MATH60082
Lab Class 7

Contents

7.1 Example - Simple SOR 1
7.2 Crank-Nicolson Method 4
7.3 Direct methods 10

7.1 Example - Simple SOR

Consider the matrix equation

$$A\mathbf{x} = \mathbf{d}.$$

where

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 1 \end{pmatrix}, \qquad \mathbf{x} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \qquad \text{and} \qquad \mathbf{d} = \begin{pmatrix} 1 \\ \frac{1}{4} \\ \frac{1}{2} \\ 0 \end{pmatrix}$$

Open the example code on my website:- CLICK HERE TO DOWNLOAD FULL CODE

```
1 /*
2  *  A simple example of the sor method for you to fill in in class.
3  *  The ideas here are easily extendable to Finite Difference methods
4  */
```

This code uses SOR to find the solution x. It assumes that the matrix can be rewritten with diagonal elements as vectors, so that

$$A = \begin{pmatrix} b_0 & c_0 & & \\ a_1 & b_1 & c_1 & \\ & a_2 & b_2 & c_2 \\ & & a_2 & b_2 \end{pmatrix}$$

for vectors a, b and c.

First copy the code into a new project, check it compiles and runs. Update your guess for x by rearranging each line of the matrix equation to get an equation for x_0, x_1, x_2 and x_3. For example

$$x_0 = (d_0 - c_0 v_1)/b_0$$

Put the four expressions for each value of x into the code at the appropriate place. See my implementation here:

Click here to download full code

```
1      x[0]=(d[0] - c[0]*x[1])/b[0];
2      x[1]=(d[1] - a[1]*x[0] - c[1]*x[2])/b[1];
3      x[2]=(d[2] - a[2]*x[1] - c[2]*x[3])/b[2];
4      x[3]=(d[3] - a[3]*x[2])/b[3];
```

Tasks

7.1 Run the code, does the solution converge?

7.2 Try outputing the error by summing up the residuals. There is one for each equation, for the first one it is given by

$$r_0 = d_0 - b_0 x_0 - c_0 x_1$$

Can you see the residual going down as your guess approaches the solution?

7.3 Now try to use the parameter ω to overrelax your new guess with the formula

$$x_j^{q+1} = x_j^q + \omega(x_j^{q+1} - x_j^q)$$

Can you choose ω to speed up convergence?

Solution

The value of the solution is

$$x = \begin{pmatrix} 1 \\ 1\frac{3}{4} \\ 2\frac{1}{4} \\ 2\frac{1}{4} \end{pmatrix}.$$

See my solution for SOR in a function:-

Click here to download full code

```
1 void sorSolve(const std::vector<double> &a,const std::vector<double> &b,const
      std::vector<double> &c,const std::vector<double> &rhs,
```

```
2                    std::vector<double> &x, int iterMax, double tol, double omega, int
                       &sor )
```

Alternatively you look at a direct solver such as the Thomas Algorithm, for an implementation of this see the following code:- CLICK HERE TO DOWNLOAD FULL CODE

```
1 std::vector<double> thomasSolve(const std::vector<double> &a, const std::
      vector<double> &b_, const std::vector<double> &c, std::vector<double> &d)
```

Tasks

7.4 Create a new code and copy my SOR function and tridag solver function. Check that you can solve the problem.

7.5 Try solving the problem

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & \dots & \dots & \dots \\ -1 & 2 & -1 & 0 & \dots & \dots & \dots \\ \vdots & \vdots & \vdots & \vdots & \dots & \dots & \dots \\ 0 & \dots & a_j = -1 & b_j = 2 & c_j = -1 & \dots & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \dots & \dots \\ \vdots & \vdots & \vdots & 0 & -1 & 2 & -1 \\ \vdots & \vdots & \vdots & 0 & 0 & -1 & 1 \end{pmatrix}$$

where A is an $(n+1) \times (n+1)$ matrix. The right hand side of the equation is

$$\mathbf{d} = \begin{pmatrix} 1 \\ \frac{1}{2} \\ \vdots \\ d_j = \frac{1}{2} \\ \vdots \\ \frac{1}{2} \\ 0 \end{pmatrix}$$

for different values of n with both SOR and direct solver. Which method is more efficient?

7.2 Crank-Nicolson Method

For the Crank-Nicolson 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 ΔS and Δt
- Vectors to store:
  - stock price
  - old option values
  - new option values
  - three diagonal elements (a, b, and c)
  - the right hand side of the matrix equation

Again, the easiest thing to do is just to declare these at the top of the program before you start doing anything with them.

A sample program structure for the Crank-Nicolson method may look something like what is shown in figure 1, and this general layout is given in the example code (click here). I have initialised the variables and put the loop in to save time. At each timestep we must solve the matrix equation

$$A\mathbf{V} = b.$$

Since the matrix is tridiagonal we need only use three vectors to store all values in the matrix A.

