Assignment Quantitative Finance 2023-2024

Quantitative Finance (35V5A1-B-6)
Group 1

Andrei Agapie
Nikodem Baehr
Samuel Friedlaender

Department of Econometrics and Operations Research
Tilburg University
The Netherlands
December 2023

1 Question 1

(a) In order to estimate mean daily returns from the historical data, we've first created the daily returns by subtracting from each day's closing price its open price. Then we've computed the mean over all daily returns and obtained mean returns. As for standard deviation, we've used Numpy's `.std` function on the already generated daily returns. The obtained results for mean daily returns and standard deviation were '-0.0002857' and '0.0066153', respectively.

(b) We've implemented the Black-Scholes formula for the price of a put option with the specified strike price as a function of the appropriate inputs. We've created a function called `monte_carlo_option_price`, which uses random picks from a standard normal distribution where the assumed Brownian motion occurs. Having applied the function we got a confidence interval for the option price to be (91.48749842600122, 91.92494287138965).

(c) We've used a simulation approach called bump and reprice in order to derive the delta of the option, that is the effect of a change in price of the underlying stock on the price of the option. For this approach, we specified the initial price of the stock, the time frame, interest rate, the strike price and structure of the option, as well as the desired size of the "bump" increment. After we implement the code in a similar way to the pseudocode presented in the course, the estimated delta is '3096.89' for the common simulations, with a confidence interval of '+/- 0.9866778843758122', while the delta for the non-common numbers is '3096.64' with a confidence interval of '+/- 8.314627703414978e-15'. The very small, almost negligible confidence interval observed in the common numbers approach results from the use of the same randomly generated numbers for both the original and bumped price.

2 Question 2

(a) We assume Black-Scholes market, where $\mu = 10\%$, $\sigma = 25\%$, $S_0 = 100$, $B_0 = 1$, and $r = 3\%$. Denote the Delta of an European put option by Δ_{put}. Let the strike price be denoted by K being equal to 100 and the maturity by T being equal to 2. We have the following equations from the slides:

$$\Delta_{\text{put}} = -\Phi(-d_1) \tag{1}$$

$$d_1 = \frac{\ln\left(\frac{S_0}{K}\right) + \left(r + \frac{1}{2}\sigma^2\right)(T-t)}{\sigma\sqrt{T-t}} \tag{2}$$

Therefore, we can plug the values that are given onto those equations and calculate the Delta of an European options with K = 100 and T = 2. Δ_{put} is equal to approximately -0.3645.

For the Bump and reprice approximation of the Delta for the European put option, we consider in code (found in Appendix) only the scenario of common random numbers as we can take h as small as we want and it will not affect the varience. The estimate we got is -0.3671.

For the pathwise approximation we got estimate -0.3650. For the Likelihood Ratio Method our estimate of Δ_{put} is -0.3764.

(b) The exact gamma value of the European put option is $\Gamma_{put} = 0.0106$. We use the Gamma approximation under bump and reprice using the equation in the lecture slides for the second order Greeks the approximation estimate is 0.0152. Since the second order derivative have kinks and its not continuous the pathwise approximation is not possible. We obtain the Likelihood Ratio method approximation by getting a second order partial derivative of the score function with respect to S_0. The LRM approximation estimate of Gamma is 0.0123.

(c) We use vectorization using NumPy to obtain 2500 simulations without computationally expensive "for loops".

