imprecise than common numbers. Indicative is the width of the confidence interval. The estimate generated via common random numbers falls into the confidence interval above as well. We attribute the imperfection of this method to the fact that it involves too many distinct standard normal random variables drawn at each simulation for each time point. Such a practice allows for the emergence of wider mismatch between the bumped and not bumped trajectories, which would justify the balloon in the standard deviation we observe. Again, we see that for the bigger part the confidence interval is into the negatives, which supports the reasoning provided before - spikes in stock price presuppose devaluation of the option of interest. In summary, the former estimation method performed considerably worse than common random numbers on our dataset.

2 Question 2

In question 2, we assume the Black-Scholes market with parameters $\mu = 10\%$, $\sigma = 25$ %, $S_0 = 100$, $B_0 = 1$ and $r = 3\%$. We consider a European put option with strike price $K = 100$ and maturity $T = 2$

2.1 Item A

The task in item A is to compute the delta at time 0 and to get all possible numerical approximations of it.
First we obtain the exact value:

$$\Delta_{putexact} = -\Phi(-d_1) \tag{8}$$

where

$$d_1 = \frac{\ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t)}{\sigma\sqrt{T-t}} \tag{9}$$

The value we get is -0.36449013
Secondly, bump and reprice
For the bump and reprice, we simulated 100000 paths of the stock price, both with normal price S_0 and with the perturbed prices $S_0 + \epsilon$ where epsilon is equal to path amount to the power of -1/4 to minimize mse . Having generated the stock paths, we calculated the option prices and got the final estimation as

$$\Delta_{\text{Bump-and-reprice}} = \frac{mean(Bumpedprices) - mean(prices)}{\epsilon} \tag{10}$$

The value we get is around -0.38697062
Thirdly, the pathwise method We again simulated 100000 paths (the Z variable) of the stock price and where

$$\Delta_{\text{pathwise}} = mean(1_{\{S_T>K\}} \exp\left(r - \frac{2\sigma^2}{2}\right) T + \sigma T Z) \tag{11}$$

The value we get is around -0.36536321
Lastly, likelihood ratio method, where we again simulated 100000 paths to firstly get the density of S at time T and then calculated delta

$$\Delta_{\text{likelihood ratio}} = e^{-rT} E_q \left[\frac{F(S_T) \cdot Z}{\sigma\sqrt{T} S_0}\right] \tag{12}$$

The value we get is around -0.36843843

2.2 Item B

The task in item B is to compute the gamma at time 0 and to get all possible numerical approximations of it. Since gamma is a second order greek, it cannot be approximated by path wise method, only by bump and reprice and likelihood ratio method. First we obtain the exact value:

$$\Gamma = \frac{\phi(d_1)}{S \cdot \sigma \cdot \sqrt{T}} \tag{13}$$

where

$$d_1 = \frac{\ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right) T}{\sigma\sqrt{T}} \tag{14}$$

the value we get is 0.0106264

For the bump and reprice, we simulated 10000 paths of the stock price, both with normal price S_0 and with the perturbed prices $S_0 + \epsilon$ and $S_0 - \epsilon$. Having generated the stock paths, we calculated the option prices and got the final estimation as

$$\Gamma_{\text{Bump-and-reprice}} = \frac{mean(Bumpedprices) + mean(Oppositebumpedprices) - 2 \cdot mean(Prices)}{\epsilon^2} \tag{15}$$

the estimation we get is 0.0106263

Finally, for the likelihood ratio method, first we obtain the normal pdf value of d1 and use it in the formula

$$\Gamma_{\text{Likelihood Ratio}} = \frac{\phi(d_1)}{S \cdot \sigma \cdot \sqrt{T}} \cdot \left(1 - d_1 \cdot \frac{\phi(d_1)}{\Phi(d_1)}\right) \tag{16}$$

The estimation we calculate using the likelihood ratio method is 0.0084497, which is noticeable worse than the Bump and reprice method.

2.3 Item C

In this part, we are selling 1000 put options based on the stock described in order to fund a delta-neutral hedge position on the stock and the bond in our portfolio. The general idea is to make the portfolio delta-neutral so we first determine the position in the risky asset, and then from this

determine the position in the bonds in order to make the net cashflow zero until maturity.

The basic form of the code follows similarly to the code presented in discrete-time delta hedging strategy on github, but then vectorized to avoid for loops when doing the Monte Carlo simulations.

First we import the required packages and then initialize the variables, including the number of put options being sold, the step-size and the number of Monte Carlo simulations. Then we define a function that calculates and returns the price of the put option, the delta, and the gamma, given the input parameters.

Then we define another function that performs the discrete-time hedging according to the logic defined briefly above. It initializes the vector/matrix spaces and then the initial starting values respectively. Then it loops through the time-grid determining, the put values, the position in S and B such that the portfolio is delta-neutral and there is no net cashflow. At the end it returns the matrices, which we will use to find the quantiles.

The next function we define takes the matrices from the last function and determines the 5th, 50th and 95th quantiles for each step in the time-grid for the positions in S and B, as well as the gamma of the portfolio. Then we plot these in 3 graphs.

We report the positions in S and B at time 0 are -364.49... and 47353.89... respectively. After a random instance of running the Monte Carlo simulations, we yielded a mean portfolio value of -3.68... and standard deviation of 855.39..., at maturity T. We would charge roughly 6.6 for the put option, as we received values around that for the mean price (discounted) of the put option just before maturity, after running the MC simulations multiple times. Below we provide the graphs.

We also provide the graph of the first Monte Carlo replication for some background, as well as the histogram of the portfolio values at one step before maturity.

