Chapter 7

The Convection-Diffusion Equation

In this chapter, we describe finite-difference methods to discretize the convection-diffusion equation in one and two spatial dimensions. In particular, we use explicit, implicit, and semi-implicit schemes to march from the current time step to the next one and the upwind scheme to discretize the spatial derivatives. The entire algorithm is implemented in C++ using a hierarchy of objects: from the time-space grid at the high level to the individual grid lines at the low level, with the difference operators that act upon them.

7.1 Initial-Boundary-Value Problems

We are now ready to use the programming tools developed above in the numerical solution of initial-boundary-value problems. Here, we consider one of the most important parabolic PDEs, which is commonly used to model physical processes: the convection-diffusion equation [54], [60]. In one spatial dimension, this equation takes the form

$$u_t(t,x) - \varepsilon u_{xx}(t,x) + C(t,x)u_x(t,x) = F(t,x), \quad 0 < t < T,\ 0 < x < L.$$

Here, T is the maximal time, L is the length of the x-interval, $F(t,x)$ is the given right-hand side, ε is the small diffusion coefficient, $C(t,x)$ is the given convection coefficient, and $u(t,x)$ is the unknown solution.

We further assume that initial conditions are given in the following form:

$$u(0,x) = u^{(0)}(x), \quad 0 < x < L,$$

where $u^{(0)}(x)$ is a given function. We also assume that Dirichlet boundary conditions are imposed on the right edge:

$$u(t,L) = G(t,L), \quad 0 < t < T,$$

and mixed boundary conditions are imposed on the left edge:

$$\alpha(t,0)u(t,0) + u_n(t,0) = G(t,0), \quad 0 < t < T,$$

where $\alpha()$ and $G()$ are given functions and $n = -x$ is the direction that points away from the interval $[0,L]$ at its left edge 0 (the outer normal direction).

The above initial and boundary conditions must be compatible with each other; that is, they must agree with each other at the corner points $(0,0)$ and $(0,L)$, so that u can be defined continuously there. With these conditions, the problem is well posed in the sense that it indeed has a unique solution $u(t,x)$.

7.2 Finite Differences

PDEs can be solved analytically only in very few model cases. In general, the PDE is solved numerically on a discrete grid, which is just a finite set of points in the x-interval $[0,L]$. For example, if N is the number of points in a uniform grid with meshsize $h = L/N$, then the grid is just the set of points

$$(0, h, 2h, \ldots, (N-1)h).$$

To be solved numerically, the PDE must be approximated (discretized) on the grid. A common way to do this is by finite differences. In this approach, a derivative is approximated by the corresponding difference, and the entire differential equation is approximated by the corresponding system of difference equations, which can then be solved numerically on a computer.

Let us now describe the finite-difference discretization method in some more detail. Let the time t be fixed, and let u_j be an approximation to the solution at the jth grid point:

$$u_j \doteq u(t, jh), \quad 0 \le j < N$$

(where the symbol $\doteq$ stands for "approximately equal to"). The derivative of u at the midpoint between the jth and $(j+1)$th grid points is approximated by the divided finite difference

$$u_x(t,(j+1/2)h) \doteq h^{-1}(u_{j+1} - u_j).$$

Similarly, the derivative of u at $(j-1/2)h$ is approximated by

$$u_x(t,(j-1/2)h) \doteq h^{-1}(u_j - u_{j-1}).$$

By subtracting the latter approximation from the former and dividing by h, we obtain the approximation to the second derivative of u at $x = jh$:

$$u_{xx}(t, jh) \doteq h^{-1}(u_x(t,(j+1/2)h) - u_x(t,(j-1/2)h)) \doteq h^{-2}(u_{j+1} - 2u_j + u_{j-1}).$$

This way, the diffusion term u_{xx} can be approximated by the above finite differences on the discrete grid.

The above discretization uses symmetric differencing; that is, the divided difference between two grid points approximates the derivative at their midpoint. This scheme is of second-order accuracy; that is, the discretization error $u_j - u(t, jh)$ is as small as h^2 as $h \to 0$. However, below we'll see that accuracy is not always the most important property of a numerical scheme. In some cases, more important properties such as adequacy should also be verified before the numerical scheme is used.

7.3 The Upwind Scheme

Let us now discretize the convection term $C(t,x)u_x$. The naive way to do this is by symmetric finite differencing as before:

$$C(t,jh)u_x(t,jh) \doteq (2h)^{-1}C(t,jh)(u_{j+1}-u_{j-1}).$$

This approach, however, is inappropriate because it excludes the jth grid point. In fact, it only uses differences between two even-numbered grid points or two odd-numbered grid points, so it is completely unaware of frequent oscillations such as

$$(1,-1,1,-1,\ldots),$$

which is overlooked by the scheme. Indeed, because the scheme uses only the $(j-1)$th and $(j+1)$th grid points, it produces the same discrete convection regardless of whether or not the discrete solution contains a component of the above oscillation. Because this oscillation depends highly on h, it has nothing to do with the solution of the original PDE, $u(t,x)$, which must be independent of h. It is therefore known as a nonphysical oscillation.

In theory, the above symmetric differencing is of second-order accuracy as $h \to 0$. In practice, however, h must take a small positive value. When the diffusion coefficient ε is very small (particularly, smaller than h), the resulting scheme becomes inadequate in the sense that it produces a numerical solution that has nothing to do with the required solution of the original PDE. A more stable and adequate scheme is needed to discretize the convection term properly.

Such a scheme is the "upwind" scheme. This scheme also uses u_j in the approximation to $u_x(t,jh)$, thus avoiding nonphysical oscillations. More specifically, the scheme uses backward differencing at grid points j for which $C(t,jh) > 0$:

$$C(t,jh)u_x(t,jh) \doteq h^{-1}C(t,jh)(u_j-u_{j-1}),$$

and forward differencing at grid points j for which $C(t,jh) < 0$:

$$C(t,jh)u_x(t,jh) \doteq h^{-1}C(t,jh)(u_{j+1}-u_j).$$

This way, the coefficient of u_j in the discrete approximation of the convection term Cu_x is always positive, as in the discrete approximation to the diffusion term $-\varepsilon u_{xx}$ in Section 7.2 above. These coefficients add up to produce a substantial positive coefficient to u_j in the difference equation, which guarantees stability and no nonphysical oscillations.