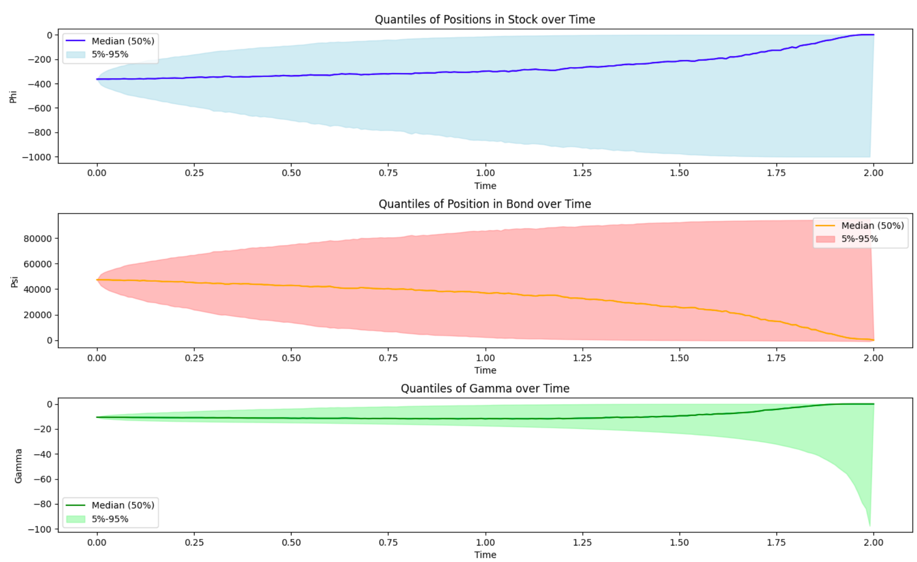

Figure 1: Plots corresponding to the simulations of delta hedging

Mean portfolio value at maturity T: -13.373344852820102
Standard deviation of portfolio value at maturity T: 866.612632692398

To answer the question What price would we charge for the option we have to first derive the price we would charge for 1000 options which corresponds to the discounted value of the portfolio at maturity. However, if we would

only take the mean value of the portfolio, we would be risking that the value of the portfolio will end up below the price of the option and we are at loss. Thus we should adjust for at least 1 standard deviation to reduce the risk of loss.
Price we would charge for the put option: -0.8287395837035515

(di) We implement the gamma-delta hedging, with first determining the position in call option to be gamma neutral, then position in stock to be delta neutral and finally the position in Bonds to have cash flows only at beginning at then at T. Since we are doing gamma hedging, we are basically setting gamma equal to zero. That is why in this exercise we instead plot the position we are going to hold in the call option.

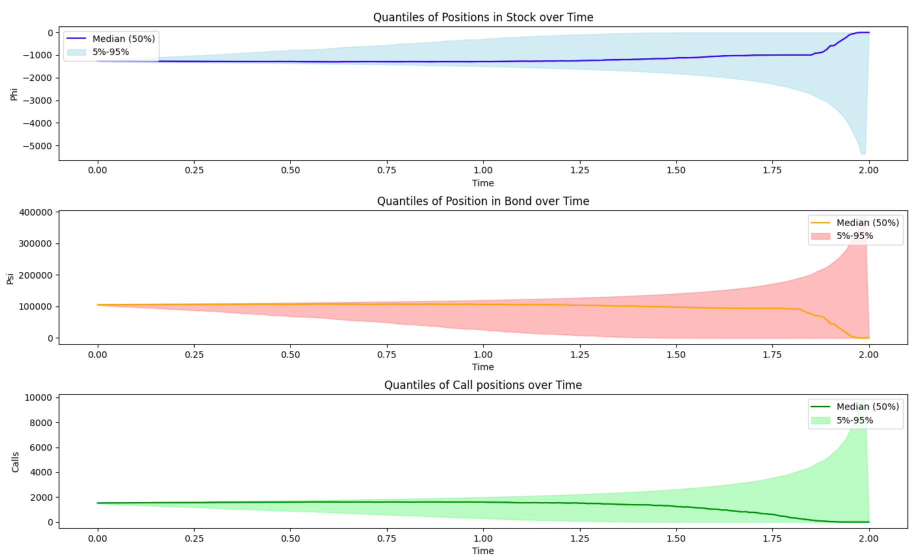

Figure 2: Plots corresponding to simulations of delta-gamma hedging

Mean portfolio value at maturity T: 0.5598944990822537
Standard deviation of portfolio value at maturity T: 128.7319970245847
Put option price: -0.12070794035344233

(dii) The fact that there is a second stock traded; however is not sufficient to be able to do delta-gamma hedging. The problem is that we are considering only put option (no call options like in Exercise 2-di). Moreover the position in put option is given by the exercise and cannot be changed and we have calculated that given this position in call option the gamma of the portfolio is (as seen in the plot for part c) between approximately -10. Since there is no other option we cannot do the gamma hedging to set gamma equal to 0. So it is not possible to do the simulations of delta-gamma hedging with just one option with given position and 2 stocks.

