Chapter 16

Implementing One-Factor Black Scholes in C++

16.1 Introduction and Objectives

In this chapter we design and implement the one-factor Black Scholes equation using finite difference methods (see Duffy 2004, Duffy 2006 for a detailed discussion of this technique). We provide the reader with a C++ framework that he or she can use to numerically price an option as well as offering the facility to extend the framework in a non-intrusive way. For example, it is possible to extend the framework as follows:

- Define your own one-factor financial models
- Design and implement new finite-difference schemes

The current software provides support for general PDEs and the well-known Black Scholes PDE and its accompanying data (such as strike price K, expiry T and so on). Furthermore, we have implemented the explicit and implicit Euler schemes (first-order accurate).

The UML class model for the current problem is shown in figure 16.1 and it is similar in structure and intent to the corresponding diagram that we discussed in chapter 15. The major difference is the fact that we now have a problem whose functions have two input parameters instead of one parameter. This makes the problem more difficult to program at the function level (lots of 'nitty-gritty' detail) but conceptually the architecture is the same as in chapter 15. We can even generalise the results of this chapter to n-factor derivatives problems by using the design in figure 16.1 as a baseline.

We now discuss the details of the proposed solution.

Figure 16.1 UML Class Diagram of Finite Difference Solution

16.2 Scope and Assumptions

As already stated we focus on one-factor models in this chapter. In other words, we consider initial boundary value problems for second-order parabolic differential equations in one space dimension. The general problem is described as follows:

$$
\begin{aligned}
&Lu \equiv -\tfrac{\partial u}{\partial t} + \sigma(x,t)\tfrac{\partial^2 u}{\partial x^2} + \mu(x,t)\tfrac{\partial u}{\partial x} + b(x,t)u = f(x,t) \text{ in } D \\
&u(x,0) = \varphi(x), \quad x \,\epsilon\, \Omega \\
&u(A,t) = g_0(t), \quad u(B,t) = g_1(t), \quad t \,\epsilon\, (0,T) \\
&\text{where } \Omega = (A,B) \text{ and } D = \Omega \times (0,T)
\end{aligned}
\tag{16.1}
$$

We note that this problem subsumes many special cases and in particular it is able to support the original Black Scholes model as discussed in the literature (Hull 2006, Wilmott 1998) as well as more general problems, for example problems with non-constant coefficients and inhomogeneous right-hand term in the PDE in system (16.1).

Furthermore, we assume that the boundary conditions are of Dirichlet type, that is the value of the solution u is known at the boundary points. A discussion of Neumann, Robin and the linearity/convexity boundary conditions is outside the scope of this chapter. For a detailed mathematical discussion of these issues, see Duffy 2006.

16.3 Assembling the C++ Building Blocks

Since we take a modular approach in this book, we design and implement the UML class diagram in figure 16.1 using already developed classes and design patterns (as already discussed in previous chapters):

- Template classes for vectors and matrices. These classes have built-in functionality for

mathematical operations and we provide a number of libraries for more advanced mathematical computation.

- The classes in figure 16.1 deploy a number of design patterns. First, we apply the *Bridge* pattern in area C to allow us to construct multiple implementations of the class `IBVP`, the class that models the components of the initial boundary value problem described by system (16.1). For example, we have provided two implementations (as can be seen in areas A and B), namely a standard implementation and an implementation that is a faithful representation of the Black Scholes model. We note from area (H) that the parameters of the Black Scholes model can be generated using an *Abstract Factory* patterns. The code in this book provides specific functionality for entering data using the `iostream` library (using the overloaded operators << and >>).

The second pattern that we use is the *Template Method* pattern as can be seen in area F. Here we create an abstract class called `FDM` containing default data (for example, generated mesh data from `Mesher` class) as well as the basic algorithmic framework for constructing the approximate solution to the pricing problem. Specific parts of the algorithms are implemented in the derived classes in area (E). The user can add his or her own favourite finite difference schemes by the specialisation process.