In summary, the difference approximation to the entire convection-diffusion term takes the form

$$D_{j,j-1}u_{j-1} + D_{j,j}u_j + D_{j,j+1}u_{j+1} \doteq -\varepsilon u_{xx}(t,jh) + C(t,jh)u_x(t,jh),$$

where

$$\begin{aligned} D_{j,j-1} &= -\varepsilon h^{-2} - \frac{|C(t,jh)| + C(t,jh)}{2h}, \\ D_{j,j} &= 2\varepsilon h^{-2} + |C(t,jh)|h^{-1}, \\ D_{j,j+1} &= -\varepsilon h^{-2} - \frac{|C(t,jh)| - C(t,jh)}{2h}. \end{aligned}$$

The convection-diffusion equation is solved numerically on a time-space grid. This is a rectangular $M \times N$ grid, containing M rows of N points each. The rows have the running index $i = 1, 2, 3, \ldots, M$ to order them from the bottom row to the top row. (The bottom row is numbered by 1, and the top row is numbered by M.) The ith row corresponds to the ith time step (time level) in the time marching in the scheme.

The above matrix D that contains the discrete convection-diffusion term may change from time step to time step according to the particular convection coefficient $C()$ at the corresponding time. Thus, we also use the superscript i to indicate the relevant time step. This way, $D^{(i)}$ denotes the matrix corresponding to the ith time step (or time level, or row) in the time-space grid ($1 \le i \le M$).

7.4 Discrete Boundary Conditions

At the endpoint of the grid, the above discretization cannot take place because the $(j+1)$th or $(j-1)$th grid point is missing. For example, for $j = N-1$, the $(j+1)$th point lies outside the grid, so the last equation

$$D_{N-1,N-2}u_{N-2} + D_{N-1,N-1}u_{N-1} + D_{N-1,N}u_N = \cdots$$

is not well defined, because it uses the dummy unknown u_N. In order to fix this, one should use the Dirichlet boundary condition available at the right edge of the domain:

$$u_N = G(t, L).$$

Once this equation is multiplied by $D_{N-1,N}$ and subtracted from the previous one, the dummy u_N unknown is eliminated, and the equation is well defined.

Similarly, the dummy unknown u_{-1} is used in the first equation:

$$D_{0,-1}u_{-1} + D_{0,0}u_0 + D_{0,1}u_1 = \cdots.$$

Fortunately, one can still use the discrete mixed boundary conditions to eliminate this unknown. Indeed, the discrete boundary conditions on the left can be written as

$$\alpha(t,0)u_0 + h^{-1}(u_{-1} - u_0) = G(t,0).$$

Once this equation is multiplied by $hD_{0,-1}$ and subtracted from the previous one, the dummy u_{-1} unknown is eliminated, and the first equation is also well defined. By the way, the above subtraction also increases the $D_{0,0}$ coefficient by

$$-hD_{0,-1}\alpha(t,0) + D_{0,-1}.$$

The above discretization of the convection-diffusion terms actually produces the matrix (i.e., difference operator) D. This operator maps any N-dimensional vector $v \equiv (v_0, v_1, \ldots, v_{N-1})$ to another N-dimensional vector Dv, defined by

$$\begin{aligned}
(Dv)_0 &= D_{0,0}v_0 + D_{0,1}v_1, \\
(Dv)_j &= D_{j,j-1}v_{j-1} + D_{j,j}v_j + D_{j,j+1}v_{j+1} \quad (0 < j < N-1), \\
(Dv)_{N-1} &= D_{N-1,N-2}v_{N-2} + D_{N-1,N-1}v_{N-1}.
\end{aligned}$$

Note that the matrix or difference operator D depends on the time t, because the functions $C()$ and $\alpha()$ depend on t. Therefore, it should actually be denoted by $D(t)$ or $D^{(i)}$, where i is the index of the time step. In the above, however, we omit the time indication for the sake of simplicity.

7.5 The Explicit Scheme

The time derivative u_t in the convection-diffusion equation is also discretized by a finite difference. For this purpose, the entire x-t domain should be first discretized or approximated by a two-dimensional uniform grid with M rows of N points each (Figure 7.1). The ith row ($1 \le i \le M$), also known as the ith time level or time step, approximates the solution of the original PDE at a particular time. More specifically, the numerical solution at the ith time step, denoted by $u^{(i)}$, approximates $u(t,x)$ on the discrete grid:

$$u_j^{(i)} \doteq u(i\triangle t, jh),$$

where $\triangle t = T/M$ is the cell size in the time direction. Clearly, for $i = 0$, the approximate solution is available from the initial conditions

$$u_j^{(0)} = u^{(0)}(jh),$$

where $u^{(0)}$ on the left is the vector of values on the grid at the zeroth time level, and $u^{(0)}$ on the right is the given function that specifies the initial condition at $t = 0$.

For $i > 0$, the numerical solution at the ith time level is computed by marching across time levels (time marching). In other words, the numerical solution at the current time level is obtained from the numerical solution computed before on the previous time level and the boundary conditions.

Time marching can be done in (at least) three different ways, which depend on the discretization of the time derivative u_t: forward differencing leads to explicit time marching (or the explicit scheme), backward differencing leads to implicit time marching (the implicit scheme), and midpoint differencing leads to semi-implicit time marching (the semi-implicit scheme).

In the explicit scheme, forward differencing is used to discretize u_t. This way, the numerical solution at the current time level is computed from a discrete approximation to the

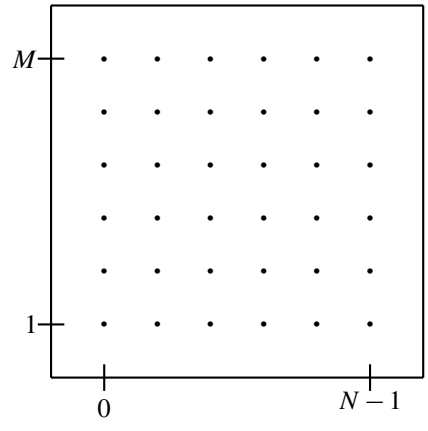

Figure 7.1. *The uniform $M \times N$ time-space grid for the discretization of the convection-diffusion equation in the time-space domain $0 < x < L$, $0 < t < T$.*

original PDE, in which the convection-diffusion term is evaluated at the previous time level:

$$u_t(t,x) - \varepsilon u_{xx}(t,x) + C(t,x)u_x(t,x)$$
$$\doteq (\triangle t)^{-1}\left(u_j^{(i)} - u_j^{(i-1)}\right) + \left(D^{(i-1)}u^{(i-1)}\right)_j = F(t, jh),$$

where $D^{(i-1)}$ is the discrete convection-diffusion term, evaluated at time $t = (i-1)\triangle t$ as in Section 7.3. This evaluation uses the numerical solution $u^{(i-1)}$ at the previous or $(i-1)$th time level to calculate the numerical solution $u^{(i)}$ at the current or ith time level. This is why this scheme is known as the explicit scheme.