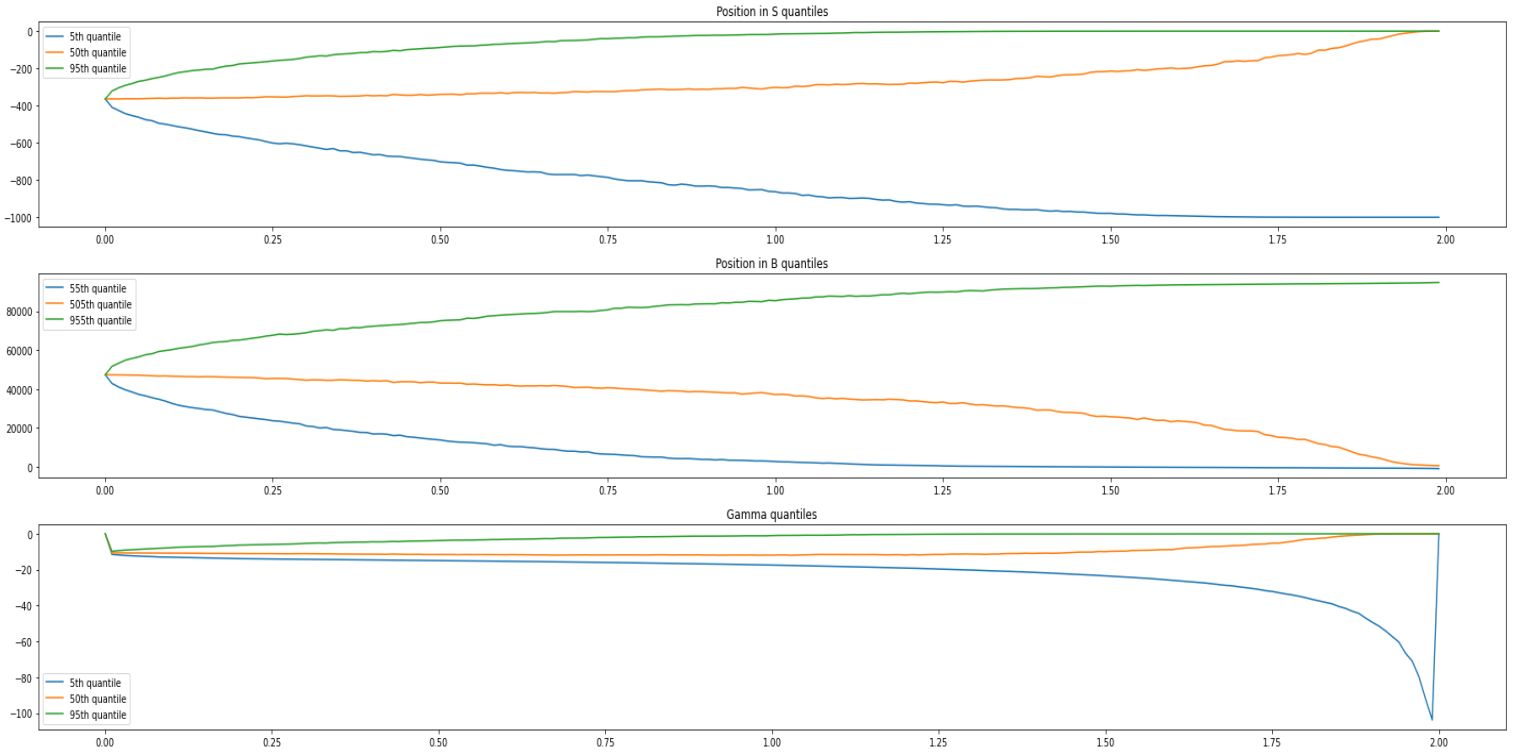

Figure 1: Quantiles of position in S and B and portfolio gamma

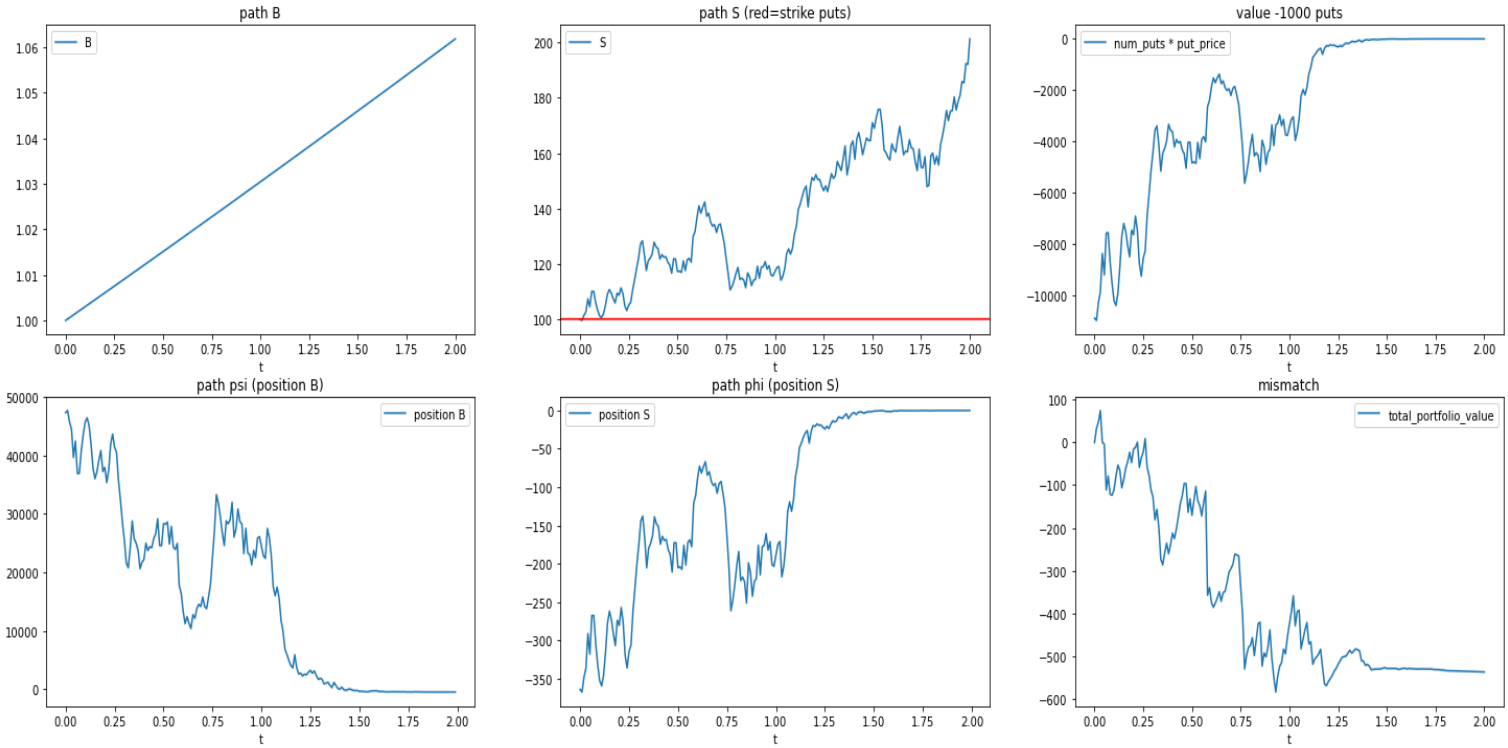

Figure 2: Graphs on first Monte Carlo simulation

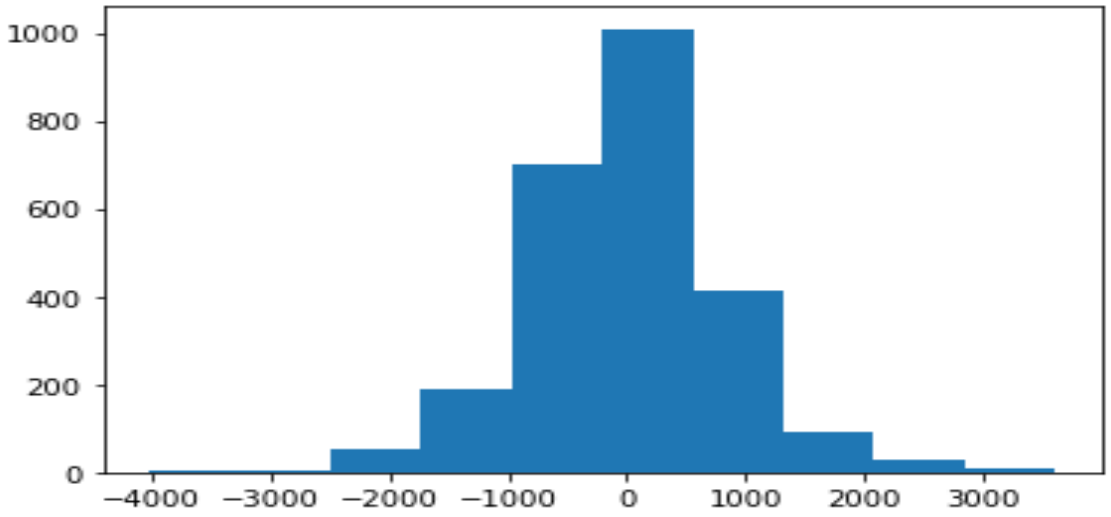

Figure 3: Histogram of portfolio values just before maturity

2.4 Item Di

In this question we also consider a call option based on the same stock, but with a maturity of 5 and a strike price of 120. And we repeat the process in item C but with delta-gamma hedging. The logic behind this is to first make the portfolio gamma-neutral, which dictates the position in the call option, and then using that we can make the portfolio delta-neutral, which determines the position in S. Then finally we make sure that there is no net cashflow before maturity $T = 2$.

The code is fairly similar to the code used in item C, but an extra function for determining call option price, delta and gamma was written, and the main function that returns the matrices of data containing the positions, Stock prices, bond prices, portfolio deltas and gammas, has been altered such that the portfolio has both position in the stock and the call option, and the bonds.

We report the positions in S and B at time 0 are -1261.39..., 1526.06..., and 105347.73... respectively. After the Monte Carlo simulations, we yielded a mean portfolio value of 6.61... and standard deviation of 118.15..., at maturity T. Below we provide the graphs.