Example - A European Put

Now, assuming that our vectors are indexed from 0 to $iMax$ (and therefore have $iMax + 1$ elements) I will go through an example for a European put with $\sigma = 0.4$, $r = 0.05$, $X = 2$, $dS = 1$, $dt = 0.25$, and $jMax = 4$. Note here that these calculations could have been done by hand, and trying to replicate an example of this scale with a code is a good way to start, and also to check for bugs/errors.

First copy the example code from the web CLICK HERE TO DOWNLOAD FULL CODE

```
1 /* Template code for the Crank Nicolson Finite Difference
2  */
```

check initial values and the initial setup phase to assign values to vectors such as stock values (that remain constant throughout) and the final payoff condition to the option values are all working correctly.

Check that you understand the time loop from the example code, inside the loop the code is broken into two sections, matrix setup and matrix solver, both of which may be written into functions later. The following section will outline with an example how to setup and solve the matrix equations.

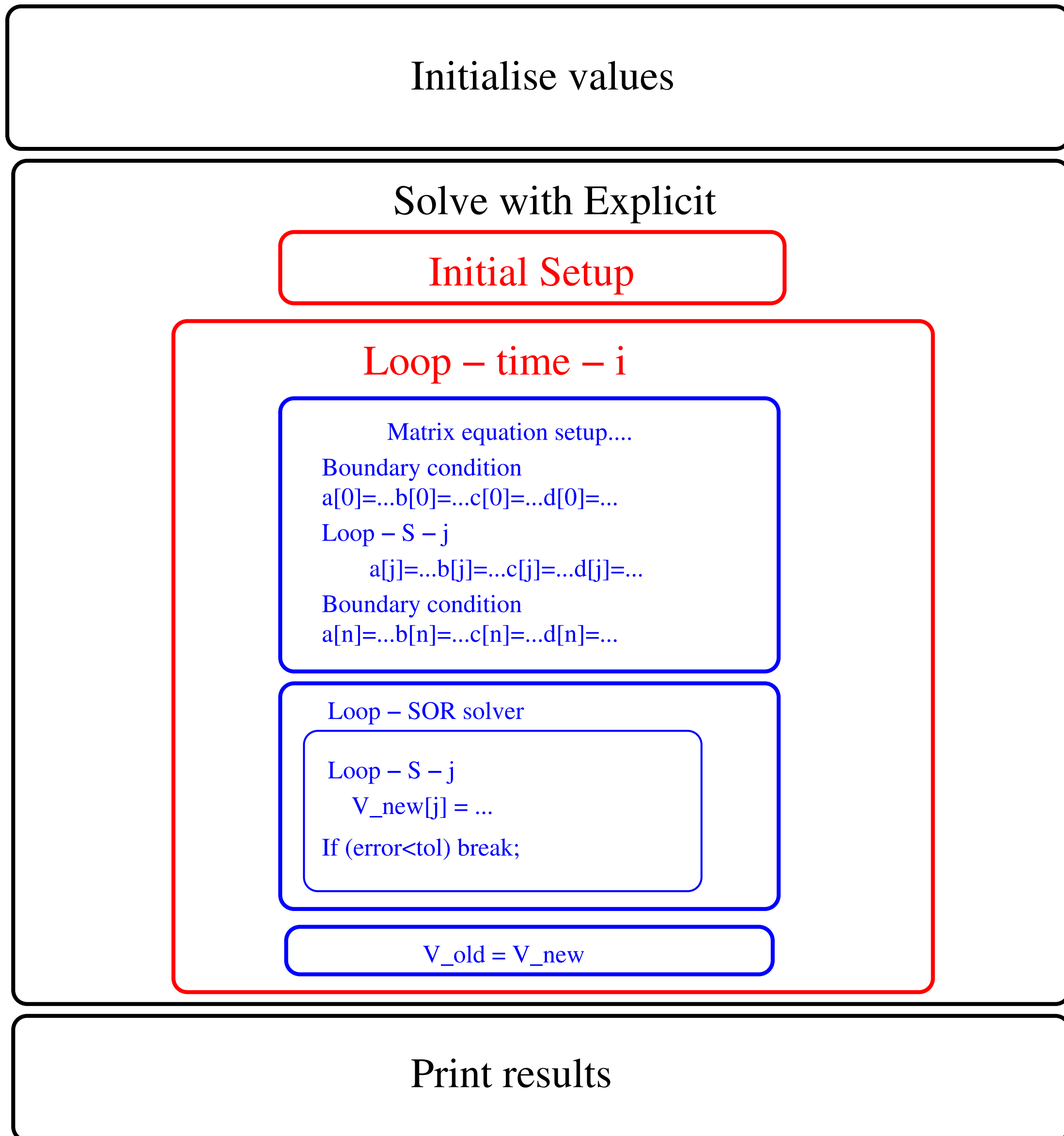

Figure 1: The program structure for a Crank-Nicolson code

Matrix Setup:

