Computational Finance - Case Study - 2

Sowrya Gali - sg4150

March 2023

From the question, the values of the strike price, barrier, initial stock price, etc., are encoded in the following listing 1.

```
import numpy as np
import math
import scipy.special as sc
import timeit
from tqdm import tqdm

S_0 = 1900
K = 2000
Barrier = 2200
r = 0.0025
q = 0.015
T = 0.5
base_vol = 0.25
nu = 0.31
theta = -0.25
Y = 0.4
SMIN = 500
Rebate = 0.0

xMin, xMax = np.log(SMIN), np.log(Barrier)
N = 800
M = 100
dx = (xMax - xMin) / N
dt = T / M
EPS = dx
tau = dt * np.arange(1, M+1)
x = xMin + np.arange(N+1) * dx
```

Listing 1: Common values and libraries

Then we define the $g1$ and $g2$; the functions that are used for discretizing the integral are in the following listing 2:

```
def g1(x, alpha):
    return sc.gammaincc(1-alpha, x) * sc.gamma(1-alpha)

def g2(x, alpha):
    return ((np.exp(-x) * (x**(-alpha)) / alpha)) - g1(x, alpha) / alpha

def sig_calculator(l):
    return (l**(Y-2)) * (-(l*EPS)**(1-Y) * np.exp(-l*EPS) + (1-Y) * (g1(0, Y) - g1(l
    *EPS, Y))) / nu

lambda_n = math.sqrt((theta**2/base_vol**4) + (2.0/(base_vol**2 * nu))) + theta/
    base_vol**2
lambda_p = math.sqrt((theta**2/base_vol**4) + (2.0/(base_vol**2 * nu))) - theta/
    base_vol**2
```

Listing 2: Discretization functions

The following functions are calculated in the listing 2

$$
\begin{aligned}
\sigma^2(\epsilon) &= \frac{1}{\nu}\lambda_p^{Y-2}(-(\lambda_p\epsilon)^{1-Y}e^{-\lambda_p\epsilon} + (1-Y)(g_1(0) - g_1(\lambda_p\epsilon))) \\
&\quad + \frac{1}{\nu}\lambda_n^{Y-2}(-(\lambda_n\epsilon)^{1-Y}e^{-\lambda_n\epsilon} + (1-Y)(g_1(0) - g_1(\lambda_n\epsilon))) \\
g_1(\xi) &= \int_\xi^\infty \frac{e^{-z}}{z^\alpha}dz \\
g_2(\xi) &= \int_\xi^\infty \frac{e^{-z}}{z^{1+\alpha}}dz \\
\lambda_p &= (\frac{\theta^2}{\sigma^4} + \frac{2}{\sigma^2\nu})^{\frac{1}{2}} - \frac{\theta}{\sigma^2} \\
\lambda_n &= (\frac{\theta^2}{\sigma^4} + \frac{2}{\sigma^2\nu})^{\frac{1}{2}} + \frac{\theta}{\sigma^2}
\end{aligned}
\tag{1}
$$

We have some pre-calculated vectors of the g_1 and g_2 functions. The following listing 3 calculates them, apart from B_l and B_u.

```
kx = np.arange(1, N+1) * dx
g1_n = g1(kx * lambda_n, Y)
g1_p = g1(kx * lambda_p, Y)
g2_n = g2(kx * lambda_n, Y)
g2_p = g2(kx * lambda_p, Y)
g2_n_plus = g2(kx * (lambda_n+1), Y)
g2_p_minus = g2(kx * (lambda_p-1), Y)

sigma = sig_calculator(lambda_n) + sig_calculator(lambda_p)
omega = ((lambda_p**Y) * g2(lambda_p*EPS, Y) - ((lambda_p-1)**Y * g2((lambda_p-1)*EPS, Y)) \
+ (lambda_n**Y) * g2(lambda_n*EPS, Y)  - ((lambda_n+1)**Y * g2((lambda_n+1)*EPS, Y))) / nu

alpha = sigma * dt / (2 * dx**2)
beta = r - q + omega - (sigma / 2)

Bl = alpha - beta * dt / (2*dx)
Bu = alpha + beta * dt / (2*dx)
```

Listing 3: Pre-calculated vectors B_l and B_u

The triDiag() and sol() methods are used for solving the matrix equation to get the vector of the call option prices. The following listing 4 shows the methods.

```
def triDiag(LL, DD, UU, rhs):
    n = len(rhs)
    v = np.zeros(n)
    y = np.zeros(n)
    w = DD[0]
    y[0] = 1.0 * rhs[0] / w

    for i in range(1, n):
        v[i-1] = 1. * UU[i-1] / w
        w = DD[i] - LL[i] * v[i-1]
        y[i] = 1. * (rhs[i] - LL[i] * y[i-1]) / w

```

```python
    for j in range(n-2, -1, -1):
        y[j] = y[j] - v[j] * y[j+1]

    return y


def sol(w):
    ans = np.zeros(N-1)
    for i in range(1, N):
        if i == 1 or i == N-1:
            ans[i-1] = 0
        else:
            for k in range(1, i):
                ans[i-1] += lambda_n**Y * (w[i-k] - w[i] - k * (w[i-k-1] - w[i-k])) * (g2_n[k-1] - g2_n[k])
                ans[i-1] += (w[i-k-1] - w[i-k]) * (g1_n[k-1] - g1_n[k]) / ((lambda_n ** (1-Y)) * dx)

            for k in range(1, N-i):
                ans[i-1] += lambda_p**Y * (w[i+k] - w[i] - k * (w[i+k+1] - w[i+k])) * (g2_p[k-1] - g2_p[k])
                ans[i-1] += (w[i+k-1] - w[i+k]) * (g1_p[k-1] - g1_p[k]) / ((lambda_p ** (1-Y)) * dx)
        ans[i-1] += K * lambda_n**Y * g2_n[i-1] - np.exp(x[i]) * (lambda_n + 1)**Y * g2_n_plus[i-1]
    return ans
```

Listing 4: Solvers and Diagonalizations

The final result can be calculated using the following snippet 5.

```python
l = np.ones(N-1) * (-Bl)
u = np.ones(N-1) * (-Bu)
d = 1 + r*dt + Bu + Bl + dt * (lambda_n**Y * g2_n[:N-1] + lambda_p**Y * g2_p[::-1][:N-1]) / nu

u[-1] =  0
l[0] = 0

s = np.exp(x)
vCall = np.maximum(s - K, 0) * (s < Barrier)

start = timeit.default_timer()
for j in tqdm(range(M)):
    rhs = (dt * sol(vCall) / nu) + vCall[1:N]
    inner = triDiag(l, d, u, rhs)
    vCall = np.pad(inner, (1, 1), 'constant', constant_values=(0, 0))
stop = timeit.default_timer()
print('Time: ', stop - start)

uoc_imp = np.interp(np.log(S_0), x, vCall)
print('Price of the UOC option:', uoc_imp)
```

Listing 5: Final calculation

The output is the following:

```
sigma:  0.0001691930835144435
omega:  0.4748657171079357
Bl:  -0.5007064081899342
```

```
Bu:  0.747348947466064
Time:  142.288243292
Price of the UOC option: 33.41145860422139
```

Listing 6: Output