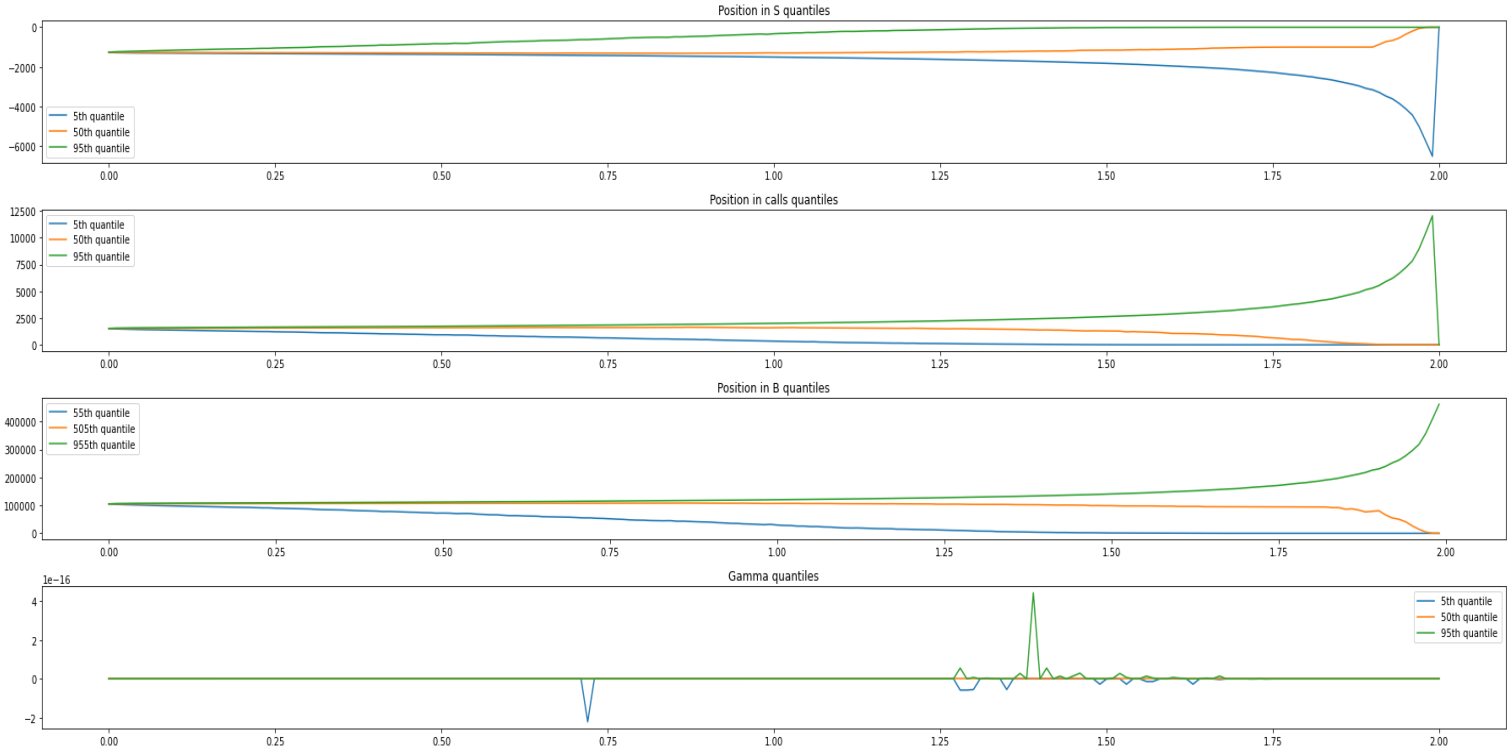

Figure 4: Quantiles of position in S, call option and B and portfolio gamma

We also provide the graph of the first Monte Carlo replication for some background, as well as the histogram of the portfolio values at one step before maturity.

For the results, we can see that delta-gamma hedging leads to a significantly lower standard deviation of the portfolio value, while retaining high stability of the mean value.

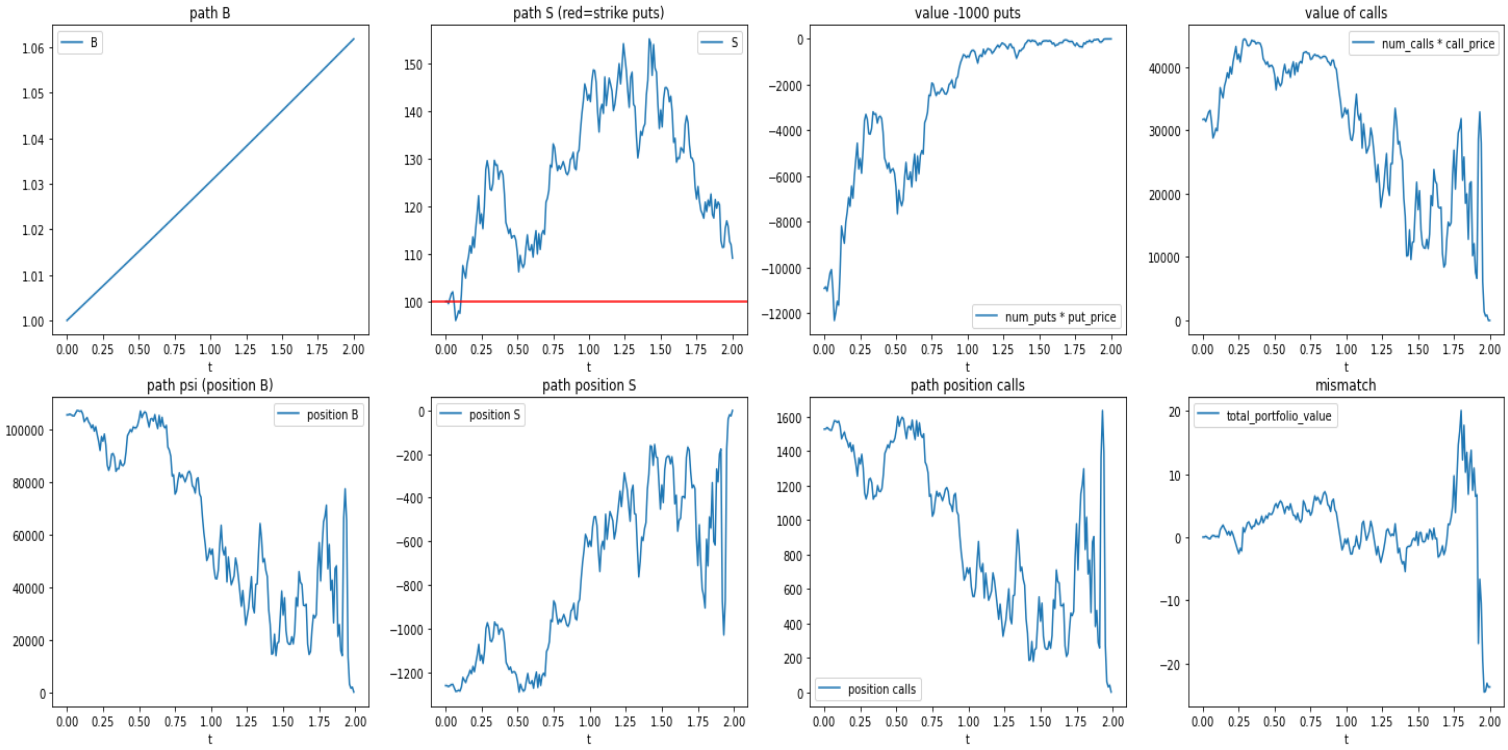

Figure 5: Graphs of the first Monte Carlo simulation

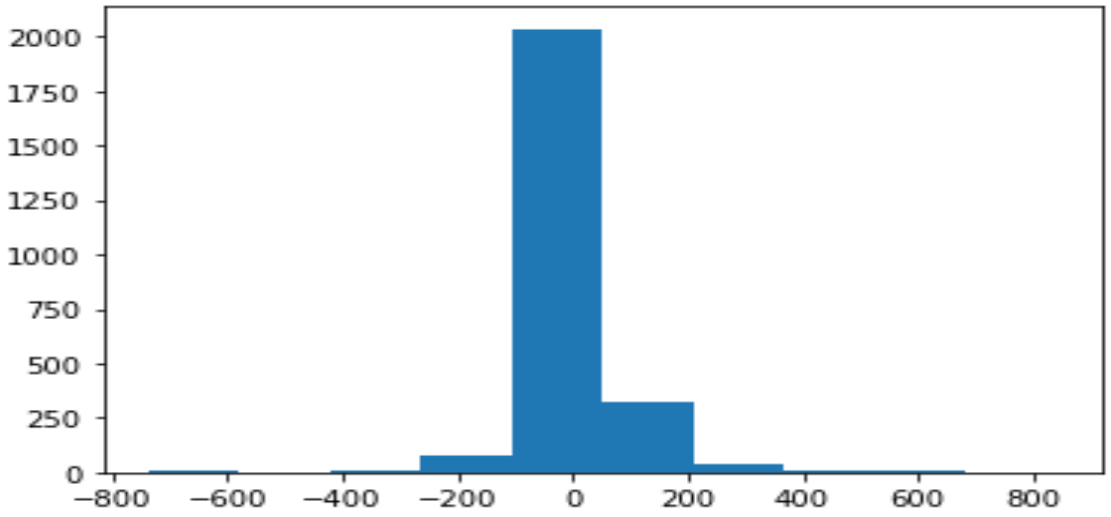

Figure 6: Histogram of the portfolio values just before maturity

2.5 Item Dii

In this exercise, we are asked to delta-gamma-hedge a portfolio consisting of 2 stocks and a single option. To begin with gamma-hedging, we know that stocks have a gamma of 0. The only security with non-zero gamma in our portfolio is the option. In order to achieve gamma-neutrality, we must have no options in our portfolio. Now, to delta-hedge, our position in stocks must be such that the portfolio has a delta of zero. We know the delta of a stock is 1. Therefore, we need to have as many stocks in short-position as we have in long-position. If we denote our position in the option with x_{option} and the positions in stocks 1 and 2 with x_{s1} and x_{s2}, respectively, we would gamma-delta-hedge our portfolio if