Finally, we use the *Visitor* pattern to present the output from the finite difference schemes in various media. This is shown in area G and in particular we are interested in displaying output in Excel in the form of cell data and line charts, for example.

16.4 Modelling the Black Scholes PDE

The C++ classes in this part of the design are similar to those in chapter 15 except that we need to model all the components in system 16.1:

- The region in which the IBVP is defined
- The coefficients of the PDE in 16.1
- The (Dirichlet) boundary conditions
- The initial conditions

The class interface for the class `IBVPImp` is given by:

```
class IBVPImp
{
public:

     virtual double diffusion(double x, double t) const = 0;
     virtual double convection(double x, double t) const = 0;
     virtual double zeroterm(double x, double t) const = 0;
     virtual double RHS(double x, double t) const = 0;
     virtual double BCL(double t) const = 0;
     virtual double BCR(double t) const = 0;
     virtual double IC(double x) const = 0;
};
```

Realising that this is in fact a template (in the broadest sense of the word) for many kinds of specific problems we can implement it using a Bridge pattern. An important case indeed is when we wish to model the original Black Scholes equation and to this end we create a class called `BSIBVPImp` that actually implements the components in system 16.1. Furthermore, the well-known Black Scholes parameters are modelled in a class called `Option` and this has an embedded `Payoff` object (we already have discussed the C++ payoff hierarchy).

The structure of the class `BSIBVPImp` is defined as follows:

```
class BSIBVPImp : public IBVPImp
{
public:
     Option* opt;
     BSIBVPImp(Option& option) {opt = &option;}
     double diffusion(double x, double t) const
     { // simulates diffusion
     double v = (opt -> sig);
          return 0.5 * v * v * x * x;
     }
     // Define the other coefficients here
};
```

Notice that there is an embedded `Option` pointer whose interface is given by:

```
class Instrument
{
public: // Abstract Base class for all Derivatives


};


class Option : public Instrument
{
public: // Data public for convenience only
     double r;      // Interest rate
```

```
    double sig;    // Volatility
    double K;     // Strike price
    double T;     // Expiry date
    double b;     // Cost of carry

    Option() {}

    // An option uses a polymorphic payoff object
    OneFactorPayoff OptionPayoff;
};
```

We are now ready to describe the actual implementations of the coefficients, boundary conditions and initial conditions (payoff) for a call option. First, the coefficients are implemented as follows:

```
    double diffusion(double x, double t) const
    { // simulates diffusion
        double v = (opt -> sig);
        return 0.5 * v * v * x * x;
    }
    double convection(double x, double t) const
    { // simulates drift
        return (opt -> r) * x;
    }
    double zeroterm(double x, double t) const
    {
```

```
return - (opt -> r);
}

double RHS(double x, double t) const
{
     return 0.0;
}
```

while the boundary conditions and initial condition corresponding to a call options are given by (Wilmott 1998):

```
double BCL(double t) const
{

     return 0.0;
}
double BCR(double t) const
{
     return 3.0 * K;      // Magic number
}
double IC(double x) const // Initial condition
{
     return (*opt).OptionPayoff.payoff(x);
}
```

Here is an example of code:

```
InstrumentFactory* myFactory = GetInstrumentFactory();
```

```
Option* myOption = myFactory->CreateOption();


// Derived implementation class
BSIBVPImp bs (*myOption);
bs.opt = myOption;
```

16.4.1 Creating your own one-Factor Financial Models

Looking at figure 16.1 again we see that it is possible to define new derived classes of `IBVPImp`, thus allowing us to incorporate a number of one-factor models into our framework. For example, initial boundary value problems for barrier options are a good candidate. In this case the boundary conditions are of Dirichlet type and the boundaries can be constant or time-dependent. We can also model rebates in this case. We have already discussed finite difference schemes for barrier option problems in Duffy 2006.

Another extension to the framework lies in the ability to define different kinds of payoff functions by creating new payoff classes. For example, we can model symmetric and asymmetric power options by defining their respective payoff functions.