3 Appendix

```python
import pandas as pd
import numpy as np

# Load historical data into a DataFrame, skipping the first three lines
file_path = "/Users/and/Downloads/AEX-INDEX_historical_price.txt"
data = pd.read_csv(file_path, sep=';',skiprows=4)
data['daily returns'] = (data.iloc[:, 1] - data.iloc[:, 5]) / data.iloc[:, 5]
avg_return = np.mean(data['daily returns'])
std_dev = np.std(data['daily returns'])


# Function to calculate the Black-Scholes option price
def black_scholes(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)
    call_price = S * norm.cdf(d1) - K * np.exp(-r * T) * norm.cdf(d2)
    return call_price

# Monte Carlo simulation for option pricing
def monte_carlo_option_price(S, K, T, r, sigma, num_simulations):
    np.random.seed(42)
    dt = T / 252  # Assuming 252 trading days in a year
    simulated_prices = np.zeros(num_simulations)

    for i in range(num_simulations):
        # Generate random Brownian motion increments
        dW = np.random.normal(0, np.sqrt(dt), int(T / dt))

        # Calculate the simulated price path using geometric Brownian motion
        price_path = S * np.exp(np.cumsum((r - 0.5 * sigma**2) * dt + sigma * dW))

        # Calculate the option payoff at expiration
        option_payoff = np.maximum(price_path[-1] - K, 0)

        # Discount the option payoff back to present value
        simulated_prices[i] = option_payoff * np.exp(-r * T)

    # Calculate the mean and standard deviation of the simulated option prices
    mean_price = np.mean(simulated_prices)
    std_dev = np.std(simulated_prices)

    # Calculate the confidence interval (e.g., 95% confidence interval)
```

```python
    confidence_interval = (mean_price - 1.96 * std_dev / np.sqrt(num_simulations),
                           mean_price + 1.96 * std_dev / np.sqrt(num_simulations))

    return mean_price, confidence_interval

# Parameters
S0 = 761.37  # Current stock price of AEX index
K = 740.0  # Strike price of the option
T = 5.0  # Time to expiration in years
r = 0.02  # Risk-free interest rate
sigma = std_dev  # Volatility

# Number of Monte Carlo simulations
num_simulations = 10000

# Obtain option price and confidence interval
option_price, confidence_interval = monte_carlo_option_price(S0, K, T, r, sigma, num_simulations)



print(f"Monte Carlo Estimated Option Price: {option_price:.4f}")
print(f"95% Confidence Interval: {confidence_interval}")
print(data)
print(avg_return)
print(std_dev)


n_sims = 1000000
h = n_sims**(-0.25)
import numpy as np

# Function to calculate delta using bump-and-reprice method
def calculate_delta(S0, K, r, sigma, T, n_sims, h):
    # Independent Simulations
    dW = np.sqrt(T) * np.random.normal(size=n_sims)
    S_T = S0 * np.exp((r - 0.5 * sigma**2) * T + sigma * dW)
    dW_bump = np.sqrt(T) * np.random.normal(size=n_sims)
    S_T_bump = S0 * np.exp(((r + h) - 0.5 * sigma**2) * T + sigma * dW_bump)

    discounted_payoff = np.exp(-r * T) * np.maximum(S_T - K, 0)
    discounted_payoff_bump = np.exp(-(r + h) * T) * np.maximum(S_T_bump - K, 0)

    E_call = np.mean(discounted_payoff)
    E_call_bump = np.mean(discounted_payoff_bump)

    delta_indep_mean = (E_call_bump - E_call) / h
    delta_indep_std = (np.std((discounted_payoff_bump - discounted_payoff) / h) / np.sqrt(n_sims))

    print(f'Delta (independent simulations) with 95% CI: {delta_indep_mean:.2f} +/- {1.96 * delta_indep_std:.2f}')

