CHAPTER 3

Finite Difference Methods

3.1. Difference Equations

In this chapter, we consider a numerical technique known as finite difference method capable of solving differential equations by difference equations[1]. It relies on discretizing continuous variables into grid of points that spans the domain of interest and approximating differential operators by finite differences. In this way, we can approximate a differential equation by a difference equation that relates function values at different points on the grid. Hence, it can be solved numerically through iteration starting from the initial condition. Suppose for example that variable x takes on discrete values of $\{ 0, \Delta x, 2\Delta x, \dots \}$ with grid size Δx. Derivative with respect to x can be approximated by the finite difference as

$$y'(x) \cong \frac{y(x + \Delta x) - y(x)}{\Delta x}$$

Thus, we can numerically solve a general first-order differential equation $y'(x) = g(x, y)$ by the difference equation $y(x + \Delta x) = y(x) + \Delta x\, g(x, y(x))$ given an initial value says $y(0) = c$.

The Black-Scholes partial differential equation in (3.1) describes the option price $F(S, t)$ with respect to its underlying asset price S and time t.

$$\frac{\partial}{\partial t}F(S, t) + rS\frac{\partial}{\partial S}F(S, t) + \tfrac{1}{2}\sigma^2 S^2 \frac{\partial^2}{\partial S^2}F(S, t) = rF(S, t) \quad , \quad F(S, T) = \psi(S) \tag{3.1}$$

Together with the differential equation, there are payoff condition $\psi(S)$ at maturity T and intermediate boundary conditions at any time t prior to maturity. The parameters r and σ are, respectively, the risk-free interest rate and volatility of the logarithmic price return of the asset. In particular, we are interesting in solving the Black-Scholes equation for the current values $F(S, 0)$ based on difference equation in this method. Note that the asset price S is always positive and the option ceases to exist after it has matured. In finite difference method, we partition the domain of interest in asset price and time using a two-dimensional grid with sizes ΔS and Δt as shown in Figure 3.1:

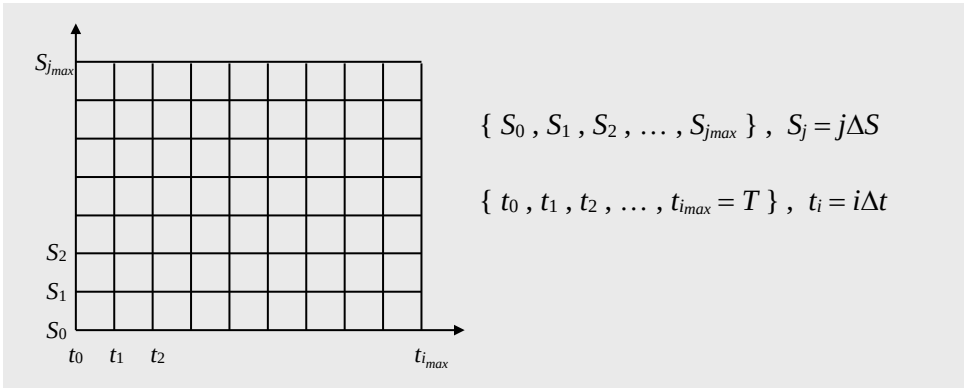

Figure 3.1: Two-dimensional grid for finite difference method.

[1] See Chapter 19 of W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in C* : *The Art of Scientific Computing*, 2nd Edition, Cambridge University Press, 1997. For application on Black-Scholes pricings, see Chapter 17.8 of John C. Hull, *Options, Futures, and Other Derivatives*, 6th Edition, Prentice Hall, 2006.

For interior point (S_j, t_i) on the grid, partial derivatives with respect to asset price in equation (3.1) can be approximated to the second order of price grid ΔS by the finite differences as

$$\frac{\partial}{\partial S}F(S_j, t_i) \cong \frac{F(S_{j+1}, t_i) - F(S_{j-1}, t_i)}{2\Delta S} \tag{3.2}$$

$$\frac{\partial^2}{\partial S^2}F(S_j, t_i) \cong \frac{F(S_{j+1}, t_i) - 2F(S_j, t_i) + F(S_{j-1}, t_i)}{(\Delta S)^2} \tag{3.3}$$

For the derivative with time, we adopt a forward difference approximation to the first order of time grid Δt given by

$$\frac{\partial}{\partial t}F(S_j, t_i) \cong \frac{F(S_j, t_{i+1}) - F(S_j, t_i)}{\Delta t} \tag{3.4}$$

