

# **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 pseudo-code 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  $\Delta_{\text{put}}$ . 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:

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

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

Therefore, we can plug the values that are given onto those equations and calculate the Delta of an European options with  $K = 100$  and  $T = 2$ .  $\Delta_{\text{put}}$  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 variance. The estimate we got is  $-0.3671$ .

For the pathwise approximation we got estimate  $-0.3650$ . For the Likelihood Ratio Method our estimate of  $\Delta_{put}$  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".

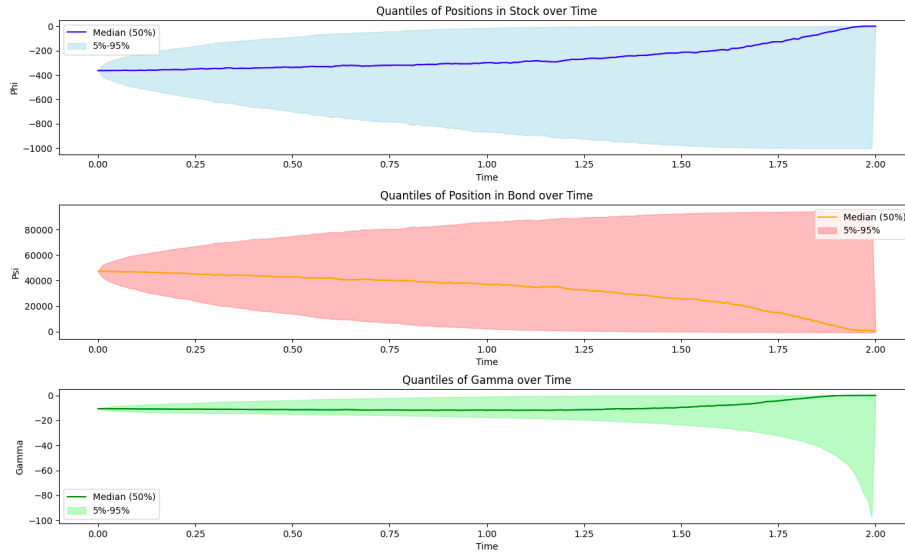


Figure 1: Plots corresponding to the simulations of delta hedging

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.