I tend to use a, b and c to represent the tridagonal matrix A, and d for the right hand side of the equation. Just remember that it makes things easier if you are consistent with your notation. Declare vector storage for the terms a, b, c and d, and initialise them to the same size as `vnew`. Click here to download full code

```
1    // declare vectors for matrix equations
2    vector<double> a(jMax+1),b(jMax+1),c(jMax+1),d(jMax+1);
```

Next setup the boundary condition

$$V(S = 0) = Xe^{-r(T-t)}$$

by writing the code Click here to download full code

```
1    a[0] = 0.;
2    b[0] = 1.;
3    c[0] = 0.;
4    d[0] = // FILL IT IN -- this is boundary at S=0
```

Next write a for loop to assign each of the inner equations. Here we shall let $V_{new}[j] = V_j^i$ and $V_{old}[j] = V_j^{i+1}$. The inner equations may be written

$$a_j V_{j-1}^i + b_j V_j^i + c_j V_{j+1}^i = d_j$$

so inside the loop write something like: Click here to download full code

```
1    for(int j=1;j<=jMax-1;j++)
2    {
3        a[j] = // FILL IT IN -- this term multiplies V\_new[j-1] in the
             equation...
4        b[j] = // FILL IT IN -- this term multiplies  V\_new[j] in the
             equation...}
5        c[j] = // FILL IT IN -- this term multiplies  V\_new[j+1] in the
             equation...}
6        d[j] = // FILL IT IN -- this term will involve V\_old...}
7    }
```

After the loop is finished we still must set up the final boundary condition:
Click here to download full code

```
1    a[jMax] = 0.;
2    b[jMax] = 1.;
3    c[jMax] = 0.;
4    d[jMax] = // FILL IT IN -- this is boundary at S=S_max
```

Now we should have finished setting up the matrix equations, they should look something like (at the first timestep)

j	a[j]	b[j]	c[j]	d[j]
0	0	1	0	1.97516
1	0.0275	-4.105	0.0525	-3.95
2	0.135	-4.345	0.185	-0.135
3	0.3225	-4.745	0.3975	-0
4	0	1	0	0

A solution to generate this is given here:- CLICK HERE TO DOWNLOAD FULL CODE

```
1       // output matrix
2       for(int j=0;j<=jMax;j++)
3       {
4           cout << j << " " << a[j] << " " << b[j] << " " << c[j] << " " <<
                d[j] << endl;
5       }
```

Matrix Solver

Still inside the time loop, we now need to solve the matrix equation. This can be done using either iterative methods (SOR) or direct methods (Thomas's solver). We shall go through the SOR method here as it is more easily adapted to American options.

The method is as follows:-

1 update the value of V_j^i for each j using the SOR formula

2 check whether the residual is less than the tolerance:

- Yes - exit as we have found the solution
- No - go back to step one

The SOR Loop

Create a loop in your code after the matrix has been setup, to iterate with SOR until a solution is found. It is a good idea to check whether or not convergence has been achieved, so we store a variable `sor` so that we know whether or not `iterMax` iterations have been reached. CLICK HERE TO DOWNLOAD FULL CODE

```
1       // solve matrix equations with SOR
2       int sor,iterMax=10000;
3       double tol = 1.e-8,omega=1.;
4       for(sor=0;sor<iterMax;sor++)
5       {
6           // SOR equations in here
7
8           // calculate residual
9
10          // check for convergence and exit loop if converged
11
12      }
```

Here `iterMax` is the maximum number of iterations you permit (may need to be as high as 10000). The variable `tol` is the small number we check for convergence against. The variable `omega` can be used to over-relax and converge quicker to a solution.

Like the simple example from before, you need to write an equation to update the the value at the boundary $j = 0$, then a loop of equations to update all values in between $1 \leq j < jMax$ and then another at the boundary $j = jMax$. Your code might look like this: CLICK HERE TO DOWNLOAD FULL CODE

```
1          // SOR equations in here
2          {
3              double y = (d[0] - c[0]*vNew[1])/b[0];
4              vNew[0] = vNew[0] + omega*(y-vNew[0]);
5              cout << " ( " << vNew[0] << " , ";
6          }
7          for(int j=1;j<jMax;j++)
8          {
9              double y = (d[j] - a[j]*vNew[j-1] - c[j]*vNew[j+1])/b[j];
10             vNew[j] = vNew[j] + omega*(y-vNew[j]);
11             cout << vNew[j] << " , ";
12         }
13         {
14             double y = (d[jMax] - a[jMax]*vNew[jMax-1])/b[jMax];
15             vNew[jMax] = vNew[jMax] + omega*(y-vNew[jMax]);
16             cout << vNew[jMax] << " )\n";
17         }
```

After sufficient iterations the value of V_{new} and V_{old} should be:

i	j	V_old[j]	V_new[j]
3	0	2	1.97516
3	1	1	0.976261
3	2	0	0.061581
3	3	0	0.00418543
3	4	0	0

If you don't want to be waiting around a long time, you need to calculate residual and exit the loop when converged. This might look like this:- CLICK HERE TO DOWNLOAD FULL CODE

```
1          // calculate residual
2          double error=0.;
3          error += fabs(d[0] - b[0]*vNew[0] - c[0]*vNew[1]);
4          for(int j=1;j<jMax;j++)
5              error += fabs(d[j] - a[j]*vNew[j-1] - b[j]*vNew[j] - c[j]*
                   vNew[j+1]);
6          error += fabs(d[jMax] - a[jMax]*vNew[jMax-1] - b[jMax]*vNew[jMax
               ]);
7          // check for convergence and exit loop if converged
8          if(error<tol)
```

```
9             break;
```

All that is left to do now is to check convergence, and set the old values equal to the new and you're done!! CLICK HERE TO DOWNLOAD FULL CODE

```
1     if(sor==iterMax)
2         cout << "\n NOT CONVERGED \n";
3     cout << " Solved after " << sor << " iterations.\n";
```

The full set of results for the example is below...

i	j	V_old[j]	V_new[j]
3	0	2	1.97516
3	1	1	0.976261
3	2	0	0.061581
3	3	0	0.00418543
3	4	0	0
2	0	1.97516	1.95062
2	1	0.976261	0.954845
2	2	0.061581	0.112606
2	3	0.00418543	0.01471
2	4	0	0
1	0	1.95062	1.92639
1	1	0.954845	0.935397
1	2	0.112606	0.155285
1	3	0.01471	0.0282984
1	4	0	0
0	0	1.92639	1.90246
0	1	0.935397	0.917626
0	2	0.155285	0.191233
0	3	0.0282984	0.0429639
0	4	0	0

Load in the code (unless you have written your own version) from the last section CLICK HERE TO DOWNLOAD FULL CODE

```
1 /* Solution code for the Crank Nicolson Finite Difference
2  */
```

and remove any `cout` statements we might not need, and just output the value of the option at the end.

You can use the linear interpolation function from the last lab class, CLICK HERE TO SEE IT. CLICK HERE TO DOWNLOAD FULL CODE

```
    // output the estimated option price
    double optionValue;
    {
        int jStar=S0/dS;
        double sum=0.;
        sum+=(S0 - S[jStar])/dS * vNew[jStar+1];
        sum+=(S[jStar+1] - S0)/dS * vNew[jStar];
        optionValue = sum;
    }
    cout << "Value of the option V(S="<<S0<<") = " <<  optionValue << "\n";
}
```

Tasks

7.6 Try moving the algorithm into a function of its own, you will need to supply the algorithm with a number of time steps, space steps, S_max, tolerance, omega and maximum iterations in addition to the usual Black-Scholes parameters. At the end return the value of the option at S_0.

7.7 Include the analytic solution for the European put option in your code, now compare the value of the option derived from the Crank-Nicolson method at $V(S = X, t = 0)$. What happens as you change the value of $iMax$ and $jMax$?

7.8 Try varying the other numerical parameters, S_max, tolerance, omega and maximum iterations, can you verify the effect they have on the solution?

7.9 Now choosing an appropriate set of numerical parameters, investigate the convergence rate of the option value, see (click here) for an example of the analysis you might do. I've also written a notebook with more details:- (click here).

7.10 Try implementing an extrapolation using the result of a calculation with successive grids ($iMax = jMax = n$ and $iMax = jMax = 2n$), is the solution more accurate than either of the original values? What happens if $S \neq X$?

7.3 Direct methods

- The direct method (Thomas' algorithm) can be implemented at the solve stage, everything preceeding that may be kept the same

- Since both methods solve **exactly** the same equation, they should both post **exactly** the same results, provided the tolerance in the SOR method is taken small enough.
- Therefore the solution of the European put option can be used to check your iterative method as well as a direct method.

Tasks

7.11 Try rewriting your European option pricing code to use the Thomas algorithm, an example of which is given earlier in this sheet (click here).

7.12 What effect does using the direct method have on the efficiency of your code, i.e. the computation times?