MF796hw5

March 26, 2024

0.1 MF 796: Computational Methods of Mathematical Finance

Professors Christopher Kelliher and Eugene Sorets Spring 2024

0.1.1 Problem set 5

Problem 1: Numerical PDEs: Suppose that the underlying security SPY evolves according to standard geometric brownian motion. Then its derivatives obey the Black-Scholes equation:

$$\frac{\partial c}{\partial t} + \frac{1}{2}\sigma^2 s^2 \frac{\partial^2 c}{\partial s^2} + rs\frac{\partial c}{\partial s} - rc = 0 \quad (1)$$

Assume SPY closed at 514 on March 14, 2024 and that the risk-free rate was 5.0%.
We are going to find the price of a call spread with the right of early exercise. The two strikes of the call spread are $K_1 = 525$ and $K_2 = 540$ and the expiry is September 20, 2024.

(a) Explain why this instrument is not the same as being long an American call with strike 525 and short an American call with strike 540, both with expiry September 20, 2024 The difference lies in the early exercise rights.

When someone holds a long position in an American call with strike 525 and a short position in an American call with strike 540, each option can be exercised independently by the respective option holders. For example, this means that the short position can be assigned at any time, regardless of exercise the long call or not.

On the other hand, a call spread with the right of early exercise is a single instrument. The holder have the right to exercise the spread as a whole.

(b) Let's assume that we are not able to find by calibrating to the European call spread price and must find it by other means. Find a way to pick the , explain why you chose this method, and then find the Use historical volatility as an estimate

[100]:
```
import numpy as np
import pandas as pd
import yfinance as yf

symbol = "SPY"
start_date = "2023-11-15"
end_date = "2024-03-14"

stock_data = yf.download(symbol, start=start_date, end=end_date)
```

```
log_returns = np.log(stock_data["Close"] / stock_data["Close"].shift(1))

daily_std = log_returns.std()

historical_volatility = daily_std * np.sqrt(252)

print(f"historical volatility of SPY: {historical_volatility:.4f}")
```

```
[**********************100%%**********************]  1 of 1 completed

historical volatility of SPY: 0.1037
```

(c) Set up an explicit Euler discretization of (1). You will need to make decisions about the choice of s_{max}, h_s, h_t , etc. Please explain how you arrived at each of these choices. To set up an explicit Euler discretization of the Black-Scholes equation (1), we need to make decisions about the following parameters:

1. s_{max}: The maximum stock price in the grid.
2. h_s: The stock price step size.
3. h_t: The time step size.

Below is the numerical value of these parameters, and the reason why we choose this number.

1. s_{max}: We want to choose an s_{max} that is sufficiently large to cover the range of stock prices we are interested in. For this question, since the highest strike price is 540, we can choose $s_{max} = 1620$.
2. h_s: The stock price step size should be small enough to provide accurate results but not so small that it significantly increases the computational time. Let's choose $h_s = 5$.
3. h_t: The time step size should be chosen to ensure stability and accuracy of the numerical scheme. For the explicit Euler method, the stability condition is

$$h_t \leq \frac{h_s^2}{\sigma^2 s_{max}^2}$$

With $h_s = 1$ and $= 0.1037$, we get $h_t \leq 0.00003$.

[73]:
```
S0 = 514.0
K1 = 525.0
K2 = 540.0
sigma = 0.1037  # Volatility
smax = 3 * K2
r = 0.05  # Annual risk-free rate
T = 0.5  # 6 months
hs = 5.0  # Spatial step size
ht = 0.0001  # Time step size, adjusted for stability and accuracy
```

[88]:
```
S = np.arange(hs, smax + hs, hs)
si = S[:-1]  # Exclude the last point to match the intended size
M = len(si)  # Correctly adjusting M to match the size of si

# Coefficients for the tridiagonal matrix
ai = 1 - (sigma**2) * (si**2) * (ht/hs**2) - r * ht
li = ((sigma**2) * (si**2)/2) * (ht/hs**2) - (r * si * ht)/(2*hs)
ui = ((sigma**2) * (si**2)/2) * (ht/hs**2) + (r * si * ht)/(2*hs)

# Initialize the tridiagonal matrix A
A = np.diag(ai)

# Adjustments for the tridiagonal matrix
l = li[1:]
u = ui[:-1]
for i in range(M-1):  # Adjusting the range to match the dimensions of A
    if i < M-2:  # Ensure we don't exceed the bounds
        A[i, i+1] = u[i]
    if i > 0:
        A[i, i-1] = l[i-1]
A
```