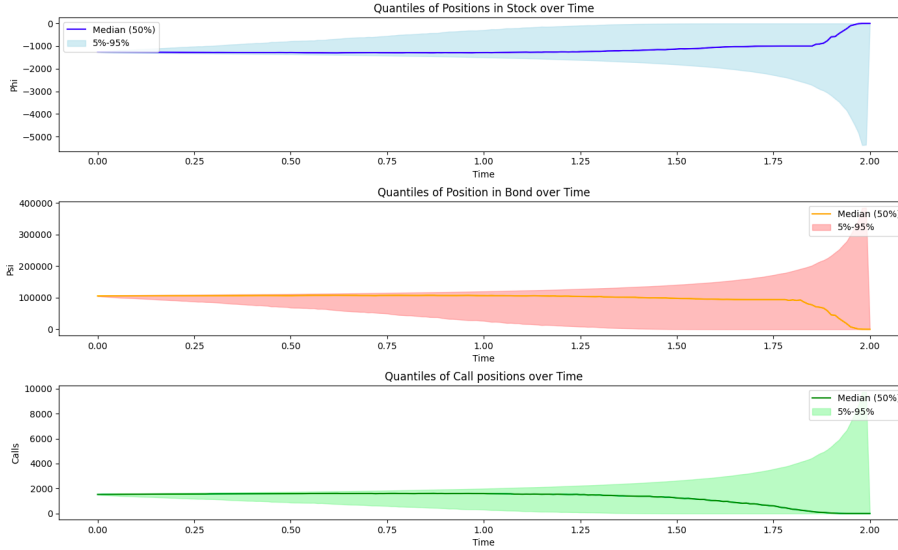


Figure 2: Plots corresponding to simulations of delta-gamma hedging

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

```

1 import pandas as pd
2 import numpy as np
3
4 # Load historical data into a DataFrame, skipping the first three
  lines
5 file_path = "/Users/and/Downloads/AEX-INDEX_historical_price.txt"
6 data = pd.read_csv(file_path, sep=';', skiprows=4)
7 data['daily returns'] = (data.iloc[:, 1] - data.iloc[:, 5]) / data.
  iloc[:, 5]
8 avg_return = np.mean(data['daily returns'])
9 std_dev = np.std(data['daily returns'])
10
11
12 # Function to calculate the Black-Scholes option price
13 def black_scholes(S, K, T, r, sigma):
14     d1 = (np.log(S / K) + (r + 0.5 * sigma**2) * T) / (sigma * np.
  sqrt(T))
15     d2 = d1 - sigma * np.sqrt(T)
16     call_price = S * norm.cdf(d1) - K * np.exp(-r * T) * norm.cdf(
  d2)
17     return call_price
18
19 # Monte Carlo simulation for option pricing
20 def monte_carlo_option_price(S, K, T, r, sigma, num_simulations):
21     np.random.seed(42)
22     dt = T / 252 # Assuming 252 trading days in a year
23     simulated_prices = np.zeros(num_simulations)
24
25     for i in range(num_simulations):
26         # Generate random Brownian motion increments
27         dW = np.random.normal(0, np.sqrt(dt), int(T / dt))
28
29         # Calculate the simulated price path using geometric
  Brownian motion
30         price_path = S * np.exp(np.cumsum((r - 0.5 * sigma**2) * dt
  + sigma * dW))
31
32         # Calculate the option payoff at expiration
33         option_payoff = np.maximum(price_path[-1] - K, 0)
34
35         # Discount the option payoff back to present value
36         simulated_prices[i] = option_payoff * np.exp(-r * T)
37
38     # Calculate the mean and standard deviation of the simulated
  option prices
39     mean_price = np.mean(simulated_prices)
40     std_dev = np.std(simulated_prices)
41
42     # Calculate the confidence interval (e.g., 95% confidence
  interval)

```

```

43     confidence_interval = (mean_price - 1.96 * std_dev / np.sqrt(
44         num_simulations),
45                             mean_price + 1.96 * std_dev / np.sqrt(
46                                 num_simulations))
47
48     return mean_price, confidence_interval
49
50 # Parameters
51 S0 = 761.37 # Current stock price of AEX index
52 K = 740.0 # Strike price of the option
53 T = 5.0 # Time to expiration in years
54 r = 0.02 # Risk-free interest rate
55 sigma = std_dev # Volatility
56
57 # Number of Monte Carlo simulations
58 num_simulations = 10000
59
60 # Obtain option price and confidence interval
61 option_price, confidence_interval = monte_carlo_option_price(S0, K,
62     T, r, sigma, num_simulations)
63
64 print(f"Monte Carlo Estimated Option Price: {option_price:.4f}")
65 print(f"95% Confidence Interval: {confidence_interval}")
66 print(data)
67 print(avg_return)
68 print(std_dev)
69
70 n_sims = 1000000
71 h = n_sims**(-0.25)
72 import numpy as np
73
74 # Function to calculate delta using bump-and-reprice method
75 def calculate_delta(S0, K, r, sigma, T, n_sims, h):
76     # Independent Simulations
77     dW = np.sqrt(T) * np.random.normal(size=n_sims)
78     S_T = S0 * np.exp((r - 0.5 * sigma**2) * T + sigma * dW)
79     dW_bump = np.sqrt(T) * np.random.normal(size=n_sims)
80     S_T_bump = S0 * np.exp(((r + h) - 0.5 * sigma**2) * T + sigma *
81         dW_bump)
82
83     discounted_payoff = np.exp(-r * T) * np.maximum(S_T - K, 0)
84     discounted_payoff_bump = np.exp(-(r + h) * T) * np.maximum(
85         S_T_bump - K, 0)
86
87     E_call = np.mean(discounted_payoff)
88     E_call_bump = np.mean(discounted_payoff_bump)
89
90     delta_indep_mean = (E_call_bump - E_call) / h
91     delta_indep_std = (np.std((discounted_payoff_bump -
92         discounted_payoff) / h) / np.sqrt(n_sims))
93
94     print(f'Delta (independent simulations) with 95% CI: {
95         delta_indep_mean:.2f} +/- {1.96 * delta_indep_std:.2f}')
96