```

```python
    # Common Simulations
    dW = np.sqrt(T) * np.random.normal(size=n_sims)
    S_T = S0 * np.exp((r - 0.5 * sigma**2) * T + sigma * dW)
    S_T_bump = S0 * np.exp(((r + h) - 0.5 * sigma**2) * T + sigma * dW)

    discounted_payoff = np.exp(-r * T) * np.maximum(S_T - K, 0)
    discounted_payoff_bump = np.exp(-(r + h) * T) * np.maximum(S_T_bump - K, 0)

    E_call = np.mean(discounted_payoff)
    E_call_bump = np.mean(discounted_payoff_bump)

    delta_com_mean = (E_call_bump - E_call) / h
    delta_com_std = (np.std((discounted_payoff_bump - discounted_payoff) / h) / np.sqrt(n_sims))

    print(f'Delta (common simulations) with 95% CI: {delta_com_mean:.2f} +/- {1.96 * delta_com_std:.2f}')

# Define parameters


# Call the function to calculate delta
calculate_delta(S0, K, r, sigma, T, n_sims, h)


#Exercise 2
import pandas as pd
import numpy as np
from scipy.stats import norm
# Parameters
mu = 0.1
sigma = 0.25
S_0 = 100
B_0 = 1
r = 0.03
K = 100
T = 2



#a)
# Number of Monte Carlo simulations
num_simulations = 10000
h = 0.000000001

d1 = (np.log(S_0 / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
delta_exact = -norm.cdf(-d1)
print("Exact delta: " + str(delta_exact))

def common_rand_bump(S_0, K, r, sigma, T, num_simulations, h):
    dW = np.sqrt(T) * np.random.normal(size=num_simulations)
    S_T = S_0 * np.exp((r - 0.5 * sigma**2) * T + sigma * dW)
    S_T_bump = (S_0+h) * np.exp((r - 0.5 * sigma**2) * T + sigma * dW)
```

```python

    discounted_payoff = np.exp(-r * T) * np.maximum(K - S_T, 0)
    discounted_payoff_bump = np.exp(-r * T) * np.maximum(K - (
        S_T_bump), 0)

    E_call = np.mean(discounted_payoff)
    E_call_bump = np.mean(discounted_payoff_bump)

    delta_com_mean = (E_call_bump - E_call) / h
    print("Bump and reprice delta: " + str(delta_com_mean))

common_rand_bump(S_0, K, r, sigma, T, num_simulations, h)

def pathwise(S_0, K, r, sigma, T, num_simulations):
    dW = np.sqrt(T) * np.random.normal(size=num_simulations)
    S_T = S_0 * np.exp((r - 0.5 * sigma**2) * T + sigma * dW)
    indicator_function = lambda K, S_T: np.where(K > S_T, 1, 0)
    f = - np.exp(-r * T) * indicator_function(K,S_T) * np.exp((r -
        0.5 * sigma**2) * T + sigma * dW)
    print("Patwise approximation delta: " + str(np.mean(f)))


pathwise(S_0, K, r, sigma, T, num_simulations)

def LRM(S_0, K, r, sigma, T, num_simulations):
    dW = np.sqrt(T) * np.random.normal(size=num_simulations)
    S_T = S_0 * np.exp((r - 0.5 * sigma**2) * T + sigma * dW)
    score_func = (np.log(S_T/S_0) - (r-0.5*sigma**2)*T)/(sigma ** 2
        * T * S_0)
    approx = np.maximum(K - S_T, 0) * score_func
    discounted_mean = np.exp(-r * T) * np.mean(approx)
    print("LRM approximated delta " + str(discounted_mean))

LRM(S_0, K, r, sigma, T, num_simulations)

#b)
h = num_simulations ** (-0.25)
# Calculate gamma
gamma_exact = 1 / (S_0 * sigma * np.sqrt(T)) * norm.pdf(d1)

print("Exact gamma: " + str(gamma_exact))
def gamma_bump_approx(S_0, K, r, sigma, T, num_simulations, h):
    dW = np.sqrt(T) * np.random.normal(size=num_simulations)
    S_T = S_0 * np.exp((r - 0.5 * sigma**2) * T + sigma * dW)
    S_T_bump = (S_0+h) * np.exp((r - 0.5 * sigma**2) * T + sigma *
        dW)
    S_T_minusbump = (S_0-h) * np.exp((r - 0.5 * sigma**2) * T +
        sigma * dW)

    discounted_payoff = np.exp(-r * T) * np.maximum(K - S_T, 0)
    discounted_payoff_bump = np.exp(-r * T) * np.maximum(K - (
        S_T_bump), 0)
    discounted_payoff_minusbump = np.exp(-r * T) * np.maximum(K - (
        S_T_minusbump), 0)

    f_theta = np.mean(discounted_payoff)
    f_theta_bump = np.mean(discounted_payoff_bump)
```

```
    f_theta_minusbump = np.mean(discounted_payoff_minusbump)

    gamma_com_mean = (f_theta_bump - 2 * f_theta + f_theta_minusbump) / h ** 2
    print("Bump and reprice gamma: " + str(gamma_com_mean))

gamma_bump_approx(S_0, K, r, sigma, T, num_simulations, h)

#Pathwise not possible

def gamma_LRM_approx(S_0, K, r, sigma, T, num_simulations):
    dW = np.sqrt(T) * np.random.normal(size=num_simulations)
    S_T = S_0 * np.exp((r - 0.5 * sigma**2) * T + sigma * dW)
    score_func = (1 - np.log(S_T/S_0)+(r-0.5*sigma**2)*T)/(sigma**2*T*S_0**2)
    approx = np.maximum(K - S_T, 0) * score_func
    discounted_mean = np.exp(-r * T) * np.mean(approx)
    print("LRM approximated gamma " + str(discounted_mean))

gamma_LRM_approx(S_0, K, r, sigma, T, num_simulations)

#c)
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
import matplotlib.pyplot as plt
# Parameters
mu = 0.1
sigma = 0.25
S_0 = 100
B_0 = 1
r = 0.03
K = 100
T = 2
num_simulations = int(2500)
delta_t = 0.01
num_puts=-1000

def get_delta(S_c, T):
    d1 = (np.log(S_c / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    delta_exact = -norm.cdf(-d1)
    return(delta_exact)

def get_put_price(S_c,T):
    d1 = (np.log(S_c / 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_c*norm.cdf(-d1)
    return(put_price)

def get_gamma(S_c,T):
    d1 = (np.log(S_c / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    gamma_exact = norm.pdf(d1) / (S_c * sigma * np.sqrt(T))
    return gamma_exact