Replacing also the asset price terms in (3.1) by $S_j = j\Delta S$, we can approximate the Black-Scholes partial differential equation by the difference equation:

$$F(S_j, t_{i+1}) = a_j F(S_{j-1}, t_i) + b_j F(S_j, t_i) + c_j F(S_{j+1}, t_i) \quad , \quad \text{for } j = 1, \dots, j_{max} - 1, \; i = 0, \dots, i_{max} - 1 \tag{3.5}$$

where $a_j = \tfrac{1}{2} rj\Delta t - \tfrac{1}{2}\sigma^2 j^2 \Delta t$

$b_j = 1 + r\Delta t + \sigma^2 j^2 \Delta t$

$c_j = -\tfrac{1}{2} rj\Delta t - \tfrac{1}{2}\sigma^2 j^2 \Delta t$

Note that the boundary values $F(S_0, t_{i+1})$ and $F(S_{jmax}, t_{i+1})$ are both missing in the iteration. For completeness, we include in equation (3.5) the corresponding transformations

$$F(S_0, t_{i+1}) = b_0 F(S_0, t_i) \tag{3.6}$$

$$F(S_{jmax}, t_{i+1}) = b_{jmax} F(S_{jmax}, t_i) \tag{3.7}$$

where $b_{jmax} = 1$ and $b_0 = e^{r\Delta t}$ or 1 for European or American-style options[2], respectively. The difference equation can now be written in matrix representation as

$$\begin{pmatrix} F(S_0, t_{i+1}) \\ F(S_1, t_{i+1}) \\ \vdots \\ F(S_{jmax-1}, t_{i+1}) \\ F(S_{jmax}, t_{i+1}) \end{pmatrix} = \boldsymbol{G} \begin{pmatrix} F(S_0, t_i) \\ F(S_1, t_i) \\ \vdots \\ F(S_{jmax-1}, t_i) \\ F(S_{jmax}, t_i) \end{pmatrix} \tag{3.8}$$

[2] Assume S_{jmax} to be sufficiently large such that the change in time premium between t_i and t_{i+1} is insignificant. This gives

$$b_{jmax} = F(S_{jmax}, t_{i+1})/F(S_{jmax}, t_i) \cong 1$$

For European and American call options, we have $F(S_0, t_i) = 0$ and b_0 is thus arbitrary. For European put options, $F(S_0, t_i) = Ke^{-r(T-t_i)}$ from put-call parity. This gives $b_0 = e^{r\Delta t}$. For American put options, $F(S_0, t_i) = K$ due to early exercise and $b_0 = 1$.

where $\boldsymbol{G}$ is a $(j_{max} + 1) \times (j_{max} + 1)$ tridiagonal matrix given by

$$\begin{pmatrix} b_0 & 0 & 0 & \dots & & & & \\ a_1 & b_1 & c_1 & 0 & \dots & & & \\ 0 & a_2 & b_2 & c_2 & 0 & \dots & & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ & & & & \dots & a_{j_{max}-1} & b_{j_{max}-1} & c_{j_{max}-1} \\ & & & & \dots & 0 & 0 & b_{j_{max}} \end{pmatrix} \tag{3.9}$$

It is clear that equation (3.8) is iterating forward in time. To solve the Black-Scholes equation, we need to iterate backward in time instead starting from the option's payoff values at maturity to its current values at t_0. This can be done *implicitly* by reverting equation (3.8) through the inverse matrix $\boldsymbol{G}^{-1}$ such that numerically it is given by the difference equation[3]

$$\begin{pmatrix} F(S_0, t_i) \\ F(S_1, t_i) \\ \vdots \\ F(S_{j_{max}-1}, t_i) \\ F(S_{j_{max}}, t_i) \end{pmatrix} = \boldsymbol{G}^{-1} \begin{pmatrix} F(S_0, t_{i+1}) \\ F(S_1, t_{i+1}) \\ \vdots \\ F(S_{j_{max}-1}, t_{i+1}) \\ F(S_{j_{max}}, t_{i+1}) \end{pmatrix} \tag{3.10}$$

for $i = 0, \dots, i_{max} - 1$ and with payoff condition $F(S_j, t_{i_{max}}) = \psi(S_j)$ at maturity. For exotic options with intermediate boundary conditions, we need to adjust each $F(S_j, t_i)$ on the left side of equation (3.10) according to the boundary conditions before the next iteration with $\boldsymbol{G}^{-1}$ to an earlier time. For example, an American put option has the payoff condition at maturity given by

$$\psi(S_j) = max\{ K - S_j , 0 \}$$

with a strike price K. The option can be exercised at any time prior to its maturity based on the same payoff function. We should therefore compare each $F(S_j, t_i)$ with its intrinsic value $\psi(S_j)$ and perform the update according to the early exercising condition as

$$F(S_j, t_i) = max\{ F(S_j, t_i) , \psi(S_j) \}$$

The errors involved in the use of equation (3.10) are proportional to the time grid Δt and to the square of the price grid ΔS. Numerically, the iteration is unconditionally stable in the sense that the solution remains well behaved for arbitrarily large values of Δt and ΔS despite being less precise. A more accurate procedure is given by the Crank-Nicholson scheme for which the errors are proportional to $(\Delta t)^2$ and $(\Delta S)^2$. The difference equation in this scheme can be found in Appendix 3.1 and the iteration is unconditionally stable but it is numerically more intensive.