$$x_{\text{option}} = 0 \tag{17}$$

and

$$x_{\text{s1}} = -x_{\text{s2}} \tag{18}$$

Listing 1: Python Code

```
import numpy as np
import pandas as pd
import requests
from scipy.stats import norm
import math
import matplotlib.pyplot as plt

## First, we set a common seed so that our results are reproducible at
    later stages.
np.random.seed(1)

#
    ================================================================================

# Exercise 1
# Item A)
#
    ================================================================================

## Using the requests package, we import our data from Github. The set
    consists of
## daily records for the stock price paths for the past 2 years (from
    December 2021
## onwards).
github_url = 'https://raw.githubusercontent.com/emojsovic/qf/main/data.
    csv'
response = requests.get(github_url)
open('local_filename.csv', 'wb').write(response.content)
data_raw = pd.read_csv('local_filename.csv',sep = ';')

## We generate an empty dataframe where we store the closing stock price
     for each day as
## the pricewe use in our calculations in the follow-up. According to
    the research we
## did, this is the conventional way to tackle such option estimation
    tasks.
data=pd.DataFrame()
data['Price'] = (data_raw['Close'])

## Then, to facilitate the parameter estimation procedure, we take the
    log returns of
## our dataset. We accomplish this via taking the natural log of the
    quotient of the
## stock price at time t divided by the price one unit of time backward.
     That's the
## motivation behind using the shift(1) specification.
data['Log-Returns'] = np.log(data['Price']/data['Price'].shift(1))
```

```
## Adhering to the GBM assumption, we see that the log(St/S0) has the
    same volatility
## term as the differential equaton for St, so we esimate the sigma by
    taking the
## standard devation of log returns. However, the aforementioned
    differental equation
## for log returns features a different drift term than the one in dSt.
    Namely,
## the drift here is given by (mu-0.5*sigma**2), where mu is the drift
    of the original
## equation. Hence, feeding in the sigma obtained before, we can
    estimate mu as well.
## As a final note we acknowledge that due to the import of daily data
    into our project,
## we need to account for the fact we're investigating annual periods
    and adjust the
## estimated parameters by shifting them by a factor of 252. We take the
     252 factor
## for by convention there are roughly 252 trading days a year.
sigma_daily = data['Log Returns'].std()
mu_daily = (data['Log Returns'].mean() + (sigma_daily**2/2) )

sigma_annual= sigma_daily*np.sqrt(252)
mu_annual = mu_daily*252

print("Exercise 1, Item A")
print("-----Sigma",sigma_annual,"\n------Mu",mu_annual)
#
    ================================================================


# Item B)
#
    ================================================================


## In this item, we generate the option price at t=0 via a sequence of
    steps.
## Routinely, we first list all parameters in the function definition we
     will
## need later on.

def option_price_estimate(T,K,S0,r,sigma,n,dt):

    ## Now, we generate empty arrays where we intend to store the stock
        price paths
    ## and, eventually, the option price estimates.
    stock_price = np.empty((n,T+1))
    option_price = np.empty(n)
```

```
## For each stage of the MC simulations, we set the initial stock
    price, S0, to be
## the S0 we feed into the function (it's predetermined, not random)
    .
for i in range(0,n):
    stock_price[i,0] = S0

    ## For each year to maturity, we generate a new standard normal
        variable for the
    ## Wiener motion in the SDE and update the stock price using the
        GBM equation.
    for time in range(dt,T+1,dt):
        Z = np.random.normal()
        stock_price[i,time] = stock_price[i,time-dt]*np.exp((r-0.5*
            sigma**2)*dt + sigma*np.sqrt(dt)*Z)

    ## To obtain the option price for each simulation given the
        provided option
    ## price functional prescription, we take the average of the 5
        generated stock
    ## price paths for t = 1,2,3,4,5, and take the maximum of 0 and
        strike-average.
    ## Note that we disregard S0 from the mean for it's not included
        in the
    ## option pricing formula!!!
    mean_stock = np.mean(stock_price[:,1:], axis = 1)
    option_price[i] = np.maximum(0, K-mean_stock[i])

## Finally, to retrieve the estimate as desired, we discount the
    price obtained
## above and take the average of the discounted prices.
discounted_price = np.exp(r*T)*option_price
proxyPrice = np.mean(discounted_price)

## As for the confidence interval, adhering to the Central Limit
    Theorem for the
## functional prescription, we use the standard deviation of the
    discounted option
## prices and use the 97.5% quantile (1.96) to obtain a 95% CI for
    the option price.
deviationPrice = np.std(discounted_price)
confidenceIntervalLeft = proxyPrice - 1.96 * deviationPrice / np.
    sqrt(n)
confidenceIntervalRight = proxyPrice + 1.96 * deviationPrice / np.
    sqrt(n)
confidenceInterval = (confidenceIntervalLeft,
    confidenceIntervalRight)
```

```
    return proxyPrice, confidenceInterval


## To test the performance of the procedure defined above, we take the
    last record
## in our dataset as S0, the estimated sigma in 1a as the volatility
    parameter and
## 2% as the risk-free interest rate (conventional assumption in the
    field). The other
## parameters we use as given in the exercise. We pick the number of
    simulations to be
## 100000, as we postulate that it would suffice to provide various
    option prices based
## on the underlying stock fluctuations and should converge to the
    actual option price
## if our assumptionas and model are correct.
S0 = data['Price'].iloc[1]
K = 740
r = 0.02
sigma  = sigma_annual
T = 5
dt = 1
n = 10000
eta = n**(-0.25)
option_price, confidence_interval = option_price_estimate(T,K,S0,r,sigma
    ,n,dt)
print("\nExercise 1, Item B")
print("-----Option Price:", option_price)
print("-----95% Confidence Interval:", confidence_interval)

#
    ================================================================


# Item C)
#
    ================================================================


## In this item, we define a number of functions to improve the
    readability of our code.

## We start off with the stock price generating function for the 5 years
     of question.
## This first function doesn't include bumps.
def stock_price(T,K,S0,r,sigma,n,dt):

    ## Just as in 1a, we start by defining an empty array for the stock
        prices to be
    ## stored in upon generation in each simulation.
```

```
    stock_price = np.empty((n,T+1))
    for i in range(0,n):
        ## Again, we set the initial stock price to be deterministic,
            namely the last
        ## available record in our dataset.
        stock_price[i,0] = S0
        for time in range(dt,T+1,dt):
            ## For each time point, we generate a new standard normal
                variable
            ## and recalculate the new stock price using the GBM pricing
                 formula.
            ## The stock price obtained at period t is used as the
                initial price
            ## when determining the price of the stock at t+1. We can do
                 this since
            ## the time step is one year, so our functional