16.5 Finite Difference Schemes

We now discuss areas D, E and F in figure 16.1. First, the class `FDM` is the base class for all finite difference schemes that we will use to model one-factor problems. It contains common data and functions that are inherited by all derived classes. In particular, it is a repository (or blackboard data structure) containing the following kinds of data:

- Mesh data (step-sizes, mesh arrays)
- Vectors and matrices to hold the calculations (they hold option prices)

- Other redundant data

The data in the base class is declared as protected because we wish to be able to access this data in derived classes. The data members of the `FDM` class are given by:

```
protected:
     IBVP* ibvp;            // Pointer to 'parent'
     long N;                // Number of t subdivisions of interval
     double k;              // Step length; redundant data

     long J;                // The number of x subdivisions
     double h, h2;          // Step length; redundant data

     double tprev, tnow;
     long currentIndex, maxIndex;
     Mesher m;
     Vector<double, long> xarr;

     NumericMatrix<double, long> res;  // Contain the results

     // Other data
     long n;                           // Current counter
     Vector<double, long> vecOld;
     Vector<double, long> vecNew;
};
```

We have implemented the Template Method pattern by defining invariant and variants parts of an algorithm:

```
class IBVPFDM
{
pubic:
     // N.B. !!! Resulting output of size N+1 and start index 1
     NumericMatrix<double, long>& result(); // Result of calculation

     // Hook function for Template Method pattern
     virtual void calculateBC() = 0; // Tells calculate sol. at n+1
     virtual void calculate() = 0; // Tells calculate sol. at n+1
};
```

The body of the 'key' invariant algorithm is given by:

```
NumericMatrix<double, long>& IBVPFDM::result()
{ // The result of the calculation

L1:
     tnow = tprev + k;
     // Template Method pattern here
     // Variant part
     calculateBC(); // Calculate the BC at n+1

     // Variant part; pure virtual hook functions
     calculate (); // Calculate the solution at n+1
```

```
    if (currentIndex < maxIndex)
    {
        tprev = tnow;
        currentIndex++;
        // Now postprocess
        res.Row(currentIndex, vecNew);
        vecOld = vecNew;
        goto L1;
    }
    return res;
}
```

Here we see that we 'march' in time and calculate the approximate solution at each time level. The derived classes must define the 'hook' functions `calculateBC()` and `calculate()`. It will be the responsibility of each derived class to do this. To this end, we now discuss how to effect this for explicit and implicit Euler schemes.

16.5.1 Explicit Schemes

We discuss the application of the well-known explicit Euler scheme to the initial boundary value problem (16.1). In other words, we use Forward Differencing in Time and Centred Differencing in Space (FTCS). After some manipulation and arithmetic we can show how to determine the value at time level $n+1$ in terms of the value at time level n:

$$u_j^{n+1} = (A_j^n u_{j-1}^n + B_j^n u_j^n + C_j^n u_{j+1}^n)/h^2 - k f_j^n$$

where

$$\begin{cases} A_j^n = (k\sigma_j^n - \frac{kh}{2}\mu_j^n) \\ B_j^n = (h^2 - 2k\sigma_j^n + kh^2 b_j^n) \\ C_j^n = (k\sigma_j^n + \frac{kh}{2}\mu_j^n) \end{cases} \tag{16.2}$$

$$1 \leq j \leq J-1, \quad n \geq 0$$

Thus, the solution can be calculated without the need to solve a matrix system at each time level. Of course, the boundary conditions must be defined:

```
void calculateBC()
{ // Tells how to calculate sol. at n+1
     vecNew[vecNew.MinIndex()] = ibvp->BCL(tprev);
     vecNew[vecNew.MaxIndex()] = ibvp->BCR(tprev);
}
```

