MATH60082
Lab Class 6

Contents

6.1 Explicit Finite Difference For Option Pricing

In this example we are going to price a European call option with explicit finite difference.

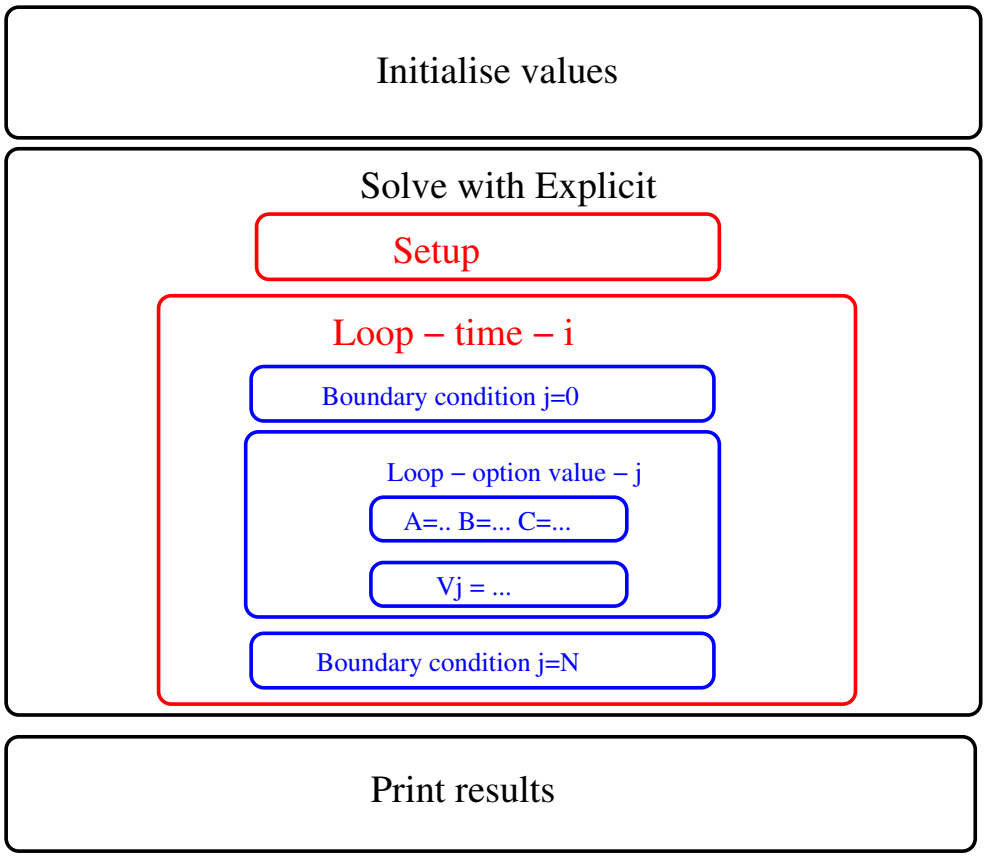
Initial Setup

For the explicit method we shall need:

- All parameters for the option, such as X and S_0 etc.
- The number of divisions in stock, $jMax$, and divisions in time $iMax$
- The size of the divisions $\Delta S = S_{max}/jMax$ and $\Delta t = T/iMax$
- A vector to store the stock prices, and two to store the option values at the current and previous time level.

It will make things easier if you just declare these at the top of the program before you start doing anything with them.

The program structure will look something like this:



Use the code below to as a template to get started.

CLICK HERE TO DOWNLOAD FULL CODE

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <vector>
#include <algorithm>
using namespace std;

/* Template code for the Explicit Finite Difference
 */
```

Inside the template the comments list the rough layout of your program. First declare the parameters for the problem, local grid variables and the vectors required. Set the values of the parameters to the following: $\sigma = 0.4$, $r = 0.05$, $X = 2$, $T = 1$, and $iMax = jMax = 4$. Next we need to note that in option pricing problems we must for S over the semi inifinite domain, therefore numerically we need to choose an appropriate S_{max}. In general, you should set this according to mathematical reasoning if you want your code to be generic and work in all cases. Often we just choose $S_{max} = K \times X$ where K is some constant (say 5) but it should really be linked to the probability distribution of the underlying asset. For simplicity here we choose $S_{max} = 2X$ so that small grids still work, for any decent level of accuracy S_{max} will need to set much higher. Next assuming we have set the correct level

of storage to your vector, calculate the values of ΔS and Δt for your grid and using them to assign initial values to the stock price vector, and the option value vectors.

$$S_j = j\Delta S,$$

$$V_j = payoff(S_j).$$

It is a good idea at this stage to output the stock values and option values to check that the results are as you would expect them to be. The lines added to our template might look something like this

CLICK HERE TO DOWNLOAD FULL CODE

