Computational Finance exam: Report.

Viktoriia Egorova (matricola 879882)

23.06.2020

Task: to price an American options with the Crank-Nicolson finite difference method.

Finite-difference methods are discretizations which are used for solving differential equations by approximating them with difference equation (finite differences approximate the derivatives). In particular, they are used for pricing options. During the lectures explicit and fully implicit methods were considered and implemented in Python. Crank−Nicolson method is another finite difference method, implicit in time, which can be thought of as an average of the explicit method and the fully implicit method.
The project consists of three python files: "CrankNicolson.py" (the implementation of Crank-Nicolson method), "lesson.py" (implementation of explicit and fully implicit methods from lessons), "Main.py" (the comparison of explicit, fully implicit and Crank-Nicolson methods).

1 Preliminary theory

The goal is to solve the Black-Scholes PDE:

$$\frac{\partial}{\partial t}U(S,t)+\frac{1}{2}\sigma^2S^2\cdot\frac{\partial^2}{\partial S^2}U(S,t)+rS\cdot\frac{\partial}{\partial S}U(S,t)-r\cdot U(S,t)=0,$$

or, in more general form

$$\frac{\partial}{\partial t}U(S,t)+a(S)\cdot\frac{\partial^2}{\partial S^2}U(S,t)+b(S)\cdot\frac{\partial}{\partial S}U(S,t)+c(S)\cdot U(S,t)=0. \tag{1}$$

Stock-time plane is discretized in to a grid of nodes with examined time points $(0,\Delta t,...,K\Delta t)$ and examined stock prices $(0,\Delta S,...,I\Delta S)$. We are going backward in time starting from the expiration and back to time 0. To find a numerical solution means to find the option value at each of the grid points

$$U_{i,k}=U(i\Delta S,T-k\Delta t)\text{ where }0\le i\le I,\ 0\le k\le K \tag{2}$$

In order to solve the PDE numerically we need to specify boundary and final (initial) conditions.

1. At expiry the option value is just the payoff function. Then final condition is

$$U_{i,0}=\text{Payoff}(i\Delta S)=\begin{cases}\max(i\Delta S-K,0)\text{ for a call option with strike K}\\ \max(K-i\Delta S,0)\text{ for a put option with strike K}\end{cases},0\le i\le I. \tag{3}$$

2. We also need to specify the solution value at the extremes of the region, that is to prescribe the option value at $S=0$ and $S=I\Delta S$. The boundary conditions are

$$\text{For the call option: }\begin{cases}S=0\Rightarrow U_{0,k}=0\\ \text{Large }S\Rightarrow U_{I,k}=I\Delta S-Ke^{-rk\Delta t}\end{cases} \tag{4}$$

$$\text{For the put option: }\begin{cases}S=0\Rightarrow U_{0,k}=Ke^{-rk\Delta t}\\ S=0\Rightarrow U_{I,k}=0\end{cases} \tag{5}$$

1.1 Explicit method

By choosing the backward difference approximation at time k for the time derivative and central difference approximation for the space derivative, equation (1) can be rewritten in the following form:

$$\frac{U_{i,k}-U_{i,k+1}}{\Delta t}+a\frac{U_{i+1,k}-2U_{i,k}+U_{i-1,k}}{\Delta S^2}+b\frac{U_{i+1,k}-U_{i-1,k}}{2\Delta S}+cU_{i,k}=0.$$

By reorganising coefficients we get

$$U_{i,k+1} = A_i U_{i+1,k} + B_i U_{i,k} + C_i U_{i-1,k}.$$

The knowledge at time step k of values $U_{i,k}$ for all i allows to explicitly calculate $U_{i,k+1}$, si it can be easily generalized to handle the early exercise of an American options. Explicit method is easy to implement, but it does not always converge and there are restrictions on the timestep, so it also can be slower than other schemes.