Then, the C++ code implementing algorithm (16.2) is given by:

```
void calculate()
{ // Tells how to calculate sol. at n+1
     // Explicit Euler schemes
     for (long i = vecNew.MinIndex()+1;
          i <= vecNew.MaxIndex()-1; i++)
     {
          tmp1 = k*(ibvp->diffusion(xarr[i], tprev));
```

```
            tmp2 = (k*h*0.5 *
              (ibvp->convection(xarr[i], tprev)));

            // Coefficients of the U terms
            alpha = tmp1 - tmp2;
            beta = h2 - (2.0*tmp1) +
               (k*h2*(ibvp->zeroterm(xarr[i], tprev)));
            gamma = tmp1 + tmp2;
            vecNew[i] = ( (alpha * vecOld[i-1])
                        + (beta * vecOld[i])
                             + (gamma * vecOld[i+1]) ) / h2
                             - (k*(ibvp -> RHS(xarr[i], tprev)));
        }
    }
```

An example of using the explicit scheme is:

```
BSIBVPImp bs (*myOption);

// Now the client class IBVP
IBVPImp* myImp3 = &bs;
IBVP i3(*myImp3, rangeX, rangeT);
ExplicitEulerIBVP fdm3(i3, N, J);
print(fdm3.result());
```

16.5.2 Implicit Schemes

In this section we take the implicit Euler scheme, sometimes called the Backward in Time Centred in Space (BTCS) scheme. After some manipulation and arithmetic we can show how to determine the value at time level $n+1$ in terms of the value at time level n:

$$A_j^{n+1}u_{j-1}^{n+1} + B_j^{n+1}u_j^{n+1} + C_{j+1}^{n+1}u_{j+1}^{n+1} = h^2(kf_j^{n+1} - u_j^n)$$

where

$$\begin{cases} A_j^{n+1} = (k\sigma_j^{n+1} - \frac{kh}{2}\mu_j^{n+1}) \\ B_j^{n+1} = (-h^2 - 2k\sigma_j^{n+1} + kh^2 b_j^{n+1}) \\ C_j^{n+1} = (k\sigma_j^{n+1} + \frac{kh}{2}\mu_j^{n+1}) \end{cases} \tag{16.3}$$

$$1 \leq j \leq J-1, \quad n \geq 0$$

The problem now is that we have three unknown values on the left-hand side of equation (16.3). In order to solve this problem we can employ a number of matrix techniques, for example LU decomposition (Keller 1992, Duffy 2004). To this end we write system (16.3) in the vector form:

$$A^{n+1}\underline{U}^{n+1} = \underline{r}^{n+1}$$

where

$$\underline{U} = {}^t(u_1, \ldots, u_{J-1})$$

$$r = {}^t(r_1, \ldots, r_{J-1})$$

$$A = \begin{pmatrix} B_1 & C_1 & & \\ A_2 & \ddots & \ddots & 0 \\ 0 & \ddots & \ddots & C_{J-2} \\ & & A_{J-1} & B_{j-1} \end{pmatrix} \tag{16.4}$$

