The Mathematics *of* Financial Derivatives

A Student Introduction

PAUL WILMOTT • SAM HOWISON
JEFF DEWYNNE

8 Finite-difference Methods

8.1 Introduction

Finite-difference methods are a means of obtaining numerical solutions to partial differential equations (as we see in this chapter) and linear complementarity problems (as we see in the following chapter). They constitute a very powerful and flexible technique and, if applied correctly, are capable of generating accurate numerical solutions to all of the models derived in this book, as well as to many other partial differential equations arising in both the physical and financial sciences. Needless to say, in such a brief introduction to the subject as we give here, we can only touch on the basics of finite differences; for more, see *Option Pricing*. Nevertheless, the underlying ideas generalise in a relatively straightforward manner to many more complicated problems.

As we saw in Chapter 5, once the Black–Scholes equation is reduced to the diffusion equation it is a relatively simple matter to find exact solutions (and then convert these back into financial variables). This is, of course, because the diffusion equation is a far simpler and less cluttered equation than the Black–Scholes equation. For precisely this reason also, it is a much simpler exercise to find numerical solutions of the diffusion equation and then, by a change of variables, to convert these into numerical solutions of the Black–Scholes equation, than it is to solve the Black–Scholes equation itself numerically. In this chapter, therefore, we concentrate on solving the diffusion equation using finite-differences. This allows us to introduce the fundamental ideas in as uncluttered an environment as possible.

This is not to say that one should not use finite differences to solve the Black–Scholes equation directly. There are many examples (particularly of multi-factor models) where it is not feasible or even not possible to reduce the problem to a constant coefficient diffusion equation (in one or more dimensions); in this case there is little choice but to use finite differences on the generalisations of the Black–Scholes equation. Direct application of finite-difference methods to the Black–Scholes equation is left to the exercises at the end of the chapter; the reader who has understood the underlying principles of finite differences should have no difficulty with these.

Recall from Sections 5.4 and 7.7.1 that, by using the change of variables (5.9) the Black–Scholes equation (3.9) for any European option can be transformed into the diffusion equation

$$\frac{\partial u}{\partial \tau} = \frac{\partial^2 u}{\partial x^2}.$$

The payoff function for the option determines the initial conditions for $u(x,\tau)$, and the boundary conditions for the option determine the conditions at infinity for $u(x,\tau)$ (that is, as $x \to \pm\infty$); for a put they are given by (3.13), (3.14) and (3.15); for a call they are given by (3.10), (3.11) and (3.12); and for a cash-or-nothing call the payoff is given on pages 38 and 82.

The values of the option $V(S,t)$, in financial variables, may be recovered from the non-dimensional $u(x,\tau)$ using (5.9), yielding

$$V = E^{\frac{1}{2}(1+k)} S^{\frac{1}{2}(1-k)} e^{\frac{1}{8}(k+1)^2\sigma^2(T-t)} u\left(\log(S/E), \tfrac{1}{2}\sigma^2(T-t)\right),$$

where $k = r/\frac{1}{2}\sigma^2$.

8.2 Finite-difference Approximations

The idea underlying finite-difference methods is to replace the partial derivatives occurring in partial differential equations by approximations based on Taylor series expansions of functions near the point or points of interest. For example, the partial derivative $\partial u/\partial\tau$ may be defined to be the limiting difference

$$\frac{\partial u}{\partial \tau}(x,\tau) = \lim_{\delta\tau \to 0} \frac{u(x,\tau+\delta\tau) - u(x,\tau)}{\delta\tau}.$$

If, instead of taking the limit $\delta\tau \to 0$, we regard $\delta\tau$ as nonzero but small, we obtain the approximation

$$\frac{\partial u}{\partial \tau}(x,\tau) \approx \frac{u(x,\tau+\delta\tau) - u(x,\tau)}{\delta\tau} + O(\delta\tau). \qquad (8.1)$$

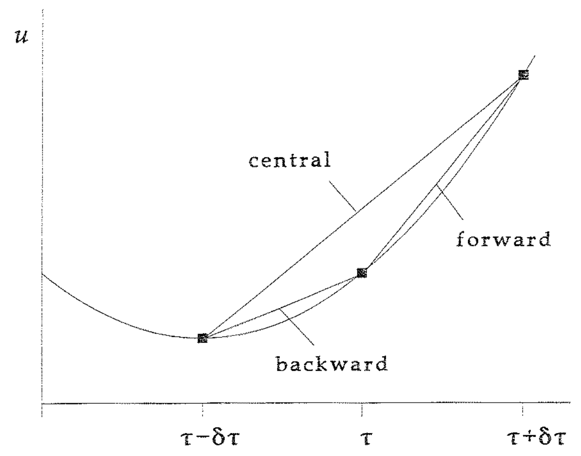

Figure 8.1 Forward-, backward- and central-difference approximations. The slopes of the lines are approximations to the tangent at (x,τ).

This is called a **finite-difference approximation** or a **finite difference** of $\partial u/\partial\tau$ because it involves small, but not infinitesimal, differences of the dependent variable u. This particular finite-difference approximation is called a **forward difference**, since the differencing is in the forward τ direction; only the values of u at τ and $\tau+\delta\tau$ are used. As the $O(\delta\tau)$ term suggests, the smaller $\delta\tau$ is, the more accurate the approximation.[1]

We also have

$$\frac{\partial u}{\partial \tau}(x,\tau) = \lim_{\delta\tau \to 0} \frac{u(x,\tau) - u(x,\tau-\delta\tau)}{\delta\tau},$$

so that the approximation

$$\frac{\partial u}{\partial \tau}(x,\tau) \approx \frac{u(x,\tau) - u(x,\tau-\delta\tau)}{\delta\tau} + O(\delta\tau) \qquad (8.2)$$

is likewise a finite-difference approximation for $\partial u/\partial\tau$. We call this finite-difference approximation a **backward difference**.

We can also define **central differences** by noting that

$$\frac{\partial u}{\partial \tau}(x,\tau) = \lim_{\delta\tau \to 0} \frac{u(x,\tau+\delta\tau) - u(x,\tau-\delta\tau)}{2\,\delta\tau}.$$

[1] The $O(\delta\tau)$ term arises from a Taylor series expansion of $u(x,\tau+\delta\tau)$ about (x,τ); see Exercises 1–3.

This gives rise to the approximation

$$\frac{\partial u}{\partial \tau}(x,\tau) \approx \frac{u(x,\tau+\delta\tau)-u(x,\tau-\delta\tau)}{2\,\delta\tau} + O\Big((\delta\tau)^2\Big). \tag{8.3}$$

Figure 8.1 shows a geometric interpretation of these three types of finite differences. Note that central differences are more accurate (for small $\delta\tau$) than either forward or backward differences; this is also suggested by Figure 8.1. (For the analysis of the accuracy of finite-difference approximations, see Exercises 1–3 at the end of the chapter.)

When applied to the the diffusion equation, forward- and backward-difference approximations for $\partial u/\partial\tau$ lead to **explicit** and **fully implicit** finite-difference schemes, respectively. Central differences of the form (8.3) are never used in practice because they always lead to bad numerical schemes (specifically, schemes that are inherently unstable). Central differences of the form

$$\frac{\partial u}{\partial \tau} \approx \frac{u(x,\tau+\delta\tau/2)-u(x,\tau-\delta\tau/2)}{\delta\tau} + O\Big((\delta\tau)^2\Big) \tag{8.4}$$

arise in the **Crank–Nicolson** finite-difference scheme.

We can define finite-difference approximations for the x-partial derivative of u in exactly the same way. For example, the central finite-difference approximation is easily seen to be[2]

$$\frac{\partial u}{\partial x}(x,\tau) \approx \frac{u(x+\delta x,\tau)-u(x-\delta x,\tau)}{2\,\delta x} + O\Big((\delta x)^2\Big). \tag{8.5}$$

For second partial derivatives, such as $\partial^2 u/\partial x^2$, we can define a symmetric finite-difference approximation as the forward difference of backward-difference approximations to the first derivative or as the backward difference of forward-difference approximations to the first derivative. In either case we obtain the **symmetric central-difference** approximation

$$\frac{\partial^2 u}{\partial x^2}(x,\tau) \approx \frac{u(x+\delta x,\tau)-2u(x,\tau)+u(x-\delta x,\tau)}{(\delta x)^2} + O\Big((\delta x)^2\Big). \tag{8.6}$$

Although there are other approximations, this approximation to $\partial^2 u/\partial x^2$ is preferred, as its symmetry preserves the reflectional symmetry of the second partial derivative; it is left invariant by reflections of the form $x \mapsto -x$. It is also more accurate than other similar approximations.

[2] Although central differences of the form (8.3) are never used for τ- or t-partial derivatives, differences of the form (8.5) for x- or S-partial derivatives are used; see, for example, Exercises 9 and 13.

