MATH60082
Example Sheet 6
Explicit Finite Difference

Dr P Johnson

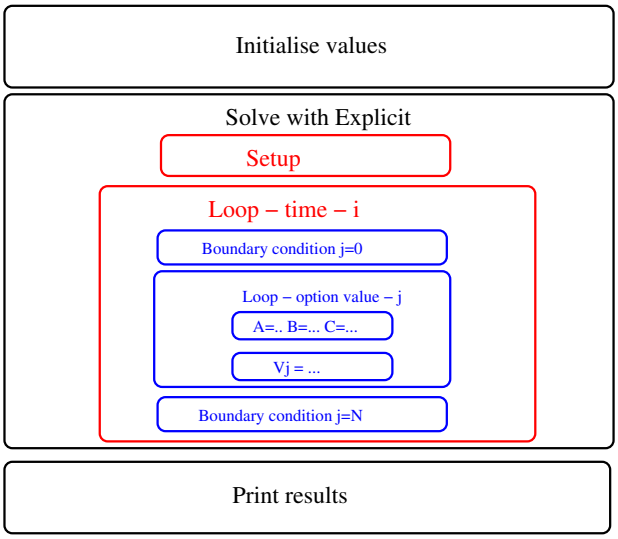
Initial Setup

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:



The list above should give you an idea of what you need to initialise at the start of the program. The setup stage involves assigning the correct level of storage to your vector (if not already done inside the declaration), then setup the values ΔS and Δt for your grid and use them to assign initial values to the stock price vector, and the option value vectors.

$$S_j = j\Delta S,$$

$$V_j = payoff(S_j).$$

Exercises

6.1 Create a code with parameter storage as outlined above, choosing $\sigma = 0.4$, $r = 0.05$, $X = 2$, $dS = 1$, $T = 1$, $dt = 0.25$ and $iMax = jMax = 4$.

6.2 Use vectors to store stock prices (called `S`) and option values (called `vold` and `vnew`) and update them with their initial values.

6.3 Write stock value and option value to a file "initial-conditions.csv". Check that the results are as you would expect them to be by plotting them in excel.

Timestep Calculation

Next we must setup two loops to count backwards through time, and at each timestep through all stock price values (except the boundaries). Now as stated earlier we wish to use two time levels of storage for V, which we shall call V_{new} and V_{old}. Here let V^i be represented by the vector V_{new}, and V^{i+1} be represented by the vector V_{old}. 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.

Exercises

6.4 In your code, write two loops over `i` and `j`, to move backwards through time and through all stock prices (except the boundaries). See binomial code for an example of how this may be done. For your first attempt, use only five nodes in space and five nodes in time, this will allow you to check calculations by hand.

6.5 Using a variable for time t, initialised with $t = T$, update time at each timestep and print to screen. Check that time starts at T and finishes at 0.

6.6 Now inside the time step loop, you must first implement the boundary condition at $j = 0$. For example, for a put we may write

$$V_0^i = Xe^{-r(T-t_i)},$$

and remember that $V_{new} = V^i$.
Then we may declare variables to store the coefficients multiplying V as A, B and C. Now inside the loop through stock price values, we wish to find what V_j is. Then simply calculate the values of A, B and C for the current value of j, then write

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

Finally implement the boundary condition at $j = n$. **Always** remember to set the old values equal to new at the end of the timestep. You may wish to print out the value of the option at each timestep to check what is going on.

6.7 Once all code is checked and working try to run grid checks (change the grid size and check the effect on the solution) to satisfy yourself that the code is correct.