$$r_1^{n+1} = h^2(kf_1^{n+1} - u_1^n) - A_1^{n+1}u_0^{n+1}$$

$$r_j^{n+1} = h^2(kf_j^{n+1} - u_j^n), \quad 2 \leq j \leq J-2$$

$$r_{J-1}^{n+1} = h^2(kf_{J-1}^{n+1} - u_{J-1}^n) - C_{J-1}^{n+1}u_J^{n+1}$$

We can solve system (16.4) using the implementation for the LU solver code on the accompanying CD. The complete code for the current derived class is given by:

```
void calculateBC()
{ // Tells how to calculate sol. at n+1
     vecNew[vecNew.MinIndex()] = ibvp->BCL(tnow);
     vecNew[vecNew.MaxIndex()] = ibvp->BCR(tnow);
}
```

```
void calculate()
{ // Tells how to calculate sol. at n+1
     // In general we need to solve a tridiagonal system
     double tmp1, tmp2;

     for (long i = 1; i <= J-1; i++)
     {
          tmp1 = (k*ibvp->diffusion(xarr[i],tnow ));
          tmp2 = (0.5 * k * h*
            (ibvp->convection(xarr[i], tnow)));

          // Coefficients of the U terms
          A[i] = tmp1 - tmp2;
          B[i] = -h2 - (2.0*tmp1) +
            (k*h2*(ibvp->zeroterm(xarr[i],tnow)));
          C[i] = tmp1 + tmp2;

          F[i] = h2*(k *
            (ibvp -> RHS(xarr[i], tnow)) - vecOld[i+1]);
     }

     // Correction term for RHS
     F[1] -= A[1] * vecNew[vecNew.MinIndex()];
     F[J-1] -= C[J-1] * vecNew[vecNew.MaxIndex()] ;
```

```
        // Now solve the system of equations
        LUTridiagonalSolver<double, long> mySolver(A, B, C, F);


        Vector <double, long> solution = mySolver.solve();


        for (long ii = vecNew.MinIndex()+1;
               ii <= vecNew.MaxIndex()-1; ii++)
        {
             vecNew[ii] = solution[ii-1];
        }
    }
```

16.5.3 A Note on Exception Handling

In the case of the implicit Euler method (see system (16.4)) it is important that the solution can be found at each time level. In general, we demand that system (16.4) has a unique solution. A sufficient condition is that the matrix A is positive definite. But how does this requirement translate to C++ code? In this case we call the `assert` macro (defined in `<assert.h>`). If the matrix is diagonally dominant the program will stop executing:

```
// Now solve the system of equations
LUTridiagonalSolver<double, long> mySolver(A, B, C, F);


// The matrix must be diagonally dominant; we call the
// assert macro and the programs stops
assert (mySolver.diagonallyDominant() == true);
```

```
Vector <double, long> solution = mySolver.solve();
```

A more elegant solution is to us the exception handling mechanism in C++.

16.5.4 Other Schemes

In this chapter we have discussed two schemes to approximate the solution of system (16.1). As can be seen from figure 16.1 we can devise other schemes by creating derived classes from **FDM** and implementing the appropriate pure virtual 'hook' functions. For example, the following schemes can be programmed:

- Crank-Nicolson scheme (this is gotten by taking the average of the explicit and implicit schemes in equations (16.2) and (16.3))

- Richardson extrapolation: we apply the scheme (16.3) on two meshes of sizes k and $k/2$. Then we use the second-order approximation:

$$W_j^n \equiv U_j^n - V_{2j}^{2n}, \quad 1 \leq j \leq J-1 \tag{16.5}$$

  **(16.5)

  where U and V are the values corresponding to the finite difference schemes on mesh sizes k and k/w, respectively. We have actually written the code for this scheme for the case of initial value problems in chapter 15.

- Special schemes, for example schemes for problems with low volatility and/or large convection terms (Duffy 1980, Duffy 2004, Duffy 2006).

We discuss these schemes as exercises at the end of this chapter.

16.6 Test Cases and Presentation in Excel

The classes in this framework allow us to calculate the price of an option for arrays of stock prices and at a discrete set of time levels up to and including the expiry date T. In this chapter we have assembled these values in a `NumericMatrix` object. We can then present this matrix in Excel using the `ExcelDriver` class.

16.6.1 Creating the User Interface Dialogue

In the special case of the initial boundary value problem that models the one-factor Black Scholes problem we have employed an Abstract Factory pattern to realise the input of the specific parameters (such as the strike price K, expiry date T and so on). The client main program does not have to worry about the details of how data is created but instead it delegates to a factory object that takes care of such details.

We now discuss how we have implemented the Abstract Factory pattern in the current context and how it is used in the client program. In this chapter we concentrate on inputting data using the `iostream` library. In fact, you can define your own factory classes (for example, an MFC or Microsoft SDK) to allow you to input data in other ways as well.