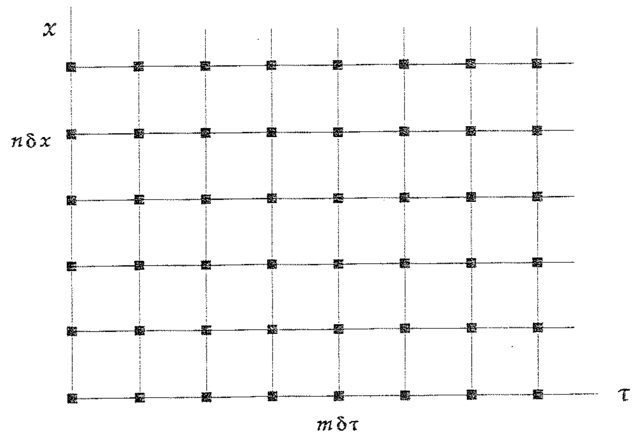

Figure 8.2 The mesh for a finite-difference approximation.

8.3 The Finite-difference Mesh

To continue with the finite-difference approximation to the diffusion equation we divide the x-axis into equally spaced **nodes** a distance δx apart, and the τ-axis into equally spaced nodes a distance $\delta\tau$ apart. This divides the (x,τ) plane into a mesh, where the **mesh points** have the form $(n\,\delta x, m\,\delta\tau)$; see Figure 8.2. We then concern ourselves only with the values of $u(x,\tau)$ at mesh points $(n\,\delta x, m\,\delta\tau)$; see Figure 8.3. We write

$$u_n^m = u(n\,\delta x, m\,\delta\tau) \tag{8.7}$$

for the value of $u(x,\tau)$ at the mesh point $(n\,\delta x, m\,\delta\tau)$.

8.4 The Explicit Finite-difference Method

Consider the general form of the transformed Black–Scholes model for the value of a European option,

$$\frac{\partial u}{\partial \tau} = \frac{\partial^2 u}{\partial x^2},$$

with boundary and initial conditions

$$\begin{gathered} u(x,\tau) \sim u_{-\infty}(x,\tau), \quad u(x,\tau) \sim u_{\infty}(x,\tau) \quad \text{as} \quad x \to \pm\infty \\ u(x,0) = u_0(x). \end{gathered} \tag{8.8}$$

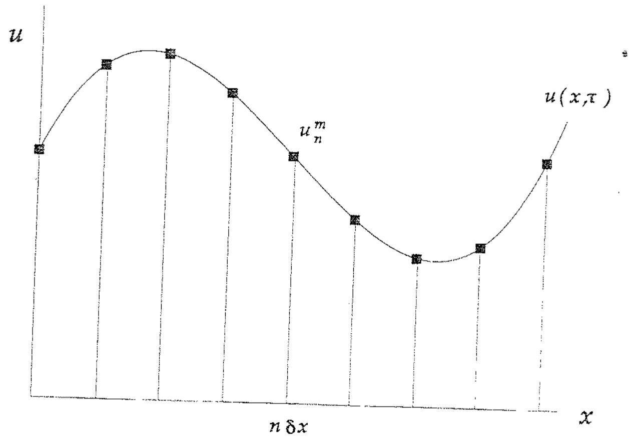

Figure 8.3 The finite-difference approximation at a fixed time-step.

We use the notation $u_{-\infty}(\tau)$, $u_{\infty}(\tau)$ and $u_0(x)$ to emphasise that the following does not in any way depend on the particular boundary and initial conditions involved. (For puts, calls and cash-or-nothing calls these are given as above.)

Confining our attention to values of u at mesh points, and using a forward difference for $\partial u/\partial\tau$, equation (8.1), and a symmetric central difference for $\partial^2 u/\partial x^2$, equation (8.6), we find that the diffusion equation becomes

$$\frac{u_n^{m+1}-u_n^m}{\delta\tau}+O(\delta\tau)=\frac{u_{n+1}^m-2u_n^m+u_{n-1}^m}{(\delta x)^2}+O((\delta x)^2). \qquad (8.9)$$

Ignoring terms of $O(\delta\tau)$ and $O\big((\delta x)^2\big)$, we can rearrange this to give the difference equations

$$u_n^{m+1}=\alpha u_{n+1}^m+(1-2\alpha)u_n^m+\alpha u_{n-1}^m \qquad (8.10)$$

where

$$\alpha=\frac{\delta\tau}{(\delta x)^2}. \qquad (8.11)$$

(Note that, whereas (8.9) is exact, albeit vague about the error terms, (8.10) is only approximate.)

If, at time-step m, we know u_n^m for all values of n we can explicitly calculate u_n^{m+1}. This is why this method is called explicit. Note that u_n^{m+1} depends only on u_{n+1}^m, u_n^m and u_{n-1}^m, as shown in Figure 8.4. This

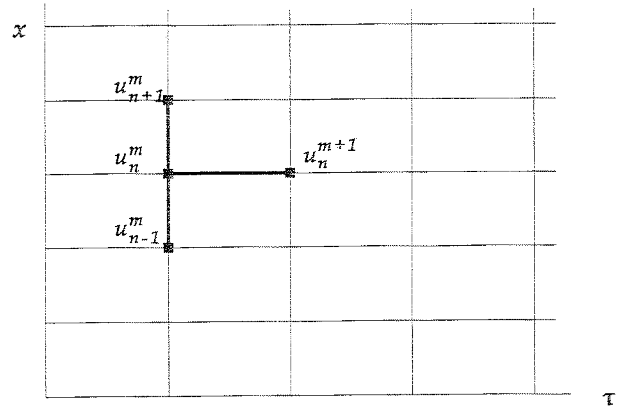

Figure 8.4 Explicit finite-difference discretisation.

figure also suggests that (8.10) may be considered as a random walk on a regular lattice, where u_n^m denotes the probability of a marker being at position n at time-step m, and α denotes the probability of it moving to the right or left by one unit and $(1-2\alpha)$ is the probability of it staying put.

If we choose a constant x-spacing δx, we cannot solve the problem for all $-\infty < x < \infty$ without taking an infinite number of x-steps. We get around this problem by taking a finite, but suitably large, number of x-steps. We restrict our attention to the interval

$$N^-\delta x \le x \le N^+\delta x$$

where $-N^-$ and N^+ are large positive integers.

To obtain the finite-difference solution for the option price, we divide the non-dimensional time to expiry of the option, $\frac{1}{2}\sigma^2 T$, into M equal time-steps so that

$$\delta\tau=\tfrac{1}{2}\sigma^2 T/M.$$

We then solve the difference equations (8.10) for $N^- < n < N^+$ and $0 < m \le M$, and use the boundary conditions from (8.8) to determine $u_{N^+}^m$ and $u_{N^-}^m$:

$$\begin{aligned} u_{N^-}^m &= u_{-\infty}(N^-\delta x, m\,\delta\tau), \quad 0<m\le M,\\ u_{N^+}^m &= u_{\infty}(N^+\delta x, m\,\delta\tau), \quad 0<m\le M. \end{aligned} \qquad (8.12)$$

```
explicit_fd( values,dx,dt,M,Nplus,Nminus )
{
  a = dt/(dx*dx);

  for( n=Nminus; n<=Nplus; ++n )
    oldu[n] = pay_off( n*dx );

  for( m=1; m<=M; ++m )
  {
    tau = m*dt;

    newu[Nminus] = u_m_inf( Nminus*dx,tau );
    newu[ Nplus] = u_p_inf(  Nplus*dx,tau );

    for( n=Nminus+1; n<Nplus; ++n )
      newu[n] = oldu[n]
              + a*(oldu[n-1]-2*oldu[n]+oldu[n+1]);

    for( n=Nminus; n<=Nplus; ++n )
      oldu[n] = newu[n];
  }

  for( n=Nminus; n<=Nplus; ++n )
    values[n] = oldu[n];
}
```

Figure 8.5 Pseudo-code for the explicit finite-difference solution of the diffusion equation. The variables are `a` $= \alpha$, `tau` $= \tau$, `Nminus` $= N^-$, `Nplus` $= N^+$. The values of u_n^m are stored in the array `oldu[]`, and the values of u_n^{m+1} are stored in the array `newu[]`. Initially the values of u_n^0 are put in `oldu[]`. Once all of the u_n^{m+1} are found they are copied back into `oldu[]` and the process is repeated until all the required time-steps have been completed. The numerical solution is copied into `values[]` and returned to the calling program.

To start the iterative procedure we use the initial condition from (8.8):

$$u_n^0 = u_0(n\,\delta x), \quad N^- \le n \le N^+. \tag{8.13}$$

As the equations determining u_n^{m+1} in terms of the u_n^m are explicit, this process can be easily coded for a computer to solve; a pseudo-code is given in Figure 8.5.