```

```

93     # Common Simulations
94     dW = np.sqrt(T) * np.random.normal(size=n_sims)
95     S_T = S0 * np.exp((r - 0.5 * sigma**2) * T + sigma * dW)
96     S_T_bump = S0 * np.exp(((r + h) - 0.5 * sigma**2) * T + sigma *
97         dW)
98
99     discounted_payoff = np.exp(-r * T) * np.maximum(S_T - K, 0)
100     discounted_payoff_bump = np.exp(-(r + h) * T) * np.maximum(
101         S_T_bump - K, 0)
102
103     E_call = np.mean(discounted_payoff)
104     E_call_bump = np.mean(discounted_payoff_bump)
105
106     delta_com_mean = (E_call_bump - E_call) / h
107     delta_com_std = (np.std((discounted_payoff_bump -
108         discounted_payoff) / h) / np.sqrt(n_sims))
109
110     print(f'Delta (common simulations) with 95% CI: {delta_com_mean
111         :.2f} +/- {1.96 * delta_com_std:.2f}')
112
113 # Define parameters
114
115
116 # Call the function to calculate delta
117 calculate_delta(S0, K, r, sigma, T, n_sims, h)
118
119
120 #Exercise 2
121 import pandas as pd
122 import numpy as np
123 from scipy.stats import norm
124 # Parameters
125 mu = 0.1
126 sigma = 0.25
127 S_0 = 100
128 B_0 = 1
129 r = 0.03
130 K = 100
131 T = 2
132
133
134 #a)
135 # Number of Monte Carlo simulations
136 num_simulations = 10000
137 h = 0.000000001
138
139 d1 = (np.log(S_0 / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.
140     sqrt(T))
141 delta_exact = -norm.cdf(-d1)
142 print("Exact delta: " + str(delta_exact))
143
144 def common_rand_bump(S_0, K, r, sigma, T, num_simulations, h):
145     dW = np.sqrt(T) * np.random.normal(size=num_simulations)
146     S_T = S_0 * np.exp((r - 0.5 * sigma**2) * T + sigma * dW)
147     S_T_bump = (S_0+h) * np.exp((r - 0.5 * sigma**2) * T + sigma *
148         dW)