```

```python
def simulate_function_vectorized(K, T, S_0, mu, sigma, B_0, r, delta_t, num_puts, num_simulations):
    num_time_steps_total = int(T/delta_t)

    # Initialize arrays to store results for each simulation
    all_times = np.linspace(0, T, num_time_steps_total + 1)
    all_S = np.zeros((num_time_steps_total + 1, num_simulations))
    all_B = np.zeros((num_time_steps_total + 1, num_simulations))
    all_phi = np.zeros((num_time_steps_total + 1, num_simulations))
    all_psi = np.zeros((num_time_steps_total + 1, num_simulations))
    all_price_puts =np.zeros((num_time_steps_total + 1, num_simulations))
    all_total_portfolio_value = np.zeros((num_time_steps_total + 1, num_simulations))
    all_gamma = np.zeros((num_time_steps_total + 1, num_simulations))

    # Determine initial positions
    put_price_initial = get_put_price(S_c=S_0, T=T)
    price_puts_initial = num_puts * put_price_initial
    phi_initial = -num_puts * get_delta(S_c=S_0, T=T)
    psi_initial = -(price_puts_initial + phi_initial * S_0) / B_0
    total_portfolio_value_initial = price_puts_initial + phi_initial * S_0 + psi_initial * B_0
    gamma_initial = num_puts * get_gamma(S_c=S_0, T=T)

    # Assign initial values to arrays
    all_S[0, :] = S_0
    all_B[0, :] = B_0
    all_phi[0, :] = phi_initial
    all_psi[0, :] = psi_initial
    all_price_puts[0, :] = price_puts_initial
    all_total_portfolio_value[0, :] = total_portfolio_value_initial
    all_gamma[0, :] = gamma_initial

    # Iterate over discrete-time grid
    for k in range(1, num_time_steps_total + 1):
        # New asset prices
        all_B[k, :] = all_B[k-1, :] * np.exp(r * delta_t)
        all_S[k, :] = all_S[k-1, :] * np.exp((mu - 0.5 * sigma ** 2) * delta_t + sigma * np.sqrt(delta_t) * norm.rvs(size=num_simulations))

        # Current value of (S,B) portfolio from previous point-in-time (below we will rebalance)
        value = all_phi[k-1, :] * all_S[k, :] + all_psi[k-1, :] * all_B[k,:]

        # New value puts
        if all_times[k] == T:
            all_price_puts[k, :] = num_puts * np.maximum(K - all_S[k, :], 0)
            all_total_portfolio_value[k, :] = all_price_puts[k, :] + value
            break