In Figure 8.6 we compare explicit finite-difference solutions for a European put with the exact Black–Scholes formula (note that we have transformed back into financial variables using (5.9)). We have deliberately chosen to regard α and $\delta\tau$ as variable, rather than the more obvious choice of δx and $\delta\tau$, to illustrate an extremely important point. When $\alpha = 0.25$ and $\alpha = 0.5$ there is good agreement between computed

S	$\alpha = 0.25$	$\alpha = 0.50$	$\alpha = 0.52$	exact
0.00	9.7531	9.7531	9.7531	9.7531
2.00	7.7531	7.7531	7.7531	7.7531
4.00	5.7531	5.7531	5.7531	5.7531
6.00	3.7532	3.7532	2.9498	3.7532
7.00	2.7567	2.7567	−17.4192	2.7568
8.00	1.7986	1.7985	95.3210	1.7987
9.00	0.9879	0.9879	350.5603	0.9880
10.00	0.4418	0.4419	625.0347	0.4420
11.00	0.1605	0.1607	−457.3122	0.1606
12.00	0.0483	0.0483	−208.9135	0.0483
13.00	0.0124	0.0123	40.5813	0.0124
14.00	0.0028	0.0027	−15.2150	0.0028
15.00	0.0006	0.0005	−3.1582	0.0006
16.00	0.0001	0.0001	0.7365	0.0001

Figure 8.6 Comparison of exact Black–Scholes solution and explicit finite-difference solutions for a European put with $E = 10$, $r = 0.05$, $\sigma = 0.20$ and with six months to expiry. Note the effect of taking $\alpha > \frac{1}{2}$.

and exact solutions, whereas when $\alpha = 0.52$, the computed solution is nonsensical. This illustrates the **stability problem** for explicit finite differences.

The stability problem arises because we are using *finite precision computer arithmetic* to solve the difference equations (8.10). This introduces rounding errors into the *numerical* solution of (8.10). The system (8.10) is said to be **stable** if these rounding errors are not magnified at each iteration. If the rounding errors do grow in magnitude at each iteration of the solution procedure, then (8.10) is said to be **unstable**.

It can be shown (see Exercise 5 at the end of the chapter) that the system (8.10) is:

- stable if $0 < \alpha \le \frac{1}{2}$ (**stability condition**);
- unstable if $\alpha > \frac{1}{2}$ (**instability condition**).

If we regard the explicit finite-difference equations in terms of a random walk, instability corresponds to the presence of negative probabilities (specifically, the probability of a marker staying put, $1 - 2\alpha$, is negative).

The stability condition puts severe constraints on the size of time-steps. For stability we must have

$$0 < \frac{\delta\tau}{(\delta x)^2} \leq \tfrac{1}{2}.$$

Thus if we start with a stable solution on a mesh and double the number of x-mesh points, for example to improve accuracy, we must quarter the size of the time-step. Each time-step then takes twice as long (twice as many x-mesh spacings) and there are four times as many time-steps. Thus, doubling the number of x-mesh points means that finding the solution takes eight times as long.

It can be shown that the numerical solution of the finite-difference equations converges to the exact solution of the diffusion equation as $\delta x \to 0$ and $\delta\tau \to 0$, in the sense that

$$u_n^m \to u(n\,\delta x, m\,\delta\tau),$$

if and only if the explicit finite-difference method is stable; the proof of this is beyond the scope of this text and we refer the reader to *Option Pricing*.

Note that because of the method's independence from the initial and boundary conditions, the explicit finite-difference method is easily adapted to deal with more general binary and barrier options (see Exercise 7).

8.5 Implicit Finite-difference Methods

Implicit finite-difference methods are used to overcome the stability limitations imposed by the restriction $0 < \alpha \leq \frac{1}{2}$, which applies to the explicit method. Implicit methods allow us to use a large number of x-mesh points without having to take ridiculously small time-steps.

Implicit methods require the solution of *systems* of equations. We consider the techniques of LU decomposition and SOR for solving numerically these systems. By using these techniques implicit methods may be made almost as efficient as the explicit method in terms of arithmetical operations per time-step.[3] As fewer time-steps need to be taken, implicit finite-difference methods are usually more efficient overall than explicit methods. We shall consider both the fully implicit and Crank–Nicolson methods.

[3] In their most efficient forms explicit and implicit methods require $O(2N)$ and $O(4N)$ arithmetical operations per time-step, respectively, where N is the number of x-grid points.

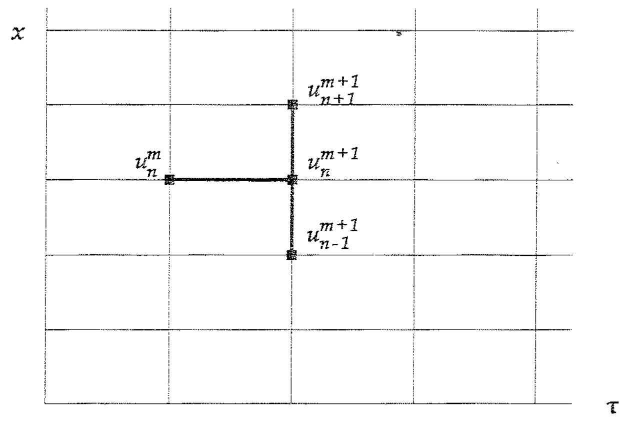

Figure 8.7 Implicit finite-difference discretisation.

8.6 The Fully-implicit Method

The fully-implicit finite-difference scheme, which is usually known as the **implicit finite-difference** method, uses the backward-difference approximation (8.2) for the $\partial u/\partial\tau$ term and the symmetric central-difference approximation (8.6) for the $\partial^2 u/\partial x^2$ term. This leads to the equation

$$\frac{u_n^{m+1} - u_n^m}{\delta\tau} + O(\delta\tau) = \frac{u_{n+1}^{m+1} - 2u_n^{m+1} + u_{n-1}^{m+1}}{(\delta x)^2} + O\left((\delta x)^2\right),$$

where we are using the same notation as in the previous sections. Again, we neglect terms of $O(\delta\tau)$, $O((\delta x)^2)$ and higher and, after some rearrangement, we find the implicit finite-difference equations

$$-\alpha u_{n-1}^{m+1} + (1+2\alpha)u_n^{m+1} - \alpha u_{n+1}^{m+1} = u_n^m. \qquad (8.14)$$

As before, the space-step and the time-step are related through the parameter α, defined by (8.11). In the implicit finite-difference equation (8.14), u_n^{m+1}, u_{n-1}^{m+1} and u_{n+1}^{m+1} all depend on u_n^m in an *implicit* manner; the new values cannot immediately be separated out and solved for explicitly in terms of the old values. The scheme is illustrated in Figure 8.7.

Let us consider the European option problem discussed in the previous sections. We assume that we can truncate the infinite mesh at $x = N^-\delta x$ and $x = N^+\delta x$, and take N^- and N^+ sufficiently large so that no significant errors are introduced. As before we find the u_n^0 using (8.13) and $u_{N^-}^{m+1}$ and $u_{N^+}^{m+1}$ using (8.12). The problem is then to find the u_n^{m+1} for $m \geq 0$ and $N^- < n < N^+$ from (8.14).

We can write (8.14) as the linear system

$$\begin{pmatrix} 1+2\alpha & -\alpha & 0 & \cdots & 0 \\ -\alpha & 1+2\alpha & -\alpha & & 0 \\ 0 & -\alpha & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & -\alpha \\ 0 & 0 & & -\alpha & 1+2\alpha \end{pmatrix} \begin{pmatrix} u_{N^-+1}^{m+1} \\ \vdots \\ u_0^{m+1} \\ \vdots \\ u_{N^+-1}^{m+1} \end{pmatrix}$$

$$= \begin{pmatrix} u_{N^-+1}^{m} \\ \vdots \\ u_0^{m} \\ \vdots \\ u_{N^+-1}^{m} \end{pmatrix} + \alpha \begin{pmatrix} u_{N^-}^{m+1} \\ 0 \\ \vdots \\ 0 \\ u_{N^+}^{m+1} \end{pmatrix} = \begin{pmatrix} b_{N^-+1}^{m} \\ \vdots \\ b_0^{m} \\ \vdots \\ b_{N^+-1}^{m} \end{pmatrix}, \quad \text{say.} \qquad (8.15)$$

The vector on the right-hand side of the middle term in this equation arises from the end equations, for example

$$(1+2\alpha)u_{N^+-1}^{m+1} - \alpha u_{N^+-2}^{m+1} = u_{N^+-1}^{m} + \alpha u_{N^+}^{m+1}.$$

We can write (8.15) in the more compact form

$$\mathbf{M}\boldsymbol{u}^{m+1} = \boldsymbol{b}^m \qquad (8.16)$$

where $\boldsymbol{u}^{m+1}$ and $\boldsymbol{b}^m$ denote the $(N^+ - N^- - 1)$-dimensional vectors

$$\boldsymbol{u}^{m+1} = (u_{N^-+1}^{m+1}, \ldots, u_{N^+-1}^{m+1}), \quad \boldsymbol{b}^m = \boldsymbol{u}^m + \alpha(u_{N^-}^{m+1}, 0, 0, \ldots, 0, u_{N^+}^{m+1}),$$

and $\mathbf{M}$ is the $(N^+ - N^- - 1)$-square symmetric matrix given in (8.15).