[3] For backward difference approximation of the time derivative in equation (3.1),

$$\partial F(S_j, t_i)/\partial t \cong [F(S_j, t_i) - F(S_j, t_{i-1})]/\Delta t$$

The resulting difference equation is *explicitly* iterating backward in time starting from option's maturity with known payoff condition. However, it is numerically stable only with very small time grid $\Delta t < (\sigma^2 j_{max}^2)^{-1}$. The scaling is also practically inconvenient as doubling the price grids would require quadrupling the time grids to maintain stability.

3.2. EXCEL Implementation

Figures 3.2 to 3.4 illustrate how the difference equation in (3.10) could be implemented in EXCEL spreadsheet [4]. As shown in Figure 3.2, option parameters (T, K, r, σ) are input through cells B4, B5, B6, and B7, respectively. In finite difference method, price grid is configurated by the grid number j_{max} and the grid size ΔS. There is also the requirement that boundary value S_{jmax} should presumably be far away from the strike price K. Thus, the grid size cannot be chosen arbitrary in order to improve accuracy. Numerically, it is sufficient to consider $S_{jmax} > 2K$ such that we have a soft lower bound on the grid size as

$$\Delta S > 2K / j_{max}$$

It is then clear that the grid number j_{max} should be defined to the full extent of EXCEL in order to maximize precision. As j_{max} also governs the size of the matrix $\boldsymbol{G}$, we can at most take $j_{max} = 50$ under the maximum capacity of matrix operations in EXCEL. Thus, $j_{max} = 50$ in B2 is a rigid parameter and the grid size ΔS is input through B8 with reference to its lower bound in E8 = 2∗B5/B2. For the time grid configurated by i_{max} and Δt, the boundary value t_{imax} is defined to be the option maturity. Similarly, a large grid number $i_{max} \leq 250$ should be inserted in B3 utilizing the full column space in EXCEL. The corresponding grid size Δt is presumably small and determined in B9 = B4/B3.

	A	B	C	D	E
1					
2	*jmax* =	50			
3	*imax* =	100			
4	*Maturity (T)* =	5	*(year)*		
5	*Strike (K)* =	5			
6	*Risk-free Rate (r)* =	0.05	*(per year)*		
7	*Volatility (σ)* =	0.25	*(per year)*		
8	*Price Inc. (ΔS)* =	0.25		*Min :*	0.2
9	*Time Inc. (Δt)* =	0.05	*(year)*		
10					

Figure 3.2: Option parameters and grid configuration.

We now construct the tridiagonal matrix $\boldsymbol{G}$ as given by (3.9). We may define several named cells to enhance the readability of the formulae: F5(strike), F6(riskfree), F7(sigma), F8(dprice), and F9(dtime). Note that $j_{max} = 50$ is a rigid setup in the implementation, the size of the matrix is 51×51 and it is defined in cells B66:AZ116 from top-left to bottom-right. It is convenient to first create the row and column labels of $\boldsymbol{G}$ such that all entries can be defined with reference to the labeling. As partially shown in Figure 3.3, cells B65:AZ65 label the column number L_c : (0 – 50) of the matrix, while A66:A116 label its row number L_r : (0 – 50) of the matrix. For the top row and bottom row of the matrix, the nonzero entries are defined according to (3.9) as

$$G(L_r = 0, L_c = 0) = b_0 = e^{r\Delta t} \text{ or } 1$$

$$G(L_r = 50, L_c = 50) = b_{jmax} = 1$$

Consider the case of a European-style option, we set B66 = **EXP**(riskfree∗dtime), AZ116 = 1, and elsewhere zero in the two rows. For all the interior rows of the matrix, the entries can be defined according to the rules:

[4] Refer to implicitfd_ep.xls.

$$\text{If } (L_c = L_r - 1) \text{ then } \; G(L_r, L_c) = a_{j=L_r} \tag{3.11}$$

$$\text{If } (L_c = L_r) \text{ then } \; G(L_r, L_c) = b_{j=L_r}$$

$$\text{If } (L_c = L_r + 1) \text{ then } \; G(L_r, L_c) = c_{j=L_r}$$

$$\text{Elsewhere } G(L_r, L_c) = 0$$

where a_j, b_j, and c_j are given by equation (3.5). We can use the row and column labels as reference and apply the following expression for B67 to each of the cells B67:AZ115 in the interior rows.

IF(B$65 = $A67 – 1 , 0.5 ∗ riskfree ∗ $A67 ∗ dtime – 0.5 ∗ sigma^2 ∗ ($A67)^2 ∗ dtime ,
IF(B$65 = $A67 , 1 + riskfree ∗ dtime + sigma^2 ∗ ($A67)^2 ∗ dtime ,
IF(B$65 = $A67 + 1 , – 0.5 ∗ riskfree ∗ $A67 ∗ dtime – 0.5 ∗ sigma^2 ∗ ($A67)^2 ∗ dtime , 0)))

The difference equation in (3.10) requires instead the inverse matrix $\boldsymbol{G}^{-1}$. It is efficient to explicitly calculate all the entries in $\boldsymbol{G}^{-1}$ for successive usages as it is static in the iteration. The size of $\boldsymbol{G}^{-1}$ is also 51×51 and supposes it is defined in cells B119:AZ169. The entries can be determined based on the matrix inverse operation adopted in these cells as **MINVERSE**(B66:AZ116) where B66:AZ116 denotes the input matrix $\boldsymbol{G}$. It is also convenient to name $\boldsymbol{G}^{-1}$ in cells B119:AZ169 as inverseG.

