Chapter 5 - Finite difference methods

Modelling Derivatives in C++

1 Explicit difference methods

Rather than using three nodes to represent price changes - why not create $2N_j + 1$ nodes instead? Consider the BS PDE:

$$\begin{aligned} \frac{\partial f}{\partial t} + (r-q)S\frac{\partial f}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 f}{\partial S^2} &= rf \\ x &:= ln(S) \\ u(x,t) &:= e^{r(T-t)} f(e^x, t) \\ \frac{1}{2}\sigma^2 \frac{\partial^2 u}{\partial x^2} + \mu \frac{\partial u}{\partial x} &= -\frac{\partial \mu}{\partial t} \end{aligned}$$

With some discretisation and manipulation, we get the following series of equations:

$$\begin{aligned} \alpha &= \frac{\Delta t}{(\Delta x)^2} \\ \beta &= \frac{\mu \Delta t}{\Delta x} \\ \tilde{p}_u &= \frac{1}{2}(\sigma^2 \alpha + \beta) \\ \tilde{p}_m &= 1 - \sigma^2 \alpha \\ \tilde{p}_d &= \frac{1}{2}(\sigma^2 \alpha - \beta) \\ f_{i,j} &= e^{-r(T-t_i)} u_{i,j} = e^{-r\Delta t}(\tilde{p}_u f_{i+1,j+1} + \tilde{p}_m f_{i+1,j} + \tilde{p}_d f_{i+1,j-1}) \end{aligned}$$

2 Explicit finite-difference implementation

This is done in the Python code implementation.

3 Implicit difference method

In this section we assess how $2N_j + 1$ scenarios can be presented in a tridiagonal matrix form of linear difference equations, to be solved using an LU decomposition method. Boundary conditions will need to be set, and the resultant computation is quite complex and tedious to be written. This can be presented more easily in a Python implementation.

4 LU decomposition method

Similar as above.

5 Implicit difference method implementation

Similar as above.

6 Object-oriented finite-difference implementation

An object-oriented implementation needs to be carefully designed - as very large array sizes need to be allocated to implement large grids in practice. In Python, numpy arrays are usually the best and easiest way to implement this. Though we would also have to create classes for tridiagonal operators to implement the implicit difference method, and the boundary class to set boundaries for systems of linear equations as well. Finite difference models and options are fed through separate classes. A final step condition class allows for computation along the lattice.

A segment of this is replicated in Python code - however, we swap out some custom built structures with numpy arrays.

7 Iterative methods

An iterative method - unlike the LU decompositions provide initial guesses of the solution of the SDE. It is slower to implement - but it is easier to write and generalise to more complex derivatives.

For example, the successive overrelaxation method (SOR) relies on the fact that $u_{i+1,j}$ can be written as:

$$u_{i+1,j} = \frac{1}{1+2\alpha}(b_{i,j} + \alpha(u_{i+1,j-1} + u_{i+1,j+1})$$

And guesses can be made for $u_{i+1,j}$ until the above satisfies the following condition:

$$\sum_{j=N+1}^{N-1} (u_{i+1,j}^{k+1} - u_{i+1,j}^{k})^2 < \epsilon$$

For a certain $\alpha > 0$. A simple SOR technique is provided in the Python code.

8 Crank-Nicolson scheme

The Crank-Nicolson scheme improves upon the stability and convergence restrictions on explicit and implicit finite-difference schemes. Which produces another set of linear equations (not written here due to simplicity) that can be solved using the LU decomposition method.

9 Alternating direction implicit method

For multiple state variables (i.e., multiple underlyings), the methods mentioned above can be extremely computationally heavy. The alternating direction implicit method allows for a split of a PDE with multiple securities into state variables that alternate between each other in a forward and backward implementation. This reduces the computational cost, and stability of the method.