It can be shown that, for $\alpha \geq 0$, $\mathbf{M}$ is invertible and so in principle

$$\boldsymbol{u}^{m+1} = \mathbf{M}^{-1}\boldsymbol{b}^m, \qquad (8.17)$$

where $\mathbf{M}^{-1}$ is the inverse of $\mathbf{M}$. We can therefore find $\boldsymbol{u}^{m+1}$ given $\boldsymbol{b}^m$, which in turn may be found from $\boldsymbol{u}^m$ and the boundary conditions. As the initial condition determines $\boldsymbol{u}^0$, we can find each $\boldsymbol{u}^{m+1}$ sequentially.

8.6.1 Practical Considerations

In practice there are far more efficient solution techniques than matrix inversion. The matrix $\mathbf{M}$ has the property that it is **tridiagonal**; that

is, only the diagonal, super-diagonal and sub-diagonal elements are non-zero. This has a number of important consequences.

First, it means that we do not have to store all the zeros, just the non-zero elements. The inverse of $\mathbf{M}$, $\mathbf{M}^{-1}$, is not tridiagonal and requires a great deal more storage.[4]

Second, the tridiagonal structure of $\mathbf{M}$ means that there are highly efficient algorithms for solving (8.16) in $O(N)$ arithmetic operations per solution (specifically, about $4N$ operations). We now discuss two of these algorithms, **LU decomposition** and **SOR**.

8.6.2 The LU Method

In LU decomposition we look for a decomposition of the matrix $\mathbf{M}$ into a product of a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$, namely $\mathbf{M} = \mathbf{LU}$, of the form

$$\begin{pmatrix} 1+2\alpha & -\alpha & 0 & \cdots & 0 \\ -\alpha & 1+2\alpha & -\alpha & & \vdots \\ 0 & -\alpha & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & -\alpha \\ 0 & \cdots & 0 & -\alpha & 1+2\alpha \end{pmatrix} =$$

$$\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ \ell_{N^-+1} & 1 & \ddots & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \ell_{N^+-2} & 1 \end{pmatrix} \begin{pmatrix} y_{N^-+1} & z_{N^-+1} & 0 & \cdots & 0 \\ 0 & y_{N^-+2} & \ddots & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & z_{N^+-2} \\ 0 & \cdots & 0 & 0 & y_{N^+-1} \end{pmatrix}. \qquad (8.18)$$

In order to determine the quantities ℓ_n, y_n and z_n (and observe that these have only to be calculated once) we simply multiply together the two matrices on the right-hand side of (8.18) and equate the result to the left-hand side. After some simple manipulation we find that

$$\begin{aligned} y_{N^-+1} &= (1+2\alpha), \\ y_n &= (1+2\alpha) - \alpha^2/y_{n-1}, \quad n = N^- + 2, \ldots, N^+ - 1, \\ z_n &= -\alpha, \quad \ell_n = -\alpha/y_n, \quad n = N^- + 1, \ldots, N^+ - 2. \end{aligned} \qquad (8.19)$$

[4] If N is the dimension of the system, storing $\mathbf{M}^{-1}$ requires N^2 real numbers, whereas storing the non-zero elements of $\mathbf{M}$ requires $3N - 2$. Furthermore, the most efficient means of inverting $\mathbf{M}$ requires $O(N^2)$ operations, and the matrix multiplication required to find $\mathbf{M}^{-1}\boldsymbol{b}^m$ requires a further $O(N^2)$ operations.

This also shows that the only quantities we need to calculate and save are the y_n, $n = N^- + 1, \ldots, N^+ - 1$.

The original problem $\mathbf{M}\boldsymbol{u}^{m+1} = \boldsymbol{b}^m$ can be written as $\mathbf{L}(\mathbf{U}\boldsymbol{u}^{m+1}) = \boldsymbol{b}^m$, which may then be broken down into two simpler subproblems,

$$\mathbf{L}\boldsymbol{q}^m = \boldsymbol{b}^m, \quad \mathbf{U}\boldsymbol{u}^{m+1} = \boldsymbol{q}^m,$$

where $\boldsymbol{q}^m$ is an intermediate vector. Thus, having eliminated the ℓ_n from the lower triangular matrix and the z_n from the upper triangular matrix using (8.19), the solution procedure is to solve the two subproblems

$$\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ -\dfrac{\alpha}{y_{N^-+1}} & 1 & 0 & & \vdots \\ 0 & -\dfrac{\alpha}{y_{N^-+2}} & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & -\dfrac{\alpha}{y_{N^+-2}} & 1 \end{pmatrix} \begin{pmatrix} q^m_{N^-+1} \\ q^m_{N^-+2} \\ \vdots \\ q^m_{N^+-2} \\ q^m_{N^+-1} \end{pmatrix} = \begin{pmatrix} b^m_{N^-+1} \\ b^m_{N^-+2} \\ \vdots \\ b^m_{N^+-2} \\ b^m_{N^+-1} \end{pmatrix} \tag{8.20}$$

and

$$\begin{pmatrix} y_{N^-+1} & -\alpha & 0 & \cdots & 0 \\ 0 & y_{N^-+2} & -\alpha & & \vdots \\ 0 & 0 & \ddots & \ddots & 0 \\ \vdots & & \ddots & y_{N^+-2} & -\alpha \\ 0 & \cdots & 0 & 0 & y_{N^+-1} \end{pmatrix} \begin{pmatrix} u^{m+1}_{N^-+1} \\ u^{m+1}_{N^-+2} \\ \vdots \\ u^{m+1}_{N^+-2} \\ u^{m+1}_{N^+-1} \end{pmatrix} = \begin{pmatrix} q^m_{N^-+1} \\ q^m_{N^-+2} \\ \vdots \\ q^m_{N^+-2} \\ q^m_{N^+-1} \end{pmatrix}. \tag{8.21}$$

The intermediate quantities q^m_n are easily found by forward substitution. We can read off the value of $q^m_{N^-+1}$ directly, while any other equation in the system relates only q^m_n and q^m_{n-1}. If we solve the system in increasing n-indicial order, we have q^m_{n-1} available at the time we have to solve for q^m_n. Thus we can find q^m_n easily:

$$q^m_{N^-+1} = b^m_{N^-+1}, \quad q^m_n = b^m_n + \frac{\alpha q^m_{n-1}}{y_{n-1}}, \quad n = N^-+2, \ldots, N^+-1. \tag{8.22}$$

Similarly, solving (8.21) for the u^m_n (given that we have found the intermediate q^m_n) is easily achieved by backward substitution. This time it is $u^{m+1}_{N^+-1}$ that can be read off directly, and if we solve in decreasing

```
lu_find_y( y,a,Nminus,Nplus )
{
  asq = a*a;
  y[Nminus+1] = 1+2*a;

  for( n=Nminus+2; n<Nplus; ++n )
  {
    y[n] = 1+2*a - asq/y[n-1];
    if (y[n]==0) return(SINGULAR);
  }
  return( OK );
}

lu_solver( u,b,y,a,Nminus,Nplus )
{
/* Must call lu_find_y before using this */

  q[Nminus+1] = b[Nminus+1];

  for( n=Nminus+2; n<Nplus; ++n )
    q[n] = b[n]+a*q[n-1]/y[n-1];

  u[Nplus-1] = q[Nplus-1]/y[Nplus-1];

  for( n=Nplus-2; n>Nminus; --n )
    u[n] = (q[n]+a*u[n+1])/y[n];
}
```

Figure 8.8 Pseudo-code for LU tridiagonal solver. Variables are `a` $= \alpha$, `asq` $= \alpha^2$, `Nplus` $= N^+$, `Nminus` $= N^-$, `b[n]` $= b^m_n$, `q[n]` $= q^m_n$, `u[n]` $= u^{m+1}_n$, `y[n]` $= y_n$. The routine solves the problem only for `Nminus` $+ 1 \le$ `n` $\le$ `Nplus` $- 1$; the values at `Nplus` and `Nminus` are assumed to have been set by the calling routine. The calling routine must call `lu_find_y` before it calls `lu_solver` in order to set up `y[]`.

n-indicial order we can find all of the u^m_n in an equally simple manner:

$$u^{m+1}_{N^+-1} = \frac{q^m_{N^+-1}}{y_{N^+-1}}, \quad u^{m+1}_n = \frac{q^m_n + \alpha u^{m+1}_{n+1}}{y_n}, \quad n = N^+ - 2, \ldots, N^- + 1. \tag{8.23}$$

Equations (8.19), (8.22) and (8.23) define the LU algorithm:

- find the y_n using (8.19); [5]
- given the vector $\boldsymbol{b}^m$, use (8.22) to find the vector $\boldsymbol{q}^m$;
- use (8.23) to find the required solution $\boldsymbol{u}^{m+1}$.