                prescriptions remain intact.
            Z = np.random.normal()
            stock_price[i,time] = stock_price[i,time-dt]*np.exp((r-0.5*
                sigma**2)*dt + sigma*np.sqrt(dt)*Z)

    ## Finally, we return a vector of the constructed stock paths.
    return stock_price


## This function estimates the stock price, where the very first price
    is bumped.
def stock_price_bump(T,K,S0,r,sigma,n,dt,eta):

    ## We generate the empty array for the stock price paths.
    stock_price_bump = np.empty((n,T+1))
    for i in range(0,n):

        ## The first entry this time is S0 added the bump eta feeded to
            the function.
        ## The remainder of the function follows the exact same course
            as the function
        ## without bumps.
        stock_price_bump[i,0] = S0 + eta
        for time in range(dt,T+1,dt):
            Z = np.random.normal()
            stock_price_bump[i,time] = stock_price_bump[i,time-dt]*np.
                exp((r-0.5*sigma**2)*dt + sigma*np.sqrt(dt)*Z)

    ## Finally, we return the estimated stock prices given the initial
        bumpy one.
    return stock_price_bump
```

```
## The third function administers the case where (for the common random
    numbers
## estimation) we use the same standard normal Z for both the non-bumped
     and the bumped
## stock price.
def stock_price_both(T,K,S0,r,sigma,n,dt,eta):

    ## For both the case with and without nump, we perform the usual
        procedure of
    ## defining the empty arrays for the stock prices.
    stock_price = np.empty((n,T+1))
    stock_price_bump = np.empty((n,T+1))
    for i in range(0,n):

        ## For no-bump, the first record is always just S0, whereas for
            the bumped
        ## scenario we add the eta to S0.
        stock_price[i,0] = S0
        stock_price_bump[i,0] = S0 + eta
        for time in range(dt,T+1,dt):

            ## We draw a random standard normal Z, but this time the
                SAME Z is used
            ## for the estimation of both the bumped and not bumped
                stock price.
            Z = np.random.normal()
            stock_price[i,time] = stock_price[i,time-dt]*np.exp((r-0.5*
                sigma**2)*dt + sigma*np.sqrt(dt)*Z)
            stock_price_bump[i,time] = stock_price_bump[i,time-dt]*np.
                exp((r-0.5*sigma**2)*dt + sigma*np.sqrt(dt)*Z)

    ## Finally, we feed back the vectors of stock prices.
    return stock_price, stock_price_bump

## The penultimate function concerns the estimated option payoffs,
def optionPrice(S,K,r,T):
    ## We use the dimension of each stock price to figure out what n is
        (to avoid
    ## the inclusion of abundant arguments in the function definition)
    n = np.shape(S)[0]

    ## We generate arrays to store the payoffs.
    payoff = np.empty(n)
    discounted_payoff = np.empty(n)

    ## For each of the n sims, we evaluate the process following the CT
        formula.
```

```
    ## As before, we weed out S0 from the mean of S as the option price
        is defined
    ## to depend only on the trajectories for t = 1,...,5.
    ## Then, we discount the payoffs with the interest rate r.
    for i in range(0,n):
        payoff[i] = np.maximum(0,K-np.mean(S[i,1:]))
        discounted_payoff[i] = np.exp(-r*T)*payoff[i]

    ## The function returns a vector of discounted payoffs.
    return discounted_payoff

## For the common random numbers estimation, we commence the following
    procedure.
def common_numbers(T,K,S0,r,sigma,n,dt,eta):
    ## First, using the functions defined above, we obtain arrays of the
        stock
    ## prices for both bumped and not bumped.
    S, S_bump = stock_price_both(T,K,S0,r,sigma,n,dt,eta)

    ## Then, we get the discounted payoffs for both stock trajectories.
    payoff_proxy = optionPrice(S,K,r,T)
    payoff_proxy_bump = optionPrice(S_bump,K,r,T)

    ## To get a proxy for the option price in each case, we take the
        mean of the
    ## discounted payoffs.
    E_proxy = np.mean(payoff_proxy)
    E_proxy_bump = np.mean(payoff_proxy_bump)

    ## Our proxy for delta is the difference between the means above
        divided by
    ## the bump eta.
    proxyDelta = (E_proxy_bump - E_proxy) / eta

    ## To define the confidence interval, we take the standard deviation
        of the
    ## diffecence in option prices (with and without bump) and, again
        pertaining to the
    ## Central Limit Theorem, we retrieve a 95% CI for the delta using
        the 97.5% quantile
    ## of the standard normal distribution.
    deviationDelta = np.std((payoff_proxy_bump-payoff_proxy)/eta)/np.
        sqrt(n)
    confidenceIntervalLeft = proxyDelta - 1.96 * deviationDelta
    confidenceIntervalRight = proxyDelta + 1.96 * deviationDelta
    confidenceInterval = (confidenceIntervalLeft,
        confidenceIntervalRight)
```

```
    ## The function returns the estimated option delta and the desired
        CI.
    return proxyDelta, confidenceInterval

delta_common, interval_common = common_numbers(T,K,S0,r,sigma,n,dt,eta)
print("\nExercise 1, Item C")

print("----- Estimated Delta with common numbers:", delta_common)
print("----- 95% Confidence Interval:", interval_common)

## The last function addresses the non-common random numbers estimation.
def non_common_numbers(T,K,S0,r,sigma,n,dt,eta):
    ## Unlike the common case, here we generate the stock price
        trajectories based on
    ## different standard normal variables. Thus, we expect wider gaps
        to appear between
    ## the two stock prices as each path draws a different random
        standard normal variable
    ## for each of the 5 time steps, which opens the door for deviations
        .
    S = stock_price(T,K,S0,r,sigma,n,dt)
    S_bump = stock_price_bump(T,K,S0,r,sigma,n,dt,eta)

    ## Again, we obtain the discounted payoffs for both.
    payoff_proxy = optionPrice(S,K,r,T)
    payoff_proxy_bump = optionPrice(S_bump,K,r,T)

    ## We take the mean of the discounted payoffs as proxy for option
        price.
    E_proxy = np.mean(payoff_proxy)
    E_proxy_bump = np.mean(payoff_proxy_bump)

    ## The estimate of delta follows the same line of reasoning as
        before. So
    ## does the confidence interval construction. However, our
        hypothesis is that
    ## this time the deviation between the bump and no-bump trajectories
         could fetch
    ## a significantly larger effect on the width of the CI due to the
        fact that the
    ## random numbers are different for each simulation for the pair of
        stock trajectories.
    proxyDelta = (E_proxy_bump - E_proxy) / eta
    deviationDelta = np.std((payoff_proxy_bump-payoff_proxy)/eta)
    confidenceIntervalLeft = proxyDelta - 1.96 * deviationDelta / np.
        sqrt(n)
    confidenceIntervalRight = proxyDelta + 1.96 * deviationDelta / np.
        sqrt(n)
```

```
    confidenceInterval = (confidenceIntervalLeft,
        confidenceIntervalRight)

    ## Like before, we retrieve a delta estimate and a 95% CI.
    return proxyDelta, confidenceInterval

delta_non_common, interval_non_common = non_common_numbers(T,K,S0,r,
    sigma,n,dt,eta)
print("-----Estimated Delta with non-common numbers:", delta_non_common)
print("-----95% Confidence Interval:", interval_non_common)

#
    ================================================================

# Exercise 2
# Item A)
#
    ================================================================