1.2 Implicit method

Here the first and second derivatives with respect to S are calculated at time step $k+1$ instead of k.

$$\frac{U_{i,k} - U_{i,k+1}}{\Delta t} + a_i \frac{U_{i+1,k+1} - 2U_{i,k+1} + U_{i-1,k+1}}{\Delta S^2} + b_i \frac{U_{i+1,k+1} - U_{i-1,k+1}}{2\Delta S} + c_i U_{i,k+1} = 0.$$

By reorganising coefficients

$$U_{i,k} = A_i U_{i+1,k+1} + B_i U_{i,k+1} + C_i U_{i-1,k+1}.$$

Implicit method does not require the restriction on the time step, but the solution is not straightforward. After some computations, implicit system can be rewritten in the matrix form, taking into the account boundary conditions, as:

$$M_{imp} U = q.$$

Two most used solution methods for such matrix equations are

- LU Decomposition. A direct method. It means to find the exact solution of the equations in one go through matrix operations. It is not immediately applicable to American options.
- Successive Over Relaxation (SOR) method. An iterative method. It starts with a guess for the solution and successively improves it until it converges near enough to the exact solution. The solution obtained by this method is not exact, but we can find the solution to whatever accuracy is required. It is easily applied to American options.

  Matrix equation $MU = q$ with $N \times N$ matrix can be iteratively rewritten starting from some initial guess (n denotes the level of the iteration) as

$$\begin{cases} M_{11}v_1 + M_{12}v_2 + \dots + M_{1N}v_N = q_1 \\ M_{21}v_1 + M_{22}v_2 + \dots + M_{2N}v_N = q_2 \\ \dots\dots \\ M_{N1}v_1 + M_{N2}v_2 + \dots + M_{NN}v_N = q_N \end{cases} \Rightarrow \begin{cases} v_1^{n+1} = \frac{1}{M_{11}}(q_1 - (M_{12}^n v_2 + \dots + M_{1N}^n)) \\ v_2^{n+1} = \frac{1}{M_{22}}(q_2 - (M_{21}^n v_2 + \dots + M_{2N}^n)) \\ \dots\dots \\ v_N^{n+1} = \frac{1}{M_{NN}}(q_N - (M_{N1}^n v_2 + \dots)) \end{cases}$$

  Matrix $M = D + U + L$ can be written as a sum of a diagonal, upper triangular and a lower triangular matrices. In order to speed up the convergence the relaxation parameter is included:

$$v^{n+1} = v^{n+1} + \omega D^{-1}(q - Uv^n - Lv^n)$$

1.3 Crank - Nicolson Method

It is another implicit finite difference method, which can be thought of as an average of the explicit method and the fully implicit method. It uses six points: three from the previous time step and three from the current time step. The scheme is

$$\frac{U_{i,k} - U_{i,k+1}}{\Delta t} + \frac{a_i}{2} \cdot \frac{U_{i+1,k+1} - 2U_{i,k+1} + U_{i-1,k+1}}{\Delta S^2} + \frac{a_i}{2} \cdot \frac{U_{i+1,k} - 2U_{i,k} + U_{i-1,k}}{\Delta S^2} + \\ + \frac{b_i}{2} \frac{U_{i+1,k+1} - U_{i-1,k+1}}{2\Delta S} + \frac{b_i}{2} \frac{U_{i+1,k} - U_{i-1,k}}{2\Delta S} + \frac{c_i}{2} U_{i,k} + \frac{c_i}{2} U_{i,k+1} = 0.$$

This can be rewritten as

$$-A_i U_{i-1,k+1} + (1 - B_i) U_{i,k+1} - C_i U_{i+1,k+1} = A_i U_{i-1,k} + (1 + B_i) U_{i,k} + C_i U_{i+1,k}$$

where

$$\begin{cases} A = \frac{1}{2}\nu_1 a_i + \frac{1}{4}\nu_2 b_i = \frac{1}{4}ki(\sigma^2 i + r) \\ B = -\nu_1 a_i + \frac{1}{2}\Delta t c_i = -\frac{1}{2}\Delta t(\sigma^2 i^2 + r) \\ C = \frac{1}{2}\nu_1 a_i - \frac{1}{4}\nu_2 b_i = \frac{1}{4}ki(\sigma^2 i - r) \end{cases}$$