[5] Note that these depend only on the matrix $\mathbf{M}$ and not on $\boldsymbol{b}^m$. We propose solving the system $\mathbf{M}\boldsymbol{u}^{m+1} = \boldsymbol{b}^m$ for many time-steps. We have, however, to find the y_n once only; they are the same for each time-step.

The algorithm is illustrated by the pseudo-code in Figure 8.8. (Note that the ideas above can also be applied to quite general tridiagonal systems where the super-, sub- and diagonal elements vary with position in the matrix. Such matrices arise if the implicit method is used directly on the Black–Scholes equation; see Exercise 14.)

8.6.3 The SOR Method

The LU method discussed in the previous section is a *direct* method for solving the system (8.16) in that it aims to find the unknowns exactly and in one pass. An alternative strategy is to employ an *iterative* method. Iterative methods differ from direct methods in that one starts with a guess for the solution and successively improves it until it converges to the exact solution (or is near enough to the exact solution). In a direct method, one obtains the solution without any iteration. An advantage of iterative methods over direct methods is that they generalise in straightforward ways to American option problems and nonlinear models involving transaction costs, whereas direct methods do not. Another advantage is that they are easier to program. On the other hand, a disadvantage of iterative methods is that, in the context of European option problems, they are somewhat slower than direct methods.[6]

The acronym **SOR** stands for Successive Over-Relaxation, and the SOR algorithm is an example of an iterative method. It is a refinement of another iterative method known as the **Gauss–Seidel** method, which in turn is a development of the **Jacobi** method. It is easiest to explain the SOR method by first describing these other two related but simpler methods. All three iterative methods rely on the fact that the system (8.14) (or 8.15 or (8.16)) may be written in the form

$$u_n^{m+1} = \frac{1}{1+2\alpha}\left(b_n^m + \alpha(u_{n-1}^{m+1} + u_{n+1}^{m+1})\right). \tag{8.24}$$

This equation is simply a rearrangement of either of (8.14), (8.15) or (8.16) that isolates the diagonal terms in the problem on the left-hand side of the equation.

The idea behind the Jacobi method is to take some initial guess for u_n^{m+1} for $N^- + 1 \le n \le N^+ - 1$ (a good initial guess is the values of

[6] The LU algorithm described in the previous section requires about $4N$ operations per time-step. The SOR algorithm described in this section requires $4N \times$ (number of iterations) per time-step. Typically the number of iterations is of the order of two or three.

u from the previous step, i.e. u_n^m) and substitute these into the right-hand side of (8.24) to obtain a new guess for u_n^{m+1} (from the left-hand side). The process is then repeated until the approximations cease to change (or cease to change significantly). When this happens we have the solution.

Formally we can define the **Jacobi** method as follows. Let $u_n^{m+1,k}$ be the k-th iterate for u_n^{m+1}. Thus, the initial guess is denoted by $u_n^{m+1,0}$ and as $k \to \infty$ we expect $u_n^{m+1,k} \to u_n^{m+1}$. Then, given $u_n^{m+1,k}$, we calculate $u_n^{m+1,k+1}$ using a modified form of (8.24), namely,

$$u_n^{m+1,k+1} = \frac{1}{1+2\alpha}\left(b_n^m + \alpha(u_{n-1}^{m+1,k} + u_{n+1}^{m+1,k})\right), \quad N^- < n < N^+. \tag{8.25}$$

The whole process is repeated until a measure of the error such as

$$\|\boldsymbol{u}^{m+1,k+1} - \boldsymbol{u}^{m+1,k}\|^2 = \sum_n \left(u_n^{m+1,k+1} - u_n^{m+1,k}\right)^2$$

becomes sufficiently small for us to regard any further iterations as unnecessary; then we take $u_n^{m+1,k+1}$ as the value of u_n^{m+1}. The method is known to converge to the correct solution for any value of $\alpha > 0$, although a full discussion of the convergence of the algorithm is beyond the scope of this text.

The **Gauss–Seidel** method is a development of the Jacobi method. It relies on the fact that when we come to calculate $u_n^{m+1,k+1}$ in (8.25) we have already found $u_{n-1}^{m+1,k+1}$. In the Gauss–Seidel method we use this value instead of $u_{n-1}^{m+1,k}$. Thus, the Gauss–Seidel method is identical to the Jacobi method, except that (8.25) is replaced by

$$u_n^{m+1,k+1} = \frac{1}{1+2\alpha}\left(b_n^m + \alpha(u_{n-1}^{m+1,k+1} + u_{n+1}^{m+1,k})\right), \quad N^- < n < N^+. \tag{8.26}$$

The difference may be summarised by saying that, in the Gauss–Seidel method, we use an updated guess immediately when it becomes available, while in the Jacobi method we use updated guesses only when they are *all* available. One practical consequence of using the most recent information (i.e. $u_n^{m+1,k+1}$ rather than $u_n^{m+1,k}$) is that the Gauss–Seidel method converges more rapidly than the Jacobi method and is therefore more efficient. In fact, the Gauss–Seidel method is even more efficient, as the updating is achieved by overwriting old iterates, whereas in the Jacobi method the old and new iterates have to be stored separately until all the new iterates are found (and then copied over the old iterates).

```
SOR_solver( u,b,Nminus,Nplus,a,omega,eps,loops )
{
  loops = 0;
  do
  {
    error = 0.0;
    for( n=Nminus+1; n<Nplus; ++n )
    {
      y = ( b[n]+a*(u[n-1]+u[n+1]) )/(1+2*a);
      y = u[n]+omega*(y-u[n]);
      error += (u[n]-y)*(u[n]-y);
      u[n]=y;
    }
   ++loops;
  }
  while ( error > eps );
  return(loops);
}
```

Figure 8.9 Pseudo-code for SOR algorithm for a European option problem. Here a $= \alpha$, Nplus $= N^+$, Nminus $= N^-$, b[n] $= b_n^m$, u[n] $= u_n^{m+1,k}$ or $u_n^{m+1,k+1}$, y $= y_n^{m+1,k+1}$, omega $= \omega$ and eps is the desired error tolerance. The routine over-writes old iterates as soon as a new iterate is generated; thus for a given value of n in the loop, u[n $-$ 1], u[n $-$ 2], and so forth, will contain $u_{n-1}^{m+1,k+1}$, $u_{n-2}^{m+1,k+1}$, etc., whereas u[n $+$ 1], u[n $+$ 2], and so forth, will contain $u_{n+1}^{m+1,k}$, $u_{n+2}^{m+1,k}$, etc. The algorithm is considered to have converged once the sum over n of the squares of the difference between $u_n^{m+1,k+1}$ and $u_n^{m+1,k}$ is less than the error tolerance eps. At this point the array u[ ] contains the SOR solution for u_n^{m+1}. Note that the routine solves the problem only for Nminus $+ 1 \le$ n $\le$ Nplus $- 1$; the values at Nplus and Nminus are assumed to have been set by the calling routine. The routine returns the number of iterations executed in loops; this is so the calling routine may adjust omega to minimise the number of iterations.

Again, it can be shown that the Gauss–Seidel algorithm converges to the correct solution if α is positive, although we do not show it here.

The **SOR** algorithm is a refinement of the Gauss–Seidel method. We begin with the (seemingly trivial) observation that

$$u_n^{m+1,k+1} = u_n^{m+1,k} + (u_n^{m+1,k+1} - u_n^{m+1,k}).$$

As the sequence of iterates $u_n^{m+1,k}$ is intended to converge to u_n^{m+1} as $k \to \infty$, we can think of $(u_n^{m+1,k+1} - u_n^{m+1,k})$ as a correction term to be added to $u_n^{m+1,k}$ to bring it nearer to the exact value of u_n^{m+1}. The possibility then arises that we might be able to get the sequence to converge more rapidly if we over-correct; this is true if the sequence of iterates $u_n^{m+1,k} \to u_n^{m+1}$ monotonically as k increases, rather than

oscillating; this is the case for both Gauss–Seidel and SOR. That is, we put

$$\begin{aligned} y_n^{m+1,k+1} &= \frac{1}{1+2\alpha}\left(b_n^m + \alpha(u_{n-1}^{m+1,k+1} + u_{n+1}^{m+1,k})\right) \\ u_n^{m+1,k+1} &= u_n^{m+1,k} + \omega(y_n^{m+1,k+1} - u_n^{m+1,k}), \end{aligned} \tag{8.27}$$

where $\omega > 1$ is the over-correction or **over-relaxation** parameter. (Note that the term $y_n^{m+1,k+1}$ is the value that the Gauss–Seidel method would give for $u_n^{m+1,k+1}$; in SOR we view $y_n^{m+1,k+1} - u_n^{m+1,k}$ as a correction to be made to $u_n^{m+1,k}$ in order to obtain $u_n^{m+1,k+1}$.) It can be shown that the SOR algorithm converges to the correct solution of (8.14) or (8.16) if $\alpha > 0$ and provided $0 < \omega < 2$. (When $0 < \omega < 1$ the algorithm is referred to as under-relaxation rather than over-relaxation, which is used for cases where $1 < \omega < 2$.) It can be shown that there is an optimal value of ω, in the interval $1 < \omega < 2$, which leads to much more rapid convergence than other values of ω. This optimal value of ω depends on the dimension of the matrix involved and, more generally, on the details of the matrix involved. There are means of calculating or estimating the optimal value of ω, but typically these involve so many calculations that it is quicker to change ω at each time-step until a value is found that minimises the number of iterations of the SOR loop. In Figure 8.9 we give the SOR algorithm for the fully implicit finite-difference equations for the diffusion equation.