```

```
        all_price_puts[k, :] = num_puts * get_put_price(S_c=all_S[k, :], T=T - all_times[k])

        # Determine new position S for next interval (such that combination of (S, B)-portfolio and puts is delta-neutral)
        all_phi[k, :] = -num_puts * get_delta(S_c=all_S[k, :], T=T - all_times[k])

        # Determine new position B, such that there is no net cashflow in (S, B)-portfolio
        all_psi[k, :] = (value - all_phi[k, :] * all_S[k, :]) / all_B[k, :]

        # Mismatch between discrete-time delta-neutral, self-financing portfolio, and price puts
        all_total_portfolio_value[k, :] = all_price_puts[k, :] + all_phi[k, :] * all_S[k, :] + all_psi[k, :] * all_B[k, :]

        # Calculate gamma
        all_gamma[k, :] = num_puts * get_gamma(S_c=all_S[k, :], T=T - all_times[k])

    return all_times, all_S, all_B, all_phi, all_psi, all_price_puts, all_total_portfolio_value, all_gamma

# Perform simulations without using a for loop
all_times, all_S, all_B, all_phi, all_psi, all_price_puts, portfolio_value, all_gamma = simulate_function_vectorized(
    K, T, S_0, mu, sigma, B_0, r, delta_t, num_puts, num_simulations
)

mean_portfolio_at_T = np.mean(portfolio_value[-1, :])
std_portfolio_at_T = np.std(portfolio_value[-1,:])

# Print the results
print(f"Mean portfolio value at maturity T: {mean_portfolio_at_T}")
print(f"Standard deviation of portfolio value at maturity T: {std_portfolio_at_T}")

print("Our option price: ", np.exp(-r*T)*(mean_portfolio_at_T-std_portfolio_at_T)/1000)
# Calculate quantiles
quantiles_phi = np.percentile(all_phi, [5, 50, 95], axis=1)
quantiles_psi = np.percentile(all_psi, [5, 50, 95], axis=1)
quantiles_gamma = np.percentile(all_gamma, [5, 50, 95], axis=1)

# Plot quantiles for phi, psi, and gamma
plt.figure(figsize=(12, 9))

# Plot quantiles for phi
plt.subplot(3, 1, 1)
plt.plot(all_times, quantiles_phi[1], label='Median (50%)', color='blue')
```

```
plt.fill_between(all_times, quantiles_phi[0], quantiles_phi[2], color='lightblue', alpha=0.5, label='5%-95%')
plt.title('Quantiles of Positions in Stock over Time')
plt.xlabel('Time')
plt.ylabel('Phi')
plt.legend()

# Plot quantiles for psi
plt.subplot(3, 1, 2)
plt.plot(all_times, quantiles_psi[1], label='Median (50%)', color='orange')
plt.fill_between(all_times, quantiles_psi[0], quantiles_psi[2], color='lightcoral', alpha=0.5, label='5%-95%')
plt.title('Quantiles of Position in Bond over Time')
plt.xlabel('Time')
plt.ylabel('Psi')
plt.legend()

# Plot quantiles for gamma
plt.subplot(3, 1, 3)
plt.plot(all_times, quantiles_gamma[1], label='Median (50%)', color='green')
plt.fill_between(all_times, quantiles_gamma[0], quantiles_gamma[2], color='lightgreen', alpha=0.5, label='5%-95%')
plt.title('Quantiles of Gamma over Time')
plt.xlabel('Time')
plt.ylabel('Gamma')
plt.legend()

plt.tight_layout()
plt.show()




import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
import matplotlib.pyplot as plt
# Parameters
mu = 0.1
sigma = 0.25
S_0 = 100
B_0 = 1
r = 0.03
K_put = 100
T = 2
num_simulations = 2500
delta_t = 0.01
num_puts=-1000
num_time_steps_per_unit_of_time = 100

def get_delta_put(S_c, T):
    d1 = (np.log(S_c / K_put) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    delta_exact = -norm.cdf(-d1)
    return(delta_exact)
```

```python

def get_put_price(S_c, T):
    d1 = (np.log(S_c / K_put) + (r + 0.5 * sigma ** 2) * T) / (
        sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    put_price = K_put * np.exp(-r*T)*norm.cdf(-d2)-S_c*norm.cdf(-d1