```

```

144     discounted_payoff = np.exp(-r * T) * np.maximum(K - S_T, 0)
145     discounted_payoff_bump = np.exp(-r * T) * np.maximum(K - (
146         S_T_bump), 0)
147
148     E_call = np.mean(discounted_payoff)
149     E_call_bump = np.mean(discounted_payoff_bump)
150
151     delta_com_mean = (E_call_bump - E_call) / h
152     print("Bump and reprice delta: " + str(delta_com_mean))
153
154 common_rand_bump(S_0, K, r, sigma, T, num_simulations, h)
155
156 def pathwise(S_0, K, r, sigma, T, num_simulations):
157     dW = np.sqrt(T) * np.random.normal(size=num_simulations)
158     S_T = S_0 * np.exp((r - 0.5 * sigma**2) * T + sigma * dW)
159     indicator_function = lambda K, S_T: np.where(K > S_T, 1, 0)
160     f = - np.exp(-r * T) * indicator_function(K, S_T) * np.exp((r -
161         0.5 * sigma**2) * T + sigma * dW)
162     print("Patwise approximation delta: " + str(np.mean(f)))
163
164 pathwise(S_0, K, r, sigma, T, num_simulations)
165
166 def LRM(S_0, K, r, sigma, T, num_simulations):
167     dW = np.sqrt(T) * np.random.normal(size=num_simulations)
168     S_T = S_0 * np.exp((r - 0.5 * sigma**2) * T + sigma * dW)
169     score_func = (np.log(S_T/S_0) - (r-0.5*sigma**2)*T)/(sigma ** 2
170         * T * S_0)
171     approx = np.maximum(K - S_T, 0) * score_func
172     discounted_mean = np.exp(-r * T) * np.mean(approx)
173     print("LRM approximated delta " + str(discounted_mean))
174
175 LRM(S_0, K, r, sigma, T, num_simulations)
176
177 #b)
178 h = num_simulations ** (-0.25)
179 # Calculate gamma
180 gamma_exact = 1 / (S_0 * sigma * np.sqrt(T)) * norm.pdf(d1)
181
182 print("Exact gamma: " + str(gamma_exact))
183 def gamma_bump_approx(S_0, K, r, sigma, T, num_simulations, h):
184     dW = np.sqrt(T) * np.random.normal(size=num_simulations)
185     S_T = S_0 * np.exp((r - 0.5 * sigma**2) * T + sigma * dW)
186     S_T_bump = (S_0+h) * np.exp((r - 0.5 * sigma**2) * T + sigma *
187         dW)
188     S_T_minusbump = (S_0-h) * np.exp((r - 0.5 * sigma**2) * T +
189         sigma * dW)
190
191     discounted_payoff = np.exp(-r * T) * np.maximum(K - S_T, 0)
192     discounted_payoff_bump = np.exp(-r * T) * np.maximum(K - (
193         S_T_bump), 0)
194     discounted_payoff_minusbump = np.exp(-r * T) * np.maximum(K - (
195         S_T_minusbump), 0)
196
197     f_theta = np.mean(discounted_payoff)
198     f_theta_bump = np.mean(discounted_payoff_bump)

```



```

194     f_theta_minusbump = np.mean(discounted_payoff_minusbump)
195
196     gamma_com_mean = (f_theta_bump - 2 * f_theta +
197                       f_theta_minusbump) / h ** 2
198     print("Bump and reprice gamma: " + str(gamma_com_mean))
199
200     gamma_bump_approx(S_0, K, r, sigma, T, num_simulations, h)
201     #Pathwise not possible
202
203     def gamma_LRM_approx(S_0, K, r, sigma, T, num_simulations):
204         dW = np.sqrt(T) * np.random.normal(size=num_simulations)
205         S_T = S_0 * np.exp((r - 0.5 * sigma**2) * T + sigma * dW)
206         score_func = (1 - np.log(S_T/S_0)+(r-0.5*sigma**2)*T)/(sigma
207                       **2*T*S_0**2)
208         approx = np.maximum(K - S_T, 0) * score_func
209         discounted_mean = np.exp(-r * T) * np.mean(approx)
210         print("LRM approximated gamma " + str(discounted_mean))
211
212     gamma_LRM_approx(S_0, K, r, sigma, T, num_simulations)
213     #c)
214     import numpy as np
215     import matplotlib.pyplot as plt
216     from scipy.stats import norm
217     import matplotlib.pyplot as plt
218     # Parameters
219     mu = 0.1
220     sigma = 0.25
221     S_0 = 100
222     B_0 = 1
223     r = 0.03
224     K = 100
225     T = 2
226     num_simulations = int(2500)
227     delta_t = 0.01
228     num_puts=-1000
229
230     def get_delta(S_c, T):
231         d1 = (np.log(S_c / K) + (r + 0.5 * sigma ** 2) * T) / (sigma *
232                       np.sqrt(T))
233         delta_exact = -norm.cdf(-d1)
234         return(delta_exact)
235
236     def get_put_price(S_c,T):
237         d1 = (np.log(S_c / K) + (r + 0.5 * sigma ** 2) * T) / (sigma *
238                       np.sqrt(T))
239         d2 = d1 - sigma * np.sqrt(T)
240         put_price = K * np.exp(-r*T)*norm.cdf(-d2)-S_c*norm.cdf(-d1)
241         return(put_price)
242
243     def get_gamma(S_c,T):
244         d1 = (np.log(S_c / K) + (r + 0.5 * sigma ** 2) * T) / (sigma *
245                       np.sqrt(T))
246         gamma_exact = norm.pdf(d1) / (S_c * sigma * np.sqrt(T))
247         return gamma_exact