The top-level interface for the factory is given by:

```
class InstrumentFactory
{
public:

     virtual Option* CreateOption() const = 0;

};
```

At this moment we have defined an interface for one kind of financial instrument, namely an option. This interface can easily be extended to support other instrument types. The console implementation is given by:

```
class ConsoleInstrumentFactory : public InstrumentFactory
{
public:
     Option* CreateOption() const
     {
          double dr; // Interest rate
          double dsig; // Volatility
          double dK; // Strike price
          double dT; // Expiry date
          double db; // Cost of carry
          cout << "Interest rate: ";
          cin >> dr;
          cout << "Volatility: ";
          cin >> dsig;
          cout << "Strike Price: ";
          cin >> dK;
          cout << "Expiry: ";
          cin >> dT;
          cout << "Cost of carry: ";
          cin >> db;

          Option* result = new Option;
```

```
cout << "Payoff 1) Call, 2) Put: ";
int ans;
cin >> ans;
if (ans == 1)
{
     (*result).OptionPayoff
           = OneFactorPayoff(dK, MyCallPayoffFN);
}
else
{
     (*result).OptionPayoff
           OneFactorPayoff(dK, MyPutPayoffFN);
}

result->r = dr;
result->sig = dsig;
result->K = dK;
result->T = dT;
result->b = db;

cout << "ACK: Press ANY key to continue: ";
cin >> ans;

return result;
```

```
    }
};
```

Using factory objects is much the same as using 'normal' objects in C++. In general, we prefer to hide the choice of factory in a function:

```
InstrumentFactory* GetInstrumentFactory()
{

    // Only 1 factory in this version, like model T
    return new ConsoleInstrumentFactory;
}
```

Then client code has a pointer to an abstract factory at run-time. To this end, here is an example of use:

```
    InstrumentFactory* myFactory = GetInstrumentFactory();
    Option* myOption = myFactory->CreateOption();

    // Derived implementation class
    BSIBVPImp bs (*myOption);
    bs.opt = myOption;
```

16.6.2 Vector and Matrix Output

In general, the output from finite difference schemes is some kind of data structure that contains the option price for certain values of the underlying and at certain points in time. We may also be interested in calculating option delta and gamma, in which case we can take divided differences of the option price.

In the current framework we use two `Vector` instances and one `NumericMatrix` instance. The vectors hold the values at the time levels n (previous) and $n+1$ (current) while the matrix holds the values at all time levels up to N (where $Nk = T$) and all points in S space, including the boundaries. Having calculated the desired values we are then free to decide how we are to display them.

16.6.3 Presentation

There are various output display devices to which we can send our calculated values. In this chapter we are interested in displaying vectors and matrices in the following ways:

- To the console (useful for quick debugging)
- In Excel (for extended and sophisticated presentation)

The first option is easy to apply. It displays the results on the console. We have two generic functions for such presentation:

```
template <class V, class I>
     void print(const Array<V,I>& array);


template <class V, class I>
     void print(const Matrix<V,I>& array);
```

The second set of functions presents vectors and matrices in Excel. The main functions are:

- Displaying a vector in Excel
- Displaying a list of vectors in Excel
- Displaying a matrix in Excel

An example of using these functions is given by:

```
ImplicitIBVP fdm2(i2, N, J);

Mesher m(rangeX, rangeT);

Vector<double, long> XARR = m.xarr(6);

Vector<double, long> vec = fdm.result()[2];

printOneExcel(XARR, vec, string("explicit");


Vector<double, long> vec2 = fdm2.result()[2];

printOneExcel(XARR, vec2, string("implicit");
```

16.7 Summary

We have shown how to apply design patterns to the creation of a customisable framework in C++ for the one-factor Black Scholes PDE. It is possible to define your own PDE model as it is possible to apply special-purpose finite difference schemes by the specialisation process. The framework represents a stable software environment in which you can develop and test option models.

16.8 Exercises and Projects

1. (Improving performance) For the implicit Euler scheme we used the algorithm for LU decomposition to solve a system of equations at each time level. In particular, the arrays in equation (16.4) are copied into the member data of the class `LUTridiagonalSolver`. Performance can be improved by using pointers instead, as the following code suggests:

```
// Defining arrays (input)
```

```
Vector<V,I> * a;   // The lower-diagonal array [1..J]

Vector<V,I> * b;   // The diagonal array [1..J] "baseline array"

Vector<V,I> * c;   // The upper-diagonal array [1..J]

Vector<V,I> * r;   // Right-hand side of equation Au = r [1..J]
```

Client code should still keep functioning but there will be an improvement in performance.