8.6.4 The Implicit Finite-difference Algorithm

The implicit finite-difference solution scheme is to solve (8.16) (or (8.14) or (8.15)) for each time-step using either the LU solver routine described in Section (8.6.2) or the SOR algorithm described in the previous section. This allows us to time-step through and calculate the current value of the option. The algorithm using the LU method is illustrated in Figure 8.10, and the algorithm using the SOR method is illustrated in Figure 8.11.

In Figure 8.12 we compare implicit finite-difference solutions for a European put with a three month expiry time, exercise price $E = 10$, volatility $\sigma = 0.4$ and risk-free interest rate $r = 0.1$ with the exact Black–Scholes formula. As with the explicit method, the x-mesh spacing is first fixed and the time-step subsequently determined from α. We have chosen, as before, to regard α and $\delta\tau$ as variable to highlight an important point about stability. Whether $\alpha = 0.5$, $\alpha = 1.0$ or $\alpha = 5.0$,

```
implicit_fd1( values,dx,dt,M,Nminus,Nplus )
{
  a = dt/(dx*dx);

  for( n=Nminus; n<=Nplus; ++n )
    values[n] = pay_off(n*dx);

  lu_find_y( y,a,Nminus,Nplus );

  for( m=1; m<=M; ++m )
  {
    tau = m*dt;

    for( n=Nminus+1; n<Nplus; ++n )
      b[n] = values[n];

    values[Nminus] = u_m_inf( Nminus*dx, tau );
    values[ Nplus] = u_p_inf(  Nplus*dx, tau );
    b[Nminus+1] += a*values[Nminus];
    b[ Nplus-1] += a*values[ Nplus];

    lu_solver( values,b,y,a,Nminus,Nplus );
  }
}
```

Figure 8.10 Pseudo-code for an implicit solver using LU decomposition. `M` is the number of time-steps and `a` $= \alpha$. Note that we have to call `lu_find_y` once (and only once) before using the routine `lu_solver`. We then store the initial values in the array `values[]` and repeatedly call `lu_solver` to time-step through to expiry. Note the boundary value correction applied to the end values `b[Nminus + 1]` and `b[Nplus − 1]`.

the computed solution agrees quite well with the exact solution and there is certainly no evidence of the sort of instability seen in the explicit finite-difference solution when $\alpha > \frac{1}{2}$.

This illustrates the fact that the implicit scheme is stable where the explicit scheme is unstable (that is, for $\alpha > \frac{1}{2}$). In fact, we can show that the implicit finite-difference method is stable for any $\alpha > 0$; see Exercise 11. The consequence is that we can solve the diffusion equation with larger time-steps using an implicit algorithm than we can using an explicit algorithm. This leads to more efficient numerical solutions; even though each time-step takes slightly longer in the implicit method the need for fewer time-steps more than compensates for this. The convergence of the implicit finite-difference approximation to the solution of the partial differential equation can be proved. Again, like the explicit finite-difference scheme, it is convergent if and only if it is stable.

```
implicit_fd2( values,dx,dt,M,Nminus,Nplus )
{
  a = dt/(dx*dx);
  eps = 1.0e-8;
  omega = 1.0;
  domega = 0.05;
  oldloops = 10000;

  for( n=Nminus; n<=Nplus; ++n )
    values[n] = pay_off(n*dx);

  for( m=1; m<=M; ++m )
  {
    tau = m*dt;

    for( n=Nminus+1; n<Nplus; ++n )
      b[n] = values[n];

    values[Nminus] = u_m_inf( Nminus*dx, tau );
    values[ Nplus] = u_p_inf(  Nplus*dx, tau );

    SOR_solver( values,b,Nminus,Nplus,a,omega,eps,loops );
    if ( loops > oldloops ) domega *= -1.0;
    omega += domega;
    oldloops = loops;
  }
}
```

Figure 8.11 Pseudo-code for an implicit solver using the SOR method. `M` is the number of time-steps, `a` $= \alpha$, `eps` is the error tolerance and `omega` the SOR parameter. As in the previous pseudo-code, the routine first puts the initial values into the array `values[]` and then repeatedly calls `SOR_solver` to time-step through to expiry. There is no need to apply boundary value corrections to the end values `b[Nminus + 1]` and `b[Nplus − 1]`, as the SOR routine does this automatically. The routine increments `omega` by `domega` at each step. If this results in the number of iterations for the current step, `loops`, exceeding the iterations for the previous loop, `oldloops`, then the sign of `domega` is changed so that `omega` is moved back towards the value that minimises the number of iterations.

8.7 The Crank–Nicolson Method

The Crank–Nicolson finite-difference method is used to overcome the stability limitations imposed by the stability and convergence restrictions of the explicit finite-difference method, and to have $O((\delta\tau)^2)$ rates

S	$\alpha = 0.50$	$\alpha = 1.00$	$\alpha = 5.00$	exact
0.00	9.7531	9.7531	9.7531	9.7531
2.00	7.7531	7.7531	7.7531	7.7531
4.00	5.7531	5.7531	5.7530	5.7531
6.00	3.7569	3.7570	3.7573	3.7569
8.00	1.9025	1.9025	1.9030	1.9024
10.00	0.6690	0.6689	0.6675	0.6694
12.00	0.1674	0.1674	0.1670	0.1675
14.00	0.0327	0.0328	0.0332	0.0326
16.00	0.0054	0.0055	0.0058	0.0054

Figure 8.12 Comparison of exact Black–Scholes and fully implicit finite-difference solutions for a European put with $E = 10$, $r = 0.1$, $\sigma = 0.4$ and three months to expiry. Even with $\alpha = 5.0$ the results are accurate to 2 decimal places.

of convergence to the solution of the partial differential equation. (The rate of convergence of the implicit and explicit methods is $O(\delta\tau)$.)

The Crank–Nicolson implicit finite-difference scheme is essentially an average of the implicit and explicit methods. Specifically, if we use a forward-difference approximation to the time partial derivative we obtain the explicit scheme

$$\frac{u_n^{m+1} - u_n^m}{\delta\tau} + O(\delta\tau) = \frac{u_{n+1}^m - 2u_n^m + u_{n-1}^m}{(\delta x)^2} + O\left((\delta x)^2\right),$$

and if we take a backward difference we obtain the implicit scheme

$$\frac{u_n^{m+1} - u_n^m}{\delta\tau} + O(\delta\tau) = \frac{u_{n+1}^{m+1} - 2u_n^{m+1} + u_{n-1}^{m+1}}{(\delta x)^2} + O\left((\delta x)^2\right).$$

The average of these two equations is

$$\frac{u_n^{m+1} - u_n^m}{\delta\tau} + O(\delta\tau) = \frac{1}{2}\left(\frac{u_{n+1}^m - 2u_n^m + u_{n-1}^m}{(\delta x)^2} + \frac{u_{n+1}^{m+1} - 2u_n^{m+1} + u_{n-1}^{m+1}}{(\delta x)^2}\right) + O\left((\delta x)^2\right). \tag{8.28}$$

In fact, it can be shown that the terms in (8.28) are accurate to $O((\delta\tau)^2)$, rather than $O(\delta\tau)$; see Exercise 16. Ignoring the error terms leads to the Crank–Nicolson scheme

$$u_n^{m+1} - \tfrac{1}{2}\alpha(u_{n-1}^{m+1} - 2u_n^{m+1} + u_{n+1}^{m+1}) = u_n^m + \tfrac{1}{2}\alpha(u_{n-1}^m - 2u_n^m + u_{n+1}^m) \tag{8.29}$$

where, as before, $\alpha = \delta\tau/(\delta x)^2$. Note that u_n^{m+1}, u_{n-1}^{m+1} and u_{n+1}^{m+1} are now determined implicitly in terms of all of u_n^m, u_{n+1}^m and u_{n-1}^m.

