6
Numerical Computation of Solutions

Up to this point we have examined PDEs from an analytical viewpoint, often seeking a formula for the solution. Now we devote a brief chapter to solving PDEs numerically. It is a fact that in industry and applied science PDEs are almost always solved numerically on a computer; most real-world problems are too complicated to solve analytically. And, even if a problem can be solved analytically, usually the solution is in the form of a difficult integral or an infinite series, requiring a numerical calculation anyway. This chapter presents a brief introduction to one method, the finite difference method. There are many other methods, for example, the finite element method, to mention only one. Numerical methods for PDEs have been and continue to be one of the most active research areas in applied mathematics, computer science, and the applied sciences as investigators seek faster and more accurate algorithms. Another feature is that numerical methods give tremendous insight into the basic nature and theory of PDEs. The reader will find a large amount of literature on the subject; it is a field within itself. Brief introductions to finite difference methods can be found in Smith (1978), Logan (1987), and Holmes (2007).

In the **finite difference method** the idea is to replace the *continuous* PDE by a *discrete* algebraic problem that can be solved on a computer in finitely many steps. The result is a discrete solution which is known approximately at only finitely many points of the domain. Simply, partial derivatives in the equation are replaced by their difference-quotient approximations; this leads to a difference equation relating the discrete approximations of the solution. We then develop a computer code to solve for the discrete approximations. Although the procedure is straightforward, it does not come without risks.

J. D. Logan, *Applied Partial Differential Equations,* 3rd edition.
Undergraduate Texts in Mathematics, DOI 10.1007/978-3-319-12493-3_6

If care is not taken, the procedure can lead to disastrous results even when it appears that the discrete approximation to the continuous problem is very sensible.

6.1 Finite Difference Approximations

Discrete approximations for derivatives can be obtained by **Taylor's theorem**, one of the most important results in calculus. It gives both the form and the accuracy of the approximation. If we assume a function f is sufficiently differentiable, say, having $n+1$ continuous derivatives in a neighborhood of a point x, then Taylor's theorem states that for h sufficiently small, $f(x)$ can be expanded in as

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \frac{1}{3}f'''(x)h^3 + \cdots + \frac{1}{n!}f^{(n)}(x)h^n \\ + \frac{1}{(n+1)!}f^{(n+1)}(\overline{x})h^{n+1},$$

where $\overline{x}$ is some point between x and $x+h$. The last term in this expansion is the *error term*, and we say it is order $n+1$; we write it as $\mathrm{O}(h^{n+1})$, a general expression meaning that it is some constant times the factor h^{n+1}.

The reader should be familiar with this approximation strategy for initial value problems for ordinary differential equations. We review the simplest method.

Example 6.1

(**Euler's Method**) The problem is to numerically solve the initial value problem (IVP)

$$y' = f(t,y), \;\; t_0 < t < T; \;\; y(t_0) = y_0,$$

where f and f_y are continuous in both arguments in an open neighborhood containing the initial point (t_0, y_0). These conditions guarantee a unique solution to the IVP on an interval containing t_0. The first step in computing a numerical approximation is to discretize the interval $t_0 \le t \le T$ by defining a finite number of points $t_n = t_0 + nh$, $n = 0, 1, \ldots, N$, where h is the **step size** defined by $h = (T - t_0)/N$. Thus, $t_{n+1} = t_n + h$. At each discrete point t_n of the domain we determine an approximation Y_n to the exact value $y(t_n)$ of the solution at that point. From Taylor's theorem we know that

$$y(t_{n+1}) = y(t_n) + hy'(t_n) + \mathrm{O}(h^2) = y(t_n) + hf(t_n, y(t_n)) + \mathrm{O}(h^2).$$

If we replace $y(t_n)$ by its approximation Y_n and drop the $\mathrm{O}(h^2)$ term, the so-called truncation error, we get

$$Y_{n+1} = Y_n + hf(t_n, Y_n), \quad n = 0, 1, \ldots, N, \quad Y_0 = y_0.$$

This is the *Euler method*, and we can easily compute the approximations Y_1, $Y_2, \ldots$, recursively from $n = 0$ to $n = N - 1$. The Euler method is a *marching scheme* where the values are computed successively, forward in time. It is only one of many, many schemes designed for numerical computation of IVPs. A MATLAB script named *EulerMethod* at the end of the chapter carries out this iteration process for the initial value problem

$$y' = -0.1\left(y - 15 - 12\cos\left(\frac{3.14t}{12}\right)\right), \quad y(0) = 68$$

over the interval $0 \le t \le 72$ using 1000 time steps. $\square$

Solving PDEs numerically requires finite difference approximations for various partial derivatives. If $u = u(x,t)$ is a sufficiently smooth function, then Taylor's theorem implies

$$u(x+h,t) = u(x,t) + u_x(x,t)h + \frac{1}{2}u_{xx}(x,t)h^2 + \frac{1}{3!}u_{xxx}(x,t)h^3 + \mathrm{O}(h^4), \quad (6.1)$$