```
1     // declare and initialise Black Scholes parameters
2     double S0=100.,X=100.,T=1.,r=0.06,sigma=0.2;
3     // declare and initialise grid paramaters
4     int iMax=4,jMax=4;
5     // declare and initialise local variables (ds,dt)
6     double S_max=2*X;
7     double dS=S_max/jMax;
8     double dt=T/iMax;
9     // create storage for the stock price and option price (old and new)
10    vector<double> S(jMax+1),vOld(jMax+1),vNew(jMax+1);
11    // setup and initialise the stock price
12    for(int j=0;j<=jMax;j++)
13    {
14        S[j] = j*dS;
15    }
16    // setup and initialise the final conditions on the option price
17    for(int j=0;j<=jMax;j++)
18    {
19        vOld[j] = max(S[j]-X,0.);
20        vNew[j] = max(S[j]-X,0.);
21        cout << iMax << " " << j << " " << S[j] << " " << vNew[j] << " " <<
              vOld[j] << endl;
22    }
```

Timestep Calculation

Next we must setup a loop to count backwards through time, and at each timestep another loop to go through all stock price values (except the boundaries). We wish to use two time levels of storage for V, which we shall call `vNew` and `vOld`. Here let V^i be represented by the vector `vNew`, and V^{i+1} be represented by the vector `vOld`. Then at the end of each timestep we must overwrite the old values with the new ones, please refer to the solution at the end to see this in action. This allows us to move recursively through time with only two levels of storage. Since we can often have timestep constraints, it can be extremely inefficient to store all levels of time.

Firstly add in your code the timestep loop over $i = iMax - 1$ down through to $i = 0$ to step backwards through time. Now inside the loop, you must first implement the boundary

condition at $j = 0$. For example, for a call we may write

$$V_0^i = 0,$$

and remember that `vNew`$= V^i$. Next add in the condition at $j = jMax$, for a call we may write

$$V_{jMax}^i = S_{jMax} - Xe^{-r(T-i\Delta t)}.$$

Check at this stage that you are implementing the boundary conditions correctly, so output them to screen. Your code for the loop may look like

CLICK HERE TO DOWNLOAD FULL CODE

```
1     for(int i=iMax-1;i>=0;i--)
2     {
3         // apply boundary condition at S=0
4         vNew[0] = 0.;
5         cout << i << " " << 0 << " " << S[0] << " " << vNew[0] << " " << vOld
              [0] << endl;
6         // apply boundary condition at S=S_max
7         vNew[jMax] = S[jMax] - X*exp(-r*(T-i*dt));
8         cout << i << " " << jMax << " " << S[jMax] << " " << vNew[jMax] << "
              " << vOld[jMax] << endl;
9     }
```

Now because we are only using five nodes in space and five nodes in time, this will allow us to check our calculations by hand. Make sure you are checking the values outputed from your code at every step.

Next we can put a loop in between the boundary conditions to calculate for the values $1 \leq j \leq jMax - 1$. Inside the loop, to make things easier follow the notation from the notes (click here, pg 65) and declare variables to store the coefficients A, B and C. Then simply calculate the values of A, B and C for the current value of j, and write

$$V_j^i = \frac{1}{1 + r\Delta t}(AV_{j+i}^{i+1} + BV_j^{i+1} + CV_{j-i}^{i+1}).$$

Again output the values of the code and check against the solution in figure 1. You **must** remember to update the old values to new by putting a line `vOld=vNew;` at the end of the time loop.

Once all code is checked and working you should run grid checks (change the grid size and check the effect on the solution) to satisfy yourself that the code is correct. You will notice that if you increase $jMax$ by a factor of 10 you will need to increase $iMax$ by a factor of 100 to maintain stability. Please refer to the course notes and lectures for more details on this (click here).

Tasks

6.1 Complete the coding for the explicit finite difference method with $iMax = 4$ and $jMax = 4$.

6.2 Try varying the value of $iMax$ and $jMax$, what happens to the solution if $iMax$ is not very large compared to $jMax$?

6.3 Check the stability condition holds, can you set an appropriate value of $iMax$ given an input value for $jMax$?

6.4 Plot out the solution $V(S = X, t = 0)$ for different values of $jMax$, what does it look like?

6.5 Increase $jMax$ by multiples of 2 starting at $jMax = 10$, can you estimate the convergence rate?

6.6 Change $S_{\max}$ to $5X$ or $10X$ – what effect does this have on the results?

6.7 Compare the efficiency of the method to that of binomial trees from last week.

Figure 1: Solution for given parameters.

The following code generates the results found in figure 1 and at the end you have estimates for the option price at time $t = 0$ for a discrete set of values in S. The value estimates might not though correspond to the value of S_0 you need the option price at, so we will need to look at interpolation in the next section.

CLICK HERE TO DOWNLOAD FULL CODE