        )
    return(put_price)

def get_gamma(S_c, T, K):
    d1 = (np.log(S_c / K) + (r + 0.5 * sigma ** 2) * T) / (sigma *
        np.sqrt(T))
    gamma_exact = 1 / (S_c * sigma * np.sqrt(T)) * norm.pdf(d1)
    return gamma_exact

def get_delta_call(S_c, T):
    d1 = (np.log(S_c / K_call) + (r + 0.5 * sigma ** 2) * T) / (
        sigma * np.sqrt(T))
    delta_exact = norm.cdf(d1)
    return(delta_exact)

def get_call_price(S_c, T):
    d1 = (np.log(S_c / K_call) + (r + 0.5 * sigma ** 2) * T) / (
        sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    call_price = S_c*norm.cdf(d1) - K_call * np.exp(-r*T)*norm.cdf(
        d2)
    return(call_price)

K_call = 120

def simulate_function_vectorized(T, S_0, mu, sigma, B_0, r,
    num_puts, num_simulations):
    num_time_steps_total = int(T/delta_t)

    # Initialize arrays to store results for each simulation
    all_times = np.linspace(0, T, num_time_steps_total + 1)
    all_S = np.zeros((num_time_steps_total + 1, num_simulations))
    all_B = np.zeros((num_time_steps_total + 1, num_simulations))
    all_phi = np.zeros((num_time_steps_total + 1, num_simulations))
    all_psi = np.zeros((num_time_steps_total + 1, num_simulations))
    all_price_puts =np.zeros((num_time_steps_total + 1,
        num_simulations))
    all_total_portfolio_value = np.zeros((num_time_steps_total + 1,
         num_simulations))
    all_callposition = np.zeros((num_time_steps_total + 1,
        num_simulations))
    all_price_calls = np.zeros((num_time_steps_total + 1,
        num_simulations))

    # Determine initial positions
    put_price_initial = get_put_price(S_c=S_0, T=T)
    call_price_initial = get_call_price(S_c=S_0,T= T+3)
    price_puts_initial = num_puts * put_price_initial
    call_position_initial = (- num_puts * get_gamma(S_c=S_0, T=T, K
         = K_put))/ get_gamma(S_c=S_0, T=T+3, K = K_call)
    call_prices = call_position_initial * call_price_initial
```

```
    phi_initial = -num_puts * get_delta_put(S_c=S_0, T=T) - call_position_initial * get_delta_call(S_c=S_0, T=T+3)
    psi_initial = -(price_puts_initial + call_prices + phi_initial * S_0) / B_0
    total_portfolio_value_initial = price_puts_initial + call_prices + phi_initial * S_0 + psi_initial * B_0

    # Assign initial values to arrays
    all_S[0, :] = S_0
    all_B[0, :] = B_0
    all_phi[0, :] = phi_initial
    all_psi[0, :] = psi_initial
    all_price_puts[0, :] = price_puts_initial
    all_total_portfolio_value[0, :] = total_portfolio_value_initial
    all_callposition[0, :] = call_position_initial
    all_price_calls[0, :] = call_prices

    # Iterate over discrete-time grid
    for k in range(1, num_time_steps_total + 1):
        # New asset prices
        all_B[k, :] = all_B[k-1, :] * np.exp(r * delta_t)
        all_S[k, :] = all_S[k-1, :] * np.exp((mu - 0.5 * sigma ** 2) * delta_t + sigma * np.sqrt(delta_t) * norm.rvs(size=num_simulations))

        # Current value of (S,B, call) portfolio from previous point-in-time (below we will rebalance)
        value = all_phi[k-1, :] * all_S[k, :] + all_psi[k-1, :] * all_B[k, :] + all_price_calls[k, :] + all_callposition[k-1, :] * get_call_price(S_c=all_S[k, :], T=T+ 3 - all_times[k])

        # New value puts
        if all_times[k] == T:
            all_price_puts[k, :] = num_puts * np.maximum(K_put - all_S[k, :], 0)
            all_total_portfolio_value[k, :] = all_price_puts[k, :] + value
            break