```

```

246 def simulate_function_vectorized(K, T, S_0, mu, sigma, B_0, r,
247     delta_t, num_puts, num_simulations):
248     num_time_steps_total = int(T/delta_t)
249
250     # Initialize arrays to store results for each simulation
251     all_times = np.linspace(0, T, num_time_steps_total + 1)
252     all_S = np.zeros((num_time_steps_total + 1, num_simulations))
253     all_B = np.zeros((num_time_steps_total + 1, num_simulations))
254     all_phi = np.zeros((num_time_steps_total + 1, num_simulations))
255     all_psi = np.zeros((num_time_steps_total + 1, num_simulations))
256     all_price_puts = np.zeros((num_time_steps_total + 1,
257         num_simulations))
258     all_total_portfolio_value = np.zeros((num_time_steps_total + 1,
259         num_simulations))
260     all_gamma = np.zeros((num_time_steps_total + 1, num_simulations))
261
262     # Determine initial positions
263     put_price_initial = get_put_price(S_c=S_0, T=T)
264     price_puts_initial = num_puts * put_price_initial
265     phi_initial = -num_puts * get_delta(S_c=S_0, T=T)
266     psi_initial = -(price_puts_initial + phi_initial * S_0) / B_0
267     total_portfolio_value_initial = price_puts_initial +
268         phi_initial * S_0 + psi_initial * B_0
269     gamma_initial = num_puts * get_gamma(S_c=S_0, T=T)
270
271     # Assign initial values to arrays
272     all_S[0, :] = S_0
273     all_B[0, :] = B_0
274     all_phi[0, :] = phi_initial
275     all_psi[0, :] = psi_initial
276     all_price_puts[0, :] = price_puts_initial
277     all_total_portfolio_value[0, :] = total_portfolio_value_initial
278     all_gamma[0, :] = gamma_initial
279
280     # Iterate over discrete-time grid
281     for k in range(1, num_time_steps_total + 1):
282         # New asset prices
283         all_B[k, :] = all_B[k-1, :] * np.exp(r * delta_t)
284         all_S[k, :] = all_S[k-1, :] * np.exp((mu - 0.5 * sigma **
285             2) * delta_t + sigma * np.sqrt(delta_t) * norm.rvs(size
286                 =num_simulations))
287
288         # Current value of (S,B) portfolio from previous point-in-
289             time (below we will rebalance)
290         value = all_phi[k-1, :] * all_S[k, :] + all_psi[k-1, :] *
291             all_B[k, :]
292
293         # New value puts
294         if all_times[k] == T:
295             all_price_puts[k, :] = num_puts * np.maximum(K - all_S[
296                 k, :], 0)
297             all_total_portfolio_value[k, :] = all_price_puts[k, :]
298                 + value
299             break