Solving this system of equations is, in principle, no different from solving the equations (8.14) for the implicit scheme. This is because everything on the right-hand side of (8.29) can be evaluated explicitly if the u_n^m are known. The problem thus reduces to first calculating

$$Z_n^m = (1-\alpha)u_n^m + \tfrac{1}{2}\alpha(u_{n-1}^m + u_{n+1}^m), \tag{8.30}$$

which is an explicit formula for Z_n^m, and then solving

$$(1+\alpha)u_n^{m+1} - \tfrac{1}{2}\alpha(u_{n-1}^{m+1} + u_{n+1}^{m+1}) = Z_n^m. \tag{8.31}$$

This second problem is essentially the same as solving (8.14).

Again we assume that we can truncate the infinite mesh at $x = N^-\delta x$ and $x = N^+\delta x$, and take N^- and N^+ sufficiently large so that no significant errors are introduced. As before we can calculate u_n^0 using (8.13) and $u_{N^-}^{m+1}$ and $u_{N^+}^{m+1}$ from the boundary conditions (8.12).

There remains the problem of finding the u_n^{m+1} for $m \geq 0$ and $N^- < n < N^+$ from (8.31). We can write the problem as a linear system

$$\mathbf{C}\boldsymbol{u}^{m+1} = \boldsymbol{b}^m \tag{8.32}$$

where the matrix $\mathbf{C}$ is given by

$$\mathbf{C} = \begin{pmatrix} 1+\alpha & -\frac{1}{2}\alpha & 0 & \cdots & 0 \\ -\frac{1}{2}\alpha & 1+\alpha & -\frac{1}{2}\alpha & & \vdots \\ 0 & -\frac{1}{2}\alpha & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & -\frac{1}{2}\alpha \\ 0 & 0 & & -\frac{1}{2}\alpha & 1+\alpha \end{pmatrix}, \tag{8.33}$$

and the vectors $\boldsymbol{u}^{m+1}$ and $\boldsymbol{b}^m$ are given by

$$\boldsymbol{u}^{m+1} = \begin{pmatrix} u_{N^-+1}^{m+1} \\ \vdots \\ u_0^{m+1} \\ \vdots \\ u_{N^+-1}^{m+1} \end{pmatrix}, \quad \boldsymbol{b}^m = \begin{pmatrix} Z_{N^-+1}^m \\ \vdots \\ Z_0^m \\ \vdots \\ Z_{N^+-1}^m \end{pmatrix} + \tfrac{1}{2}\alpha \begin{pmatrix} u_{N^-}^{m+1} \\ 0 \\ \vdots \\ 0 \\ u_{N^+}^{m+1} \end{pmatrix}. \tag{8.34}$$

The vector on the extreme right-hand side of equation (8.34), in $\boldsymbol{b}^m$, arises from the boundary conditions applied at the ends, as in the fully implicit finite-difference method.

```
crank_fd1( val,dx,dt,M,Nminus,Nplus )
{
  a = dt/(dx*dx);
  a2 = a/2.0;

  for( n=Nminus; n<=Nplus; ++n )
    val[n] = pay_off( n*dx );

  lu_find_y( y,a2,Nminus,Nplus );

  for( m=1; m<=M; ++m )
  {
    tau = m*dt;
    for( n=Nminus+1; n<Nplus; ++n )
      b[n] = (1-a)*val[n]+a2*(val[n+1]+val[n-1]);

    val[Nminus] = u_m_inf( Nminus*dx,tau );
    val[ Nplus] = u_p_inf(  Nplus*dx,tau );
    b[Nminus+1] += a2*val[Nminus];
    b[ Nplus-1] += a2*val[ Nplus];

    lu_solver( val,b,y,a2,Nminus,Nplus );
  }
}
```

Figure 8.13 Pseudo-code for a Crank–Nicolson solver using the LU method. The solution after `M` time-steps is returned in the array `val[]`. Here `a` $= \alpha$ and `a2` $= \alpha/2$. Note the boundary corrections to `b[Nminus + 1]` and `b[Nplus − 1]`. Note also that the code is almost identical to the code in Figure 8.10; the main differences are in the use of $\alpha/2$ instead of α and in the calculation of `b[n]`.

To implement the Crank–Nicolson scheme, we first form the vector b^m using known quantities. Then we use an LU tridiagonal solver or an SOR solver to solve the system (8.32). This allows us to time-step through the solution. The only difference between the LU or SOR solvers for the Crank–Nicolson and fully implicit methods is that whenever α appears in the algorithm for the implicit scheme, we replace it by $\frac{1}{2}\alpha$ for the Crank–Nicolson scheme. In Figures 8.13 and 8.14 we give pseudo-codes for the Crank–Nicolson method using LU and SOR solution methods respectively.

In Figure 8.15 we compare Crank–Nicolson finite-difference solutions for a European put with four months to expiry, exercise price 10, volatility $\sigma = 0.45$ and risk-free interest rate $r = 0.1$ with the exact Black–Scholes formula. Notice that with $\alpha = \frac{1}{2}$, $\alpha = 1.0$ and even $\alpha = 10.0$,

```
crank_fd2( val,dx,dt,M, Nminus,Nplus )
{
  a = dt/(dx*dx);
  a2 = a/2.0;
  eps = 1.0e-8;
  omega = 1.0;
  domega = 0.05;
  oldloops = 10000;

  for( n=Nminus; n<=Nplus; ++n )
    val[n] = pay_off( n*dx );

  for( m=1; m<=M; ++m )
  {
    tau = m*dt;

    for( n=Nminus+1; n<Nplus; ++n )
      b[n] = (1-a)*val[n]+a2*(val[n+1]+val[n-1]);

    val[Nminus] = u_m_inf(Nminus*dx,tau);
    val[ Nplus] = u_p_inf( Nplus*dx,tau);

    SOR_solver( val,b,Nminus,Nplus,a2,omega,eps,loops );
    if ( loops > oldloops ) domega *= -1.0;
    omega += domega;
    oldloops = loops;
  }
}
```

Figure 8.14 Pseudo-code for a Crank–Nicolson solver using the SOR method. The solution after `M` time-steps is returned in the array `val[]`. Here `a` $= \alpha$ and `a2` $= \alpha/2$. Note the boundary corrections to `b[Nminus + 1]` and `b[Nplus − 1]`. Note also that the code is almost identical to the code in Figure 8.11; the main differences are in the use of $\alpha/2$ instead of α and in the calculation of `b[n]`.

the computed solution agrees very well with the exact solution. This demonstrates the fact that the Crank–Nicolson scheme is stable where the explicit scheme is unstable (that is, for $\alpha > \frac{1}{2}$). Moreover, its accuracy is greater than that of the fully implicit scheme.

We can show that the Crank–Nicolson scheme is both stable and convergent for all values of $\alpha > 0$. For the proof of stability see Exercise 17.

S	$\alpha = 0.50$	$\alpha = 1.00$	$\alpha = 10.00$	exact
0.00	9.6722	9.6722	9.6722	9.6722
2.00	7.6721	7.6721	7.6721	7.6722
4.00	5.6722	5.6722	5.6723	5.6723
6.00	3.6976	3.6976	3.6975	3.6977
8.00	1.9804	1.9804	1.9804	1.9806
10.00	0.8605	0.8605	0.8566	0.8610
12.00	0.3174	0.3174	0.3174	0.3174
14.00	0.1047	0.1047	0.1046	0.1046
16.00	0.0322	0.0322	0.0321	0.0322

Figure 8.15 Comparison of exact Black–Scholes and Crank–Nicolson finite-difference solutions for a European put with $E = 10$, $r = 0.1$, $\sigma = 0.45$ and four months to expiry. Even with $\alpha = 10$, the numerical and exact results differ only marginally.

Further Reading

- The books by Johnson & Riess (1982), Strang (1986) and Stoer & Bulirsch (1993) give excellent background material on some of the basic considerations of numerical analysis: accuracy of computer arithmetic, convergence and efficiency of algorithms and stability of numerical methods.
- Richtmyer & Morton (1967) and Smith (1985) are both very readable books on the subject of finite-difference methods.
- Brennan & Schwartz (1978) were the first to describe the application of finite-difference methods to option pricing.
- Geske & Shastri (1985) give a comparison of the efficiency of various finite-difference and other numerical methods for option pricing.

Exercises

1. By considering the Taylor series expansion of $u(x, \tau + \delta\tau)$ about (x, τ), show that the forward difference (8.1) satisfies
$$\frac{u(x,\tau+\delta\tau)-u(x,\tau)}{\delta\tau} = \frac{\partial u}{\partial \tau}(x,\tau) + \frac{\partial^2 u}{\partial \tau^2}(x,\tau+\lambda\,\delta\tau)\,\delta\tau$$
for some $0 \le \lambda \le 1$. Obtain a similar result for the backward-difference approximation (8.2)

2. Expand $u(x, \tau + \delta\tau)$ and $u(x, \tau - \delta\tau)$ as Taylor series about (x, τ). Deduce that the central-difference approximations (8.3) and (8.4) are indeed accurate to $O\Big((\delta\tau)^2\Big)$.