# Given parameters
S0 = 100        # Initial stock price
K = 100         # Strike price
r = 0.03        # Risk-free interest rate
sigma = 0.25    # Volatility
T = 2           # Time to maturity
B0 = 1          # Value of a risk-free bond at t=0



# Calculate d1 and d2
d1 = (np.log(S0 / K) + (r + (sigma**2) / 2) * T) / (sigma * np.sqrt(T))
d2 = d1 - sigma * np.sqrt(T)

# Calculate the Black-Scholes option price
BS_put_price = B0 * (norm.cdf(-d2) * K * np.exp(-r * T) - norm.cdf(-d1)
    * S0)

# Calculate the delta analytically
delta_analytical = -1* norm.cdf(-d1)

# Display the analytical delta
print("\nExercise 2, Item A")

print(f'-----Analytical Delta: {delta_analytical:.8f}')


num_simulations = 100000
epsilon = 1/num_simulations**0.25
```

```
# Generate random samples

Z = np.random.randn(num_simulations)
S0_bump=S0+epsilon #bump
# Calculate S_T for bumped stock price
S_T_bumped = S0_bump * np.exp((r - 0.5 * sigma**2) * T + sigma * np.sqrt
    (T) * Z)

# Calculate option prices for bumped and original stock prices
BS_put_price_bumped_MC = np.mean(np.maximum(K - S_T_bumped, 0))
BS_put_price_MC = np.mean(np.maximum(K - S0 * np.exp((r - 0.5 * sigma
    **2) * T + sigma * np.sqrt(T) * Z), 0))

# Approximate delta using Monte Carlo simulation
delta_monte_carlo = (BS_put_price_bumped_MC - BS_put_price_MC) / epsilon

# Display the result for Monte Carlo simulation
print(f'-----Delta (bump and reprice): {delta_monte_carlo:.8f}')



# Likelihood ratio method
# Generate random samples
Z = np.random.randn(num_simulations)
#define S_T
ST = S0 * np.exp((r - 0.5 * sigma**2) * T + sigma * np.sqrt(T) * Z)

#caculate pay-offs
F_ST=np.maximum(K - ST, 0)
#apply likliehood ratio method
delta_likelihood_ratio=np.exp(-r*T)*np.mean(F_ST*(Z/(sigma*np.sqrt(T)*S0
    )))

# Display the result for the likelihood ratio method
print(f'------Delta (Likelihood ratio method): {delta_likelihood_ratio
    :.8f}')

#pathwise

# Simulate stock prices at maturity
Z = np.random.randn(num_simulations)
ST = S0 * np.exp((r - 0.5 * sigma**2) * T + sigma * np.sqrt(T) * Z)


# Compute the pathwise derivative for each simulation
pathwise_derivatives = -np.where(ST < K, np.exp((r - 0.5 * sigma**2) * T
     + sigma * np.sqrt(T) * Z), 0)
```

```
# Approximate delta using the pathwise method
delta = np.mean(np.exp(-r*T)*pathwise_derivatives)

print(f'-----Delta-(Pathwise-Method):-{delta:.8f}')

#
    ================================================================

# Item B)
#
    ================================================================



mu = 0.10  # drift
sigma = 0.25  # volatility
S0 = 100  # initial stock price
B0 = 1  # initial bond price
r = 0.03  # risk-free rate
K = 100  # strike price
T = 2  # maturity
eta = 0.001  # Bump amount for S0
n = 10000  # Number of Monte Carlo paths
d = 365  # Number of time steps for each path

d1 = ((math.log(S0 / K) + (r + 0.5 * sigma ** 2) * T)) / (sigma * math.
    sqrt(T))
GammaExact = norm.pdf(d1) / (S0 * sigma * math.sqrt(T))


def simulate_stock_paths(S0, r, sigma, T, n, d):
    dt = T / d
    paths = np.zeros((n, d + 1))
    paths[:, 0] = S0
    for i in range(1, d + 1):
        Z = np.random.normal(0, 1, n)
        paths[:, i] = paths[:, i - 1] * np.exp((r - 0.5 * sigma**2) * dt
            + sigma * np.sqrt(dt) * Z)
    return paths
def black_scholes_put(S, K, T, r, sigma):
    d1 = (np.log(S / K) + (r + 0.5 * sigma**2) * T) / (sigma * np.sqrt(T
        ))
    d2 = d1 - sigma * np.sqrt(T)
    put_price = K * np.exp(-r * T) * norm.cdf(-d2) - S * norm.cdf(-d1)
    return put_price
def BaRGamma(sigma,S0,B0,r,K,T,eta,n,d):
    stock_paths = simulate_stock_paths(S0, r, sigma, T, n, d)
```

```
    initial_call_price = np.mean(black_scholes_put(stock_paths[:, 0], K,
        T, r, sigma))
    bumped_call_prices = black_scholes_put(stock_paths[:, 0] + eta, K, T
        , r, sigma)
    bumped_call_prices_opposite = black_scholes_put(stock_paths[:, 0] -
        eta, K, T, r, sigma)
    return np.mean((bumped_call_prices - 2 * initial_call_price +
        bumped_call_prices_opposite) / eta**2)

GammaBumpAndReprice = BaRGamma(sigma,S0,B0,r,K,T,eta,n,d)


def gamma_likelihood_ratio(S, K, T, r, sigma,d1):
    pdf_d1 = norm.pdf(d1)
    gamma_lr = (pdf_d1 / (S * sigma * np.sqrt(T))) * (1 - d1 * (pdf_d1 /
        norm.cdf(d1)))
    return gamma_lr

GammaLR = gamma_likelihood_ratio(S0, K, T, r, sigma,d1)


print("\nExercise 2, Item B")

print('-----Exact Gamma', GammaExact, '\n-----Bump and reprice gamma',
    GammaBumpAndReprice, '\n-----LR gamma',GammaLR)


#
    ================================================================


# Item C)
#
    ================================================================


# Black_Scholes market:
mu = 0.1
sigma = 0.25
S0 = 100
B0 = 1
r = 0.03
K = 100
T = 2
step = 0.01
num_puts = 1000


M = 2500
print("\nExercise 2, Item C")
```

```
def put_option_BS(K,r,sigma,St,time_to_maturity):


    d1 = (np.log(St/K) + (r + 0.5*(sigma)**2)*time_to_maturity)/(sigma*
        np.sqrt(time_to_maturity))
    d2 = d1 - sigma*np.sqrt(time_to_maturity)

    put_price = np.exp(-r*time_to_maturity)*K*norm.cdf(-d2) - St*norm.
        cdf(-d1)

    put_delta = -norm.cdf(-d1)

    put_gamma = norm.pdf(d1) / (St * sigma * np.sqrt(time_to_maturity))

    return put_price, put_delta, put_gamma



def put_option_hedge(K,T,S0,mu,sigma,B0,r,step,num_puts,M):

    # Calculate the number of steps using maturity and step size
    num_steps = int(T/step)

    # Initializing our variables
    phi = np.zeros((M, num_steps + 1))
    psi = np.zeros((M, num_steps + 1))
    phi[:, -1] = np.nan
    psi[:, -1] = np.nan
    price_puts = np.zeros((M, num_steps + 1))
    S = np.zeros((M, num_steps + 1))
    S[:, 0] = S0
    B = np.zeros((M, num_steps + 1))
    B[:, 0] = B0
    portfolio_value = np.zeros((M, num_steps + 1))
    time = np.linspace(0,T,num_steps + 1)
    delta = np.zeros((M, num_steps + 1))
    gamma = np.zeros((M, num_steps + 1))

    # Determining the intial positions
    put_price_initial = put_option_BS(K,r,sigma,S0,T)[0]
    price_puts[:, 0] = num_puts*put_price_initial
    phi[:, 0] = - num_puts*put_option_BS(K,r,sigma,S0,T)[1] # Making the
        total portfolio value delta_neutral
    print('-----Position in S at time 0:'+ str(phi[0,0]))
    psi[:, 0] = - (price_puts[:, 0] + phi[:, 0]*S[:, 0])/B[:, 0]
    print('-----Position in B at time 0:'+ str(psi[0,0]))
```

```
portfolio_value[:, 0] = price_puts[:, 0] + phi[:, 0]*S[:, 0] + psi
    [:, 0]*B[:, 0] # This should be 0 by construction

# Iterating over the time grid:
for t in range(1, num_steps + 1):

    # New asset prices:
    B[:, t] = B[:, t-1]*np.exp(r*step)
    S[:, t] = S[:, t-1] * (1 + np.expm1((mu - 0.5 * sigma**2) * step
        + sigma * np.sqrt(step) * np.random.normal(size=M)))
    # expm1 used as it is more accurate for values near zero

    # Current portfolio value before repositioning:
    value = phi[:, t-1]*S[:, t] + psi[:, t-1]*B[:, t]

    # New value puts:
    if time[t] == T:

        price_puts[:, t] = num_puts*np.maximum(K - S[:, t], 0)
        portfolio_value[:, t] = price_puts[:, t] + value

        # Calculating mean and standard deviation of the total
            portfolio value at maturity
        portfolio_mean = np.mean(portfolio_value[:, t])
        portfolio_std_dev = np.std(portfolio_value[:, t])
        print("-----Mean of total portfolio value at maturity:" +
            str(portfolio_mean))
        print("-----Standard deviation of total portfolio value at