Solution

Example explicit method on a European call option:

	i=0		i=1		i=2		i=3		i=4	
	V^{old}	V^{new}	V^{old}	V^{new}	V^{old}	V^{new}	V^{old}	V^{new}	V^{old}	V^{new}
j=4	2.0975	2.0975	2.0736	2.0736	2.0494	2.0494	2.0248	2.0248	2	2
j=3	1.1396	1.1396	1.0991	1.0991	1.0597	1.0597	1.0247	1.0247	1	1
j=2	0.2979	0.2979	0.2375	0.2375	0.1694	0.1694	0.0914	0.0914	0	0
j=1	0.0125	0.0125	0.0066	0.0066	0.0024	0.0024	0	0	0	0
j=0	0	0	0	0	0	0	0	0	0	0

sigma = 0.4 r=0.05 X=2 dS=1 T=1 dt=0.25 n=4

Interpolation

Here we shall go through a simple example on interpolation. We wish to interpolate to find the value of the function $y(x)$ (which has no formula) at an arbitrary position a given the following data:

i	x_i	$y(x_i)$
0	0	1.90246
1	0.4	1.50246
2	0.8	1.10507
3	1.2	0.739343
4	1.6	0.454018
5	2	0.262869
6	2.4	0.146872
7	2.8	0.0799558
8	3.2	0.0414204
9	3.6	0.0175786
10	4	0

- The first task is to write a program and input the data here into two vectors x and y.
- Then we wish to write a function `interpolate` that will take the two vectors, along with the value a and an integer to specify precision as input, then return the interpolated value $y(x = a)$.
- The method we shall use to interpolate is a simple method using a Lagrange polynomial.

Prototype for the function

The function prototype should look like:

```
double interpolate(vector<double>& x,vector<double>& y,double a,int degree)
{
// interpolate in here...
}
```

where x and y are vectors containing the data, a is the position at we wish to evaluate the function, and degree is the number of points that we wish to use.

Generating a Lagrange polynomial

The Lagrange polynomial is defined as the linear combination:

$$L(a) = \sum_{j=0}^{n} y_j l_j(a)$$

where the polynomial l is defined by:

$$l_j(a) = \Pi_{i=0, i\neq j}^{n} \frac{a - x_i}{x_j - x_i}$$

Finding where to interpolate

The function is to be split into two parts:

- finding the value of x_i closest to a,
- interpolating using the closest points to a

The idea here is that we can let degree be *smaller* than the size of the vectors x and y.

In order to find which points to use in the interpolation, let us assume that (as is the case above, and when interpolating finite difference grids)

$$x_i = i\Delta x.$$

The value of Δx here is just the difference between any two nodes in x. Then given the value a, we know that there exists some i^* such that

$$a \geq i^*\Delta x, \quad \text{and} \quad a < (i^*+1)\Delta x$$

To find i^* with using C++ we can simply say

```
int istar;
istar = a/dx;
```

However, if we wish to find the *closest* node to a then write

```
int istar;
istar = int(a/dx+0.5);
```

Try this out to see what the subtle difference between the two is.

Example - Linear Interpolation

Given the protoype for the function as given above, we shall choose degree equal to 2 as an example. There are two stages:

(1) Given the value a, find i^* (use the method to find the node below)

```
int istar;
istar = a/dx;
```

(2) Evaluate the Lagrange polynomial at $x = a$

```
double sum=0.;
sum = sum + (a - x[istar+1])/(x[istar]-x[istar+1])*y[istar];
sum = sum + (a - x[istar])/(x[istar+1]-x[istar])*y[istar+1];
return sum;
```

(3) Test and debug the code!!! You can check that the polynomial above is the equation of a line passing through the points (x_{i^*}, y_{i^*}) and (x_{i^*+1}, y_{i^*+1}).

Exercises

6.8 Test and implement the example on interpolation given here.

6.9 After testing the code for the linear case, see if you can write the code in loops (see wiki) to be extendible to higher degrees.

6.10 Depending on whether degree is odd or even, it will be more appropriate to choose the nearest node or the node below a. Think about which one is needed in each case and why.

6.11 Write the function for the general case and test it with degree equal to 4. The results above are from the European put example from the previous tutorial on Crank-Nicolson. Set $a = 1.8$ and see how accurately the results agree with the Black-Scholes formula.

Be careful! Don't choose degree too high. It should never need to be greater than 5.