2. (LU Decomposition and Complex Numbers) In some cases and applications we are interested in solving linear systems of equations where the coefficients are complex-valued. We take the example of the time-dependent linear Schrödinger equation in one dimension (Pauling 1963, Press 2002):

$$
\begin{aligned}
& i\frac{\partial \psi}{\partial t} = -\frac{\partial^2 \psi}{\partial x^2} + V(x)\psi, \quad -\infty < x < \infty \\
& \text{where} \\
& \psi = \text{Schrödinger wave function} \\
& V(x) = \text{one-dimension potential} \\
& i = \sqrt{-1}
\end{aligned}
\tag{16.6}
$$

This equation describes the scattering of a one-dimensional wave packet by the potential $V(x)$. We have assumed for convenience that Planck's constant $h = 1$ in this example. We rewrite equation (16.6) in the equivalent form:

$$
\begin{aligned}
& i\frac{\partial \psi}{\partial t} = H\psi \\
& \text{where} \\
& H = -\frac{\partial^2}{\partial x^2} + V(x)
\end{aligned}
\tag{16.7}
$$

The operator H is called the Hamiltonian operator and it determines the time variation of the system. We are now interested in approximating equation (16.7) using finite difference schemes. To this end, the explicit Euler FTCS scheme is given by:

$$\begin{aligned} & i\tfrac{\psi^{n+1}-\psi^n}{k} = H\psi^n \\ & \text{or} \\ & \psi^{n+1} = (1 - iHk)\psi^n \end{aligned} \tag{16.8}$$

while the implicit Euler BTCS scheme is given by:

$$\begin{aligned} & i\tfrac{\psi^{n+1}-\psi^n}{k} = H\psi^{n+1} \\ & \text{or} \\ & (i - Hk)\psi^{n+1} = i\psi^n \\ & \text{or} \\ & (1 + iHk)\psi^{n+1} = \psi^n \end{aligned} \tag{16.9}$$

Scheme (16.8) is unstable while (16.9) is stable. However, neither scheme is unitary in the sense of the original problem, that is, the total probability of finding the particle somewhere is 1:

$$\int_{-\infty}^{\infty} |\psi(x)|^2 dx = 1 \tag{16.10}$$

A remedy for this is to use the so-called Cayley form (this is in fact the Crank Nicolson scheme):

$$i\frac{\psi^{n+1}-\psi^n}{k} = \frac{H}{2}(\psi^{n+1}+\psi^n)$$

or

$$(1+\tfrac{i}{2}HK)\psi^{n+1} = (1-\tfrac{i}{2}Hk)\psi^n \tag{16.11}$$

This scheme is unitary; you can check this by a bit of arithmetic using complex arithmetic. Finally, we must discretise the operator H in the x direction by using a centred scheme, for example. We then get a fully discrete scheme and it can be cast in a form as in system (16.4) except the coefficients are complex-valued!

We have provided a test program to show how the template classes can be used. The objective in this exercise is to show how to program the fully-discrete version of (16.11) using the finite difference method. Here is a simple example of how to use the code in this book. For illustrative purposes we solve a 2X2 complex system:

```
J = 2;
Vector<Complex, long> A(J,1,Complex(1.0, 0.0));
Vector<Complex, long> B(J,1,Complex(0.0, 1.0));
Vector<Complex, long> C(J,1,Complex(1.0, 0.0));
Vector<Complex, long> R(J,1,Complex(0.0, 0.0));
R[1] = Complex(0.0, 2.0);
LUTridiagonalSolver<Complex, long> mySolver2(A, B, C, R);
Vector<Complex, long> result2 = mySolver2.solve();
print(result2);
```

Now you can see the classes that you need to use.