```

```

291         all_price_puts[k, :] = num_puts * get_put_price(S_c=all_S[k
292             , :], T=T - all_times[k])
293         # Determine new position S for next interval (such that
294             combination of (S, B)-portfolio and puts is delta-
295             neutral)
296         all_phi[k, :] = -num_puts * get_delta(S_c=all_S[k, :], T=T
297             - all_times[k])
298         # Determine new position B, such that there is no net
299             cashflow in (S, B)-portfolio
300         all_psi[k, :] = (value - all_phi[k, :] * all_S[k, :]) /
301             all_B[k, :]
302         # Mismatch between discrete-time delta-neutral, self-
303             financing portfolio, and price puts
304         all_total_portfolio_value[k, :] = all_price_puts[k, :] +
305             all_phi[k, :] * all_S[k, :] + all_psi[k, :] * all_B[k,
306             :]
307         # Calculate gamma
308         all_gamma[k, :] = num_puts * get_gamma(S_c=all_S[k, :], T=T
309             - all_times[k])
310         return all_times, all_S, all_B, all_phi, all_psi,
311             all_price_puts, all_total_portfolio_value, all_gamma
312     # Perform simulations without using a for loop
313     all_times, all_S, all_B, all_phi, all_psi, all_price_puts,
314     portfolio_value, all_gamma = simulate_function_vectorized(
315         K, T, S_0, mu, sigma, B_0, r, delta_t, num_puts,
316         num_simulations
317     )
318     mean_portfolio_at_T = np.mean(portfolio_value[-1, :])
319     std_portfolio_at_T = np.std(portfolio_value[-1,:])
320     # Print the results
321     print(f"Mean portfolio value at maturity T: {mean_portfolio_at_T}")
322     print(f"Standard deviation of portfolio value at maturity T: {
323         std_portfolio_at_T}")
324     print("Our option price: ", np.exp(-r*T)*(mean_portfolio_at_T-
325         std_portfolio_at_T)/1000)
326     # Calculate quantiles
327     quantiles_phi = np.percentile(all_phi, [5, 50, 95], axis=1)
328     quantiles_psi = np.percentile(all_psi, [5, 50, 95], axis=1)
329     quantiles_gamma = np.percentile(all_gamma, [5, 50, 95], axis=1)
330     # Plot quantiles for phi, psi, and gamma
331     plt.figure(figsize=(12, 9))
332     # Plot quantiles for phi
333     plt.subplot(3, 1, 1)
334     plt.plot(all_times, quantiles_phi[1], label='Median (50%)', color='
335         blue')

```

```

331 plt.fill_between(all_times, quantiles_phi[0], quantiles_phi[2],
332                  color='lightblue', alpha=0.5, label='5%-95%')
333 plt.title('Quantiles of Positions in Stock over Time')
334 plt.xlabel('Time')
335 plt.ylabel('Phi')
336 plt.legend()
337
338 # Plot quantiles for psi
339 plt.subplot(3, 1, 2)
340 plt.plot(all_times, quantiles_psi[1], label='Median (50%)', color='
orange')
341 plt.fill_between(all_times, quantiles_psi[0], quantiles_psi[2],
342                  color='lightcoral', alpha=0.5, label='5%-95%')
343 plt.title('Quantiles of Position in Bond over Time')
344 plt.xlabel('Time')
345 plt.ylabel('Psi')
346 plt.legend()
347
348 # Plot quantiles for gamma
349 plt.subplot(3, 1, 3)
350 plt.plot(all_times, quantiles_gamma[1], label='Median (50%)', color
='green')
351 plt.fill_between(all_times, quantiles_gamma[0], quantiles_gamma[2],
352                  color='lightgreen', alpha=0.5, label='5%-95%')
353 plt.title('Quantiles of Gamma over Time')
354 plt.xlabel('Time')
355 plt.ylabel('Gamma')
356 plt.legend()
357
358 plt.tight_layout()
359 plt.show()
360
361 import numpy as np
362 import matplotlib.pyplot as plt
363 from scipy.stats import norm
364 import matplotlib.pyplot as plt
365 # Parameters
366 mu = 0.1
367 sigma = 0.25
368 S_0 = 100
369 B_0 = 1
370 r = 0.03
371 K_put = 100
372 T = 2
373 num_simulations = 2500
374 delta_t = 0.01
375 num_puts=-1000
376 num_time_steps_per_unit_of_time = 100
377
378 def get_delta_put(S_c, T):
379     d1 = (np.log(S_c / K_put) + (r + 0.5 * sigma ** 2) * T) / (
380         sigma * np.sqrt(T))
381     delta_exact = -norm.cdf(-d1)
382     return(delta_exact)