            maturity:"+ str(portfolio_std_dev))

        break
    price_puts[:, t] = num_puts*put_option_BS(K,r,sigma,S[:, t],
        time_to_maturity=T-time[t])[0]

    # Determining the delta-neutral postion:
    phi[:, t] = - num_puts*put_option_BS(K,r,sigma,S[:, t],
        time_to_maturity=T-time[t])[1]

    # Determining position in B such that there is no net cashflow
    psi[:, t] = (value - phi[:, t]*S[:, t])/B[:, t]

    # Updating portfolio value:
    portfolio_value[:, t] = price_puts[:, t] + phi[:, t]*S[:, t] +
        psi[:, t]*B[:, t]

    # Calculating delta:
    delta[:, t] = num_puts*put_option_BS(K,r,sigma,S[:, t],
        time_to_maturity=T-time[t])[1] + psi[:, t]
```

```
        # Calculating gamma:
        gamma[:, t] = num_puts*put_option_BS(K,r,sigma,S[:, t],
            time_to_maturity=T-time[t])[2]


    return time, S, B, phi, psi, price_puts, portfolio_value, delta,
        gamma, portfolio_mean, portfolio_std_dev

time, S, B, phi, psi, price_puts, portfolio_value, delta, gamma,
    portfolio_mean, portfolio_std_dev = put_option_hedge(K,T,S0,mu,sigma
    ,B0,r,step,num_puts,M)

quantiles = [5, 50, 95]

def quants(quantiles, time, S, B, phi, psi, price_puts, portfolio_value,
     gamma):

    phi_quants = np.zeros((3, np.shape(phi)[1]))
    psi_quants = np.zeros((3, np.shape(psi)[1]))
    gamma_quants = np.zeros((3, np.shape(gamma)[1]))

    for t in range(0, len(time)):

        phi_quants[:, t] = np.percentile(phi[:, t], quantiles)
        psi_quants[:, t] = np.percentile(psi[:, t], quantiles)
        gamma_quants[:, t] = np.percentile(gamma[:, t], quantiles)

    return phi_quants, psi_quants, gamma_quants

phi_quants, psi_quants, gamma_quants = quants(quantiles, time, S, B, phi
    , psi, price_puts, portfolio_value, gamma)


df = pd.DataFrame(data = np.array([time, S[0,:], B[0,:], phi[0,:], psi
    [0,:], price_puts[0,:], portfolio_value[0,:]]).T, columns=["t", "S",
     "B",
                                "position_S", "position_B", "num_puts_*_
                                    put_price", "total_portfolio_value"
                                    ])
fig, ax = plt.subplots(2, 3, figsize=(25, 10))
df.plot(x="t", y="B", title="path_B", ax=ax[0, 0])
df.plot(x="t", y="S", title="path_S_(red=strike_puts)", ax=ax[0, 1])
ax[0, 1].axhline(y=K, color="r")
df.plot(x="t", y="num_puts_*_put_price", title=f"value_{num_puts}_puts",
     ax=ax[0, 2])
df.iloc[:-1].plot(x="t", y="position_B", title="path_psi_(position_B)",
    ax=ax[1, 0])
```

```
df.iloc[:-1].plot(x="t", y="position-S", title="path-phi-(position-S)",
    ax=ax[1, 1])
df.plot(x="t", y="total_portfolio_value", title="mismatch", ax=ax[1, 2])


df = pd.DataFrame(data = np.array([time, phi_quants[0,:],phi_quants
    [1,:],phi_quants[2,:], psi_quants[0,:],
                                       psi_quants[1,:], psi_quants[2,:],
                                          gamma_quants[0,:], gamma_quants
                                          [1,:], gamma_quants[2,:]]).T,
                      columns=["t", "position-S-5th-quantile", "position-S-
                          50th-quantile", "position-S-95th-quantile",
                                                 "position-B-5th-quantile"
                                                     , "position-B-50th-
                                                     quantile", "position-
                                                     B-95th-quantile",
                                                 "gamma-5th-quantile", "
                                                     gamma-50th-quantile",
                                                      "gamma-95th-quantile
                                                     "])
fig, ax = plt.subplots(3,1, figsize=(25,10))

for i in range(0,3):
    ax[0].plot(time, phi_quants[i,:], label=f'{quantiles[i]}th-quantile'
        )

ax[0].set_title('Position-in-S-quantiles')
ax[0].legend()


for i in range(0,3):
    ax[1].plot(time, psi_quants[i,:], label=f'{quantiles[i]}5th-quantile
        ')

ax[1].set_title('Position-in-B-quantiles')
ax[1].legend()

for i in range(0,3):
    ax[2].plot(time, gamma_quants[i,:], label=f'{quantiles[i]}th-
        quantile')

ax[2].set_title('Gamma-quantiles')
ax[2].legend()


plt.tight_layout()
plt.show()
```

```
plt.hist(portfolio_value[:, 200-1])

print(f"-----Mean price of the put option at just before maturity: {(np.
    mean(price_puts[:, 200-1])/-1000)*np.exp(-r*T)}")


# ============================================================================

# Item D)
# ============================================================================

print("\nExercise 2, Item Di")


mu = 0.1
sigma = 0.25
S0 = 100
B0 = 1
r = 0.03
K = 100
T = 2
step = 0.01
num_puts = -1000


M = 2500 # number of MC replications

# New strike and maturity for call option
K2 = 120
T2 = 5

def put_option_BS(K,r,sigma,St,time_to_maturity):


    d1 = (np.log(St/K) + (r + 0.5*(sigma)**2)*time_to_maturity)/(sigma*
        np.sqrt(time_to_maturity))
    d2 = d1 - sigma*np.sqrt(time_to_maturity)

    put_price = np.exp(-r*time_to_maturity)*K*norm.cdf(-d2) - St*norm.
        cdf(-d1)

    put_delta = -norm.cdf(-d1)

    put_gamma = norm.pdf(d1) / (St * sigma * np.sqrt(time_to_maturity))

    return put_price, put_delta, put_gamma
```

```
def call_option_BS(K,r,sigma,St,time_to_maturity):


    d1 = (np.log(St/K) + (r + 0.5*(sigma)**2)*time_to_maturity)/(sigma*
        np.sqrt(time_to_maturity))
    d2 = d1 - sigma*np.sqrt(time_to_maturity)

    call_price = St*norm.cdf(d1) - np.exp(-r*time_to_maturity)*K*norm.
        cdf(d2)

    call_delta = norm.cdf(d1)

    call_gamma = norm.pdf(d1) / (St * sigma * np.sqrt(time_to_maturity))

    return call_price, call_delta, call_gamma


def call_option_hedge(K,K2,T,T2,S0,mu,sigma,B0,r,step,num_puts,M):

    # Calculate the number of steps using maturity and step size
    num_steps = int(T/step)

    # Initializing our variables
    psi = np.zeros((M, num_steps + 1))
    psi[:, -1] = np.nan
    num_calls = np.zeros((M, num_steps + 1))
    S_pos = np.zeros((M, num_steps + 1))
    price_puts = np.zeros((M, num_steps + 1))
    price_calls = np.zeros((M, num_steps + 1))
    S = np.zeros((M, num_steps + 1))
    S[:, 0] = S0
    B = np.zeros((M, num_steps + 1))
    B[:, 0] = B0
    portfolio_value = np.zeros((M, num_steps + 1))
    time = np.linspace(0,T,num_steps + 1)
    delta = np.zeros((M, num_steps + 1))
    gamma = np.zeros((M, num_steps + 1))

    # Determining the initial positions
    put_price_initial = put_option_BS(K,r,sigma,S0,T)[0]
    call_price_initial = call_option_BS(K2,r,sigma,S0,T2)[0]
    price_puts[:, 0] = num_puts*put_price_initial

    num_calls[:, 0] = -(num_puts*put_option_BS(K,r,sigma,S0,T)[2])/
        call_option_BS(K2,r,sigma,S0,T2)[2] # Gamma neutral
    price_calls[:, 0] = num_calls[:, 0]*call_price_initial
```

```
S_pos[:, 0] = - num_puts*put_option_BS(K, r, sigma, S0, T)[1] -
    num_calls[:, 0]*call_option_BS(K2, r, sigma, S0, T2)[1] # Delta
    neutral
print('-----Position in S at time 0:'+ str(S_pos[0,0]))
psi[:, 0] = - (price_puts[:, 0] + S_pos[:, 0]*S[:, 0] + price_calls
    [:, 0])/B[:, 0]
print('-----Position in calls at time 0:'+ str(num_calls[0,0]))
print('-----Position in B at time 0:'+ str(psi[0,0]))
portfolio_value[:, 0] = price_puts[:, 0] + S_pos[:, 0]*S[:, 0] +
    num_calls[:, 0]*call_price_initial + psi[:, 0]*B[:, 0] # This
    should be 0 by construction