```
1    for(int j=1;j<=jMax-1;j++)
2    {
3      double A,B,C;
4      A=0.5*sigma*sigma*j*j*dt+0.5*r*j*dt;
5      B=1.-sigma*sigma*j*j*dt;
6      C=0.5*sigma*sigma*j*j*dt-0.5*r*j*dt;
7      vNew[j] = 1./(1.+r*dt)*(A*vOld[j+1]+B*vOld[j]+C*vOld[j-1]);
8      cout << i << " " << j << " " << S[j] << " " << vNew[j] << " " << vOld[j
           ] << endl;
9    }
```

6.2 Interpolation

Example - Linear Interpolation

We show here a simple example of how to take the code detailed above and output a value for the option at a given stock price using linear interpolation. We use a style here which allows for a more generic higher order Lagrange polynomial approximation to be derived. We assume that we have the set of points S_j and $vNew_j$ and we wish to find the value of the interpolated function $V(S)$. The linear approximation says that for points $j*$ and $j*+1$ we should choose

$$V(S) = \frac{S - S_{j*+1}}{S_{j*} - S_{j*+1}} vNew_{j*} + \frac{S - S_{j*}}{S_{j*+1} - S_{j*}} vNew_{j*+1}$$

There are three stages:

(1) Given the value S_0, find j^* such that $S_0 \in [j * \Delta S, (j*+1)\Delta S]$

CLICK HERE TO DOWNLOAD FULL CODE

```
1    // return the value of the option at S0
2    int jstar;
3    jstar = S0/dS;
```

(2) Evaluate the Lagrange polynomial at $S = S_0$

CLICK HERE TO DOWNLOAD FULL CODE

```
    double sum=0.;
    sum = sum + (S0 - S[jstar+1])/(S[jstar]-S[jstar+1])*vNew[jstar];
    sum = sum + (S0 - S[jstar])/(S[jstar+1]-S[jstar])*vNew[jstar+1];
    cout << " V(S="<<S0<<") = " << sum << endl;
    /* OUTPUT
     * V(S=1.639) = 0.194858
     */
```

(3) Test and debug the code!!! You can check that the polynomial above is the equation of a line passing through the points $(S_{j^*}, vNew_{j^*})$ and $(S_{j^*+1}, vNew_{j^*+1})$. Try different values of S_0 and check they look ok.

Once you are convinced the code is working move everything out into a function. You will now be able to pass in S_0 as an argument to the function, simply return the value from the interpolation.

The final code might look like this

CLICK HERE TO DOWNLOAD FULL CODE

```
int main()
{
  // declare and initialise Black Scholes parameters
  double S0=1.639,X=2.,T=1.,r=0.05,sigma=0.4;
  // declare and initialise grid paramaters
  int iMax=4,jMax=4;
  cout << explicitCallOption(S0,X,T,r,sigma,iMax,jMax) << endl;
  /* OUTPUT
0.194858
   */
}
```

Tasks

6.8 Run the code with linear interpolation.

6.9 Setting an appropriate value for $iMax$ each time, plot out the solution $V(S0 = 1.639, t = 0)$ for different values of $jMax$.

6.10 Increase $jMax$ by multiples of 2 starting at $jMax = 10$, can you estimate the convergence rate?

6.11 Try implementing a higher order interpolation, does this affect the convergence rate?