3. Show that the central-symmetric-difference approximation (8.6) is accurate to $O\Big((\delta x)^2\Big)$.

4. Find the minimum number of arithmetical operations (divisions and multiplications) required per time-step of the explicit algorithm (8.10).

5. **Stability I.** Suppose that e_n^m are the finite-precision errors introduced into the solution of (8.10) because of initial rounding errors, e_n^0, in the exact initial values when represented on a computer. Why do the e_n^m also satisfy (8.10)? As a consequence of Fourier analysis, we may assume (without any loss of generality) that the errors take the form $e_n^m = \lambda^m \sin(n\omega)$ for some given frequency ω. Deduce from (8.10) that
$$\lambda = 1 - 4\alpha \sin^2(\tfrac{1}{2}\omega)$$
and infer that unless $0 \le \alpha \le \frac{1}{2}$ there are always frequencies ω for which the error grows without bound.

6. For the explicit finite-difference scheme, show that if we increase the number of x-points by a factor of K, then the number of calculations performed increases by a factor of K^3 (assuming α remains constant).

7. What changes would be necessary to the boundary and initial conditions for the explicit finite-difference method to value a bullish vertical spread? What modifications would be needed to the pseudo-code in Figure 8.5 ?

8. Write a computer program that implements the explicit finite-difference method for calls, puts, cash-or-nothing calls and cash-or-nothing puts.

9. Consider the untransformed Black–Scholes equation (6.3). Use a mesh of equal S-steps of size δS and equal time-steps of size δt, central differences for S derivatives and backward differences for time derivatives to obtain the explicit finite-difference equations
$$V_n^m = a_n V_{n-1}^{m+1} + b_n V_n^{m+1} + c_n V_{n+1}^{m+1}, \quad n = 0, 1, 2, 3, \ldots$$
where V_n^m is the finite-difference approximation to $V(n\,\delta S, m\,\delta t)$ and
$$\begin{aligned} a_n &= \tfrac{1}{2}\left(\sigma^2 n^2 - (r - D_0)n\right)\delta t \\ b_n &= 1 - \left(\sigma^2 n^2 + r\right)\delta t \\ c_n &= \tfrac{1}{2}\left(\sigma^2 n^2 + (r - D_0)n\right)\delta t. \end{aligned}$$

Why is this an *explicit* method? What boundary and initial or final conditions are appropriate? What stability problems can you see arising?

10. Find the minimum number of arithmetical operations (divisions and multiplications) required per time-step of the implicit algorithm (8.14) assuming that an LU solver is used.

11. **Stability II.** Suppose that e_n^m are the finite-precision arithmetical errors introduced into the solution of (8.14) because of initial rounding errors, e_n^0, in the exact initial values when represented on a computer. Following the analysis of Exercise 5, show that if the errors take the form $e_n^m = \lambda^m \sin(n\omega)$ for some given frequency ω then, from (8.14),
$$\lambda = \frac{1}{1 + 4\alpha \sin^2(\frac{1}{2}\omega)}.$$
Deduce that if $\alpha > 0$ errors of any frequency ω do not grow in time.

12. Write a computer program that implements the fully implicit finite-difference method for calls, puts, cash-or-nothing calls and puts.

13. With the same notation as in Exercise 9 show that the implicit discretisation of the Black–Scholes equation may be written as
$$B_0 V_0^m + C_0 V_1^m = V_0^{m+1},$$
$$A_n V_{n-1}^m + B_n V_n^m + C_n V_{n+1}^m = V_n^{m+1}, \quad n = 1, 2, 3, \ldots, N,$$
where
$$\begin{aligned} A_n &= -\tfrac{1}{2}\left(\sigma^2 n^2 - (r - D_0)n\right)\delta t \\ B_n &= 1 + \left(\sigma^2 n^2 + r\right)\delta t \\ C_n &= -\tfrac{1}{2}\left(\sigma^2 n^2 + (r - D_0)n\right)\delta t. \end{aligned}$$

14. Show that LU decomposition for the system
$$\begin{pmatrix} B_0 & C_0 & 0 & \cdots & 0 \\ A_1 & B_1 & C_1 & & \vdots \\ 0 & A_2 & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & C_{N-1} \\ 0 & \cdots & 0 & A_N & B_N \end{pmatrix} \begin{pmatrix} V_0^m \\ V_1^m \\ \vdots \\ V_{N-1}^m \\ V_N^m \end{pmatrix} = \begin{pmatrix} V_0^{m+1} \\ V_1^{m+1} \\ \vdots \\ V_{N-1}^{m+1} \\ V_N^{m+1} \end{pmatrix},$$
of Exercise 13, leads to the algorithm
$$q_0^m = V_0^{m+1}, \qquad q_n^m = V_n^{m+1} - A_n q_{n-1}^m / F_{n-1}, \quad n = 1, 2, \ldots, N$$
$$V_N^m = q_N^m / F_N, \qquad V_n^m = (q_n^m - C_n V_{n+1}^m)/F_n, \quad n = N-1, \ldots, 2, 1,$$

where the q_n are intermediate quantities and the F_n are calculated from the matrix above using
$$F_0 = B_0, \qquad F_n = B_n - C_{n-1} A_n / F_{n-1}, \quad n = 1, 2, \ldots, N.$$

15. Describe the Jacobi, Gauss-Seidel and SOR algorithms for the system in Exercises 13 and 14.

16. Show that the right-hand side of (8.28) is a finite-difference approximation to $\frac{1}{2}\left(\partial^2 u/\partial x^2(x, \tau + \delta\tau) + \partial^2 u/\partial x^2(x, \tau)\right)$. By using Taylor's theorem deduce that
$$\frac{\partial^2 u}{\partial x^2}(x, \tau + \delta\tau/2) = \frac{1}{2}\left(\frac{\partial^2 u}{\partial x^2}(x, \tau) + \frac{\partial^2 u}{\partial x^2}(x, \tau + \delta\tau)\right) + O\left((\delta\tau)^2\right).$$
Use Exercise 2 to show that
$$\frac{\partial u}{\partial \tau}(x, \tau + \delta\tau/2) = \frac{u(x, \tau + \delta\tau) - u(x, \tau)}{\delta\tau} + O\left((\delta\tau)^2\right).$$
Deduce that (8.28), viewed as a finite-difference approximation to
$$\frac{\partial u}{\partial \tau}(x, \tau + \delta\tau/2) = \frac{\partial^2 u}{\partial x^2}(x, \tau + \delta\tau/2),$$
is accurate to $O\Big((\delta\tau)^2 + (\delta x)^2\Big)$.

17. **Stability III.** Repeat the stability calculations in Exercises 5 and 11, but in this case for the Crank–Nicolson equations (8.31). Show that an error term $e_n^m = \lambda^m \sin(n\omega)$ implies
$$\lambda = \frac{1 - 2\alpha \sin^2 \frac{1}{2}\omega}{1 + 2\alpha \sin^2 \frac{1}{2}\omega}.$$
Deduce that the Crank–Nicolson scheme is stable for all $\alpha > 0$.

18. Write down the system of equations resulting from the direct application of the Crank–Nicolson method to the Black–Scholes equation. Describe the LU decomposition and SOR algorithms for its solution.

19. Consider the diffusion equation problem (this problem arises, for example, from an option whose payoff depends on the difference in prices of

two uncorrelated assets)

$$
\begin{aligned}
\frac{\partial u}{\partial \tau} &= \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}, \\
u(x, y, 0) &= u_0(x, y), \\
\text{as } x \to -\infty, \quad u(x, y, \tau) &\sim u^1_{-\infty}(y, \tau), \\
\text{as } x \to \infty, \quad u(x, y, \tau) &\sim u^1_{\infty}(y, \tau), \\
\text{as } y \to -\infty, \quad u(x, y, \tau) &\sim u^2_{-\infty}(x, \tau), \\
\text{as } y \to \infty, \quad u(x, y, \tau) &\sim u^2_{\infty}(x, \tau).
\end{aligned}
$$

Assume equal x- and y-step sizes, $\delta x = \delta y$, a square grid $N^- \delta x \leq i\,\delta x \leq N^+ \delta x$ and $N^- \delta y \leq j\,\delta y \leq N^+ \delta y$, and let u^m_{ij} denote the finite-difference approximation to $u(i\,\delta x, j\,\delta y, m\,\delta \tau)$. Write down the explicit, fully implicit and Crank–Nicolson finite-difference schemes for this problem. What stability restrictions apply to the explicit method?