3. (Barrier options) The finite difference schemes in this chapter are suitable for modelling

one-factor barrier option problems with continuous monitoring. We can support both single and double barrier problems because Dirichlet boundary conditions are applicable in these cases. Constant, time-dependent and exponential barriers can easily be modelled as well as the ability to include rebates functions into the boundary conditions. We have found that the implicit Euler scheme gives good results, especially near the barrier. Test the code with some examples of barrier options.

In many cases the boundary conditions may be discontinuous at a finite number of points in S space. For example, rebates are sometimes defined as step-functions for given ranges of S and in these cases it may be necessary to average-out or smooth these discontinuous functions in order to avoid inaccuracies in the approximate solution.

Which classes in the framework would you need to modify if you wish to model barrier options with discrete monitoring? Recall that we must determine the jump conditions at the monitoring dates. An algorithmic approach to approximating this problem is given in Duffy 2006. The major addition to the code is that the marching process in time must be modified.

4. (American options) The code for this chapter is for European options only but the code can easily be extended to American put (and call) option. In this case we must ensure that the constraint for a put option price P satisfies:

$$P(S,t) \geq \max(K-S,0)$$

where K is the strike price and S is the price of the underlying. We wish to ssatisfy the same constraints in the difference schemes, for example. One possible solution is to check the constraint at time level $n+1$:

$$u_j^{n+1} = \max(u_j^{n+1}, \max(K - S_j, 0)) \tag{16.12}$$

Where would you incorporate this feature into the framework while at the same time not 'disturbing' the current functionality for European options?

5. (Second-order Accuracy) The implicit Euler BTCS scheme (16.3) is first-order accurate in time. It is possible to achieve second-order accuracy by two applications of BTCS on meshes of size k and k/2. We have coded this in the case of a scalar initial value problem in chapter. Recall the code:

```
void calculate()
{
     // Extrapolated implicit Euler; create two solutions on k/2
     // and k called v(k/2) and v(k), respectively.  Then form
     // the new array 2*v(k/2) - v(k). Code can be optimised (later)

     // Refined mesh and solution
     Vector<double, long> res2 (2*N + 1, 1);
     double k2 = k * 0.5;
     res2[res2.MinIndex()] = (ivp -> startValue());
     for (long i = res2.MinIndex() + 1;
          i <= res2.MaxIndex(); i++)
     {
          res2[i] = ( res2[i-1] + (k2 * ivp->f(i*k2)) ) /
            ( 1.0 + (k2 * ivp->a(i*k2)) );
     }
```

```
        // Rougher mesh
        Vector<double, long> res1 (N + 1, 1);
        res1[res1.MinIndex()] = (ivp -> startValue());
        for (long ii = res1.MinIndex() + 1;
             ii <= res1.MaxIndex(); ii++)
        {
             res1[ii] = ( res1[ii-1] + (k* ivp->f(ii*k)) ) /
                ( 1.0 + (k* ivp->a(ii*k)) );
        }

        // Extrapolated solution
        for (long iii = res1.MinIndex() + 1;
             iii <= res1.MaxIndex(); iii++)
        {
             res[iii] = (2.0 * res2[(2*iii)]) - res1[iii];
        }
    }
```

The objective of this exercise is to generalise this code to the current initial boundary value problem. Create the code for this problem and integrate it into the framework. Test your work by modelling a Black Scholes benchmark example whose solution you know.

6. (Approximating the Greeks) The C++ code in this chapter produces a matrix of values; the rows represent time levels while the columns are discrete values of the underlying price. Knowing this fact, we can use divided differences to find approximate values for the delta, gamma and theta of the option price. Implement the code for this problem and

integrate it in the framework. What data structure do you intend to use?

7. (Linear Interpolation) You may wish to calculate the option price at some point between two given mesh points. You should first determine in which sub-interval of the mesh that this point falls and then use linear interpolation to calculate the option price. Finally, if you decide to use cubic spline interpolation, you can use the LU solver (Dahlquist 1974).