```

```

382
383 def get_put_price(S_c, T):
384     d1 = (np.log(S_c / K_put) + (r + 0.5 * sigma ** 2) * T) / (
385         sigma * np.sqrt(T))
386     d2 = d1 - sigma * np.sqrt(T)
387     put_price = K_put * np.exp(-r*T)*norm.cdf(-d2)-S_c*norm.cdf(-d1)
388     return(put_price)
389
390 def get_gamma(S_c, T, K):
391     d1 = (np.log(S_c / K) + (r + 0.5 * sigma ** 2) * T) / (sigma *
392         np.sqrt(T))
393     gamma_exact = 1 / (S_c * sigma * np.sqrt(T)) * norm.pdf(d1)
394     return gamma_exact
395
396 def get_delta_call(S_c, T):
397     d1 = (np.log(S_c / K_call) + (r + 0.5 * sigma ** 2) * T) / (
398         sigma * np.sqrt(T))
399     delta_exact = norm.cdf(d1)
400     return(delta_exact)
401
402 def get_call_price(S_c, T):
403     d1 = (np.log(S_c / K_call) + (r + 0.5 * sigma ** 2) * T) / (
404         sigma * np.sqrt(T))
405     d2 = d1 - sigma * np.sqrt(T)
406     call_price = S_c*norm.cdf(d1) - K_call * np.exp(-r*T)*norm.cdf(
407         d2)
408     return(call_price)
409
410 K_call = 120
411
412 def simulate_function_vectorized(T, S_0, mu, sigma, B_0, r,
413     num_puts, num_simulations):
414     num_time_steps_total = int(T/delta_t)
415
416     # Initialize arrays to store results for each simulation
417     all_times = np.linspace(0, T, num_time_steps_total + 1)
418     all_S = np.zeros((num_time_steps_total + 1, num_simulations))
419     all_B = np.zeros((num_time_steps_total + 1, num_simulations))
420     all_phi = np.zeros((num_time_steps_total + 1, num_simulations))
421     all_psi = np.zeros((num_time_steps_total + 1, num_simulations))
422     all_price_puts = np.zeros((num_time_steps_total + 1,
423         num_simulations))
424     all_total_portfolio_value = np.zeros((num_time_steps_total + 1,
425         num_simulations))
426     all_callposition = np.zeros((num_time_steps_total + 1,
427         num_simulations))
428     all_price_calls = np.zeros((num_time_steps_total + 1,
429         num_simulations))
430
431     # Determine initial positions
432     put_price_initial = get_put_price(S_c=S_0, T=T)
433     call_price_initial = get_call_price(S_c=S_0, T= T+3)
434     price_puts_initial = num_puts * put_price_initial
435     call_position_initial = (- num_puts * get_gamma(S_c=S_0, T=T, K
436         = K_put))/ get_gamma(S_c=S_0, T=T+3, K = K_call)
437     call_prices = call_position_initial * call_price_initial

```

```

427 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)
428 psi_initial = -(price_puts_initial + call_prices + phi_initial
      * S_0) / B_0
429 total_portfolio_value_initial = price_puts_initial +
      call_prices + phi_initial * S_0 + psi_initial * B_0
430
431 # Assign initial values to arrays
432 all_S[0, :] = S_0
433 all_B[0, :] = B_0
434 all_phi[0, :] = phi_initial
435 all_psi[0, :] = psi_initial
436 all_price_puts[0, :] = price_puts_initial
437 all_total_portfolio_value[0, :] = total_portfolio_value_initial
438 all_callposition[0, :] = call_position_initial
439 all_price_calls[0, :] = call_prices
440
441 # Iterate over discrete-time grid
442 for k in range(1, num_time_steps_total + 1):
443     # New asset prices
444     all_B[k, :] = all_B[k-1, :] * np.exp(r * delta_t)
445     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))
446
447     # Current value of (S,B, call) portfolio from previous
      point-in-time (below we will rebalance)
448     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])
449
450     # New value puts
451     if all_times[k] == T:
452         all_price_puts[k, :] = num_puts * np.maximum(K_put -
      all_S[k, :], 0)
453         all_total_portfolio_value[k, :] = all_price_puts[k, :]
      + value
454         break
455
456     all_price_puts[k, :] = num_puts * get_put_price(S_c=all_S[k
      , :], T=T - all_times[k])
457
458     # New position in call option such that portfolio is gamma
      neutral
459     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)
460     # Determine new position S for next interval (such that
      combination of (S, B)-portfolio and puts is delta-
      neutral)
461     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])
462
463     all_price_calls[k, :] = all_callposition[k, :] *
      get_call_price(S_c=all_S[k, :], T=T+ 3 - all_times[k])