Note that, when $j = 0$ or $j = N-1$, the right-hand side should be incremented by the appropriate contribution from the boundary-condition function $G(t,x)$. In fact, the right-hand side is the N-dimensional vector $f^{(i)}$ defined by

$$f_0^{(i)} = F(i\triangle t, 0) - hD_{0,-1}^{(i)} G(i\triangle t, 0),$$
$$f_j^{(i)} = F(i\triangle t, jh) \quad (0 < j < N-1),$$
$$f_{N-1}^{(i)} = F(i\triangle t, (N-1)h) - D_{N-1,N}^{(i)} G(i\triangle t, L).$$

The above formulas lead to the explicit time marching

$$u^{(i)} = u^{(i-1)} - (\triangle t)D^{(i-1)}u^{(i-1)} + (\triangle t)f^{(i-1)}.$$

This equation defines the numerical solution at the current or ith time level in terms of the numerical solution at the previous or $(i-1)$th time level.

The explicit scheme can actually be viewed as the analogue of the scheme in Chapter 6, Section 8, with the first-order Taylor approximation

$$\exp\left(-(\triangle t)D^{(i-1)}\right) \doteq I - (\triangle t)D^{(i-1)},$$

where I is the identity matrix.

The implementation of this marching in C++ requires not only "vector" objects to store and manipulate the data in the time levels but also "difference" objects to handle the difference operators that act upon vectors. This is the subject of Section 7.12 below.

In the above explicit scheme, the numerical solution $u^{(i)}$ is computed by evaluating the discrete convection-diffusion terms at the $(i-1)$th time level, where the numerical solution $u^{(i-1)}$ is already known from the previous step. Clearly, this can be done in a straightforward loop over the time levels, from the first one to the Mth one. Although the cost of each step in this loop is fairly low, its total cost may be rather large. Indeed, as discussed in Chapter 8, Section 3, the parameter $\triangle t$ used in the explicit scheme must be as small as h^2 to prevent nonphysical oscillations from accumulating during the time marching. Thus, the number of time levels M must be as large as Th^{-2}, leading to a prohibitively lengthy process. More stable schemes, which use fewer time levels, are clearly needed.

7.6 The Implicit Scheme

A more stable scheme, which requires fewer time levels to discretize the same time-space domain, is the implicit scheme. In this scheme, the discrete convection-diffusion terms are

evaluated at the current (or ith) time level rather than the previous (or $(i-1)$th) time level:

$$\begin{aligned} &u_t(t,x) - \varepsilon u_{xx}(t,x) + C(t,x)u_x(t,x) \\ &\doteq (\triangle t)^{-1}\left(u_j^{(i)} - u_j^{(i-1)}\right) + \left(D^{(i)}u^{(i)}\right)_j = F(i\triangle t, jh) \end{aligned}$$

where $t = i\triangle t$. Using this formula, the implicit scheme can be written compactly in terms of N-dimensional vectors as:

$$\left(I + (\triangle t)D^{(i)}\right)u^{(i)} = u^{(i-1)} + (\triangle t)f^{(i)},$$

where I is the identity matrix of order N.

The calculation of $u^{(i)}$ requires the "inversion" of the tridiagonal matrix $I + (\triangle t)D^{(i)}$. Of course, the inverse of this matrix is never calculated explicitly, because this would be too expensive. By "inversion" we mean solving the above linear system for the unknown vector $u^{(i)}$. The algorithm for doing this is discussed below. This completes the definition of the implicit time marching. The detailed implementation will be discussed later.

Although the computation of each individual $u^{(i)}$ is more expensive than in the explicit scheme, the entire loop over the entire time-space grid is usually far less expensive, because M can be much smaller. Indeed, since the implicit scheme is unconditionally stable (Chapter 8, Section 8.3), it can use $\triangle t$ as large as h. This reduces considerably the number of time levels M and the total computational cost.

The implicit scheme can actually be viewed as the analogue of the scheme in Chapter 6, Section 8, with the first-order Taylor approximation

$$\exp\left(-(\triangle t)D^{(i)}\right) = \exp\left((\triangle t)D^{(i)}\right)^{-1} \doteq \left(I + (\triangle t)D^{(i)}\right)^{-1}.$$

7.7 The Semi-Implicit Scheme

The semi-implicit scheme, also known as the Crank–Nicolson or "midpoint" scheme, can be viewed as a compromise between the explicit and implicit schemes. In this scheme, the discrete convection-diffusion term is the average of the evaluation at the previous time level as in the explicit scheme and the evaluation at the current time level as in the implicit scheme:

$$\begin{aligned} &u_t(t,x) - \varepsilon u_{xx}(t,x) + C(t,x)u_x(t,x) \\ &\doteq (\triangle t)^{-1}\left(u_j^{(i)} - u_j^{(i-1)}\right) + \frac{1}{2}\left(D^{(i)}u^{(i)} + D^{(i-1)}u^{(i-1)}\right)_j \\ &= F((i-1/2)\triangle t, jh) \end{aligned}$$

at the "half" time level $t = (i-1/2)\triangle t$. Because the time derivative u_t is discretized symmetrically around this half time level, the scheme is of second-order accuracy in time wherever the solution to the original PDE is indeed smooth in time.

The semi-implicit scheme can be written compactly as the vector equation

$$\left(I + \frac{\triangle t}{2}D^{(i)}\right)u^{(i)} = \left(I - \frac{\triangle t}{2}D^{(i-1)}\right)u^{(i-1)} + (\triangle t)f^{(i-1/2)}.$$

The algorithm to solve this system for the unknown vector $u^{(i)}$ will be provided later. This completes the definition of the semi-implicit scheme. The complete implementation will be given later.

Like the implicit scheme, the semi-implicit scheme is unconditionally stable (Chapter 8, Section 8.3). Thus, $\triangle t$ can be relatively large without producing any unstable nonphysical oscillations. Thus, M and the computational cost are kept rather low.

The semi-implicit scheme can actually be viewed as the analogue of the scheme in Chapter 6, Section 8, with the first-order diagonal Padé approximation (Chapter 1, Section 22)

$$\exp\left(-(\triangle t)D^{(i-1/2)}\right) \doteq \left(I + (\triangle t/2)D^{(i)}\right)^{-1}\left(I - (\triangle t/2)D^{(i-1)}\right).$$

In the next sections, we use the present C++ framework to implement the semi-implicit scheme. The explicit and implicit schemes can be implemented in a similar way.