	A	B	C	D	E	F	G	H	I	J
64	**Matrix G :**									
65	$L_r \setminus L_c$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
66	**0**	1.0025	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
67	**1**	-0.0003	1.0056	-0.0028	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
68	**2**	0.0000	-0.0038	1.0150	-0.0088	0.0000	0.0000	0.0000	0.0000	0.0000
69	**3**	0.0000	0.0000	-0.0103	1.0306	-0.0178	0.0000	0.0000	0.0000	0.0000
70	**4**	0.0000	0.0000	0.0000	-0.0200	1.0525	-0.0300	0.0000	0.0000	0.0000
71	**5**	0.0000	0.0000	0.0000	0.0000	-0.0328	1.0806	-0.0453	0.0000	0.0000
72	**6**	0.0000	0.0000	0.0000	0.0000	0.0000	-0.0488	1.1150	-0.0638	0.0000
73	**7**	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	-0.0678	1.1556	-0.0853
74	**8**	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	-0.0900	1.2025

Figure 3.3: The matrix $\boldsymbol{G}$ (partially) together with row and column labels.

To iterate equation (3.10) backward in time, we first construct a two-dimensional structure catering for arrays of option prices evaluated at different time. As partially shown in Figure 3.4, the row labels L_r : (0 – 50) of the array are defined in cells A12:A62, while the underlying asset prices are determined in B12:B62 according to the common expression B12 = $A12∗dprice. We also create the time labels starting from D11 and run toward the right end of the spreadsheet. We assign D11 = B4 to be the option maturity $t_{i_{max}}$ and subtract off one Δt per step to the right by applying recursively the common expression E11 = **ROUND**(D11 – dtime, 8). Here, the **ROUND** function rounds off each time label to 8 decimal places to avoid possible floating point problem in the procedure below. The time label will hit the current time t_0 at column offset from D11 with the grid number i_{max} , and will become negative thereafter.

To initiate the iteration, we define in D12:D62 the option's payoff values at maturity with respect to the asset prices in B12:B62. Suppose it is a put option with strike price defined in B5, the payoff values can be defined using the common expression

D12 = **MAX**(strike – B12 , 0)

For one Δt prior to maturity, the option values in E12:E62 can be determined according to equation (3.10) through the matrix multiplication of $\boldsymbol{G}^{-1}$ with the values D12:D62 at maturity. Iterating

backward in time in the same way, option values at different time labels can be determined through its previous values in the immediate left-hand array using the same $\boldsymbol{G}^{-1}$. For the arrays of option values under the time labels starting from E11, we apply the common matrix operation as

$$\{\ \text{E12:E62}\ = \textbf{IF}(\ \text{E\$11} >= 0\ ,\ \textbf{MMULT}(\ \text{inverseG}\ ,\ \text{D\$12:D\$62}\ )\ ,\ ""\ \ )\ \} \qquad (3.12)$$

where {…} denotes the condition that it is applied to the array in a collective sense through the **Ctrl-Shift-Enter** key. The iteration will terminate at the point when the time label reaches the current time of exactly zero as ensured by the **ROUND** function[5]. As discussed above, this will appear at column offset of i_{max} from D11. The current option values are then displayed in C12:C62 adjacent to the asset prices based on the common expression C12 = **OFFSET**(D12, 0, B3).

	A	B	C	D	E	F	G	H	I	J
11	L_τ	*Asset*	*Option*	**5.00**	**4.95**	**4.90**	**4.85**	**4.80**	**4.75**	**4.70**
12	**0**	**0.00**	3.89	5.00	4.99	4.98	4.96	4.95	4.94	4.93
13	**1**	**0.25**	3.65	4.75	4.74	4.73	4.71	4.70	4.69	4.68
14	**2**	**0.50**	3.40	4.50	4.49	4.48	4.46	4.45	4.44	4.43
15	**3**	**0.75**	3.15	4.25	4.24	4.23	4.21	4.20	4.19	4.18
16	**4**	**1.00**	2.90	4.00	3.99	3.98	3.96	3.95	3.94	3.93
17	**5**	**1.25**	2.66	3.75	3.74	3.73	3.71	3.70	3.69	3.68
18	**6**	**1.50**	2.42	3.50	3.49	3.48	3.46	3.45	3.44	3.43
19	**7**	**1.75**	2.19	3.25	3.24	3.23	3.21	3.20	3.19	3.18
20	**8**	**2.00**	1.98	3.00	2.99	2.98	2.96	2.95	2.94	2.93
21	**9**	**2.25**	1.79	2.75	2.74	2.73	2.71	2.70	2.69	2.68
22	**10**	**2.50**	1.60	2.50	2.49	2.48	2.46	2.45	2.44	2.43
23	**11**	**2.75**	1.44	2.25	2.24	2.23	2.21	2.20	2.19	2.18

Figure 3.4: The two-dimensional structure for price iteration together with time and column labels.