These equations are hold only for $1 \leq i \leq I-1$. Two missing equations are from boundary conditions. We can write matrix equation with two matrices with $I-1$ rows (equations) and $I+1$ columns (unknowns):

$$\begin{bmatrix} -A_1 & 1-B_1 & -C_1 & 0 & \cdots & 0 \\ 0 & -A_2 & 1-B_2 & -C_2 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & \cdots & -A_{I-1} & 1-B_{I-1} & -C_{I-1} \end{bmatrix} \cdot \begin{bmatrix} U_{0,k+1} \\ U_{1,k+1} \\ U_{2,k+1} \\ \cdots \\ U_{I-1,k+1} \\ U_{I,k+1} \end{bmatrix} =$$

$$\begin{bmatrix} A_1 & 1+B_1 & C_1 & 0 & \cdots & 0 \\ 0 & A_2 & 1+B_2 & C_2 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & \cdots & A_{I-1} & 1+B_{I-1} & C_{I-1} \end{bmatrix} \cdot \begin{bmatrix} U_{0,k} \\ U_{1,k} \\ U_{2,k} \\ \cdots \\ U_{I-1,k} \\ U_{I,k} \end{bmatrix} \tag{6}$$

From boundary conditions $U_{0,k+1}$ and $U_{I,k+1}$ are given, so left-hand side of (6) can be written as

$$\begin{bmatrix} 1-B_1 & -C_1 & 0 & \cdots \\ -A_2 & 1-B_2 & -C_2 & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & -A_{I-1} & 1-B_{I-1} \end{bmatrix} \cdot \begin{bmatrix} U_{1,k+1} \\ U_{2,k+1} \\ \cdots \\ U_{I-1,k+1} \end{bmatrix} + \begin{bmatrix} -A_1 U_{0,k+1} \\ 0 \\ \cdots \\ -C_{I-1} U_{I,k+1} \end{bmatrix} = M_L^{k+1} U_{k+1} + R_k = M_R^k U_k$$

where vector $M_R^k U_k$ is known right-hand side of (6). Finally

$$M_L^{k+1} U_{k+1} = M_R^k U_k - R_k = RHS^k \tag{7}$$

The solution of this matrix can be obtained using the SOR method.

2 Implementation of Crank - Nicolson method

Crank - Nicolson method is implemented (file "CrankNicolson.py") in function taking as input data the type of an option, short rate, strike, initial stock level, volatility, dividend yield, time to maturity and indicator of American/European option:

```python
def CrankNicolsonSOR(payout, r, K, S0, sigma, delta, T, american = True)
```

In order to solve the numerical problem we need to prepare the grid (as described in (2)), that is to specify the time step and space step:

```python
S_min = 0.0;
S_max = 150.0;
L = S_max - S_min;
N = 1002;  # Number of time steps
dt = float(T) / float(N);  # time step size
I = 100;  # Number of space steps
dS = float(L) / float(I);  # space step size
```

As a starting point we need to set initial condition according to (3). We consider both possible cases, for call and put option. Also the payoff is taking into account (it will be useful for considering the early exercise opportunity):

```python
# Initial values - vector at time zero is fully known
for i in range(0, I + 1):
S[i] = i * dS
if payout == 'call':
  Uk[i] = np.maximum(S[i] - K, 0)
else:
  Uk[i] = np.maximum(K - S[i], 0)
Payoff = Uk
```

Next we enter in cycle on time step.

```python
for k in range(1, N - 1):
   sys.stdout.write("\r" + "Now running time step nr: " + str(k) + "/" + str(N - 2))
   sys.stdout.flush()
   t = k * dt
  m = 0
  # initially U at the current step is set equal to U at the previous step
   Uk_next[1:I - 1] = Uk[1:I - 1]
```