```

```

464         # Determine new position B, such that there is no net
         # cashflow in (S, B, call)-portfolio
465         all_psi[k, :] = (value - all_price_calls[k, :] - all_phi[k
         , :] * all_S[k, :]) / all_B[k, :]
466
467         # Mismatch between discrete-time delta-neutral, self-
         # financing portfolio, and price puts
468         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, :]
469
470         return all_times, all_S, all_B, all_callposition, all_phi,
         all_psi, all_price_puts, all_price_calls,
         all_total_portfolio_value
471
472     # Perform simulations without using a for loop
473     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(
474         T, S_0, mu, sigma, B_0, r, num_puts, num_simulations
475     )
476
477     mean_portfolio_at_T = np.mean(all_total_portfolio_value[-1, :])
478     std_portfolio_at_T = np.std(all_total_portfolio_value[-1, :])
479     # Print the results
480     print(f"Mean portfolio value at maturity T: {mean_portfolio_at_T}")
481     print(f"Standard deviation of portfolio value at maturity T: {
         std_portfolio_at_T}")
482
483     print("Put option price: ", np.exp(-r*T)*(mean_portfolio_at_T-
         std_portfolio_at_T)/1000)
484
485     # Calculate quantiles
486     quantiles_phi = np.percentile(all_phi, [5, 50, 95], axis=1)
487     quantiles_psi = np.percentile(all_psi, [5, 50, 95], axis=1)
488     quantiles_callpos = np.percentile(all_callposition, [5, 50, 95],
         axis=1)
489     # Plot quantiles for phi, psi, and call position
490     plt.figure(figsize=(12, 9))
491
492     # Plot quantiles for phi
493     plt.subplot(3, 1, 1)
494     plt.plot(all_times, quantiles_phi[1], label='Median (50%)', color='
         blue')
495     plt.fill_between(all_times, quantiles_phi[0], quantiles_phi[2],
         color='lightblue', alpha=0.5, label='5%-95%')
496     plt.title('Quantiles of Positions in Stock over Time')
497     plt.xlabel('Time')
498     plt.ylabel('Phi')
499     plt.legend()
500
501     # Plot quantiles for psi
502     plt.subplot(3, 1, 2)
503     plt.plot(all_times, quantiles_psi[1], label='Median (50%)', color='
         orange')
504     plt.fill_between(all_times, quantiles_psi[0], quantiles_psi[2],
         color='lightcoral', alpha=0.5, label='5%-95%')

```

```

505 plt.title('Quantiles of Position in Bond over Time')
506 plt.xlabel('Time')
507 plt.ylabel('Psi')
508 plt.legend()
509
510 # Plot quantiles for call positions
511 plt.subplot(3, 1, 3)
512 plt.plot(all_times, quantiles_callpos[1], label='Median (50%)',
513          color='green')
514 plt.fill_between(all_times, quantiles_callpos[0], quantiles_callpos
515                  [2], color='lightgreen', alpha=0.5, label='5%-95%')
516 plt.title('Quantiles of Call positions over Time')
517 plt.xlabel('Time')
518 plt.ylabel('Calls')
519 plt.legend()
520 plt.tight_layout()
521 plt.show()

```