Figure 3.4 depicts only the pricing of a plain vanilla European-style option. For exotic options with intermediate boundary conditions, we need to adjust the generated option values in (3.12) according to some extra conditions depending on the underlying asset prices in B12:B62. Since (3.12) is collectively applied to the array of option prices, it is quite difficult to include in the expression any dynamical update condition with reference to the row labels. However, we can replace the collective structure in (3.12) by a flexible expression given by

$$\text{E12} = \textbf{IF}(\ \text{E\$11} >= 0\ ,\ \textbf{INDEX}(\ \textbf{MMULT}(\ \text{inverseG}\ ,\ \text{D\$12:D\$62}\ )\ ,\ \text{\$A12} + 1\ ,\ 1\ )\ \ ,\ ""\ \ )$$

where the **INDEX** function returns exclusively the entry defined by the row labels within a single column in the generated array. In this way, intermediate condition such as early exercising condition in American-style option can be included easily as

$$\begin{aligned}\text{E12} = \textbf{IF}(\ &\text{E\$11} >= 0\ ,\ \textbf{MAX}(\ \textbf{INDEX}(\ \textbf{MMULT}(\ \text{inverseG}\ ,\ \text{D\$12:D\$62}\ )\ ,\ \text{\$A12} + 1\ ,\ 1\ )\ ,\\ &\textbf{MAX}(\ \text{strike} - \text{\$B12}\ ,\ 0\ )\ )\ ,\ ""\ \ )\end{aligned}$$

The common expression is applied to the entire array in cells E12:E62 and to every array of option values under the time labels. The same piece of matrix multiplication will be repeated in each cell along an array creating a numerical burden in the implementation[6]. Thus, it is efficient to implement the difference equation in (3.10) with intermediate conditions through VBA.

[5] Under double precision, there is possible floating point problem that zero will only be quoted up to 15 decimal places with undesirable negative sign such as –0.0000000000000011.

[6] Refer to implicitfd_ap.xls.

3.3. VBA Implementation

In general, the same matrix operations can also be performed under the VBA environment. It has been shown below that the difference equation in (3.10) can be implemented in an effective way for which intermediate boundary conditions can also be included efficiently in the iteration. We can adopt the same input screen as depicted in Figure 3.2 for both the option parameters (T, K, r, σ) and grid configuration (i_{max} , j_{max} , ΔS). We then develop a routine called CalOptionPrices( ) that takes the above inputs and returns in an array of the current option values $F(L_r = 0: j_{max})$ with respect to asset prices { S_0 , S_1 , S_2 , … , S_{jmax} }. The pseudo code of CalOptionPrices( ) is given by Code 3.1 as follows:

CalOptionPrices(T , K , r , σ , i_{max} , j_{max} , ΔS , $F(0 : j_{max})$)

determine the grid size of time
  $\Delta t = T / i_{max}$

construct the matrix $\boldsymbol{G}$ according to the rules in (3.11)
  $G(0 , 0) = Exp(r\Delta t)$ or 1
  For ($L_c = 1$ to j_{max}) { $G(0 , L_c) = 0$ }

  $G(j_{max} , j_{max}) = 1$
  For ($L_c = 0$ to $j_{max} - 1$) { $G(j_{max} , L_c) = 0$ }

  For ($L_r = 1$ to $j_{max} - 1$) {
      For ($L_c = 0$ to j_{max}) { If($L_c = L_r - 1$) then
                $G(L_r , L_c) = \frac{1}{2} r L_r \Delta t - \frac{1}{2} \sigma^2 L_r^2 \Delta t$
              Elseif($L_c = L_r$) then
                $G(L_r , L_c) = 1 + r \Delta t + \sigma^2 L_r^2 \Delta t$
              Elseif($L_c = L_r + 1$) then
                $G(L_r , L_c) = -\frac{1}{2} r L_r \Delta t - \frac{1}{2} \sigma^2 L_r^2 \Delta t$
              Else
                $G(L_r , L_c) = 0$
              Endif }
  }

# initiate the option values at maturity based on the payoff condition
  For ($L_r = 0$ to j_{max}) { $F(L_r) = payoff(K , L_r \Delta S)$ }

# perform the backward iterations in (3.10) i_{max} number of times up to current option values in conjunction with update according to intermediate boundary conditions
  For ($i = 1$ to i_{max}) { $\boldsymbol{F}(0 : j_{max}) = \boldsymbol{G}^{-1} (0 : j_{max} , 0 : j_{max}) \boldsymbol{F}(0 : j_{max})$

      call *Boundary*($F(0 : j_{max}) , j_{max} , \Delta S , K$) }

Code 3.1: Pseudo code of the CalOptionPrices( ) routine.