For each time step the boundary conditions according to (4)-(5) should be specified:

```python
  # 1
  # Boundary conditions
   if payout == 'call':
     Uk_next[0] = 0
     Uk_next[I] = dS * I * math.exp(-delta * t) - K * math.exp(-r * t)
   else:
     Uk_next[0] = K * math.exp(-r * t)
     Uk_next[I] = 0
  # Now we completely know current vector of values Uk_next
```

For each time step, we need to find the coefficients of matrix M_L^{k+1}, M_R^k and RHS^k from (7). Notice that from (6) we have a vector with $I-1$ rows:

$$M_R^k U_k = \begin{bmatrix} A_1 U_{0,k} + (1+B_1)U_{1,k} + C_1 U_{2,k} \\ A_2 U_{1,k} + (1+B_2)U_{2,k} + C_2 U_{3,k} \\ \dots \\ A_{I-1} U_{I-2,k} + (1+B_{I-1})U_{I-1,k} + C_{I-1} U_{I,k} \end{bmatrix}$$

Now from (7) we get

$$RHS^k = \begin{bmatrix} A_1 U_{0,k} + (1+B_1)U_{1,k} + C_1 U_{2,k} + A_1 U_{0,k+1} \\ A_2 U_{1,k} + (1+B_2)U_{2,k} + C_2 U_{3,k} \\ \dots \\ A_{I-1} U_{I-2,k} + (1+B_{I-1})U_{I-1,k} + C_{I-1} U_{I,k} + C_{I-1} U_{I,k+1} \end{bmatrix}$$

So the right part is fully known due to boundary condition. For the left part we compute coefficients of M_L^{k+1} dividing them into 3 parts corresponding to coefficients A, B, C. These 3 array represents the diagonal upper triangular and lower triangular matrices of M_L^{k+1}.

```python
# Matrices A, b = RHS
RHS = np.zeros((I - 1))
mA = np.zeros((I - 1))
mB = np.zeros((I - 1))
mC = np.zeros((I - 1))
# A = np.zeros((I - 1, I - 1))
 for i in range(1, I):
   a = 0.25 * dt * i * (sigma * sigma * i - r)
   b = - 0.5 * dt * (r + sigma * sigma * i * i)
   c = 0.25 * dt * i * (sigma * sigma * i + r)
   A_right = a
   B_right = 1 + b
   C_right = c
   A_left = -a
   B_left = 1 - b
   C_left = -c
   if i == 1:
     RHS[i - 1] = A_right * Uk[i - 1] + B_right * Uk[i] + C_right * Uk[i + 1] - A_left * Uk_next[0]
     mB[i - 1] = B_left
     mC[i - 1] = C_left
   elif i == I - 1:
     RHS[i - 1] = A_right * Uk[i - 1] + B_right * Uk[i] + C_right * Uk[i + 1] - C_left * Uk_next[I]
     mB[i - 1] = B_left
     mA[i - 1] = A_left
   else:
   RHS[i - 1] = A_right * Uk[i - 1] + B_right * Uk[i] + C_right * Uk[i + 1]
     mB[i - 1] = B_left
     mA[i - 1] = A_left
     mC[i - 1] = C_left
# Now we have equation A*U_{k+1} = RHS
```

Now we have matrix equation $M_L U_{k+1} = RHS$ with square matrix and we solve this equation using SOR method according to Section 1.2.

```
# 3
# SOR
while (error > EPS) and (m < MAXITER):
  error = 0
  for i in range(1, I):
    y = (RHS[i - 1] - mA[i - 1] * Uk_next[i - 1] - mC[i - 1] * Uk_next[i + 1]) / mB[i - 1]
    diff = y - Uk_next[i]
    error = error + diff * diff
    Uk_next[i] = Uk_next[i] + omega * diff
    # early exercise opportunity
    if american:
      Uk_next[i] = max(Uk_next[i], Payoff[i])
  m = m + 1
#
Uk[1:I - 1] = Uk_next[1:I - 1]
```

The program "CrankNicolson.py" prints the price of an American option with given parameters of the option, and the program "Main.py" prints as price of an American option with different methods such as explicit finite difference, fully implicit finite difference and Crank-Nicolson finite difference.