        all_price_puts[k, :] = num_puts * get_put_price(S_c=all_S[k, :], T=T - all_times[k])

        # New position in call option such that portfolio is gamma neutral
        all_callposition[k, :] = (- num_puts * get_gamma(S_c=all_S[k, :], T=T - all_times[k], K = K_put))/ get_gamma(S_c=all_S[k, :], T=T+ 3 - all_times[k], K = K_call)
        # Determine new position S for next interval (such that combination of (S, B)-portfolio and puts is delta-neutral)
        all_phi[k, :] = -num_puts * get_delta_put(S_c=all_S[k, :], T=T - all_times[k]) - all_callposition[k, :] * get_delta_call(S_c=all_S[k, :], T=T+ 3 - all_times[k])

        all_price_calls[k, :] = all_callposition[k, :] * get_call_price(S_c=all_S[k, :], T=T+ 3 - all_times[k])
```

```
        # Determine new position B, such that there is no net
            cashflow in (S, B, call)-portfolio
        all_psi[k, :] = (value -  all_price_calls[k, :] - all_phi[k
            , :] * all_S[k, :]) / all_B[k, :]

        # Mismatch between discrete-time delta-neutral, self-
            financing portfolio, and price puts
        all_total_portfolio_value[k, :] = all_price_puts[k, :] +
            all_price_calls[k, :] + all_phi[k, :] * all_S[k, :] +
            all_psi[k, :] * all_B[k, :]

    return all_times, all_S, all_B, all_callposition, all_phi,
        all_psi, all_price_puts, all_price_calls,
        all_total_portfolio_value

# Perform simulations without using a for loop
all_times, all_S, all_B, all_callposition, all_phi, all_psi,
    all_price_puts, all_price_calls, all_total_portfolio_value =
    simulate_function_vectorized(
    T, S_0, mu, sigma, B_0, r, num_puts, num_simulations
)

mean_portfolio_at_T = np.mean(all_total_portfolio_value[-1, :])
std_portfolio_at_T = np.std(all_total_portfolio_value[-1, :])
# Print the results
print(f"Mean portfolio value at maturity T: {mean_portfolio_at_T}")
print(f"Standard deviation of portfolio value at maturity T: {
    std_portfolio_at_T}")

print("Put option price: ", np.exp(-r*T)*(mean_portfolio_at_T-
    std_portfolio_at_T)/1000)

# Calculate quantiles
quantiles_phi = np.percentile(all_phi, [5, 50, 95], axis=1)
quantiles_psi = np.percentile(all_psi, [5, 50, 95], axis=1)
quantiles_callpos = np.percentile(all_callposition, [5, 50, 95],
    axis=1)
# Plot quantiles for phi, psi, and call position
plt.figure(figsize=(12, 9))

# Plot quantiles for phi
plt.subplot(3, 1, 1)
plt.plot(all_times, quantiles_phi[1], label='Median (50%)', color='
    blue')
plt.fill_between(all_times, quantiles_phi[0], quantiles_phi[2],
    color='lightblue', alpha=0.5, label='5%-95%')
plt.title('Quantiles of Positions in Stock over Time')
plt.xlabel('Time')
plt.ylabel('Phi')
plt.legend()

# Plot quantiles for psi
plt.subplot(3, 1, 2)
plt.plot(all_times, quantiles_psi[1], label='Median (50%)', color='
    orange')
plt.fill_between(all_times, quantiles_psi[0], quantiles_psi[2],
    color='lightcoral', alpha=0.5, label='5%-95%')
```

```
plt.title('Quantiles of Position in Bond over Time')
plt.xlabel('Time')
plt.ylabel('Psi')
plt.legend()

# Plot quantiles for call positions
plt.subplot(3, 1, 3)
plt.plot(all_times, quantiles_callpos[1], label='Median (50%)',
    color='green')
plt.fill_between(all_times, quantiles_callpos[0], quantiles_callpos
    [2], color='lightgreen', alpha=0.5, label='5%-95%')
plt.title('Quantiles of Call positions over Time')
plt.xlabel('Time')
plt.ylabel('Calls')
plt.legend()

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