In Code 3.1, the matrix multiplication is only performed once per iteration in time followed by an update of the entire array of option values. Intermediate conditions are thus executed efficiently in the implementation. To be flexible, we define both the payoff and intermediate boundary conditions external to this routine. The payoff condition is defined through the user function payoff(K, S) that

evaluates according to the strike and asset prices. The intermediate conditions are defined using the routine Boundary( ) that updates an input array of option values. In case of an American-style option with early exercising condition, for example, the pseudo code of the Boundary( ) routine is given by Code 3.2 as follows:

Boundary($F(0 : j_{max})$, j_{max} , ΔS , K)

For ($L_r = 0$ to j_{max}) { $F(L_r) = \text{MAX}(F(L_r), payoff(K, L_r \Delta S))$ }

Code 3.2: Pseudo code of the Boundary( ) routine for early exercising condition.

Figure 3.5 depicts the spreadsheet design for this VBA implementation[7]. The input section from row 1 to row 10 is taken to be the same as in Figure 3.2 with a new button labeled "Calculate" that triggers the underlying VBA procedures. The option pricings are displayed from row 12 onward in the same way as the first three columns in Figure 3.4. The main VBA routine IFD( ) can be invoked through the "Calculate" button in the spreadsheet. As shown in Code 3.3, we have divided the whole algorithm into three parts handling the input, matrix calculation and output tasks. The input statements will read in both the option parameters and grid configuration from cells B2:B8. The matrix calculations are performed through a single call to the CalOptionPrices( ) routine. The output statements will then return the resulting option prices to column C starting from row 12 with respect to the underlying asset prices in column B.

	A	B	C	D
1				
2	*jmax* =	50		
3	*imax* =	100		
4	*Maturity* (*T*) =	5	(*year*)	
5	*Strike* (*K*) =	5		
6	*Risk-free Rate* (*r*) =	0.05	(*per year*)	
7	*Volatility* (σ) =	0.25	(*per year*)	
8	*Price Inc.* (Δ*S*) =	0.25	*Min* :	0.2
9	*Time Inc.* (Δ*t*) =	0.05	(*year*)	
10			Calculate	
11	L_r	*Asset*	*American Put Option*	
12	**0**	0.00	5.000000	
13	**1**	0.25	4.750000	
14	**2**	0.50	4.500000	
15	**3**	0.75	4.250000	
16	**4**	1.00	4.000000	
17	**5**	1.25	3.750000	
18	**6**	1.50	3.500000	

Figure 3.5: Spreadsheet design for the implementation of finite difference method.

The CalOptionPrices( ) routine first generates the matrix $\boldsymbol{G}$ according to the rules in (3.11). By default, all elements in a declared VBA array are initialized to zero. Thus, we need to assign only the tridiagonal entries of $\boldsymbol{G}$ in the actual coding. The routine then initiates and iterates array of option prices together with intermediate update conditions. The matrix operation $\boldsymbol{G}^{-1}\boldsymbol{F}$ for the iteration is performed by calling a user-defined routine SolveAxb( ) that calculates the column vector $\boldsymbol{x}(n \times 1) =$

[7] Refer to implicitfd_ap_vba.xls

$\boldsymbol{A}^{-1}\boldsymbol{b}$ given square matrix $\boldsymbol{A}(n \times n)$ and column vector $\boldsymbol{b}(n \times 1)$. The two external functions Payoff( ) and Boundary( ) will serve to define the type of option to be considered. Here, we consider an example of an American put option with early exercising boundary condition. For convenient, we have also defined the function Max( ) to handle the maximum operation in the payoff function.