[88]:
```
array([[ 9.99993925e-01,  3.03768450e-06,  0.00000000e+00, …,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [-2.84926200e-06,  9.99990699e-01,  7.15073800e-06, …,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 0.00000000e+00, -2.66083950e-06,  9.99985322e-01, …,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       …,
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00, …,
         8.89187903e-01,  5.62060486e-02,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00, …,
         5.49442797e-02,  8.88496441e-01,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00, …,
         0.00000000e+00,  0.00000000e+00,  8.87802828e-01]])
```

(d) Let A be the update matrix that you created in the previous step. Find out its eigenvalues and check their absolute values. If necessary, make adjustments to h_s, h_t , and anything else you need.

[67]:
```
egv,egl = np.linalg.eig(A)
```

[68]:
```
egl
```

[68]:
```
array([[ 5.13877331e-13,  4.95169662e-13,  3.81468765e-13, …,
         5.05007509e-23, -5.29210335e-23,  0.00000000e+00],
       [-1.07672573e-12, -3.54359409e-12, -1.05359657e-11, …,
         4.75297028e-22, -4.88366217e-22,  0.00000000e+00],
```

```
       [ 7.52587670e-11,  2.12235575e-10,  5.89493906e-10, …,
        -4.52987840e-22,  4.76349375e-22,  0.00000000e+00],
       …,
       [-2.48439834e-07,  2.17212249e-07, -1.89349393e-07, …,
        -2.02474926e-01,  1.79610148e-01,  0.00000000e+00],
       [-1.22928846e-07,  1.07452867e-07, -9.36488256e-08, …,
         1.27026690e-01, -1.20263039e-01,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00, …,
         0.00000000e+00,  0.00000000e+00,  1.00000000e+00]])
```

