Chapter 20

Finite difference schemes for the heat equation in one dimension

Prerequisites: *Chapters 7, 8*

20.1 ▪ The basic idea of finite differences

In this chapter we apply a variety of finite difference techniques to approximate the solutions of initial/boundary value problems associated with the *heat equation*

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}. \tag{20.1}$$

The unknown $u = u(x, t)$ is a function of space x and time t.

The partial differential equation (20.1) arises in a variety of contexts in mathematical physics, probability theory, digital image processing, chemistry, and financial mathematics. Perhaps the most accessible instance is as a model of the temperature in a one-dimensional heat-conducting rod which is thermally insulated all around except for its ends, where it interacts with the outside world. If we hold a flame to one end, heat will propagate through the rod and affect the temperature everywhere. The function $u(x, t)$ is the temperature at the point x at time t. The partial differential equation (20.1) accounts for the conservation of thermal energy within the rod.

We obtain a well-posed heat conduction problem if we specify the rod's temperature at time zero, that is, $u(x, 0)$, and prescribe the temperatures at its ends at all times, that is, in $u(a, t)$ and $u(b, t)$. Here I am assuming that the rod coincides with the interval (a, b) on the x axis. This information, along with (20.1), should suffice to determine the temperature $u(x, t)$ at all points $x \in (a, b)$ and all times $t > 0$. We state this formally as the following initial/boundary value problem:

Find $u = u(x, t)$ so that

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}, \qquad x \in (a, b),\ t > 0, \tag{20.2a}$$

$$u(x, 0) = u_0(x), \qquad x \in (a, b), \tag{20.2b}$$

$$u(a, t) = \alpha(t), \quad u(b, t) = \beta(t), \qquad t > 0. \tag{20.2c}$$

The *initial condition* $u_0(x)$ and the left and right *boundary conditions* $\alpha(t)$ and $\beta(t)$ are

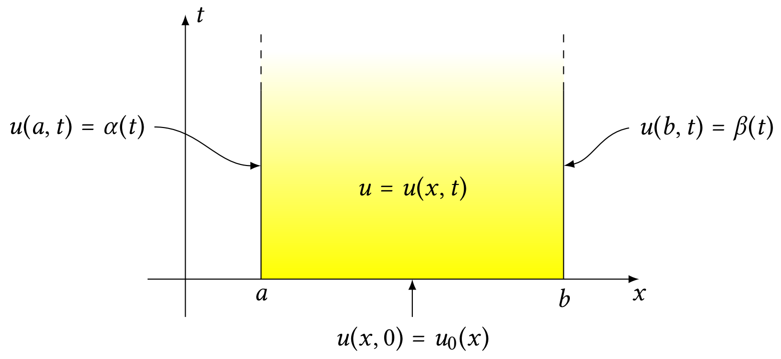

Figure 20.1: The initial condition $u_0(x)$ and boundary conditions $\alpha(t)$ and $\beta(t)$ determine the solution of the heat equation in the shaded semi-infinite strip.

prescribed. They serve to define a unique[79] solution $u(x,t)$ in the semi-infinite strip $a \le x \le b$ and $t > 0$ in the x-t plane. See Figure 20.1.

In the finite differences worldview, space and time are discrete. The interval $a \le x \le b$ is divided into n subintervals the dividing $n+1$ points $x_0 < x_1 < \cdots < x_{n-1}, x_n$, where $x_0 = a$ and $x_n = b$.

We call x_0 and x_n the *boundary points* and the remaining $n-1$, that is, $x_1, x_2, \dots, x_{n-1}$, the *internal points.* We assume, for simplicity's sake, that the $n+1$ dividing points are evenly spaced, and thus they partition the interval $[a,b]$ into n subintervals of length $\Delta x = (b-a)/n$ each. Consequently, $x_j = a + j\Delta x$, $j = 0, 1, \dots, n$.

Similarly, the time is discretized into "time-slices" $t_0 < t_1 < t_2 \cdots$, where $t_0 = 0$. We assume that the time-slices are evenly spaced at a prescribed Δt intervals, and therefore $t_k = k\Delta t$, $k = 0, 1, 2, \dots$.

The discretization of the space and time replaces the shaded strip of Figure 20.1 by a grid of points (x_j, t_k), as depicted in Figure 20.2. The task of finding the function $u(x,t)$ is replaced by the task of computing its values $u(x_j, t_k)$ at the grid points. For convenience we introduce the notation u_j^k for $u(x_j, t_k)$ since it is more compact and easier to parse. I trust that it's clear that k is a superscript here, not an exponent!

The derivative $\partial u/\partial t$ at (x_j, t_k) may be approximated by either of the following two ways:

$$\left.\frac{\partial u}{\partial t}\right|_{(x_j,t_k)} \approx \frac{u_j^{k+1} - u_j^k}{\Delta t} \qquad \text{or} \qquad \left.\frac{\partial u}{\partial t}\right|_{(x_j,t_k)} \approx \frac{u_j^k - u_j^{k-1}}{\Delta t}. \tag{20.3}$$

The first variant is called a *forward difference approximation* since it looks up the value of u at a future time. The second variant is called a *backward difference approximation* since it looks up the value of u at a previous time. Both have their uses, as we shall see.

To approximate the second derivative $\partial^2 u/\partial x^2$ at (x_j, t_k), let us look at the Taylor expansion of $u(x, t_k)$ about $x = x_j$:

$$u(x,t_k) = u(x_j,t_k) + \left.\frac{\partial u}{\partial x}\right|_{(x_j,t_k)}(x - x_j) + \frac{1}{2}\left.\frac{\partial^2 u}{\partial x^2}\right|_{(x_j,t_k)}(x - x_j)^2 + \cdots.$$

[79] I am hiding some technical details here. The existence and uniqueness of a solution u depend on the regularity and integrability of the functions u_0, α, β; see e.g., Friedman [21]. Those details, however, hardly matter for the purposes of this chapter.

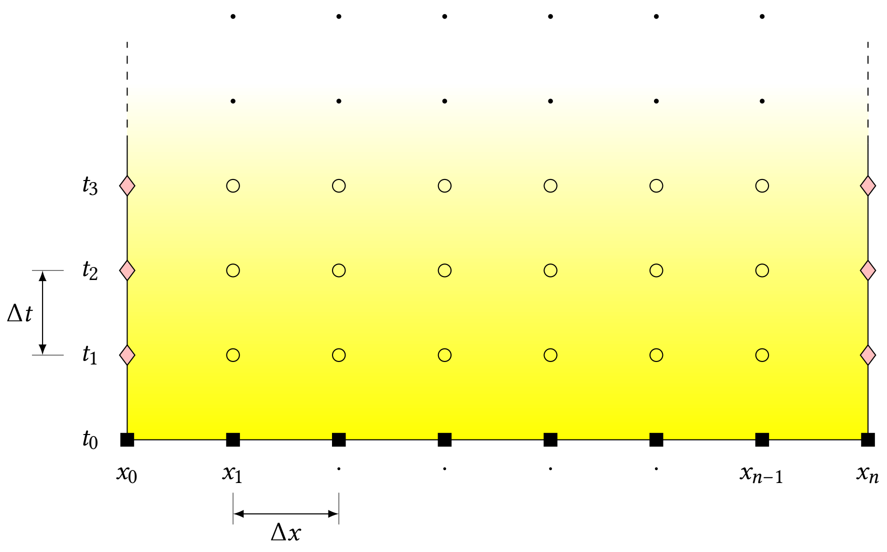

Figure 20.2: The finite difference grid consists of $n+1$ points $x_0, x_1, \dots, x_n$ in the x direction and a sequence of time-slices $t_0, t_1, t_2, \dots$ in the t direction. The initial condition determines the solution at the squares ■. The boundary conditions determine the solution at the diamonds ◊. The finite difference algorithm determines the solution at the rest of the grid points marked with the hollow circles ○.

Evaluating this at $x = x_{j-1}$ and $x = x_{j+1}$ we get

$$u(x_{j-1}, t_k) = u(x_j, t_k) + \frac{\partial u}{\partial x}\Big|_{(x_j,t_k)}(x_{j-1} - x_j) + \frac{1}{2}\frac{\partial^2 u}{\partial x^2}\Big|_{(x_j,t_k)}(x_{j-1} - x_j)^2 + \cdots,$$
$$u(x_{j+1}, t_k) = u(x_j, t_k) + \frac{\partial u}{\partial x}\Big|_{(x_j,t_k)}(x_{j+1} - x_j) + \frac{1}{2}\frac{\partial^2 u}{\partial x^2}\Big|_{(x_j,t_k)}(x_{j+1} + x_j)^2 + \cdots.$$

We then substitute $x_{j+1} + x_j = \Delta x$ and $x_{j-1} + x_j = -\Delta x$, add up the resulting equations, switch to the compact notation u_j^k introduced above, and arrive at

$$u_{j-1}^k + u_{j+1}^k = 2u_j^k + \frac{\partial^2 u}{\partial x^2}\Big|_{(x_j,t_k)}(\Delta x)^2 + \cdots,$$

whence

$$\frac{\partial^2 u}{\partial x^2}\Big|_{(x_j,t_k)} \approx \frac{u_{j-1}^k - 2u_j^k + u_{j+1}^k}{(\Delta x)^2}. \tag{20.4}$$

The approximations in (20.3) and (20.4) are the main tools in the field of finite differences. They may be applied in a variety of ways to approximate the problem (20.2) with discrete versions. We will study a few possibilities in the following sections. To learn more about the subject, you may start with one of [71, 33, 29].

20.2 ▪ An explicit scheme for the heat equation

In the heat equation (20.2a), replace the right-hand side by the approximation given in (20.4) and the left-hand side by the *forward difference approximation* defined in (20.3). We obtain

$$\frac{u_j^{k+1} - u_j^k}{\Delta t} = \frac{u_{j-1}^k - 2u_j^k + u_{j+1}^k}{(\Delta x)^2}, \quad j = 1, 2, \dots, n-1.$$

We see that the space and time increments, Δx and Δt, enter in the form of the combination $\Delta t/(\Delta x)^2$; therefore it makes sense to introduce the notation

$$r = \frac{\Delta t}{(\Delta x)^2} \tag{20.5}$$

and express the equation in terms of r, as in

$$u_j^{k+1} - u_j^k = r(u_{j-1}^k - 2u_j^k + u_{j+1}^k), \quad j = 1, 2, \dots, n-1, \tag{20.6}$$

or the rearranged form

$$u_j^{k+1} = ru_{j-1}^k + (1-2r)u_j^k + ru_{j+1}^k, \quad j = 1, 2, \dots, n-1. \tag{20.7}$$

The result is very revealing. It says that u_j^{k+1}, that is, the value of u at the time-slice $k+1$, may be computed from the values of u at the time-slice k. Since the values of u_j^k at the time-slice $t = 0$ are known—that's what the initial condition is for—we may apply (20.7) recursively, one time-slice at a time, to march forward through the time-slices and determine u_j^k at all times. If you mark the formula's entries on the finite difference grid, as is done in two instances in Figure 20.3, they arrange themselves in a $\perp$-shaped pattern. That pattern, which is a characteristic of the finite difference scheme (20.7), is called the scheme's *stencil.* If the stencil contacts the left or right boundary, as it has in one of the instances shown, it picks up the user-supplied boundary condition there. That's where equations (20.2c) come in.

The recursion formula (20.7) is called an *explicit scheme* since it provides the value of u_j^{k+1} explicitly, with no fuss, in terms of given or previously computed data. In that sense it is quite trivial to implement it in a program—just put the formula in a **`for`**-loop and compute away. We will do exactly that in our implementation. For the conceptual understanding and analysis, however, the matrix form of that formula provides a deeper insight. To obtain the matrix form, it helps to write out several instances of the formula explicitly,

$$\begin{aligned}
j = 1 : \qquad & u_1^{k+1} = ru_0^k + (1-2r)u_1^k + ru_2^k, \\
j = 2 : \qquad & u_2^{k+1} = ru_1^k + (1-2r)u_2^k + ru_3^k, \\
\cdots \qquad & \cdots \\
j = n-1 : \qquad & u_{n-1}^{k+1} = ru_{n-2}^k + (1-2r)u_{n-1}^k + ru_n^k,
\end{aligned}$$

and then, after letting $s = 1 - 2r$, pack the equations into a matrix-vector form:

$$\begin{pmatrix} u_1^{k+1} \\ u_2^{k+1} \\ \vdots \\ u_{n-2}^{k+1} \\ u_{n-1}^{k+1} \end{pmatrix} = \begin{pmatrix} s & r & & & \\ r & s & r & & \\ & \ddots & \ddots & \ddots & \\ & & r & s & r \\ & & & r & s \end{pmatrix} \begin{pmatrix} u_1^k \\ u_2^k \\ \vdots \\ u_{n-2}^k \\ u_{n-1}^k \end{pmatrix} + \begin{pmatrix} ru_0^k \\ 0 \\ \vdots \\ 0 \\ ru_n^k \end{pmatrix}. \tag{20.8}$$

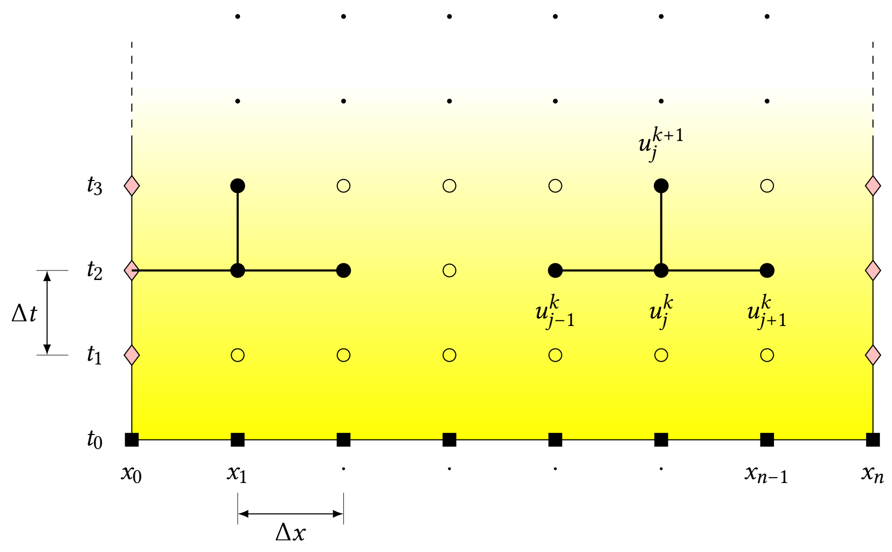

Figure 20.3: The explicit finite difference scheme acts on a ⊥-shaped stencil. It determines the values of u_j^{k+1} at the time-slice $k+1$ in terms of the three values of u from the previous time-slice. When the stencil hits the left or right boundary, its picks up the prescribed boundary value from there.

The matrix is tridiagonal, having $s = 1-2r$ on its main diagonal and r on its first upper and lower subdiagonals. All other entries are zeros. We note that the matrix acts on the grid's internal nodes at the time-slice k. The additive vector in the equation's extreme right imports the prescribed data from the boundary nodes x_0 and x_n. In effect, the equation defines a transition operator that maps the solution from the time-slice k to the time-slice $k+1$.

The recursion scheme (20.7), or its matrix equivalent (20.8), provides an extremely quick and simple method for solving the heat equation and its relatives (general parabolic equations). Of course it is natural to ask the following: Does it produce a good approximation? Does the approximation improve as Δx and Δt go to zero? It turns out that the answers to both questions are a qualified "yes". The catch is that you cannot take Δx and Δt entirely independently of each other. The method is guaranteed to work only if $r \le 1/2$, where r is defined in (20.5).

To get a feel for the source of the trouble, consider the special case where the boundary conditions $\alpha(t)$ and $\beta(t)$ in (20.2c) are zero. Then we expect the rod's temperature to go to zero in the long run, regardless of the initial condition, since its ends are being kept at zero temperature. The finite difference scheme should confirm that. But does it?

When the boundary conditions are zero, the iteration scheme in (20.8) takes the form $\boldsymbol{u}^{k+1} = A\boldsymbol{u}^k$, where A is the tridiagonal matrix in that equation, and $\boldsymbol{u}^k$ is the vector that it multiplies. We see that $\boldsymbol{u}^1 = A\boldsymbol{u}^0$, $\boldsymbol{u}^2 = A\boldsymbol{u}^1$, etc., and consequently $\boldsymbol{u}^k = A^k\boldsymbol{u}^0$. It can be shown (see [33], for instance) that the eigenvalues of A are

$$\lambda_j = 1 - 2r\left(1 - \cos\frac{j\pi}{n}\right), \quad j = 1, 2, \dots, n-1. \tag{20.9}$$

Therefore, if $r \le 1/2$, then all eigenvalues are strictly less than 1, and consequently $A^k \longrightarrow 0$ as $k \longrightarrow \infty$, confirming our expectation. If $r > 1/2$, however, and n is sufficiently large, then it is possible for some of the eigenvalues to be greater than 1; therefore $A^k\boldsymbol{u}^0$ will grow unbounded as $k \longrightarrow \infty$, at least for some choices of $\boldsymbol{u}^0$. That is a totally unreasonable way for heat to behave; the iteration is producing junk!

An iteration scheme is said to be *stable* if small perturbations in the problem's data result only in small perturbations in the solution. Otherwise it is said to be *unstable*. The explicit finite difference scheme defined by (20.7)—or the equivalent (20.8)—is *conditionally stable*: it is stable when $r \leq 1/2$.

Why should conditional stability concern us? Even with the most generous choice of $r = 1/2$, (20.5) tells us that $\Delta t = \frac{1}{2}(\Delta x)^2$. This implies that if Δx is small, then Δt will be uncomfortably small. For instance, if $\Delta x = 1/10$, then $\Delta t = 1/200$. Thus, to compute the solution up to time $t = 1$, we will have to march through 200 time-slices. If we change Δx to 1/100 to achieve a higher accuracy, then Δt changes to $1/20,000$, forcing us to plod through 20,000(!) time-slices to traverse the time interval 0 to 1. That's an inordinate amount of work.

The implicit finite difference scheme, introduced in the next section, removes that restriction on r.

20.3 ▪ An implicit scheme for the heat equation

In the heat equation (20.2a), replace the right-hand side by the approximation given in (20.4) and the left-hand side by the *backward difference approximation* defined in (20.3). We obtain

$$\frac{u_j^k - u_j^{k-1}}{\Delta t} = \frac{u_{j-1}^k - 2u_j^k + u_{j+1}^k}{(\Delta x)^2}, \quad j = 1, 2, \dots, n-1.$$

For notational consistency with the previous section, shift the superscript k up by one, and also set $r = \Delta t/(\Delta x)^2$, as before, to get

$$u_j^{k+1} - u_j^k = r(u_{j-1}^{k+1} - 2u_j^{k+1} + u_{j+1}^{k+1}). \quad j = 1, 2, \dots, n-1, \tag{20.10}$$

Then rearrange/group terms to arrive at

$$-ru_{j-1}^{k+1} + (1 + 2r)u_j^{k+1} - ru_{k+1}^{j+1} = u_j^k, \quad j = 1, 2, \dots, n-1. \tag{20.11}$$

The formula's entries form a ⊤-shaped stencil on the finite difference grid. Two instances of the stencil are shown in Figure 20.4. If the stencil contacts the left or right boundary, as it has in one of the instances shown, it picks up the user-supplied boundary condition there.

On the surface, this looks very similar to the previous section's (20.7). There is, however, something fundamentally different here. Equation (20.7) expresses u at the time-slice $k+1$ explicitly in terms of the values of u at the previous time-slice. Equation (20.11), however, does not do that. It expresses a certain *combination* of the values of u at the time-slice $k+1$ in terms of u at the previous time-slice. It is called an *implicit scheme* for that reason.

To grasp fully what the implicit scheme (20.11) represents, let us write it out in detail:

$$\begin{array}{ll}
j = 1: & -ru_0^{k+1} + (1+2r)u_1^{k+1} - ru_2^{k+1} = u_1^k, \\
j = 2: & -ru_1^{k+1} + (1+2r)u_2^{k+1} - ru_3^{k+1} = u_2^k, \\
\quad \cdots & \qquad \cdots \\
j = n-1: & -ru_{n-2}^{k+1} + (1+2r)u_{n-1}^{k+1} - ru_n^{k+1} = u_{n-1}^k.
\end{array}$$

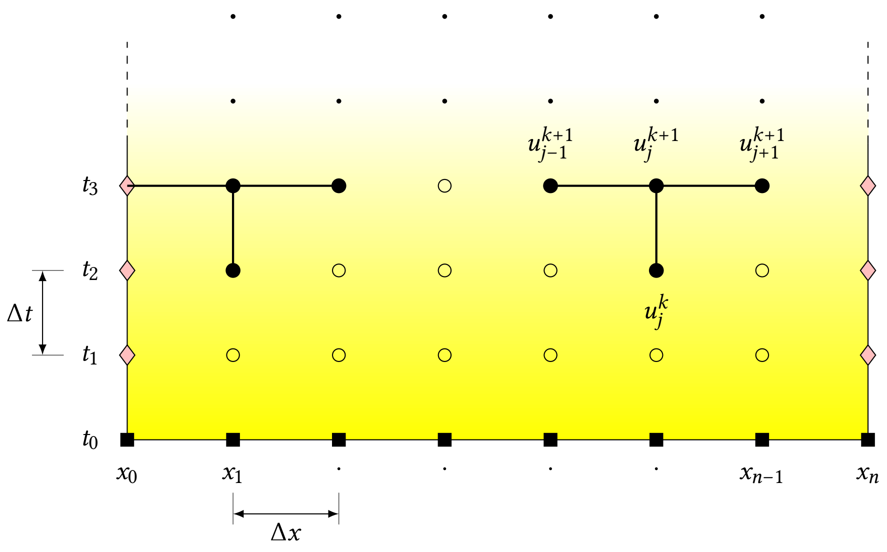

Figure 20.4: The implicit finite difference scheme acts on a ⊤-shaped stencil. It relates a linear combination of three values of u_j^{k+1} at the time-slice $k+1$ to a value of u from the previous time-slice. When the stencil hits the left or right boundary, its picks up the prescribed boundary value from there.

Then, after letting[80] $s = 1 + 2r$, pack it into a matrix-vector form:

$$\begin{pmatrix} s & -r & & & \\ -r & s & -r & & \\ & \ddots & \ddots & \ddots & \\ & & -r & s & -r \\ & & & -r & s \end{pmatrix} \begin{pmatrix} u_1^{k+1} \\ u_2^{k+1} \\ \vdots \\ u_{n-2}^{k+1} \\ u_{n-1}^{k+1} \end{pmatrix} = \begin{pmatrix} u_1^k \\ u_2^k \\ \vdots \\ u_{n-2}^k \\ u_{n-1}^k \end{pmatrix} + \begin{pmatrix} r u_0^{k+1} \\ 0 \\ \vdots \\ 0 \\ r u_n^{k+1} \end{pmatrix}. \tag{20.12}$$

The matrix is tridiagonal, having $s = 1 + 2r$ on its main diagonal and $-r$ on its first upper and lower subdiagonals. All other entries are zeros. As a whole, the equation relates the state of the solution at the grid's internal nodes at the time-slice $k + 1$ to those at the time-slice k. The additive vector in the equation's extreme right imports the prescribed data from the boundary nodes x_0 and x_n.

Computing the solution at the time-slice $k + 1$ calls for solving the system of linear equations (20.12). The task is quite simple on account of the coefficient matrix being tridiagonal. We will study the solution algorithm in subsection 20.9.1. For now, let us look at the scheme's stability.

As in the previous section, consider the special case where the boundary conditions $\alpha(t)$ and $\beta(t)$ in (20.2c) are zero. Then the iteration scheme in (20.12) takes the form $B\boldsymbol{u}^{k+1} = \boldsymbol{u}^k$, where B is the tridiagonal matrix. We see that $B\boldsymbol{u}^1 = \boldsymbol{u}^0$, $B\boldsymbol{u}^2 = \boldsymbol{u}^1$, etc., and consequently $B^k\boldsymbol{u}^k = \boldsymbol{u}^0$, and therefore $\boldsymbol{u}^k = B^{-k}\boldsymbol{u}^0$.

Considering that $s = 1 + 2r$ in this section while $s = 1 - 2r$ in the previous section, it should be evident that the matrix B here is related to the previous section's matrix A through the change of variables $r \longrightarrow -r$. Therefore B's eigenvalues are obtained by

[80]Beware that $s = 1 + 2r$ is different from the previous definitions of s.

changing r to $-r$ in (20.9). Thus, B^{-1}'s eigenvalues are

$$\lambda_j = \frac{1}{1 + 2r\left(1 - \cos \frac{j\pi}{n}\right)}, \quad j = 1, 2, \dots, n-1.$$

We see that these are all strictly less than 1 regardless of the value of r (we are taking it for granted that $r > 0$). It follows that $B^{-k} \longrightarrow 0$ as $k \longrightarrow \infty$; therefore the iteration scheme (20.12) is stable for all r. One expresses this by saying that the scheme is *unconditionally stable.*

Unconditional stability is a *good thing* since it removes the worry about the right choice of r. But lest you jump to unwarranted conclusions, let me point out that things are not quite as rosy as you might expect.

Toward the end of the previous section, when discussing the stability of the explicit scheme, I noted that the requirement $r \le 1/2$ is quite inconvenient since $\Delta t = r(\Delta x)^2$ forces exceedingly small time-steps when Δx is somewhat small. The implicit scheme, however, imposes no restriction on r, so one may be tempted to take arbitrarily large time-steps Δt, uncoupled from the size of Δx. This, however, does not work as well as we may wish. It can be shown (see [33] for instance) that the discretization error in the implicit scheme is of the order of magnitude of $\Delta t + (\Delta x)^2$. For best results we want to keep Δt and $(\Delta x)^2$ in more or less comparable sizes; otherwise the larger of the two will dominate the error. This, in effect, limits Δt to something of the order of magnitude of $(\Delta x)^2$ even though there is no restriction on r. The implicit scheme, therefore, has not released us from the bind of small time-steps.

The Crank–Nicolson scheme, introduced in the next section, gets around this dilemma.

20.4 ▪ The Crank–Nicolson scheme for the heat equation

The Crank–Nicolson scheme is obtained by summing the explicit and implicit differencing formulas (20.6) and (20.10),

$$2(u_j^{k+1} - u_j^k) = r\left(u_{j-1}^k - 2u_j^k + u_{j+1}^k + u_{j-1}^{k+1} - 2u_j^{k+1} + u_{j+1}^{k+1}\right),$$

and regrouping:

$$-ru_{j-1}^{k+1} + 2(1+r)u_j^{k+1} - ru_{j+1}^{k+1} = ru_{j-1}^k + 2(1-r)u_j^k + ru_{j+1}^k. \quad j = 1, 2, \dots, n-1. \tag{20.13}$$

The formula's entries form a ⌶-shaped stencil on the finite difference grid. Two instances of the stencil are shown in Figure 20.5. If the stencil contacts the left or right boundary, as it has in one of the instances shown, it picks up the user-supplied boundary conditions there.

To fully grasp what the Crank–Nicolson scheme (20.13) represents, let us write it out in detail:

$$\begin{aligned}
j = 1 : \quad & -ru_0^{k+1} + 2(1+r)u_1^{k+1} - ru_2^{k+1} = ru_0^k + 2(1-r)u_1^k + ru_2^k,\\
j = 2 : \quad & -ru_1^{k+1} + 2(1+r)u_2^{k+1} - ru_3^{k+1} = ru_1^k + 2(1-r)u_2^k + ru_3^k,\\
\cdots \quad & \cdots\\
j = n-1 : \quad & -ru_{n-2}^{k+1} + 2(1+r)u_{n-1}^{k+1} - ru_n^{k+1} = ru_{n-2}^k + 2(1-r)u_{n-1}^k + ru_n^k.
\end{aligned}$$

After setting[81] $s = 2(1+r)$ and $s' = 2(1-r)$, it takes the form:

[81] Beware that $s = 2(1+r)$ is different from the previous definitions of s.

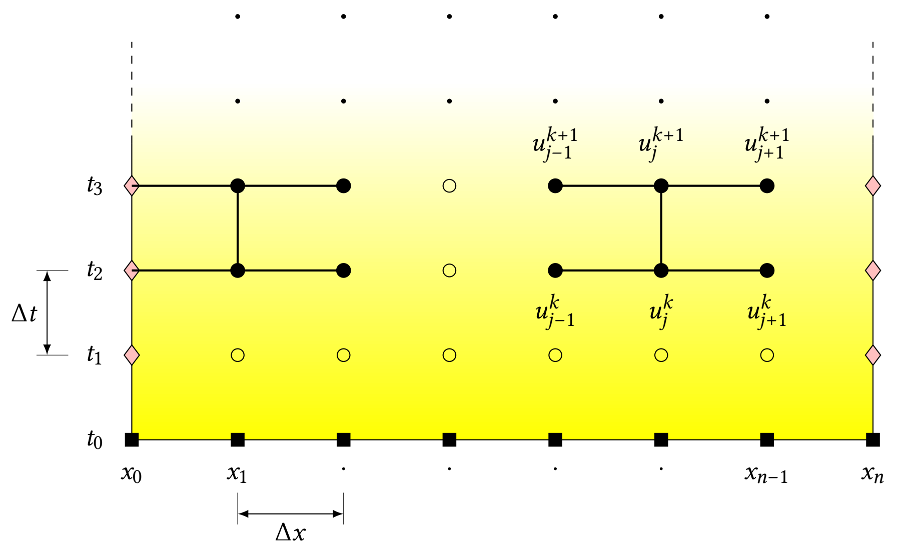

Figure 20.5: The Crank–Nicolson finite difference scheme acts on an ⌶-shaped stencil. It relates a linear combination of three values of u_j^{k+1} at the time-slice $k+1$ to three values of u from the previous time-slice. When the stencil hits the left or right boundary, its picks up the prescribed boundary values from there.

$$\begin{pmatrix} s & -r & & & \\ -r & s & -r & & \\ & \ddots & \ddots & \ddots & \\ & & -r & s & -r \\ & & & -r & s \end{pmatrix} \begin{pmatrix} u_1^{k+1} \\ u_2^{k+1} \\ \vdots \\ u_{n-2}^{k+1} \\ u_{n-1}^{k+1} \end{pmatrix}$$

$$= \begin{pmatrix} s' & r & & & \\ r & s' & r & & \\ & \ddots & \ddots & \ddots & \\ & & r & s' & r \\ & & & r & s' \end{pmatrix} \begin{pmatrix} u_1^k \\ u_2^k \\ \vdots \\ u_{n-2}^k \\ u_{n-1}^k \end{pmatrix} + \begin{pmatrix} ru_0^k + ru_0^{k+1} \\ 0 \\ \vdots \\ 0 \\ ru_n^k + ru_n^{k+1} \end{pmatrix}. \qquad (20.14)$$

It can be shown that the Crank–Nicolson scheme is unconditionally stable for all $r > 0$. That's good news. Even better news is that the discretization error of this scheme is of the order $(\Delta t)^2 + (\Delta x)^2$. Unlike the previous section's implicit scheme, here Δt and Δx occur in equal powers. This allows taking time-steps of the same order of magnitude as the space discretization. For this reason, Crank–Nicolson should be your default finite difference scheme unless other considerations prevail.

Computing the solution at the time-slice $k+1$ calls for solving the system of linear equations (20.14), which, as in the case of the previous section's implicit method, is tridiagonal and can be handled with the algorithm suggested in subsection 20.9.1.

20.5 ▪ The Seidman sweep scheme for the heat equation

In this section I introduce a lesser known difference scheme developed by Seidman [58] which has the dual advantages of being both *explicit* and *unconditionally stable.* As in the previous sections, I will explain the scheme in the context of the heat equation (20.2). The scheme's unique strength, however, is in the ease with which it may be implemented to solve nonlinear problems, as we shall see in Chapter 21.

I must add that the theory and analysis in [58] is developed in the context of general second order parabolic equations in n dimensions on irregular grids. What I present here is a very special case.

The *Seidman sweep*, as I shall call it, is very similar to the explicit scheme of Section 20.2 in that it approximates the time derivative by a forward difference. However, it calculates u_j^{k+1} in a sweeping motion, from left to right (forward sweep) or from right to left (reverse sweep), that is, in increasing or decreasing sequences of j. In the forward sweep, when processing node j, it takes advantage of the availability of u_{j-1}^{k+1} and uses it instead of u_{j-1}^{k}. In the reverse sweep it takes advantage of the availability of u_{j+1}^{k+1} and uses it instead of u_{j-1}^{k}. The idea is reminiscent of the *Gauss–Seidel* iterative scheme for solving linear systems of equations.

The algorithm splits the time-step Δt into two halves. A forward sweep advances time by $(\Delta t)/2$ and calculates $u_j^{k+1/2}$ from u_j^k, $j = 1, 2, \dots, n-1$. That is followed by a reverse sweep, which advances time by $(\Delta t)/2$ and calculates u_j^{k+1} from $u_j^{k+1/2}$, $j = n-1, \dots, 2, 1$. If we consider, for the moment, applying the explicit scheme of Section 20.2 to the forward and reverse sweeps, (20.6) will take the form:

$$\begin{aligned} u_j^{k+1/2} - u_j^k &= r'(u_{j-1}^k - 2u_j^k + u_{j+1}^k), && j = 1, 2, \dots, n-1, \\ u_j^{k+1} - u_j^{k+1/2} &= r'(u_{j-1}^{k+1/2} - 2u_j^{k+1/2} + u_{j+1}^{k+1/2}), && j = n-1, \dots, 2, 1, \end{aligned}$$

where

$$r' = \frac{r}{2} = \frac{\Delta t}{2(\Delta x)^2}, \tag{20.15}$$

since we are taking half steps in time now.

I could present the rest of this section using the $k + 1/2$ superscript notation, but the formulas become cumbersome and obscure the algorithm's simplicity. It works much better with a temporary convention where u_j, v_j, and w_j stand for the u_j^k, $u_j^{k+1/2}$, and u_j^{k+1}, respectively. With this notation, the pair of formulas shown above takes the form

$$\begin{aligned} v_j - u_j &= r'(u_{j-1} - 2u_j + u_{j+1}), && j = 1, 2, \dots, n-1, \\ w_j - v_j &= r'(v_{j-1} - 2v_j + v_{j+1}), && j = n-1, \dots, 2, 1, \end{aligned}$$

or equivalently,

$$\begin{aligned} v_j - u_j &= -r'(u_j - u_{j-1}) - r'(u_j - u_{j+1}), && j = 1, 2, \dots, n-1, \\ w_j - v_j &= -r'(v_j - v_{j-1}) - r'(v_j - v_{j+1}), && j = n-1, \dots, 2, 1. \end{aligned}$$

This is just the explicit scheme up to now. To change it over to the Seidman sweep, we note that in a *forward sweep* the value of v_{j-1} has been calculated prior to arriving at the node j. We take advantage of that and replace $(u_j - u_{j-1})$ in the first of the two formulas above by $(v_j - v_{j-1})$. Similarly, in a *reverse sweep* the value of w_{j+1} has been calculated

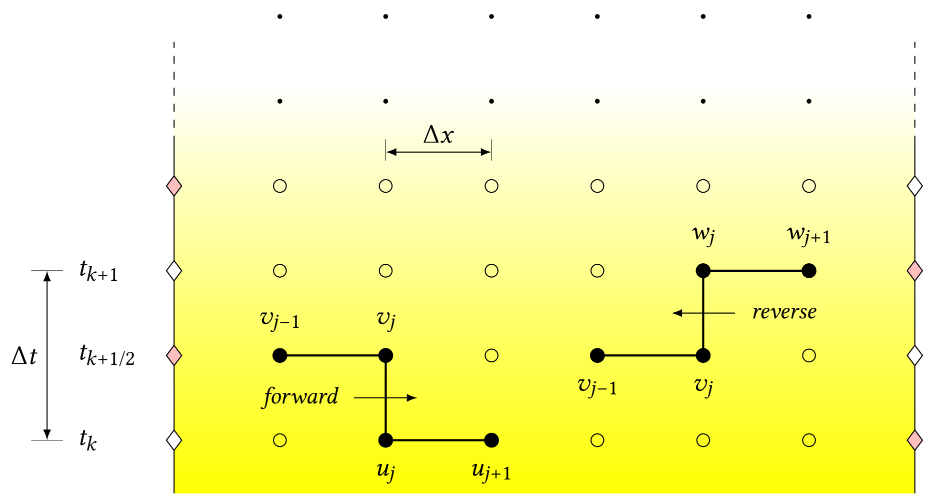

Figure 20.6: The Seidman sweep finite difference scheme advances from the time-slice k to the time-slice $k+1$ via an intermediate time-slice $k+1/2$. The forward sweep advances by half a time-step from k to $k+1/2$. The reverse sweep advances by half a time-step from $k+1/2$ to $k+1$. At any point it uses the most up-to-date data available. The boundary values at the points marked by the filled diamonds ◆ affect the solution in the interior points. The boundary values at the points marked by the hollow diamonds ◇ don't.

prior to arriving at the node j. Therefore, we replace $(v_j - v_{j+1})$ in the second of the two formulas above by $(w_j - w_{j+1})$. These result in

$$v_j - u_j = -r'(v_j - v_{j-1}) - r'(u_j - u_{j+1}), \qquad j = 1, 2, \dots, n-1, \tag{20.16a}$$
$$w_j - v_j = -r'(v_j - v_{j-1}) - r'(w_j - w_{j+1}), \qquad j = n-1, \dots, 2, 1, \tag{20.16b}$$

which we rearrange into

$$(1 + r')v_j = r' v_{j-1} + (1 - r')u_j + r' u_{j+1}, \qquad j = 1, 2, \dots, n-1, \tag{20.17a}$$
$$(1 + r')w_j = r' v_{j-1} + (1 - r')v_j + r' w_{j+1}, \qquad j = n-1, \dots, 2, 1. \tag{20.17b}$$

The pair of formulas (20.17a) and (20.17b) constitutes the *Seidman sweep* scheme. It is an *explicit scheme* since all values on the right-hand sides are available at the time when the left-hand sides are evaluated. Figure 20.6 shows the stencils for the forward and reverse sweeps.

To express the Seidman sweep as a matrix-vector equation, it is best to use the (20.16) form of the scheme. Upon inspection of (20.16a) we see that

$$\begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_{n-2} \\ v_{n-1} \end{pmatrix} - \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-2} \\ u_{n-1} \end{pmatrix} = -r' \begin{pmatrix} 1 & 0 & & & \\ -1 & 1 & 0 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 1 & 0 \\ & & & -1 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_{n-2} \\ v_{n-1} \end{pmatrix}$$
$$- r' \begin{pmatrix} 1 & -1 & & & \\ 0 & 1 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & 0 & 1 & -1 \\ & & & 0 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-2} \\ u_{n-1} \end{pmatrix} + r' \begin{pmatrix} v_0 \\ 0 \\ \vdots \\ 0 \\ u_n \end{pmatrix}.$$

Writing $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$ for the column vectors with the components u_j, v_j, w_j and letting

$$A = \begin{pmatrix} 1 & 0 & & & \\ -1 & 1 & 0 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 1 & 0 \\ & & & -1 & 1 \end{pmatrix}, \quad \boldsymbol{a} = \begin{pmatrix} v_0 \\ 0 \\ \vdots \\ 0 \\ u_n \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} v_0 \\ 0 \\ \vdots \\ 0 \\ w_n \end{pmatrix},$$

this takes on the compact form $\boldsymbol{v} - \boldsymbol{u} = -r'A\boldsymbol{v} - r'A^T\boldsymbol{u} + r'\boldsymbol{a}$, or equivalently,

$$(I + r'A)\boldsymbol{v} = (I - r'A^T)\boldsymbol{u} + r'\boldsymbol{a}, \tag{20.18a}$$

where I is the identity matrix and A^T is the transpose of A. Similar considerations regarding the reverse sweep in the formula (20.16b) lead to

$$(I + r'A^T)\boldsymbol{w} = (I - r'A)\boldsymbol{v} + r'\boldsymbol{b}. \tag{20.18b}$$

The analysis in [58] shows that the iteration scheme expressed in the pair of equations (20.18a) and (20.18b) is *unconditionally stable* for all $r' > 0$. I am not aware of a study of the scheme's *rate of convergence*; and I have not analyzed it myself. Numerical experiments—see the error graphs in Figure 20.7—point to a rate of convergence of the order $(\Delta x)^2 + (\Delta t)^2 + \frac{(\Delta t)^2}{(\Delta x)^2}$, like that of the Du Fort–Frankel scheme (see [71]), but that's a mere conjecture on my part.

In favor of the Seidman sweep, one may note the following:

1. The Seidman sweep, being an explicit method, is easy to program, as we shall see later in this chapter.
2. The Seidman sweep handles discontinuities in the initial and boundary conditions more gracefully than Crank–Nicolson.
3. The Seidman sweep has a definite advantage over implicit schemes in solving nonlinear problems. An implicit scheme, such as Crank–Nicolson, requires solving an $n \times n$ nonlinear tridiagonal system at every step. The Seidman sweep, being an explicit scheme, needs to solve $2n$ single (uncoupled) nonlinear equations in every forward/reverse sweep pair. Chapter 21 applies the Seidman sweep to solve the highly nonlinear porous medium equation.

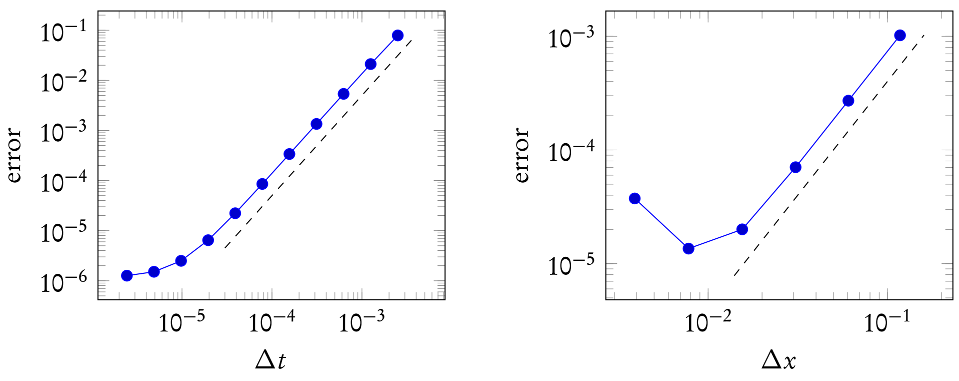

Figure 20.7: Experiments with the Seidman sweep and *problem heat1* (Section 20.6) lend support to the conjecture that the scheme's convergence rate may be $O\big((\Delta x)^2 + (\Delta t)^2 + \frac{(\Delta t)^2}{(\Delta x)^2}\big)$. On the left we have the graph of the errors versus Δt while $\Delta x = 0.004$ is kept fixed. On the right we have the graph of the errors versus Δx while $\Delta t = 5 \times 10^{-5}$ is kept fixed. The dashed lines have slopes of 2. The upturned tail in the latter graph is characteristic of the presence of a $\frac{\Delta t}{\Delta x}$ term.

20.6 ▪ Test problems

Here I introduce four simple initial/boundary value problems for the purpose of testing and demonstrating the various finite difference discretization schemes that were introduced in the previous sections. The programs which we are going to develop are general and certainly not limited to these four. You may easily modify those problems or add new ones of your own. The partial differential equation in all four problems is the heat equation (20.1) on the interval $-1 < x < 1$. Only the initial and boundary conditions are different.

The interval $-1 < x < 1$ is not hard-coded anywhere. The programs are set up to solve a finite difference problem on an interval $a < x < b$, where a and b are defined alongside the rest of the problem's data. Don't hesitate to experiment with defining and solving problems on intervals other than $-1 < x < 1$.

Problems *heat1* and *heat2* come with exact solutions. Our programs compare these against the finite difference solutions and print out the discretization errors. These two problems are constructed according to the following simple "reverse engineering" idea.

Pick any function, let's say $u_{\text{ex}}(x, t)$, that satisfies the partial differential equation (20.1) for all $-\infty < x < \infty$ and $t > 0$. Pose the following initial/boundary value problem whose data is defined in terms of $u_{\text{ex}}(x, t)$:

$$\begin{cases} \dfrac{\partial u}{\partial t} = \dfrac{\partial^2 u}{\partial x^2}, & x \in (a, b),\ t > 0, \\ u(x, 0) = u_{\text{ex}}(x, 0), & x \in (a, b), \\ u(a, t) = u_{\text{ex}}(a, t), & t > 0, \\ u(b, t) = u_{\text{ex}}(b, t), & t > 0. \end{cases} \tag{20.19}$$

Clearly $u(x, t) = u_{\text{ex}}(x, t)$ is a solution of the problem. (Actually, it's *the* solution since such problems have unique solutions.) To test the accuracy of our solvers, we have them solve (20.19) and then compare the results with $u_{\text{ex}}(x, t)$.

Problem *heat1*: You should have no difficulty in verifying that the function

$$u_{\text{ex}}(x,t) = e^{-\frac{1}{2}\pi^2 t}\cos\frac{1}{2}\pi x \tag{20.20}$$

is a solution of the heat equation (20.1). Check that for yourself! We define the problem *heat1* by plugging (20.20) into the general template (20.19) with $a = -1$, $b = 1$.

Problem *heat2*: The *error function*, erf, despite its infelicitous appellation, occurs quite frequently in the study of differential equations, probability, and statistics. It is defined as

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt.$$

The domain of erf is the entire real line. It increases monotonically from -1 at $x = -\infty$ to $+1$ at $x = +\infty$. Figure 20.8 depicts its graph. Since the derivative of $\operatorname{erf}(x)$ is $\frac{2}{\sqrt{\pi}}e^{-x^2}$, you should be able to verify without much trouble that the function

$$u_{\text{ex}}(x,t) = \operatorname{erf}\left(\frac{x}{2\sqrt{t}}\right) \tag{20.21}$$

is an exact solution of the heat equation (20.1). We wish to use this finding to "reverse-engineer" an initial/boundary value problem according to the template (20.19), but we hit a snag: The second of the equations in (20.1) calls for the value of $u_{\text{ex}}(x,0)$. Plugging $t = 0$ in (20.21) won't do since that would entail a division by zero on account of the $\sqrt{t}$ in the denominator. We get around this by observing that although the expression $u_{\text{ex}}(x,t)$ is undefined at $t = 0$, its limit as t approaches zero from above *does* exist:

$$\lim_{t\to 0^+} u_{\text{ex}}(x,t) = \begin{cases} -1 & \text{if } x < 0, \\ +1 & \text{if } x > 0. \end{cases}$$

This still leaves out the $x = 0$ case. There is no way around that since $u_{\text{ex}}(x,t)$ is irreparably discontinuous there. That's not a significant obstacle, however, since the heat equation is good at smearing out discontinuities. Any value assigned to $u_{\text{ex}}(0,0)$ will fade away quite fast. In our program we let $u_{\text{ex}}(0,0) = 0$. That's as good a choice as any. Thus, our u_{ex} really looks like this:

$$u_{\text{ex}}(x,t) = \begin{cases} \operatorname{erf}\left(\frac{x}{2\sqrt{t}}\right) & \text{if } t > 0, \\ -1 & \text{if } t = 0 \text{ and } x < 0, \\ +1 & \text{if } t = 0 \text{ and } x > 0, \\ 0 & \text{if } t = 0 \text{ and } x = 0. \end{cases} \tag{20.22}$$

With the continuity issue out of the way, we define the problem *heat2* by plugging (20.22) into the general template (20.19) with $a = -1$, $b = 1$.

Problem *heat3*: Take the "rectangular bump" function

$$u_0(x) = \begin{cases} 1, & |x| < 0.4, \\ 0 & \text{otherwise} \end{cases}$$

for the initial condition, and define the problem *heat3* as

$$\begin{cases} \dfrac{\partial u}{\partial t} = \dfrac{\partial^2 u}{\partial x^2}, & x \in (-1,1),\ t > 0, \\ u(x,0) = u_0(x), & x \in (-1,1), \\ u(-1,t) = u(1,t) = 0, & t > 0. \end{cases}$$

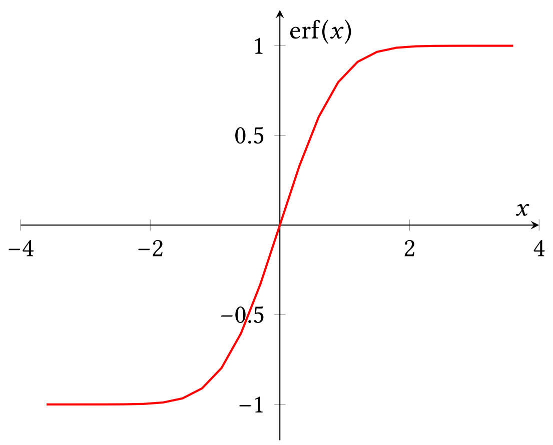

Figure 20.8: The graph of the *error function* erf(x).

Problem *heat4*: This problem is defined as

$$\begin{cases} \dfrac{\partial u}{\partial t} = \dfrac{\partial^2 u}{\partial x^2}, & x \in (-1,1),\ t > 0, \\ u(x,0) = 0, & x \in (-1,1), \\ u(-1,t) = 0, & t > 0, \\ u(1,t) = 1, & t > 0. \end{cases}$$

What does this say about the value of the solution at $x = 1$, $t = 0$?. The initial condition says $u(1,0)$ is 0. The boundary condition says $u(1,0)$ is 1. Therefore we expect the solution $u(x,t)$ to be discontinuous at $x = 1$, $t = 0$. The purpose of this problem is to examine how well the various finite difference schemes handle that discontinuity.

20.7 ▪ The program

The rest of this chapter is devoted to the details of the implementations of the four finite difference schemes introduced earlier. The file *heat-solve.c* will contain the implementations of the four schemes. Section 20.6's test problems will be encoded in the files *demo1.c*, *demo2.c*, *demo3.c*, and *demo4.c*. Each of the test problems can be solved with any of the four finite difference schemes; therefore we are going to produce a total of 16 solutions altogether. A solution is a represented as a two-dimensional array of the computed u_j^k values at the grid points. Our implementation supplies two built-in functions for post-processing the solutions:

1. If an exact solution is supplied, the program computes the largest difference (in absolute value) at the grid points between the computed and exact solutions. This is in effect the *sup norm* of the discretization error.

2. The program can write script files for plotting graphs of solutions as three-dimensional surfaces in Geomview, Maple, or Matlab, if requested.

The user, of course, may do whatever else is desired with the solution array.

In our implementation we rely on *xmalloc.[ch]* from Chapter 7 to allocate memory and the file *array.h* from Chapter 8 to construct vectors and matrices.

Thus, following the suggestions in Chapters 2 and 6, the contents of the project's directory will look like this:

```
$ cd fd1
$ ls -F
Makefile  demo1.c  demo3.c  heat-solve.c  plot3d.c  xmalloc.c@
array.h@  demo2.c  demo4.c  heat-solve.h  plot3d.h  xmalloc.h@
```

I have named my directory `fd1` since the programs here deal with finite difference schemes in a one-dimensional space.

The files *heat-solve.[ch]* provide this project's implementation of the four finite difference schemes described earlier. The demo files *demo[1–4].c* provide illustrative examples. The *plot3d.[ch]* files, whose contents I am not going to describe, are available for downloading from the book's website. These provide functions which receive the problem's solution and generate scripts for plotting the results as 3D surfaces in Geomview, Maple, and Matlab.

Here is a transcript of a sample session with the *demo1* program which solves Section 20.6 problem *heat1*:

```
$ ./demo1
Usage: ./demo1 T n m
    T : solve over 0 ≤ t ≤ T
    n : number of subintervals in the x direction
    m : number of subintervals in the t direction

$. /demo1 1 10 50

--- Explicit finite difference scheme ---
r = 0.5
Maple script written to prob1_explicit.mpl
Matlab script written to prob1_explicit.m
Geomview script written to prob1_explicit.gv
absolute error = 0.0061635

--- Implicit finite difference scheme ---
r = 0.5
Maple script written to prob1_implicit.mpl
Matlab script written to prob1_implicit.m
Geomview script written to prob1_implicit.gv
absolute error = 0.0118496

--- Crank-Nicolson finite difference scheme ---
r = 0.5
Maple script written to prob1_crank_nicolson.mpl
Matlab script written to prob1_crank_nicolson.m
Geomview script written to prob1_crank_nicolson.gv
absolute error = 0.00295428

--- Seidman Sweep finite difference scheme ---
r = 0.25
Maple script written to prob1_seidman_sweep.mpl
Matlab script written to prob1_seidman_sweep.m
Geomview script written to prob1_seidman_sweep.gv
absolute error = 0.0048706
```

The meanings of the program's arguments should be clear from the "Usage" message that appears above. The programs *demo2*, *demo3*, and *demo4* may be invoked in the same way. Figures 20.9 and 20.10 shows the results of running the four programs, each solving the four problems *heat1*, *heat2*, *heat3*, and *heat4*. The figures' captions give the command-line arguments with which the programs were invoked.

In the transcript of the interactive session given above, note that *demo1* was involved with the command-line arguments 1 10 50. In words, the problem is solved over the range $0 \le t \le 1$ through 50 time-steps; therefore $\Delta t = 1/50 = 0.02$. The x domain, which is the interval $(-1, 1)$ in this case, is divided into 10 subintervals, and therefore $\Delta x = 2/10 = 0.2$. We see that $r = (\Delta t)/(\Delta x)^2 = 0.5$, that is, the explicit scheme is within the threshold of stability; see Section 20.2. In contrast, if we run *demo1* with the arguments 0.352 20 20, we will have $r = 1.9 > 0.5$; therefore the explicit scheme will be unstable. Figure 20.11 shows the resulting calamity.

20.8 ▪ The file *heat-solve.h*

The file *heat-solve.h*, shown in it entirety in Listing 20.1, is the interface to our finite difference schemes for solving the initial boundary value problem (20.2) on page 251. Let us examines its contents.

Line 4. The user specifies the desired finite difference scheme through the symbols `FD_explicit`, etc. The symbol `FD_undefined` is intended be used as a default when the user fails to supply a scheme, in which case the program halts with an error message.

Line 7. The structure declared here holds all the necessary data for the problem's specification.

The members `a` **and** `b` hold the values of the coordinates of the endpoints of the problem's spatial domain, (a, b).

The member `T` specifies the time interval of interest, $(0, T)$

The members `n` **and** `m` are the number of subintervals of (a, b) and $(0, T)$, respectively, for the finite difference discretization.

The members `ic`**,** `bcL`**,** `bcR` are pointers to functions that specify the problem's initial, left, and right boundary conditions.

The member `method` specifies the desired finite element scheme. It takes one of the values from the **`enum`** declaration on line 4.

The member `u` is a pointer to an $(m+1)\times(n+1)$ array which will hold the computed u_j^k values of the solution at the grid points.

The member `exact_sol` is a pointer to a function which returns the problem's exact solution. If such a function is supplied, the program calculates the largest difference (in absolute value) at the grid points between the computed and exact solutions and places that value in the `error` member. Set `exact_sol` to `NULL` if no exact solution is available.

The members `maple_out`**,** `matlab_out`**,** `geomview_out` point to file names to which the program will write scripts for plotting the solution in Maple, Matlab, or Geomview. To suppress any of these outputs, set the pointer to `NULL`.

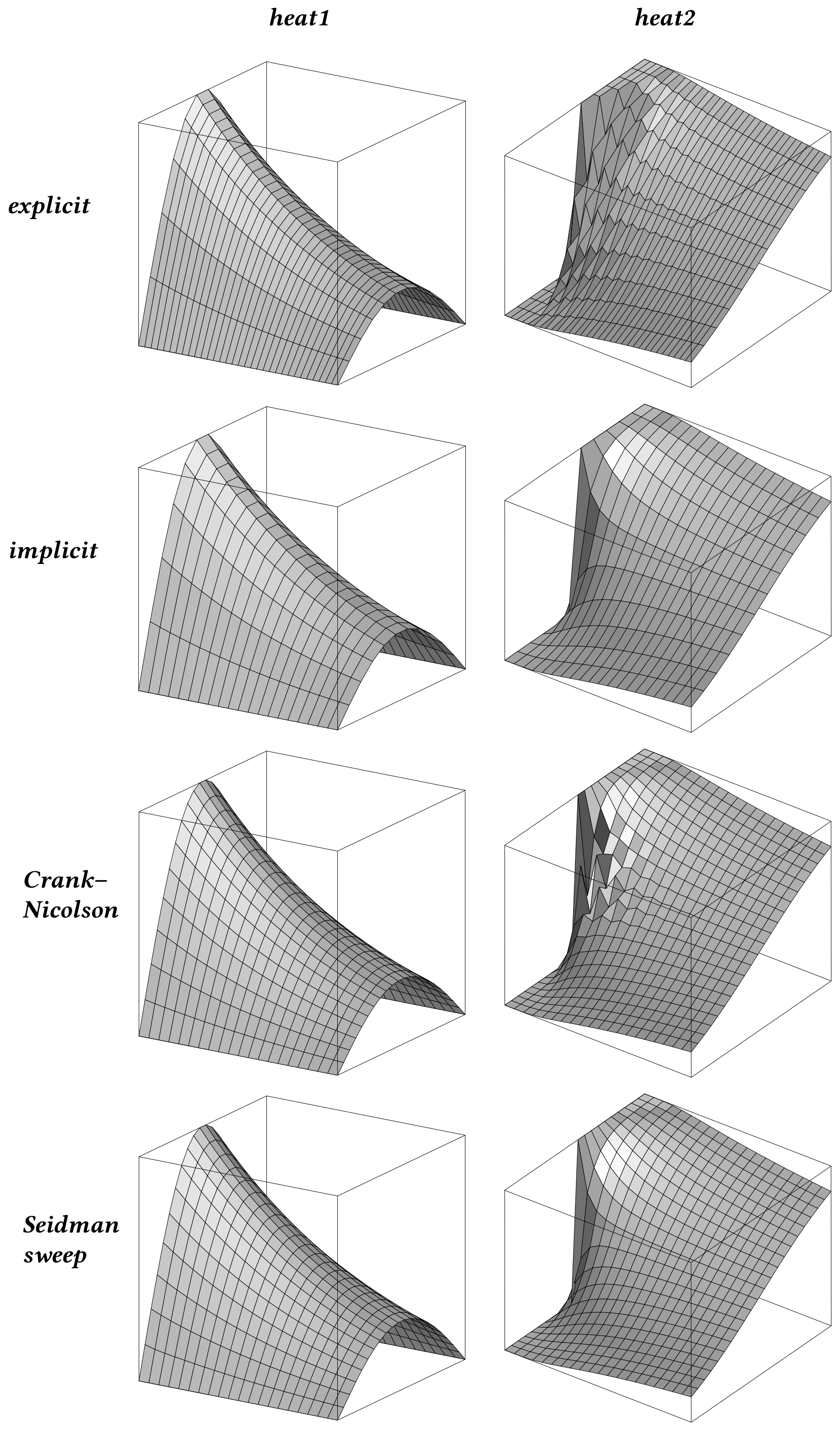

Figure 20.9: Graphs of the solutions to the problems *heat1* and *heat2* produced by the four schemes invoked as `heat-explicit 0.5 10 32`, `heat-implicit 0.5 10 20`, `heat-crank-nicolson 0.5 20 20`, `heat-seidman-sweep 0.5 20 20`.

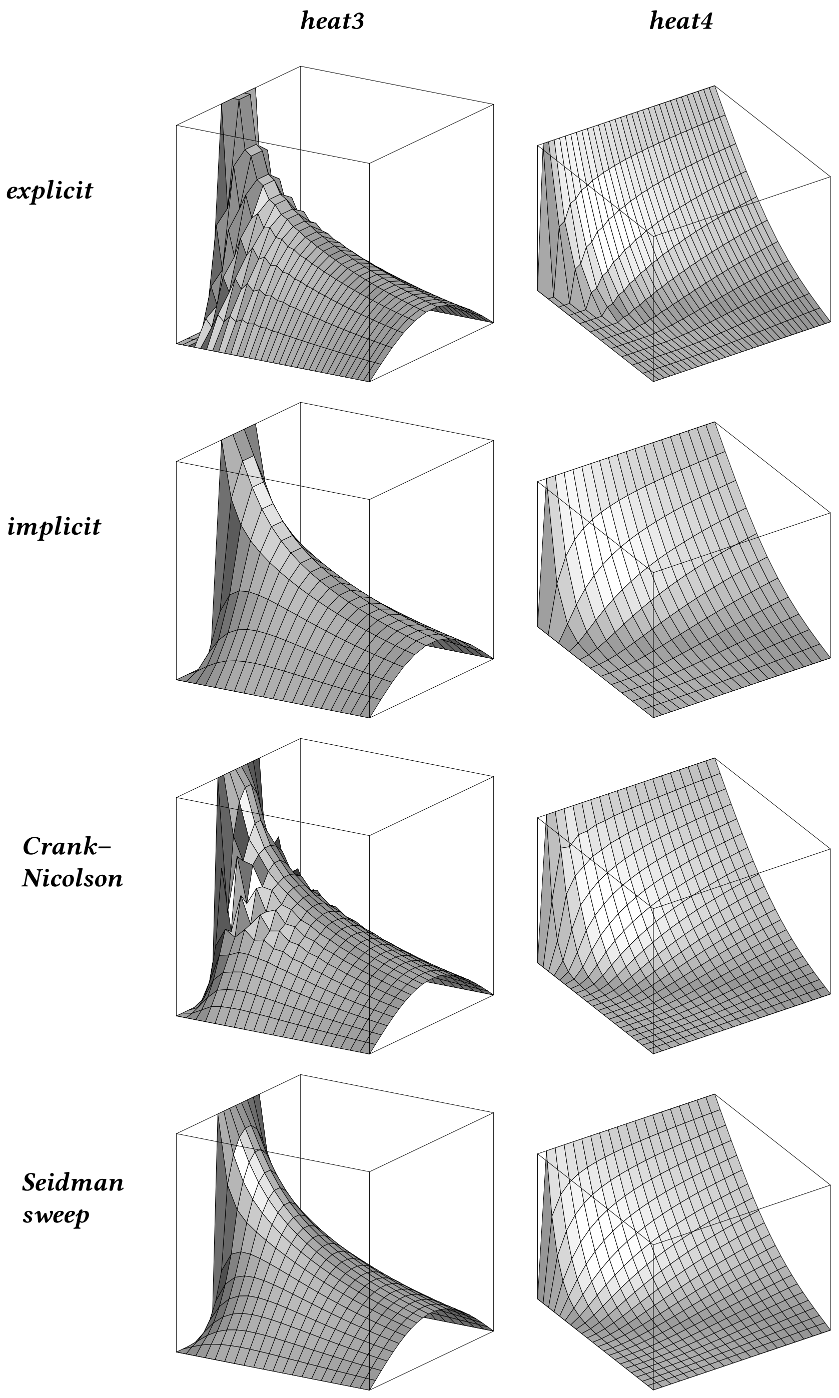

Figure 20.10: Graphs of the solutions to the problems *heat3* and *heat4* produced by the four schemes invoked as `heat-explicit 0.5 10 32`, `heat-implicit 0.5 10 20`, `heat-crank-nicolson 0.5 20 20`, `heat-seidman-sweep 0.5 20 20`.

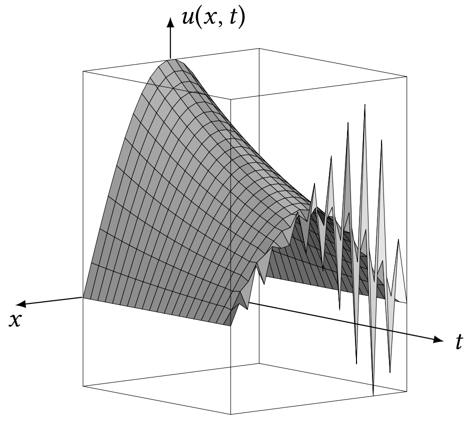

Figure 20.11: Running *demo1* with the command-line arguments `0.38 20 20` yields an unstable explicit scheme since $r = (\Delta t)/(\Delta x)^2 = 1.9 > 0.5$. Here we see the solution of the problem *heat1* begin to fall apart as we approach $t = 0.38$. Beyond this the oscillations grow very large so quickly that drawing a graph is impractical.

Line 25. The function `show_usage_and_exit()` prints a brief help message to the `stdout` and exits the program. See the transcript of the interactive session on page 266 for a sample output.

Line 26. The function `heat_solve()` is implements this project's four finite difference schemes. We turn to its implementation next.

20.9 ▪ The file *heat-solve.c*

The file *heat-solve.c* contains the implementation of this project's four finite difference schemes and a few auxiliary functions. Listing 20.2 shows an outline of this file. I will describe the details in the following subsections.

20.9.1 ▪ The function `trisolve()`

The function `trisolve()` that appears on line 2 of Listing 20.2 is a generic solver of $n \times n$ tridiagonal linear systems of equations[82] of the form

$$\begin{pmatrix} d_0 & c_0 & & & & \\ a_0 & d_1 & c_1 & & & \\ & a_1 & d_2 & c_2 & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{n-3} & d_{n-2} & c_{n-2} \\ & & & & a_{n-2} & d_{n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-2} \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-2} \\ b_{n-1} \end{pmatrix}. \tag{20.23}$$

There is no point in storing the full $n \times n$ coefficient matrix. We store only the diagonal $\langle d_0, \dots, d_{n-1} \rangle$, the subdiagonal $\langle a_0, \dots, a_{n-2} \rangle$, and the superdiagonal $\langle c_0, \dots, c_{n-2} \rangle$ as one-dimensional arrays.

[82]The n in this function should not be confused with the n that appears in the formulation of our finite difference schemes. In fact, those schemes call `trisolve()` with the argument `n` set to $n - 1$.

Listing 20.1: The contents of the file *heat-solve.h*.

```
#ifndef H_HEAT_SOLVE_H
#define H_HEAT_SOLVE_H

enum method { FD_undefined, FD_explicit, FD_implicit,
        FD_crank_nicolson, FD_seidman_sweep };

struct heat_solve {
    double a;                                      // left end at x = a
    double b;                                      // right end at x = b
    double T;                                      // solve for 0 < t < T
    int n;                                         // number of x subintervals
    int m;                                         // number of t subintervals
    double (*ic)(double x);                        // initial condition
    double (*bcL)(double t);                       // left boundary condition
    double (*bcR)(double t);                       // right boundary condition
    enum method method;                            // solution method
    double **u;                                    // solution array
    double (*exact_sol)(double x, double t); // exact solution, if any
    double error;                                  // error vs exact solution
    char *maple_out;                               // output file for maple graphics
    char *matlab_out;                              // output file for matlab graphics
    char *geomview_out;                            // output file for geomview graph-
ics
};

void show_usage_and_exit(char *progname);
void heat_solve(struct heat_solve *prob);

#endif /** H_HEAT_SOLVE_H */
```

Listing 20.2: An outline of the file *heat-solve.c*.

```
▸ #include ...
▸ static void trisolve(int n, const double *a, double *d,
        const double *c, double *b, double *x) ...
▸ static double error_vs_exact(struct heat_solve *prob) ...
▸ static void write_plotting_script(struct heat_solve *prob)
▸ void show_usage_and_exit(char *progname) ...
▸ static void heat_solve_explicit(struct heat_solve *prob) ...
▸ static void heat_solve_implicit(struct heat_solve *prob) ...
▸ static void heat_solve_crank_nicolson(struct heat_solve *prob) ...
▸ static void heat_solve_seidman_sweep(struct heat_solve *prob) ...
▸ void heat_solve(struct heat_solve *prob) ...
```

We assume that the coefficient matrix is nonsingular. Our code does not check for that. It can be shown that matrices that arise through finite difference schemes applied to parabolic problems are nonsingular.

We solve the system through a simple Gaussian elimination without pivoting. To eliminate a_0, we introduce the multiplier $m = a_0/d_0$ and then subtract m times the first equation from the second equation. This (i) reduces the entry in the a_0 position to zero;

Listing 20.3: The function `trisolve()` in the file *heat-solve.c*.

```
 1  static void trisolve(int n, const double *a, double *d,
 2          const double *c, double *b, double *x)
 3  {
 4      for (int i = 1; i < n; i++) {
 5          double m = a[i-1]/d[i-1];
 6          d[i] -= m*c[i-1];
 7          b[i] -= m*b[i-1];
 8      }
 9      x[n-1] = b[n-1]/d[n-1];
10      for (int i = n-2; i ≥ 0; i--)
11          x[i] = (b[i] - c[i]*x[i+1]) / d[i];
12  }
```

(ii) changes the entry in the d_1 position to $d_1 - mc_0$; and (iii) changes the entry in the b_1 position to $b_1 - mb_0$. The entry in the c_1 position does not change.

We repeat the procedure to eliminate the rest of the subdiagonal, in sequence, from top to bottom. When operating on row i ($i = 1, 2, \dots, n-1$) we set $m = a_{i-1}/d_{i-1}$ and then subtract m times row $i-1$ from row i. This (i) eliminates the a_{i-1} entry; (ii) changes the d_i entry to $d_i - mc_{i-1}$; and (iii) changes the b_i entry to $b_i - mb_{i-1}$.

Listing 20.3 shows my implementation of `trisolve()`. In lines 4–8 you will find the literal encoding of the statements made above.

Remark 20.1. We don't bother to zero the a_i entries in our code since we have no use for the a_i's beyond this point. Consequently, the elimination's overall effect is to change the d_i and b_i arrays, but the a_i and c_i arrays remain unchanged! We take advantage of this when calling `trisolve()`, as we will see shortly. The **`const`** qualifiers on lines 1 and 2 in Listing 20.3 assure the compiler that the entries a_i and c_i will not change.

After eliminating the subdiagonal, the system takes the form

$$\begin{pmatrix} d_0 & c_0 & & & & \\ & d_1 & c_1 & & & \\ & & d_2 & c_2 & & \\ & & & \ddots & \ddots & \\ & & & & d_{n-2} & c_{n-2} \\ & & & & & d_{n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-2} \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-2} \\ b_{n-1} \end{pmatrix}.$$

The d_i's and b_i's are different from what they were before, but the c_i's are the same. The system's last equation, that is, $d_{n-1}x_{n-1} = b_{n-1}$, yields $x_{n-1} = b_{n-1}/d_{n-1}$. We compute the rest of the x_i's through *back substitution.* To wit, the equation before the last, that is, $d_{n-2}x_{n-2} + c_{n-2}x_{n-1} = b_{n-2}$, yields the value of x_{n-2} since x_{n-1} is already known. In general, the equation in row i ($i = n-2, \dots, 0$), that is, $d_i x_i + c_i x_{i+1} = b_i$, yields $x_i = (b_i - c_i x_{i+1})/d_i$ in terms of the previously calculated x_{i+1}. Lines 9–11 in Listing 20.3 reflect this finding.

20.9.2 ▪ The function `error_vs_exact()`

The function `error_vs_exact()` that appears on line 4 of Listing 20.2 is called only when an exact solution, $u_{\text{ex}}(x, t)$, is supplied. It scans that two-dimensional array `prob→u`, compares its values with those of the exact solution at the corresponding points, determines the maximum of the absolute values of the differences, and returns that maximum value. The implementation is rather quite straightforward, so I leave that to you.

20.9.3 ▪ The functions `write_plotting_script()` `plot3d()`

The files *plot3d.[ch]*, available on the book's website, provide a facility for plotting the graphs of the solutions produced by our finite difference solver as surfaces in 3D. The file *plot3d.h* declares a structure **`struct`** `plot3d`, which is capable of holding the necessary plotting data. The file *plot3d.c* defines a function `plot3d()` which receives a **`struct`** `plot3d` and writes plotting scripts for Geomview, Maple, or Matlab. The user specifies the name of the script file in in **`struct`** `heat_solve` (see Listing 20.1). Setting a file name to `NULL` suppresses the production the corresponding plotting script.

The function `write_plotting_script()` that appears on line 5 of Listing 20.2 receives the solution produced by our finite difference solver and passes it to `plot3d()`. Here is the function `write_plotting_script()` in its entirety:

Listing 20.4: The function `write_plotting_script()` in the file *heat-solve.c*.

```
static void write_plotting_script(struct heat_solve *prob)
{
    struct plot3d data;
    data.a = prob->a;
    data.b = prob->b;
    data.T = prob->T;
    data.n = prob->n;
    data.m = prob->m;
    data.u = prob->u;
    data.maple_out = prob->maple_out;
    data.matlab_out = prob->matlab_out;
    data.geomview_out = prob->geomview_out;
    plot3d(&data);
}
```

20.9.4 ▪ The function `show_usage_and_exit()`

The function `show_usage_and_exit()` that appears on line 6 of Listing 20.2 is called when the user executes a demo program without the proper command-line arguments. It prints a brief help message to the `stdout` and exits the program. See the transcript of the interactive session on page 266 for a sample output. I will leave it to you to implement this function.

20.9.5 ▪ The function `heat_solve_implicit()`

The functions `heat_solve_explicit()`, `heat_solve_implicit()`, `heat_solve_crank_nicolson()`, and `heat_solve_seidman_sweep()` implement the four finite difference schemes introduced in this chapter. In all four solvers, we represent the solution by an $(m + 1) \times (n + 1)$ array `u`, whose `u[k][j]` entry holds the value of the finite difference solution u_j^k, $0 \le k \le m$, $0 \le j \le n$. The values of `u[k][0]` and `u[k][n]`, `k=0,1,...,m` are known from the boundary conditions, and the values of `u[0][j]`, `j=0,0,...,n` are known from the initial condition. The solver's task is to determine the values of `u[k][j]` at the grid's interior.

Here I will go through the complete contents of the function `heat_solve_implicit()`, shown in Listing 20.5. It implements the implicit finite difference scheme described in Section 20.3. I will leave the writing of the remaining three functions as projects for you.

The main task of the implicit scheme is to solve the tridiagonal system $(n-1)\times(n-1)$ tridiagonal system (20.12) on page 257. When solving for row i of the array `u`, the the first and last entries of that row's $n+1$ entries are known through the prescribed boundary conditions. The system (20.12) is concerned with the $n-1$ *interior unknowns*, that is, our vector of unknowns begins at the second element of row i of `u` and extends to one element before the last.

Our array `u` declared as **`double`** `**u` and constructed through `make_matrix()` of our *Project Vector and Matrix*. Therefore `u[i]` points to row i, and `u[i]+1` points to the second element in that row. Since tridiagonal solver `trisolve()` expects to receive a pointer to the vector of unknowns as is last argument, we pass `u[i]+1` for that argument.

Lines 1 through 12 of Listing 20.5 are self-explanatory. Let us look at the rest of the code.

Lines 14–16. Here we allocate memory for the arrays that make up the tridiagonal system 20.12, or the equivalent general form (20.23) on page 270. We see that the system is defined through the diagonal array d, the sub- and super-diagonals a and c, and the right-hand side array b. In lines 14–16 we allocate memory for these arrays. We note, however, that in the system of interest, that is (20.12), the sub- and super-diagonals are identical. Therefore we use the same array, c, for both. Two observations made in Remark 20.1 on page 272 are critical here: (a) `trisolve()` leaves the super-diagonal unchanged; and (b) although conceptually the process changes the sub-diagonal to all-zeros, `trisolve()` does not actually change the sub-diagonal at all—it just takes zero for any value that would have been read from the sub-diagonal.

Since the system is $(n-1)\times(n-1)$, the length of the main diagonal, d, the subdiagonals, c, are $n-1$ and $n-2$, respectively, as seen in lines 14 and 15. Naturally, the length of the system's right-hand side array, b, should be $n-1$, but in line 16 we allocate space for an array of length $n+1$. That makes the array `b` of the same length as the array of unknowns, `u[i]` and enables us to refer to the corresponding elements through the same index, as in `b[j]` and `u[i][j]`. Otherwise the indices of `b` and `u[i]` will be have to be out of sync. The slots `b[0]` and `b[n]` are not used, but that's small price to pay for clean code.

Line 18. As noted earlier, the array `c` is going the represent both the sub- and super-diagonals of the coefficient matrix in (20.12). The entries of these diagonals are all $-r$, and do not change by the application of `trisolve()`. Here we initialize the array `c`.

Line 21. We initialize the first row of the array `u` by applying the initial condition.

Lines 26–41. This is the main iterative loop where we march forward in time. The first row, `u[0]`, of the array `u` has been fixed. We calculate rows $i=1$ through $i=m$ by applying (20.12). In row `i`, we begin by setting the values of `u[i][0]` and `u[i][n]` by applying the boundary conditions. Then we construct the right-hand side vector `b` of (20.12) in the obvious way, On line 37 we set the entries of the diagonal array of the tridiagonal system to all $1+2r$'s, and then we call `trisolve()`

to calculate `u[i]` in the interior nodes of row `i`. Note the `b+1` and `u[i]+1` in the that call—as it was pointed out earlier, the call to `trisolve()` solves for the interior nodes of the gird, that's why we offset the pointers by one.

Finally, let us note that the array `c` is initialized outside the **`for`**-loop (line 18) since `c` does not change during the iteration, but the array `d` is initialized inside the **`for`**-loop (line 37) since the application of `trisolve()` changes `d` and therefore it needs to be reset in each iteration.

20.9.6 ▪ The function `heat_solve()`

The function `heat_solve()` that appears on line 11 of Listing 20.2 on page 270 is the front-end to our finite difference solvers. The functions `heat_solve_explicit()`, `heat_solve_implicit()`, etc., are declared **`static`** in the file *heat-solve.c* and are not user-accessible. They are accessed through `heat_solve()`. Listing 20.6 shows its implementation.

The bulk of the function consists of a **`switch`** statement which passes the control to one of the four finite difference scheme, based on the user's requested `method`. Afterward, it calls `write_plotting_script()` (see section 20.9.3) to produce plotting scripts if the user has requested any. Finally, if an exact solution is provided, it calls `error_vs_exact()` (see section 20.9.2) to calculate the absolute error, and stores the result in the `error` field of the **`struct`** `heat_solve`.

20.10 ▪ The file *demo1.c*

The file *demo1.c* sets up the initial boundary value problem described in Section 20.6. The code is straightforward but somewhat long. I have broken it into three parts on listings 20.7, 20.8, and 20.9.

Listing 20.5: The function `heat_solve_implicit()` in the file *heat-solve.c*.

```
static void heat_solve_implicit(struct heat_solve *prob)
{
    int m = prob->m;
    int n = prob->n;
    double **u = prob->u;
    double dx = (prob->b - prob->a) / n;      // space-step
    double dt = prob->T / m;                  // time-step
    double r = dt/(dx*dx);
    double *b, *d, *c;

    printf("--- Implicit finite difference scheme ---\n");
    printf("r = %g\n", r);

    make_vector(d, n-1);
    make_vector(c, n-2);
    make_vector(b, n+1);

    for (int j = 0; j < n-2; j++)
        c[j] = -r;

    for (int j = 0; j <= n; j++) {
        double x = prob->a + j*dx;
        u[0][j] = prob->ic(x);
    }

    for (int i = 1; i <= m; i++) {
        double t = i*dt;
        u[i][0] = prob->bcL(t);
        u[i][n] = prob->bcR(t);

        for (int j = 1; j <= n-1; j++)
            b[j] = u[i-1][j];

        b[1]   += r*u[i][0];
        b[n-1] += r*u[i][n];

        for (int k = 0; k < n-1; k++)
            d[k] = 1 + 2*r;

        trisolve(n-1, c, d, c, b+1, u[i]+1);
    }

    free_vector(b);
    free_vector(c);
    free_vector(d);
}
```

Listing 20.6: The function `heat_solve()` in the file *heat-solve.c*.

```c
void heat_solve(struct heat_solve *prob)
{
    switch (prob->method) {
        case FD_explicit:
            heat_solve_explicit(prob);
            break;
        case FD_implicit:
            heat_solve_implicit(prob);
            break;
        case FD_crank_nicolson:
            heat_solve_crank_nicolson(prob);
            break;
        case FD_seidman_sweep:
            heat_solve_seidman_sweep(prob);
            break;
        default:
            fprintf(stderr, "*** Missing ‘method’ specification "
                    "in struct heat_solve\n");
            exit(EXIT_FAILURE);
    }

    write_plotting_script(prob);

    if (prob->exact_sol != NULL)
        prob->error = error_vs_exact(prob);
}
```

Listing 20.7: The file *demo1.c* – part 1 or 3.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "array.h"
#include "heat-solve.h"

#define PI  4.0*atan(1.0)

/* A simple IBVP for the heat equation, with exact solution
 * u_t = u_xx          -1 < x < 1,    t > 0
 * exact_sol = e^(-pi^2*t) * cos(pi/2*x)
*/

static double exact_sol(double x, double t)
{
    return exp(-PI*PI/4*t) * cos(PI/2*x);
}

/* calulate initial and boundary conditions from the exact solution */
static double ic(double x)
{
    return exact_sol(x, 0);
}

static double bc_L(double t)
{
    return exact_sol(-1, t);
}

static double bc_R(double t)
{
    return exact_sol(1, t);
}
```

Listing 20.8: The file *demo1.c* – part 2 or 3.

```
int main(int argc, char **argv)
{
    char *endptr;

    if (argc ≠ 4)
        show_usage_and_exit(argv[0]);

    double T = strtod(argv[1], &endptr);
    if (*endptr ≠ '\0' || T ≤ 0.0)
        show_usage_and_exit(argv[0]);

    int n = strtol(argv[2], &endptr, 10);
    if (*endptr ≠ '\0' || n < 1)
        show_usage_and_exit(argv[0]);

    int m = strtol(argv[3], &endptr, 10);
    if (*endptr ≠ '\0' || m < 1)
        show_usage_and_exit(argv[0]);

    struct heat_solve prob = {
        .a      = -1,
        .b      = 1,
        .T      = T,
        .n      = n,
        .m      = m,
        .ic     = ic,
        .bcL        = bc_L,
        .bcR        = bc_R,
        .method     = FD_undefined,  // will be specified later
        .exact_sol  = exact_sol,
        .maple_out  = NULL,
        .matlab_out = NULL,
        .geomview_out   = NULL,
    };

    make_matrix(prob.u, m+1, n+1);
```

Listing 20.9: The file *demo1.c* – part 3 or 3.

```
    /* Solve the problem through four different algorithms.
     * Note: Matlab does not allow a hyphen in file name,
     * so we use underscores for all our output file names.
    */

    putchar('\n');

    prob.method = FD_explicit;
    prob.maple_out = "prob1_explicit.mpl";
    prob.matlab_out = "prob1_explicit.m";
    prob.geomview_out = "prob1_explicit.gv";
    heat_solve(&prob);
    if (prob.exact_sol ≠ NULL)
        printf("absolute error = %g\n", prob.error);
    putchar('\n');

    prob.method = FD_implicit;
    prob.maple_out = "prob1_implicit.mpl";
    prob.matlab_out = "prob1_implicit.m";
    prob.geomview_out = "prob1_implicit.gv";
    heat_solve(&prob);
    if (prob.exact_sol ≠ NULL)
        printf("absolute error = %g\n", prob.error);
    putchar('\n');

    prob.method = FD_crank_nicolson;
    prob.maple_out = "prob1_crank_nicolson.mpl";
    prob.matlab_out = "prob1_crank_nicolson.m";
    prob.geomview_out = "prob1_crank_nicolson.gv";
    heat_solve(&prob);
    if (prob.exact_sol ≠ NULL)
        printf("absolute error = %g\n", prob.error);
    putchar('\n');

    prob.method = FD_seidman_sweep;
    prob.maple_out = "prob1_seidman_sweep.mpl";
    prob.matlab_out = "prob1_seidman_sweep.m";
    prob.geomview_out = "prob1_seidman_sweep.gv";
    heat_solve(&prob);
    if (prob.exact_sol ≠ NULL)
        printf("absolute error = %g\n", prob.error);
    putchar('\n');

    free_matrix(prob.u);

    return EXIT_SUCCESS;
}
/**D: demo1**/
```