The routine SolveAxb( ) will be useful for the implementations throughout this book. The VBA coding of SolveAxb( ) is given in Code 3.4. The parameters { n ,$iptr$, $jptr$, $kptr$ } define the entries of $A(\ iptr : iptr + n - 1\ ,\ jptr : jptr + n - 1\ )$, $b(\ kptr : kptr + n - 1)$ and $x(\ kptr : kptr + n - 1)$ to be involved in the matrix calculation. The vector $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$ can be calculated very easily by making a call to the EXCEL matrix functions **MINVERSE** and **MMULT**. For EXCEL matrix functions, input and output are considered to be two-dimensional spreadsheet objects with row and column labels starting off from one. To avoid confusion on making cell reference in the use of EXCEL matrix functions, it is convenient to distinguish between VBA arrays and spreadsheet object by first making the conversion. We adopt the naming convention with prefix "ws" which will denote a spreadsheet object. In Code 3.4, we have converted the VBA matrix $\boldsymbol{A}$ and vector $\boldsymbol{b}$ into spreadsheet objects before calling the EXCEL functions. The output is a spreadsheet object that should convert back into VBA vector $\boldsymbol{x}$.

```
Sub IFD()
   'Input parameters from worksheet
   Dim iMax As Integer: iMax = Range("B3").Value
   Dim jMax As Integer: jMax = Range("B2").Value
   Dim maturity As Double: maturity = Range("B4").Value
   Dim strike As Double: strike = Range("B5").Value
   Dim riskFree As Double: riskFree = Range("B6").Value
   Dim sigma As Double: sigma = Range("B7").Value
   Dim dprice As Double: dprice = Range("B8").Value

   'Perform the matrix calculation
   Dim Fvec() As Double: ReDim Fvec(0 To jMax)
   Call CalOptionPrices(maturity, strike, riskFree, sigma, iMax, jMax, dprice, Fvec())

   'Put results back to worksheet
   Dim i As Integer
   For i = 0 To jMax: Range("C12").Offset(i, 0) = Fvec(i): Next i
End Sub
```

```
Sub CalOptionPrices(maturity As Double, strike As Double, riskFree As Double, sigma As Double, _
            iMax As Integer, jMax As Integer, dprice As Double, ByRef Fvec() As Double)

   Dim dtime As Double: dtime = maturity / iMax
   Dim Lr As Integer, i As Integer

   'Construct the matrix G
   Dim Gmatrix() As Double: ReDim Gmatrix(0 To jMax, 0 To jMax)
   Gmatrix(0, 0) = 1
   Gmatrix(jMax, jMax) = 1
   For Lr = 1 To jMax - 1
      Gmatrix(Lr, Lr - 1) = 0.5 * ((Lr * riskFree * dtime) - (Lr ^ 2 * sigma ^ 2 * dtime))
      Gmatrix(Lr, Lr) = 1 + (riskFree * dtime) + (Lr ^ 2 * sigma ^ 2 * dtime)
      Gmatrix(Lr, Lr + 1) = -0.5 * ((Lr * riskFree * dtime) + (Lr ^ 2 * sigma ^ 2 * dtime))
   Next Lr

   'Initialize the option vector according to the payoff condition
   For Lr = 0 To jMax: Fvec(Lr) = Payoff(strike, Lr * dprice): Next Lr

   'Perform the iteration
   For i = 1 To iMax
      Call SolveAxb(Gmatrix, Fvec, Fvec, jMax + 1, 0, 0, 0)
      Call Boundary(Fvec, jMax, dprice, strike)
   Next i

End Sub
```

```
'Put option payoff condition
Function Payoff(strike As Double, price As Double) As Double
   Payoff = Max(strike - price, 0)
End Function
```

```
'Early exercising condition for American-style option
Sub Boundary(ByRef Fvec() As Double, jMax As Integer, dprice As Double, strike As Double)
   Dim intrinsicValue As Double, Lr As Integer
   For Lr = 0 To jMax
      intrinsicValue = Payoff(strike, Lr * dprice)
      Fvec(Lr) = Max(Fvec(Lr), intrinsicValue)
   Next Lr
End Sub
```

```
Function Max(x As Double, y As Double) As Double
   If x > y Then Max = x Else Max = y
End Function
```

Code 3.3: VBA codes of the IFD( ) routine, CalOptionPrices( ) routine, Payoff( ) function, Boundary( ) rountine, and Max( ) function.

```
Sub SolveAxb(Amatrix() As Double, bvec() As Double, ByRef  xvec() As Double, _
                n As Integer, iptr As Integer, jptr As Integer, kptr As Integer)

   Dim wsAmatrix As Variant: ReDim wsAmatrix(1 To n, 1 To n)
   Dim row As Integer, column As Integer
   For row = 1 To n
      For column = 1 To n: wsAmatrix(row, column) = Amatrix(iptr + row - 1, jptr + column - 1): Next column
   Next row

   Dim wsbvec As Variant: ReDim wsbvec(1 To n, 1 To 1)
   For row = 1 To n: wsbvec(row, 1) = bvec(kptr + row - 1): Next row

   Dim wsxvec As Variant:
   With Application.WorksheetFunction
     wsxvec = .MMult(.MInverse(wsAmatrix), wsbvec)
   End With

   Dim i As Integer
   If  n = 1 Then
      For i = kptr To kptr + n - 1: xvec(i) = wsxvec(i - kptr + 1): Next i
   Else
      For i = kptr To kptr + n - 1: xvec(i) = wsxvec(i - kptr + 1, 1): Next i
   End If

End Sub
```

Code 3.4: VBA codes of the SolveAxb( ) routine. Note that when *n* equals one, the (1×1) spreadsheet output "wsxvec" has been degenerated into a variant with only one index.