[69]:
```
egv
```

[69]:
```
array([0.99953911, 0.9995644 , 0.99958877, 0.99963482, 0.99965654,
       0.99969742, 0.99973501, 0.99975261, 0.99976943, 0.99978549,
       0.99980079, 0.99981537, 0.99982922, 0.99984237, 0.99985484,
       0.99986663, 0.99987776, 0.99988825, 0.99989811, 0.99999108,
       0.99998823, 0.99998645, 0.99998571, 0.99998514, 0.99998412,
       0.99998268, 0.99998084, 0.9999786 , 0.99997596, 0.99997289,
       0.9999694 , 0.99996545, 0.99996105, 0.99995616, 0.99995079,
       0.99994491, 0.9999385 , 0.99990735, 0.99993156, 0.99992406,
       0.999916  , 0.99930203, 0.9993352 , 0.99936732, 0.99939842,
       0.99945762, 0.99948574, 0.9995129 , 0.99961224, 0.99971662,
       0.9996774 , 0.99895405, 0.99899731, 0.99926781, 0.99923251,
       0.99903938, 0.99919613, 0.99908028, 0.99912003, 0.99915864,
       0.99942852, 0.99802023, 0.99815096, 0.99821411, 0.99833607,
       0.99839493, 0.99845238, 0.99850846, 0.9988639 , 0.99856318,
       0.99861655, 0.99881697, 0.9986686 , 0.99871934, 0.99876879,
       0.99727303, 0.99735509, 0.99759124, 0.99766668, 0.99774052,
       0.99781277, 0.99788346, 0.99795261, 0.99808633, 0.99827581,
       0.99890958, 0.99614939, 0.99625329, 0.99635526, 0.99645531,
       0.99655346, 0.99664974, 0.99674416, 0.99751417, 0.99743546,
       0.99683676, 0.99718926, 0.99692754, 0.99701654, 0.99710378,
       0.99487179, 0.99499857, 0.99536581, 0.99604352, 0.99593567,
       0.99548394, 0.99559997, 0.99571391, 0.99582581, 0.9939206 ,
       0.99406349, 0.994478  , 0.99512315, 0.99524556, 0.99347738,
       0.99362757, 0.9937753 , 0.99420401, 0.99434217, 0.99461153,
       0.99474278, 0.9920103 , 0.9921839 , 0.9923548 , 0.9933247 ,
       0.992523  , 0.99268854, 0.99285145, 0.99316951, 0.99301177,
       0.9911004 , 0.99183394, 0.99128806, 0.9916548 , 0.99147285,
       0.99090983, 0.99071633, 0.8313836 , 0.99051986, 0.99011787,
       0.98991229, 0.9897036 , 0.98949178, 0.98927678, 0.98905858,
       0.98883713, 0.9886124 , 0.98838435, 0.98815295, 0.98791816,
       0.98767994, 0.98743825, 0.98719306, 0.98694432, 0.83476052,
       0.98669199, 0.97944765, 0.9797994 , 0.98014644, 0.98048882,
       0.98564603, 0.98537516, 0.98510046, 0.98482189, 0.98453938,
       0.98425292, 0.9808266 , 0.98115982, 0.98396244, 0.9836679 ,
       0.98336926, 0.98148855, 0.98181282, 0.9821327 , 0.98244824,
```

```
0.9748378 , 0.97525071, 0.97565818, 0.97606028, 0.97645707,
0.97684861, 0.97723496, 0.9776162 , 0.97799237, 0.97909113,
0.98643604, 0.98306647, 0.9859131 , 0.97836355, 0.84109057,
0.84972117, 0.85240582, 0.97268925, 0.97441938, 0.8469458 ,
0.97133038, 0.97178932, 0.97224225, 0.97313039, 0.97356575,
0.85752894, 0.86235952, 0.86467627, 0.86913108, 0.97039417,
0.97086536, 0.87127543, 0.87541169, 0.87740836, 0.87936025,
0.96991674, 0.96943299, 0.96586268, 0.96639306, 0.96691651,
0.96743312, 0.96794299, 0.96894284, 0.88313729, 0.88496589,
0.99032038, 0.96532528, 0.96478077, 0.83799102, 0.88675665,
0.89023029, 0.89191579, 0.89356866, 0.96422905, 0.95954339,
0.96015652, 0.9607616 , 0.96135873, 0.96194803, 0.96367001,
0.89834239, 0.89678095, 0.96252961, 0.95892208, 0.93289238,
0.93387011, 0.93483376, 0.9357836 , 0.93855286, 0.93945   ,
0.94033456, 0.94120677, 0.9376429 , 0.94206683, 0.94291497,
0.95829248, 0.94375139, 0.94619229, 0.94698377, 0.94538985,
0.94776448, 0.9485346 , 0.95765446, 0.9522321 , 0.9529421 ,
0.95501578, 0.95433379, 0.95364261, 0.94929429, 0.89987529,
0.9556887 , 0.95151248, 0.90431134, 0.90573845, 0.907141  ,
0.90851966, 0.91251869, 0.91380803, 0.9319003 , 0.92562973,
0.92671428, 0.9277826 , 0.95004373, 0.95700791, 0.92883505,
0.9684462 , 0.96310357, 0.90138053, 0.91120791, 0.91995095,
0.92112212, 0.92227519, 0.98617643, 0.9563527 , 0.91507644,
0.94457628, 0.93089359, 0.92452861, 0.95078307, 0.91755256,
0.91876124, 0.82784107, 0.84407194, 0.85997844, 0.8812693 ,
0.98275948, 0.97872978, 0.97399539, 0.888511  , 0.90285896,
0.90987509, 0.91632445, 0.92341056, 0.92987194, 0.82410868,
0.85500646, 0.87336814, 0.9367199 , 0.89519003, 0.82015485,
0.86693235, 0.7865806 , 0.815937  , 0.81139501, 0.79454587,
0.80091769, 0.80643839, 0.88780283])
```

[71]:
```
check = np.any(np.abs(egv) > 1)
check
```

[71]:
```
False
```

All the eigen value is less than 1, which means the stability condition is met

(e) Apply your discretization scheme to find today's price of the call spread without the right of early exercise. The scheme will produce a whole vector of prices at time 0. Explain how you chose the one for today's price.