7.8 The Implementation

In the remainder of this chapter, we introduce the C++ implementation of the semi-implicit scheme for the numerical solution of the convection-diffusion equation. The main advantage of this implementation in comparison with standard Fortran codes is the opportunity to specify the size of the grid in run time rather than compilation time. This flexibility is particularly important in adaptive discretizations, where the size of the grid becomes available only when the program is actually executed.

The method of adaptive mesh refinement is implemented fully in Chapter 14, Section 2. Here, we use no adaptivity, because the time-space grid is uniform. Thus, the present problem can actually be solved in Fortran or C. Nevertheless, the objects implemented here in C++ may be useful not only for this particular application but also for more complicated ones. Furthermore, the multigrid linear-system solver that can be used to solve the linear system of equations in each time step is particularly well implemented in C++, as we'll see in Chapter 10, Section 5, and Chapter 17, Section 10. In fact, in C++, one can actually develop libraries of objects, which can then be used not only in the present application but also in more advanced numerical applications and algorithms.

Let us illustrate how object-oriented languages in general and C++ in particular can be used in practice to develop the library or hierarchy of objects required in numerical applications. Suppose that a user wants to solve the convection-diffusion equation. This user is only interested in the solution of the physical model and not in the particular numerical method used for this purpose. In fact, the user may have little experience in numerical algorithms and prefer to leave this part of the job to another member of the team.

Thus, the user writes code like this:

```
int main(){
  domain D(10.,1.,.1);
  D.solveConvDif();
  return 0;
}
```

That's it, problem solved! All the details will be worked out by the numerical analyst. For this, however, the analyst must know precisely what objects to implement and what their properties or functions should be.

In order to work, the above code must have a class "domain" with a constructor that takes three "double" arguments to specify T (the size of the time interval), L (the size of the x-interval), and the required accuracy. In the above code, for example, 'D' is the time-space domain $0 \le t \le 10$, $0 \le x \le 1$, and the discretization error $|u^{(i)} - u(i\triangle t, jh)|$ should be at most 0.1.

The "domain" class must also have a member function "solveConvDif ", which solves the convection-diffusion equation (to the above accuracy) and prints the numerical solution at the final time $t = 10$ onto the screen.

The numerical analyst, in turn, also wants to make his/her life easy, so he/she assumes the existence of a template class named "xtGrid" that implements the time-space discrete grid. This class must have at least three member functions: a constructor "xtGrid(M,N)" to construct the $M \times N$ grid, "timeSteps()" to return the number of time levels, and "width()" to return the number of points in the x-direction. Then, he/she implements the "domain" class as follows:

```
class domain{
  xtGrid<double> g;
  double Time;
  double Width;
```

In this implementation, the "domain" class contains three data fields: "Time" to indicate the size of the time interval, "Width" to indicate the size of the space interval, and 'g' to contain the discrete time-space grid.

The constructor of the "domain" class takes three "double" arguments to specify the size of the time interval, the size of the space interval, and the required accuracy. This third argument is used to calculate the required numbers of time levels and grid points in the x-direction. The constructor has an empty body; all the work is done in the initialization list. In particular, the first field, the grid 'g', is constructed by calculating the required number of time levels and number of grid points in the x-direction and using them to invoke the constructor of the "xtGrid" class (whose existence is assumed):

```
public:
  domain(double T, double L, double accuracy)
    : g((int)(T/accuracy)+1,(int)(L/accuracy)+1),
      Time(T), Width(L){
  }  //  constructor
```

The numerical analyst also assumes that there exists an ordinary function named "march(g,h,$\triangle t$)", which takes "xtGrid", "double", and "double" arguments, and does the actual time marching in this grid. Then, he/she uses this function to define the function that solves the convection-diffusion equation in the "domain" class:

```
  void solveConvDif(){
    march(g,Width/g.width(),Time/g.timeSteps());
  }  //  solve the convection-diffusion equation
};
```

This completes the block of the "domain" class.

By writing the above code, the numerical analyst actually completed his/her part of the job and can pass the rest of it on to a junior numerical analyst to work out the details. The problem of solving the convection-diffusion equation has now been transformed to the numerical problem of marching in time in the time-space grid. The junior numerical analyst is asked to write the ordinary function "march(g,h,$\triangle t$)" that does the actual time marching in the time-space grid.

The junior numerical analyst also makes assumptions to make his/her life easier. In particular, he/she assumes that the "xtGrid" class is already implemented with some useful member functions. First, he/she assumes that the 'i'th row in the grid 'g' can be read as "g[i]" and accessed (for reading/writing) as "g(i)". Next, he/she assumes that the value at the 'j'th point in it can be read as "g(i,j,"read")" and accessed (for reading/writing) as "g(i,j)". Finally, he/she assumes that the member functions "timeSteps()" and "width()" return the number of time levels and the number of points in an individual time level, respectively.

The junior numerical analyst also assumes the existence of a template class "difference" that implements difference operators like $D^{(i)}$ above, along with some elementary arithmetic operations, such as "difference" plus "difference", "difference" times "dynamicVector", etc. In fact, the "difference" object 'D' is also interpreted as a tridiagonal matrix, whose "(i,j)"th element can be read as "D(i,j,"read")" and accessed for reading/writing as "D(i,j)". Finally, it is also assumed that "D.width()" returns the order of the matrix 'D'.

With these tools, the junior numerical analyst is ready to implement the semi-implicit time marching in the time-space grid. For this purpose, he/she must first define a function named "convDif()" to set the "difference" object that contains the discrete convection-diffusion term at the particular time under consideration, along with the corresponding right-hand-side vector 'f'. For this purpose, it is assumed that the functions used in the original PDE and initial-boundary conditions (F, G, α, etc.) are available as global external functions. (In the present example, most of them are set to zero for the sake of simplicity; only the initial conditions in the function "Initial()" are nonzero.)

```
double F(double, double){return 0.;}
double C(double, double){return 0.;}
double Alpha(double, double){return 0.;}
double G(double, double){return 0.;}
double Initial(double x){return 1.-x*x;}
const double Epsilon=1.;
```

It is assumed that these functions are global, so they are accessible from the "convDif()" function defined next. The first arguments in this function are the "difference" object 'd', where the discrete convection-diffusion term at time 't' is placed, and the "dynamicVector" object 'f', where the corresponding right-hand side is placed. It is assumed that both 'd' and 'f' are initially zero; in the "convDif()" function, they are set to their correct values:

```
template<class T>
void convDif(difference<T>&d,dynamicVector<T>&f,
        double h,double deltaT,double t){
    for(int j=0; j<d.width(); j++){
      if(t>deltaT/2)f(j)=F(j*h,t-deltaT/2);
      double c=C(j*h,t);
```