Appendix 3.1: Crank-Nicholson Scheme

In the Crank-Nicholson scheme, we adopt forward difference approximation for time derivative and adjust accordingly all other terms in the differential equation by forward averaging. Using the two-dimensional grid as depicted in Figure 3.1, we can approximate the time derivative in the Black-Scholes equation (3.1) by forward difference as

$$\frac{\partial}{\partial t}F(S_j, t_i) \cong \frac{F(S_j, t_{i+1}) - F(S_j, t_i)}{\Delta t}$$

and adjust the F, $(\partial F/\partial S)$, and $(\partial^2 F/\partial S^2)$ terms by averaging over the same forward time as

$$F(S_j, t_i) \to \frac{[\, F(S_j, t_{i+1}) + F(S_j, t_i)\,]}{2}$$

$$\frac{\partial}{\partial S}F(S_j, t_i) \to \frac{\partial}{\partial S}\left(\frac{[\, F(S_j, t_{i+1}) + F(S_j, t_i)\,]}{2} \right) \cong \frac{F(S_{j+1}, t_{i+1}) - F(S_{j-1}, t_{i+1}) + F(S_{j+1}, t_i) - F(S_{j-1}, t_i)}{4\Delta S}$$

$$\frac{\partial^2}{\partial S^2}F(S_j, t_i) \to \frac{\partial^2}{\partial S^2}\left(\frac{[\, F(S_j, t_{i+1}) + F(S_j, t_i)\,]}{2} \right)$$

$$\cong \frac{F(S_{j+1}, t_{i+1}) - 2F(S_j, t_{i+1}) + F(S_{j-1}, t_{i+1}) + F(S_{j+1}, t_i) - 2F(S_j, t_i) + F(S_{j-1}, t_i)}{2(\Delta S)^2}$$

The difference equation now reads

$$(-\tfrac{1}{2}a_j)F(S_{j-1}, t_{i+1}) + (1 - \tfrac{1}{2}d_j)F(S_j, t_{i+1}) + (-\tfrac{1}{2}c_j)F(S_{j+1}, t_{i+1})$$

$$= (\tfrac{1}{2}a_j)F(S_{j-1}, t_i) + (1 + \tfrac{1}{2}d_j)F(S_j, t_i) + (\tfrac{1}{2}c_j)F(S_{j+1}, t_i) \quad , \quad \text{for } \begin{matrix} j = 1, \ldots, j_{max} - 1 \\ i = 0, \ldots, i_{max} - 1 \end{matrix}$$

where $a_j = \tfrac{1}{2} rj\Delta t - \tfrac{1}{2}\sigma^2 j^2 \Delta t$

$d_j = r\Delta t + \sigma^2 j^2 \Delta t$

$c_j = -\tfrac{1}{2} rj\Delta t - \tfrac{1}{2}\sigma^2 j^2 \Delta t$

For completeness, we include also the transformations in (3.6) and (3.7) such that the difference equation can be written in matrix representation as

$$\boldsymbol{Q} \begin{pmatrix} F(S_0, t_{i+1}) \\ F(S_1, t_{i+1}) \\ \vdots \\ F(S_{j_{max}-1}, t_{i+1}) \\ F(S_{j_{max}}, t_{i+1}) \end{pmatrix} = \boldsymbol{P} \begin{pmatrix} F(S_0, t_i) \\ F(S_1, t_i) \\ \vdots \\ F(S_{j_{max}-1}, t_i) \\ F(S_{j_{max}}, t_i) \end{pmatrix}$$

where $\boldsymbol{P}$ and $\boldsymbol{Q}$ are $(j_{max} + 1) \times (j_{max} + 1)$ tridiagonal matrices given by

$$\boldsymbol{P} = \begin{pmatrix} b_0 & 0 & 0 & \dots & & & & \\ \tfrac{1}{2}a_1 & 1+\tfrac{1}{2}d_1 & \tfrac{1}{2}c_1 & 0 & \dots & & & \\ 0 & \tfrac{1}{2}a_2 & 1+\tfrac{1}{2}d_2 & \tfrac{1}{2}c_2 & 0 & \dots & & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ & & & & \dots & \tfrac{1}{2}a_{jmax-1} & 1+\tfrac{1}{2}d_{jmax-1} & \tfrac{1}{2}c_{jmax-1} \\ & & & & \dots & 0 & 0 & b_{jmax} \end{pmatrix}$$

$$\boldsymbol{Q} = \begin{pmatrix} 1 & 0 & 0 & \dots & & & & \\ -\tfrac{1}{2}a_1 & 1-\tfrac{1}{2}d_1 & -\tfrac{1}{2}c_1 & 0 & \dots & & & \\ 0 & -\tfrac{1}{2}a_2 & 1-\tfrac{1}{2}d_2 & -\tfrac{1}{2}c_2 & 0 & \dots & & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ & & & & \dots & -\tfrac{1}{2}a_{jmax-1} & 1-\tfrac{1}{2}d_{jmax-1} & -\tfrac{1}{2}c_{jmax-1} \\ & & & & \dots & 0 & 0 & 1 \end{pmatrix}$$

The difference equation can be iterated forward or backward in time by inverting $\boldsymbol{Q}$ or $\boldsymbol{P}$, respectively. It is unconditionally stable and the errors are proportional to $(\Delta t)^2$ and $(\Delta S)^2$.