[96]:
```
def European_call_spread_price(smax, sigma, r, K1, K2, T, S0, hs, ht, M, N):
    # Grid setup for the underlying asset
    S = np.arange(0, smax + hs, hs)
    si = S[:-1]

    # Coefficients for the finite difference scheme
```

```
    ai = 1 - (sigma**2) * (si**2) * (ht/hs**2) - r * ht
    li = ((sigma**2) * (si**2) / 2) * (ht/hs**2) - (r * si * ht / (2 * hs))
    ui = ((sigma**2) * (si**2) / 2) * (ht/hs**2) + (r * si * ht / (2 * hs))

    # Construction of the tridiagonal matrix
    A = np.diag(ai, 0) + np.diag(li[1:], -1) + np.diag(ui[:-1], 1)
    C = np.zeros([M+1 , N])
    pl = np.maximum(si - K1, 0)  # Payoff for a call option with strike K1
    ps = np.maximum(si - K2, 0)  # Payoff for a call option with strike K2
    C[:, -1] = pl - ps  # Initial condition: max payoff at maturity

    for j in range(N, 1, -1):
        tj = ht * j
        bj = ui[-1] * (K2 - K1) * np.exp(-r * (T - tj))  # Boundary condition
        C[:, j - 2] = A @ C[:, j - 1]  # Propagate the solution backwards
        C[-1, j - 2] += bj  # Adjust the last spatial node with the boundary
↪condition

    return np.interp(S0, si, C[:, 0])

S0 = 514.0
K1 = 525.0
K2 = 540.0
T = 0.5
r = 0.05
sigma = 0.1037
smax = 3 * K2
M = 275
N = 1000

hs = smax/M
ht = T/N

p1 = European_call_spread_price(smax, sigma, r, K1, K2, T, S0, hs, ht, M, N)
print(p1)
```

```
6.261705883665412
```

(f) Modify your code in the previous step to calculate the price of the call spread with the right of early exercise. What is the price?

[97]:
```
def American_call_spread_price(smax, sigma, r, K1, K2, T, S0, hs, ht, M, N):
    # Grid setup for the underlying asset
    S = np.arange(0, smax + hs, hs)
    si = S[:-1]

    # Coefficients for the finite difference scheme
    ai = 1 - (sigma**2) * (si**2) * (ht/hs**2) - r * ht
```

```
    li = ((sigma**2) * (si**2) / 2) * (ht/hs**2) - (r * si * ht / (2 * hs))
    ui = ((sigma**2) * (si**2) / 2) * (ht/hs**2) + (r * si * ht / (2 * hs))

    # Construction of the tridiagonal matrix
    A = np.diag(ai, 0) + np.diag(li[1:], -1) + np.diag(ui[:-1], 1)
    C = np.zeros([M+1 , N])
    pl = np.maximum(si - K1, 0)  # Payoff for a call option with strike K1
    ps = np.maximum(si - K2, 0)  # Payoff for a call option with strike K2
    C[:, -1] = pl - ps  # Initial condition: max payoff at maturity

    for j in range(N, 1, -1):
        tj = ht * j
        bj = ui[-1] * (K2 - K1) * np.exp(-r * (T - tj))  # Boundary condition
        C[:, j - 2] = A @ C[:, j - 1]  # Propagate the solution backwards
        C[-1, j - 2] += bj  # Adjust the last spatial node with the boundary␣
 ↪condition
        C[:,j-2] = np.max([C[:,j-2], pl-ps], axis=0)

    return np.interp(S0, si, C[:, 0])

p2 = American_call_spread_price(smax, sigma, r, K1, K2, T, S0, hs, ht, M, N)
print(p2)
```

```
8.75836431848145
```

(g) Calculate the early exercise premium as the difference between the American and European call spreads. Is it reasonable?

[99]:
```
diff = p2-p1
diff
```

[99]:
```
2.4966584348160374
```

The early exercise premium is about: 2.50. I think this result is reasonable. American call spreads allow people early exercise, so it is more expensive.

[ ]: