Example - Simple SOR

Consider the matrix equation

$$ Av = d. $$

where

$$ A = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{pmatrix}, \quad v = \begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix} \quad \text{and} \quad d = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} $$

Open the example code on my website [click here]. Use SOR to find the solution $v$.

7.1 Copy the code into a new project, check it compiles and runs.

7.2 Update your guess for $v$ by rearranging each line of the matrix equation to get an equation for $v_0$, $v_1$, and $v_2$. For example

$$ v_0 = (d_0 - b_0v_1 - c_0v_2)/a_0 $$

Put the three expressions for each value of $v$ into the code at the appropriate place.

7.3 Run the code, does the solution converge?

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

$$ r_0 = d_0 - a_0v_0 - b_0v_1 - c_0v_2 $$

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

7.5 Now try to use the parameter $\omega$ to overrelax your new guess with the formula

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

Can you choose $\omega$ to speed up convergence?

Solution

The value of the solution is

$$ v = \begin{pmatrix} 1 \\ 1/2 \\ 1/3 \end{pmatrix} $$

which you could verify by solving the matrix equation directly if you wish.
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 $\Delta S$ and $\Delta t$
- 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

$$AV = b.$$ 

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.

7.6 First copy the example code from the web, 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.

7.7 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.

Matrix Setup

I tend to use $a$, $b$ and $c$ to represent the tridiagonal 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.

7.8 Declare vector storage for the terms $a$, $b$, $c$ and $d$, and initialise them to the same size as $vnew$.

7.9 Carry out the steps as outlined below:
Initialise values

Solve with Explicit

Initial Setup

Loop – time – i

Matrix equation setup....
Boundary condition
a[0]=...b[0]=...c[0]=...d[0]=...
Loop – S = j
a[j]=...b[j]=...c[j]=...d[j]=...
Boundary condition
a[n]=...b[n]=...c[n]=...d[n]=...

Loop – SOR solver

Loop – S = j
V_new[j] = ...
If (error<tol) break;

V_old = V_new

Print results

Figure 1: The program structure for a Crank-Nicolson code
Setup the boundary condition

\[ V(S = 0) = X e^{-r(T-t)} \]

by writing the code

\[
\begin{align*}
    a[0] &= 0. \quad \text{// this term multiplies V\_new[-1] in the equation...} \\
    b[0] &= 1. \quad \text{// this term multiplies V\_new[0] in the equation...} \\
    c[0] &= 0. \quad \text{// this term multiplies V\_new[1] in the equation...} \\
    d[0] &= \quad \text{// fill this in...}
\end{align*}
\]

Next write a for loop to assign each of the inner equations. Here we shall let \( V_{\text{new}}[j] = V_j^i \) and \( V_{\text{old}}[j] = V_j^{i+1} \). 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:

\[
\begin{align*}
    a[j] &= \quad \text{// this term multiplies V\_new[j-1] in the equation...} \\
    b[j] &= \quad \text{// this term multiplies V\_new[j] in the equation...} \\
    c[j] &= \quad \text{// this term multiplies V\_new[j+1] in the equation...} \\
    d[j] &= \quad \text{// this term will involve V\_old...}
\end{align*}
\]

After the loop is finished we still must set up the final boundary condition:

\[
\begin{align*}
    a[j\text{Max}] &= 0. \quad \text{// this term multiplies V\_new[j\text{Max}-1] in the equation...} \\
    b[j\text{Max}] &= 1. \quad \text{// this term multiplies V\_new[j\text{Max}] in the equation...} \\
    c[j\text{Max}] &= 0. \quad \text{// this term multiplies V\_new[j\text{Max}+1] in the equation...} \\
    d[j\text{Max}] &= \quad \text{// fill this in...}
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>j</th>
<th>a[j]</th>
<th>b[j]</th>
<th>c[j]</th>
<th>d[j]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1.97516</td>
</tr>
<tr>
<td>1</td>
<td>0.0275</td>
<td>-4.105</td>
<td>0.0525</td>
<td>-3.95</td>
</tr>
<tr>
<td>2</td>
<td>0.135</td>
<td>-4.345</td>
<td>0.185</td>
<td>-0.135</td>
</tr>
<tr>
<td>3</td>
<td>0.3225</td>
<td>-4.745</td>
<td>0.3975</td>
<td>-0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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.
for(int sor=0;sor<iterMax;sor++)
{
    // implement sor in here...
}

where \texttt{iterMax} is the maximum number of iterations you permit (may need to be as high as 10000). 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 $j=jMax$. After sufficient iterations the value of $V_{\text{new}}$ and $V_{\text{old}}$ should be:

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>$V_{\text{old}}[j]$</th>
<th>$V_{\text{new}}[j]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>2</td>
<td>1.97616</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0.976261</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0.061581</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0.00418543</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

All that is left to do now is to set the old values equal to the new. Done!!

**Direct methods**

- The direct method (Thomas' algorithm) must 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 results below can be used to check your iterative method as well as your direct method.

**Solution**

The full set of results for the example is below...
<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>V_old[j]</th>
<th>V_new[j]</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>2</td>
<td>1.97516</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0.976261</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0.061581</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0.00418543</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1.97516</td>
<td>1.95062</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.976261</td>
<td>0.954845</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0.061581</td>
<td>0.112606</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0.00418543</td>
<td>0.01471</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1.95062</td>
<td>1.92639</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.954845</td>
<td>0.935397</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0.112606</td>
<td>0.155285</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0.01471</td>
<td>0.0282984</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1.92639</td>
<td>1.90246</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0.935397</td>
<td>0.917626</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0.155285</td>
<td>0.191233</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>0.0282984</td>
<td>0.0429639</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>