# Iterating over the time grid:
for t in range(1, num_steps + 1):

    # New asset prices:
    B[:, t] = B[:, t-1]*np.exp(r*step)
    S[:, t] = S[:, t-1] * (1 + np.expm1((mu - 0.5 * sigma**2) * step
        + sigma * np.sqrt(step) * np.random.normal(size=M)))
    # expm1 used as it is more accurate for values near zero

    # Current portfolio value before repositioning:
    value = S_pos[:, t-1]*S[:, t] + num_calls[:, t-1]*call_option_BS
        (K2, r, sigma, S[:, t], time_to_maturity=T2-time[t])[0] +
        psi[:, t-1]*B[:, t]

    # New value puts:
    if time[t] == T:

        price_puts[:, t] = num_puts*np.maximum(K - S[:, t], 0)
        # We don't need to consider the realisation of the call
            options because we only look on the timeline from 0 to T
            = 2,
        # while the call option matures at T2 = 5
        portfolio_value[:, t] = price_puts[:, t] + value

        # Calculating mean and standard deviation of the total
            portfolio value at maturity
        portfolio_mean = np.mean(portfolio_value[:, t])
        portfolio_std_dev = np.std(portfolio_value[:, t])
        print("-----Mean of total portfolio value at maturity:" +
            str(portfolio_mean))
        print("-----Standard deviation of total portfolio value at
            maturity:"+ str(portfolio_std_dev))

        break
    price_puts[:, t] = num_puts*put_option_BS(K,r,sigma,S[:, t],
        time_to_maturity=T-time[t])[0]
```

```
        # Determining the gamma-neutral position:
        num_calls[:, t] = -(num_puts*put_option_BS(K,r,sigma,S[:, t],
            time_to_maturity=T-time[t])[2])/call_option_BS(K2,r,sigma,S
            [:, t],time_to_maturity=T2-time[t])[2]
        price_calls[:, t] = num_calls[:, t]*call_option_BS(K2, r, sigma,
             S[:, t], time_to_maturity=T2-time[t])[0]

        # Determining the delta-neutral postion:
        S_pos[:, t] = - num_puts*put_option_BS(K,r,sigma,S[:, t],
            time_to_maturity=T-time[t])[1] - num_calls[:, t]*
            call_option_BS(K2,r,sigma,S[:, t],time_to_maturity=T2-time[t
            ])[1]

        # Determining position in B such that there is no net cashflow
        psi[:, t] = (value - S_pos[:, t]*S[:, t] - price_calls[:, t])/B
            [:, t]



        # Updating portfolio value:
        portfolio_value[:, t] = price_puts[:, t] + S_pos[:, t]*S[:, t] +
             price_calls[:, t] + psi[:, t]*B[:, t]

        # Calculating the delta:
        delta[:, t] = num_puts*put_option_BS(K,r,sigma,S[:, t],
            time_to_maturity=T-time[t])[1] + S_pos[:, t] + num_calls[:,
            t]*call_option_BS(K2, r, sigma, S[:, t], time_to_maturity=T2
             - time[t])[1]

        # Calculating gamma:
        gamma[:, t] = num_puts*put_option_BS(K,r,sigma,S[:, t],
            time_to_maturity=T-time[t])[2] + num_calls[:, t]*
            call_option_BS(K2, r, sigma, S[:, t], time_to_maturity=T2 -
            time[t])[2]


    return time, S, B, S_pos, num_calls, psi, price_puts, price_calls,
        portfolio_value, delta, gamma, portfolio_mean, portfolio_std_dev


time, S, B, S_pos, num_calls, psi, price_puts, price_calls,
    portfolio_value, delta, gamma, portfolio_mean, portfolio_std_dev =
    call_option_hedge(K,K2,T,T2,S0,mu,sigma,B0,r,step,num_puts,M)
```

```
df = pd.DataFrame(data = np.array([time, S[0,:], B[0,:], S_pos[0,:],
    num_calls[0,:], psi[0,:], price_puts[0,:], price_calls[0,:],
    portfolio_value[0,:]]).T, columns=["t", "S", "B",
                                "position S", "position calls", "
                                    position B", "num_puts * put_price",
                                     "num_calls * call_price", "
                                    total_portfolio_value"])
fig, ax = plt.subplots(2, 4, figsize=(25, 10))
df.plot(x="t", y="B", title="path B", ax=ax[0, 0])
df.plot(x="t", y="S", title="path S (red=strike puts)", ax=ax[0, 1])
ax[0, 1].axhline(y=K, color="r")
df.plot(x="t", y="num_puts * put_price", title=f"value {num_puts} puts",
     ax=ax[0, 2])
df.plot(x="t", y="num_calls * call_price", title="value of calls", ax=ax
    [0, 3])
df.iloc[:-1].plot(x="t", y="position B", title="path psi (position B)",
    ax=ax[1, 0])
df.iloc[:-1].plot(x="t", y="position S", title="path position S", ax=ax
    [1, 1])
df.iloc[:-1].plot(x="t", y="position calls", title="path position calls"
    , ax=ax[1, 2])
df.plot(x="t", y="total_portfolio_value", title="mismatch", ax=ax[1, 3])



quantiles = [5, 50, 95]

def quants(quantiles, time, S, B, S_pos, num_calls, psi, price_puts,
    portfolio_value, gamma):

    S_pos_quants = np.zeros((3, np.shape(S_pos)[1]))
    num_calls_quants = np.zeros((3, np.shape(num_calls)[1]))
    psi_quants = np.zeros((3, np.shape(psi)[1]))
    gamma_quants = np.zeros((3, np.shape(gamma)[1]))

    for t in range(0, len(time)):

        S_pos_quants[:, t] = np.percentile(S_pos[:, t], quantiles)
        num_calls_quants[:, t] = np.percentile(num_calls[:, t],
            quantiles)
        psi_quants[:, t] = np.percentile(psi[:, t], quantiles)
        gamma_quants[:, t] = np.percentile(gamma[:, t], quantiles)

    return S_pos_quants, num_calls_quants, psi_quants, gamma_quants

S_pos_quants, num_calls_quants, psi_quants, gamma_quants = quants(
    quantiles, time, S, B, S_pos, num_calls, psi, price_puts,
    portfolio_value, gamma)
```

```
df = pd.DataFrame(data = np.array([time, S_pos_quants[0,:], S_pos_quants
    [1,:], S_pos_quants[2,:],
                                   num_calls_quants[0,:],
                                       num_calls_quants[1,:],
                                       num_calls_quants[2,:],
                                   psi_quants[0,:], psi_quants[1,:],
                                       psi_quants[2,:], gamma_quants
                                       [0,:],
                                   gamma_quants[1,:], gamma_quants
                                       [2,:]]).T,
                      columns=["t", "position-S-5th-quantile", "position-S-
                          50th-quantile", "position-S-95th-quantile",
                                "position-calls-5th-quantile", "position-
                                    calls-50th-quantile", "position-calls-95
                                    th-quantile",
                                "position-B-5th-quantile", "position-B-50th-
                                    quantile", "position-B-95th-quantile",
                                "gamma-5th-quantile", "gamma-50th-quantile",
                                    "gamma-95th-quantile"])
fig, ax = plt.subplots(4,1, figsize=(25,10))

for i in range(0,3):
    ax[0].plot(time, S_pos_quants[i,:], label=f'{quantiles[i]}th-
        quantile')

ax[0].set_title('Position-in-S-quantiles')
ax[0].legend()


for i in range(0,3):
    ax[1].plot(time, num_calls_quants[i,:], label=f'{quantiles[i]}th-
        quantile')

ax[1].set_title('Position-in-calls-quantiles')
ax[1].legend()

for i in range(0,3):
    ax[2].plot(time, psi_quants[i,:], label=f'{quantiles[i]}5th-quantile
        ')

ax[2].set_title('Position-in-B-quantiles')
ax[2].legend()

for i in range(0,3):
```

```
    ax[3].plot(time, gamma_quants[i,:], label=f'{quantiles[i]}th-
        quantile')

ax[3].set_title('Gamma-quantiles')
ax[3].legend()


plt.tight_layout()
plt.show()



plt.hist(portfolio_value[:, 200-1])
print(f"-----Mean-price-of-the-put-option-at-just-before-maturity:-{(np.
    mean(price_puts[:,-200-1])/-1000)*np.exp(-r*T)}")
```