The local variable 'c' contains the convection coefficient at the 'j'th grid point at time 't'. The upwind scheme decides whether to use forward or backward differencing in the x-direction according to the sign of 'c':

```
        if(c>0.){
          d(j,j)=c/h;
          d(j,j-1)=-c/h;
          d(j,j+1)=0.;
        }
        else{
          d(j,j)=-c/h;
          d(j,j+1)=c/h;
          d(j,j-1)=0.;
        }
      }
```

So far, we have introduced the discrete convection term into the difference operator 'd'. This has been done in a loop over the grid points in the time level under consideration. Now, we add to 'd' the discrete diffusion term. This is done in one command only, using the constructor and "operator+=" to be defined later in the "difference" class:

```
    d += Epsilon/h/h * difference<T>(d.width(),-1.,2.,-1.);
```

Finally, we introduce the discrete boundary conditions into the difference operator 'd' and the right-hand-side vector 'f':

```
      d(0,0) += d(0,-1);
      d(0,0) -= d(0,-1) * h * Alpha(0,t);
      if(t>deltaT/2){
        f(0) -= d(0,-1) * h * G(0,t-deltaT/2);
        f(d.width()-1) -= d(d.width()-1,d.width())
              * G(d.width()*h,t-deltaT/2);
      }
  }  //  set the convection-diffusion matrix and right-hand side
```

The "convDif()" function is now used in the "march()" function that implements the semi-implicit time marching. For this purpose, we use a loop over the time levels, with a local integer index named "time" that goes from the first time level at the bottom to the final one at the top of the time-space grid. In this loop, we use two local "difference" objects named "current" and "previous" to store the discrete convection-diffusion terms at the current and previous time levels (the difference operators $D^{(\mathrm{time})}$ and $D^{(\mathrm{time}-1)}$):

```
  template<class T>
  void march(xtGrid<T>&g, double h, double deltaT){
    difference<T> I(g.width(),0.,1.,0.); // identity matrix
    for(int j=0; j<g.width(); j++)
      g(0,j) = Initial(j*h);
    dynamicVector<T> f(g.width());
    difference<T> previous(g.width());
    convDif(previous,f,h,deltaT,0);
    for(int time=1; time < g.timeSteps(); time++){
      difference<T> current(g.width());
      convDif(current,f,h,deltaT,time*deltaT);
```

We are now in the loop that does the actual time marching. We now advance from the previous time step to the current one. We have just put the discrete convection-diffusion spatial derivatives (evaluated at the current time) into the difference operator "current". We have also saved the discrete convection-diffusion spatial derivatives from the previous time step in the difference operator "previous". These difference operators are now used in the following command, which is the heart of the semi-implicit scheme. In this command, the '/' symbol below calls the "operator/" (to be defined later) that solves the linear system obtained in the semi-implicit scheme, which can be viewed symbolically as dividing a vector by a tridiagonal matrix or difference operator:

```
      g(time) =
          ((I-.5*deltaT*previous)*g[time-1]+deltaT*f)
          / (I + 0.5 * deltaT * current);
      previous = current;
    }
    print(g[g.timeSteps()-1]);
}  //  semi-implicit time marching
```

The actual implementation of the "xtGrid" and "difference" classes and required functions will be discussed later on.

7.9 Hierarchy of Objects

The hierarchy of objects used in the entire workplan is described in Figure 7.2. In the highest level, the "domain" object is placed, along with the function "solveConvDif()" that acts upon it. In the lower level, the "xtGrid" object is placed, along with the function "march()" that acts upon it. In fact, "solveConvDif()" only invokes "march()" to act upon the "xtGrid" object contained in the "domain" object. In the lowest level, the "dynamicVector" and "difference" objects are placed. These objects correspond to a particular time level in the entire "xtGrid" object; they are also connected by arithmetic operations between them, and the function "convDif()" sets them with the discrete convection-diffusion term at the relevant time level or time step.

7.10 List of Vectors

All that is left to do is define the "xtGrid" and "difference" template classes, with the required functions. These tasks can actually be carried out independently by two other members of the team, who are C++ programmers who don't necessarily have a background in numerical methods.

The "xtGrid" and "difference" objects can be viewed as lists of vectors. In an "xtGrid", each vector corresponds to a time level, so the number of vectors is the same as the number of time levels. In a "difference" object, on the other hand, there are only three vectors: the first contains the $(j, j-1)$ elements in the tridiagonal matrix, the second contains the (j, j) elements (the main diagonal), and the third contains the $(j, j+1)$ elements. It is thus natural to derive the "xtGrid" and "difference" classes from a list of dynamic vectors.

Because the data fields in the "list" template class in Chapter 3, Section 4, are declared "protected" rather than "private", they are accessible from derived classes. This is why it is

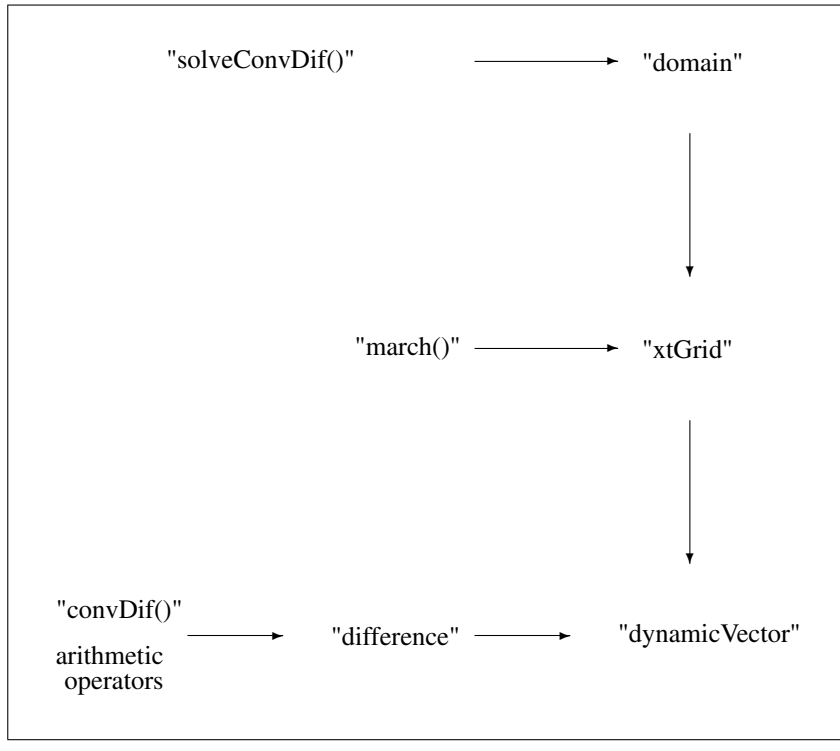

Figure 7.2. *Hierarchy of objects for the convection-diffusion equation: the "domain" object uses an "xtGrid" object, which uses "dynamicVector" and "difference" objects.*

possible to define constructors in the derived "xtGrid" and "difference" classes with integer argument to specify the dimension of the vectors in the list. Indeed, when the constructors of the "xtGrid" and "difference" classes are called, the data fields of the base "list" class are constructed first by the default constructor of that class, resulting in a trivial list with no items at all. These lists are then reconstructed in the derived-class constructor thanks to their access privilege to the data fields of the base class.

7.11 The Time-Space Grid

Here, we introduce the "xtGrid" template class that implements the time-space grid as a list of time levels, each of which is implemented as a dynamic vector.

Here is the detailed implementation of the "xtGrid" class, derived from the list of dynamic vectors:

```
template<class T>
class xtGrid : public list<dynamicVector<T> >{
   public:
    xtGrid(int,int,const T&);
```

The constructor is only declared here; its full definition will be provided later. Here are the required functions that return the number of time levels and the number of grid points in each time level:

```
    int timeSteps() const{
      return size();
    }  // number of time levels

    int width() const{
      return item[0]->dim();
    }  // width of grid
```

The individual time levels can be accessed by three different versions of "operator()". Specifically, the 'i'th time level in the "xtGrid" object 'g' can be accessed by either "g[i]" for reading only or "g(i)" for reading or writing, and the value at the 'j'th grid point in it can be accessed by "g(i,j)" for reading or writing. The compiler invokes the version that best fits the number and type of arguments in the call.

The "operator[]" is inherited from the base "list" class in Chapter 3, Section 4. The "operator()", on the other hand, although available in the "list" class, must be rewritten explicitly to prevent confusion with the other version:

```
    dynamicVector<T>& operator()(int i){
      if(item[i])return *item[i];
    }  //  ith time level (read/write)

    T& operator()(int i, int j){
      return (*item[i])(j);
    }  //  (i,j)th grid point (read/write)
};
```

This concludes the block of the "xtGrid" class. All that is left to do is to define the constructor, which is only declared in the class block above. This constructor takes two integer arguments: the first specifies the number of items in the underlying list of vectors (or the number of time levels), and the second specifies the dimension of these vectors (or the number of points in each individual time level). When the constructor is called, the underlying list of dynamic vectors is initialized automatically by the default constructor of the base "list" class to be an empty list that contains no items at all. This list is then reset in the present constructor to the required nontrivial list. This reconstruction can be done thanks to the fact that the "number" and "item" data fields are declared as "protected" rather than private in the base "list" class.

```
template<class T>
xtGrid<T>::xtGrid(int m=0,int n=0,const T&a=0){
  number = m;
  item = m ? new dynamicVector<T>*[m] : 0;
  for(int i=0; i<m; i++)
    item[i] = new dynamicVector<T>(n,a);
}  //  constructor
```

Note that the order of the above codes should actually be reversed: since the "xtGrid" class is used in the "domain" class and the "convDif" and "march" functions, it must appear before them in the program. It is only for the sake of clear presentation that the order is reversed in the above discussion.

7.12 Difference Operators

Here we define the "difference" class that implements the difference operator or the tridiagonal matrix. The "difference" class is derived from the "list<dynamicVector<T> >" template class (Figure 7.3). (Note the blank space between the two '>' symbols, which distinguishes them from the ">>" string, which has a totally different meaning in the "iostream.h" library.)

The "difference" object is actually a list of three vectors: the first vector contains the $D_{j,j-1}$ elements in the tridiagonal matrix D, the second vector contains the $D_{j,j}$ elements (the main diagonal), and the third vector contains the $D_{j,j+1}$ elements. Because the items in a "list" object are declared "protected" in Chapter 3, Section 4, they can be accessed from members of the derived "difference" class:

```
template<class T>
class difference : public list<dynamicVector<T> >{
   public:
    difference(int,const T&,const T&,const T&);
    const difference<T>& operator+=(const difference<T>&);
    const difference<T>& operator-=(const difference<T>&);
    const difference& operator*=(const T&);
```

The constructor and arithmetic operators are only declared above; they will be defined explicitly later.

Particularly important operators in the derived "difference" class are the "operator()" member functions defined below. These operators allow one to refer to the elements $D_{j,j-1}$, $D_{j,j}$, and $D_{j,j+1}$ simply as "D(j,j-1)", "D(j,j)", and "D(j,j+1)", respectively. Because this

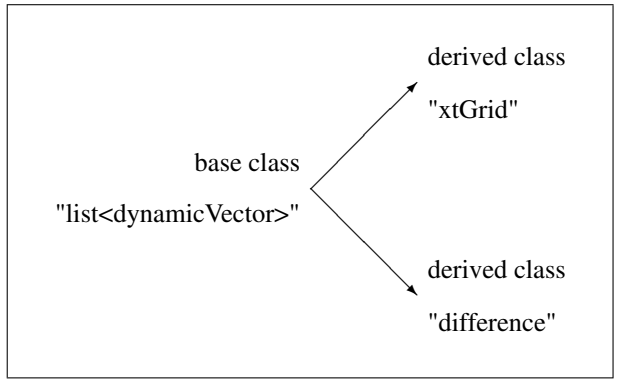

Figure 7.3. *Schematic representation of inheritance from the base class "list" (list of dynamic vectors) to the derived classes "xtGrid" and "difference".*

operator returns a nonconstant reference to the indicated element, the above calls can be used to actually change the value in it. Therefore, they must be used with caution, so that values are not changed inadvertently:

```
      T& operator()(int i,int j){
        return (*item[j-i+1])(i);
      }  //  (i,j)th element (read/write)
```

To read an element of the "difference" object 'D', one can also use the read-only version of "operator()". This version is invoked in calls like "D(j,j,"read")", which also use an extra argument of type "char*" (array of characters). Because this version returns a constant reference, the indicated element can only be read and not changed:

```
      const T& operator()(int i,int j,char*) const{
        return (*item[j-i+1])[i];
      }  //  (i,j)th element (read only)
```

Another required function is the function that returns the order of the tridiagonal matrix:

```
      int width() const{
        return item[0]->dim();
      }  //  width of grid
  };
```

This completes the block of the "difference" class. The copy constructor and assignment operator need not be redefined, because when they are called, the corresponding function in the base "list<dynamicVector<T> >" class is implicitly invoked to set the required values to data fields. Similarly, no destructor needs to be defined, because the implicitly invoked destructor of the base "list" class destroys every data field in the object.

The constructor that takes integer and 'T' arguments, on the other hand, must be redefined explicitly in the derived "difference" class. This is because it implicitly invokes the default constructor of the base "list<dynamicVector<T> >" class, with no arguments at all. This implicit call constructs a trivial list with no items in it, which must then be reset in the body of the constructor below:

```
  template<class T>
  difference<T>::difference(int n=0,
      const T&a=0,const T&b=0,const T&c=0){
    number = 3;
    item = new dynamicVector<T>*[3];
    item[0] = new dynamicVector<T>(n,a);
    item[1] = new dynamicVector<T>(n,b);
    item[2] = new dynamicVector<T>(n,c);
  }  //  constructor
```

Next, we define the required arithmetic operations with "difference" objects. These functions are defined here rather than in the base "list" class in Chapter 3, Section 4. This is because, if they had been defined there, then they would have needed to return a "list<dynamicVector<T> >" object, which would then have needed to be converted to a "difference" object. The present implementation avoids this extra conversion:

```
template<class T>
const difference<T>&
difference<T>::operator+=(const difference<T>&d){
  for(int i=0; i<number; i++)
    *item[i] += d[i];
  return *this;
}  //  adding another "difference" to the current one

template<class T>
const difference<T>&
difference<T>::operator-=(const difference<T>&d){
  for(int i=0; i<number; i++)
    *item[i] -= d[i];
  return *this;
}  //  subtracting a "difference" from the current one
```

So far, we have implemented addition and subtraction of another difference operator to or from the current one. Next, we define multiplication by a scalar:

```
template<class T>
const difference<T>&
difference<T>::operator*=(const T&t){
  for(int i=0; i<size(); i++)
    *item[i] *= t;
  return *this;
}  //  multiplying the current "difference" by a scalar T
```

Next, we implement the above arithmetic operations as (nonmember) binary operators:

```
template<class T>
const difference<T>
operator+(const difference<T>&d1,
        const difference<T>&d2){
  return difference<T>(d1) += d2;
}  //  adding two "difference" objects

template<class T>
const difference<T>
operator-(const difference<T>&d1,
        const difference<T>&d2){
  return difference<T>(d1) -= d2;
}  //  subtracting two "difference" objects

template<class T>
const difference<T>
operator*(const T&t, const difference<T>&d){
  return difference<T>(d) *= t;
}  //  scalar times "difference"
```

```
template<class T>
const difference<T>
operator*(const difference<T>&d, const T&t){
  return difference<T>(d) *= t;
}  //  "difference" times scalar
```

Next, we introduce the operator that returns the product of a difference operator and a vector, or the difference operator applied to a vector. Here, the read-only version of the "operator()" of the "difference" class and the read-only "operator[]" of the "dynamicVector" class are used in the calculations, and the read/write "operator()" of the "dynamicVector" class is then used to assign these calculated values to the appropriate components in the output vector. We also use here the "min()" and "max()" functions from Chapter 1, Section 9:

```
template<class T>
const dynamicVector<T>
operator*(const difference<T>&d,
        const dynamicVector<T>&v){
  dynamicVector<T> dv(v.dim(),0.);
  for(int i=0; i<v.dim(); i++)
    for(int j=max(0,i-1); j<=min(v.dim()-1,i+1); j++)
      dv(i) += d(i,j,"read")*v[j];
  return dv;
}  //  "difference" times vector
```

Next, we implement the operator that "divides" a vector by a difference operator, or, more precisely, solves numerically a tridiagonal linear system of the form $Dx = f$. Because the solution vector x can be expressed as $D^{-1}f$ or f/D, the binary "operator/" seems to be most suitable for denoting this operation.

For simplicity, the solution vector x is computed approximately using the Gauss–Seidel iterative method of Chapter 17, Section 3. Of course, this is not a very efficient linear-system solver; the multigrid algorithm used in Chapter 10, Section 5 and Chapter 17, Section 8, makes a much better solver that is particularly well implemented in C++. Still, the Gauss–Seidel iteration is good enough for our main purpose: writing and debugging the overall algorithm and code for the numerical solution of the convection-diffusion equation:

```
template<class T>
const dynamicVector<T>
operator/(const dynamicVector<T>&f,
        const difference<T>&d){
  dynamicVector<T> x(f);
  for(int iteration=0; iteration < 100; iteration++)
    for(int i=0; i<f.dim(); i++){
      double residual = f[i];
      for(int j=max(0,i-1); j<=min(f.dim()-1,i+1); j++)
        residual -= d(i,j,"read")*x[j];
```

```
        x(i) += residual/d(i,i,"read");
      }
    return x;
  }  //  solving d*x=f approximately by 100 GS iterations
```

7.13 Two Spatial Dimensions

We turn now to the more complicated case of the convection-diffusion equation in two spatial dimensions (the Cartesian dimensions x and y). This equation has the form

$$\begin{aligned} &u_t(t,x,y) - \varepsilon(u_{xx}(t,x,y) + u_{yy}(t,x,y)) \\ &+ C_1(t,x,y)u_x(t,x,y) + C_2(t,x,y)u_y(t,x,y) \\ &= F(t,x,y), \quad 0 < t < T,\ 0 < x < L_x,\ 0 < y < L_y, \end{aligned}$$

where T, L_x, and L_y are positive numbers denoting (respectively) the length of the time interval and the width and length of the two-dimensional spatial domain, and C_1 and C_2 are the given convection coefficients. In order to have a well-posed problem, initial and boundary conditions must also be imposed:

$$\begin{aligned} u(0,x,y) &= u^{(0)}(x,y), \quad 0 \le x \le L_x,\ 0 \le y \le L_y, \\ u(t,x,L_y) &= G(t,x,L_y), \quad 0 \le t \le T,\ 0 \le x \le L_x, \\ u(t,L_x,y) &= G(t,L_x,y), \quad 0 \le t \le T,\ 0 \le y \le L_y, \\ \alpha(t,x,0)u(t,x,0) + u_n(t,x,0) &= G(t,x,0), \quad 0 \le t \le T,\ 0 \le x \le L_x, \\ \alpha(t,0,y)u(t,0,y) + u_n(t,0,y) &= G(t,0,y), \quad 0 \le t \le T,\ 0 \le y \le L_y, \end{aligned}$$

where F, G, and α are given functions and n is the outer normal vector; that is, $n = -x$ at $y = 0$ and $n = -y$ at $x = 0$. Thus, Dirichlet boundary conditions are imposed at the right and upper edges of the rectangular spatial domain, and mixed boundary conditions are imposed on the other two edges.

The finite-difference discretization is as in Section 7.3 above, except that here both the x- and y-derivatives should be discretized. Note also that here the spatial grid is rectangular as in Figure 7.4 rather than one-dimensional as in Section 7.3.

Let us now describe the finite-difference scheme in some more detail. Let N_x and N_y denote the number of grid points in the x and y spatial directions, respectively. Let $h_x = L_x/N_x$ and $h_y = L_y/N_y$ be the corresponding meshsizes. Then, the rectangular $N_x \times N_y$ grid that approximates the spatial domain is the set of pairs

$$(0, h_x, 2h_x, \ldots, (N_x - 1)h_x) \times (0, h_y, 2h_y, \ldots, (N_y - 1)h_y)$$

(see Figure 7.4). Let the time t be fixed, and let u be the $N_x N_y$-dimensional vector that represents the numerical solution of the discrete approximation to the initial-boundary-value problem at the time level corresponding to t. More specifically, the component in u that corresponds to the (i,j)th point in the grid is denoted by $u_{i,j}$. The difference operator D that approximates the spatial derivatives in the PDE (the convection-diffusion terms) has four indices to refer to the elements in it. More specifically, the discrete approximation to

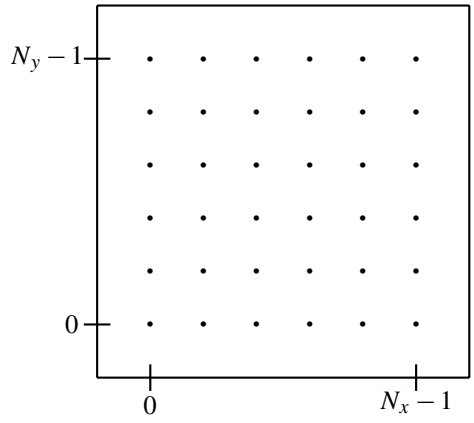

Figure 7.4. *The uniform $N_x \times N_y$ spatial grid for the discretization of the convection-diffusion terms in the unit square $0 \le x, y \le 1$.*

the convection-diffusion terms takes the form

$$\begin{aligned}
&D_{i,j,i,j-1}u_{i,j-1} + D_{i,j,i,j+1}u_{i,j+1} \\
&+ D_{i,j,i-1,j}u_{i-1,j} + D_{i,j,i+1,j}u_{i+1,j} \\
&+ D_{i,j,i,j}u_{i,j} \\
&\doteq -\varepsilon(u_{xx} + u_{yy}) + C_1 u_x + C_2 u_y,
\end{aligned}$$

where

$$\begin{aligned}
D_{i,j,i,j-1} &= -\varepsilon h_x^{-2} - \frac{|C_1(t, jh_x, ih_y)| + C_1(t, jh_x, ih_y)}{2h_x}, \\
D_{i,j,i,j+1} &= -\varepsilon h_x^{-2} - \frac{|C_1(t, jh_x, ih_y)| - C_1(t, jh_x, ih_y)}{2h_x}, \\
D_{i,j,i-1,j} &= -\varepsilon h_y^{-2} - \frac{|C_2(t, jh_x, ih_y)| + C_2(t, jh_x, ih_y)}{2h_y}, \\
D_{i,j,i+1,j} &= -\varepsilon h_y^{-2} - \frac{|C_2(t, jh_x, ih_y)| - C_2(t, jh_x, ih_y)}{2h_y}, \\
D_{i,j,i,j} &= -\left(D_{i,j,i,j-1} + D_{i,j,i,j+1} + D_{i,j,i-1,j} + D_{i,j,i+1,j}\right).
\end{aligned}$$

Of course, points that lie outside the grid must be eliminated using the discrete boundary conditions as in Section 7.4 above. For example, when $i = 0$, $u_{-1,j}$ is eliminated using the discrete mixed boundary conditions at $y = 0$ and $x = jh_x$, and so on.

The rest of the details are in principle the same as in one spatial dimension. This completes the definition of the finite-difference scheme in two spatial dimensions. The actual implementation is left as an exercise; the complete solution can be found in Section A.10 of the Appendix.

7.14 Exercises

1. Modify the code in Section 7.12 above to solve problems with mixed boundary conditions at both the $x = 0$ and $x = L$ edges of the x-interval. The vectors used in this case must be of dimension $N + 1$ rather than N, because an extra unknown at the point $Nh = L$ must also be solved for. The dummy u_{N+1} unknown should be eliminated using the discrete boundary conditions (like the dummy u_{-1} unknown).

2. The code in this chapter uses a long-memory approach, in which the numerical solution in the entire time-space grid is stored. Rewrite it using a short-memory approach, in which the numerical solution at each time step is dropped right after it is used to compute the numerical solution at the next time step, and the output of the code is the numerical solution at the final time step. Note that, with this approach, the "xtGrid" object can be avoided.

3. Modify the time-marching method in the code to use the explicit or implicit scheme rather than the semi-implicit scheme.

4. Use the "dynamicMatrix" class of Section 3.14 to implements a rectangular grid. This object is useful for implementing the individual time steps in the numerical solution of the convection-diffusion equation in two spatial dimensions in Section 7.13 above.

5. Define the "difference2" object that implements the discrete spatial derivatives in both the x- and y-directions in Section 7.13 above. The answer can be found in Section A.10 of the Appendix.

6. Define the required arithmetic operations between the "difference2" and "dynamicMatrix" objects. (In principle, they are the same as the arithmetic operators between the "difference" and "dynamicVector" objects.) The solution can be found in Section A.10 of the Appendix.

7. The "difference2" object in Section A.10 of the Appendix contains nine numbers per grid point, which is not really necessary here. Improve it to contain only five numbers per grid point.

8. Use the above objects to implement the semi-implicit scheme for the convection-diffusion equation in two spatial dimensions. The solution can be found in Section A.10 of the Appendix.

9. Modify your code to use the implicit scheme rather than the semi-implicit scheme.

10. Modify your code to use the explicit scheme rather than the semi-implicit scheme. Run your code with small $\triangle t$. Run it again with larger $\triangle t$. What happens when $\triangle t$ is too large? Does this also happen with the implicit or semi-implicit scheme?

11. Modify your code in such a way that only mixed boundary conditions are used.