where h is an increment of x. Therefore, taking only three terms on the right, we obtain the *forward difference approximation*

$$u_x(x,t) = \frac{u(x+h,t) - u(x,t)}{h} + \mathrm{O}(h).$$

In a similar way, if k is an increment of t,

$$u_t(x,t) = \frac{u(x,t+k) - u(x,t)}{k} + \mathrm{O}(h).$$

To obtain an approximation for u_{xx}, we can write (6.1) for h replaced by $-h$ to obtain

$$u(x-h,t) = u(x,t) - u_x(x,t)h + \frac{1}{2}u_{xx}(x,t)h^2 - \frac{1}{3!}u_{xxx}(x,t)h^3 + \mathrm{O}(h^4).$$

Adding this expression to (6.1) and solving for u_{xx} gives the *centered difference approximation*

$$u_{xx} = \frac{u(x-h,t) - 2u(x,t) + u(x+h,t)}{h^2} + \mathrm{O}(h^2),$$

which is an approximation for the second derivative. Similarly,

$$u_{tt} = \frac{u(x,t-k) - 2u(x,t) + u(x,t+k)}{k^2} + \mathrm{O}(k^2).$$

6.2 Explicit Scheme for the Heat Equation

The idea of marching forward in time, as in Euler's method, goes over to evolution problems in PDEs. It this section we illustrate and explicit finite difference method for an initial boundary value problem associated with the diffusion equation:

$$u_t = Du_{xx}, \quad 0 < x < 1, \quad t > 0, \tag{6.2}$$

$$u(0,t) = u(1,t) = 0, \quad t > 0, \tag{6.3}$$

$$u(x,0) = f(x), \quad 0 < x < 1. \tag{6.4}$$

The first step to discretize the region of space–time where we want to obtain a solution. In this case the region is $0 \le x \le 1$, $0 \le t \le T$. We put a bound on time because in practice we only solve a problem up until a finite time. Discretizing means defining a two-dimensional **lattice** of points in this space–time region by

$$x_j = jh, \quad t_n = nk, \quad j = 0, 1, \ldots, J; \quad n = 0, 1, \ldots, N,$$

where the fixed numbers h and k are the spatial and temporal *step sizes*, respectively, given by $h = 1/J$ and $k = 1/N$. The integer J is the number of subintervals in $0 \le x \le 1$, and N is the number of time steps to be taken. Figure 6.1 shows the lattice of points; this lattice is also called a **grid**, and the points are called **nodes**. At each lattice point (x_j, t_n) we seek an approximation, which we call U_j^n, to the exact value $u(x_j, t_n)$ of the solution. Note that the superscript n on U_j^n is an index referring to time, not an exponent; and the index subscript j refers to space. We can regard the grid function U_j^n as a two-dimensional array, or matrix, where n is a column index and j is a row index. To obtain equations for the approximations U_j^n, we replace the partial derivatives in the PDE by their difference approximations obtained in Section 6.1. So the continuous PDE (6.2) at the point (x_j, t_n) is replaced by the difference equation

$$\frac{U_j^{n+1} - U_j^n}{k} = D\frac{U_{j-1}^n - 2U_j^n + U_{j+1}^n}{h^2},$$

where we have dropped the error terms. Solving for U_j^{n+1} gives

$$U_j^{n+1} = U_j^n + r\left(U_{j-1}^n - 2U_j^n + U_{j+1}^n\right), \tag{6.5}$$

where

$$r = \frac{kD}{h^2}.$$

Observe that the difference equation (6.5) relates the approximate values of the solution at the four points $(x_{j-1}, t_n), (x_j, t_n), (x_{j-1}, t_n), (x_j, t_{n+1})$. These four

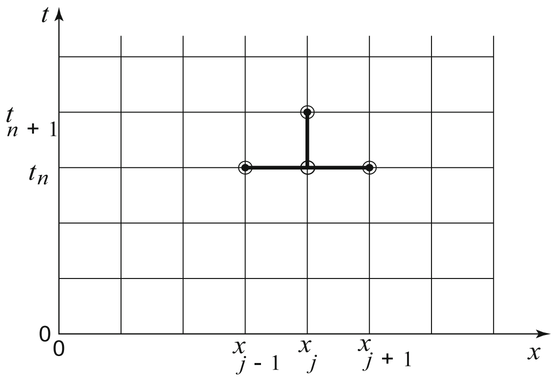

Figure 6.1 The discrete lattice, or grid, and the computational atom of the explicit scheme for diffusion equation. The atom permits calculation of U_j^{n+1} at the $(n+1)$st time level in terms of the three values U_{j-1}^n, U_j^n, U_{j+1}^n at the previous time level n. So we can fill up an entire grid row by knowing the grid values at the preceding row

points form the **computational atom** for the difference scheme (see Figure 6.1).

The difference equation (6.5) gives the approximate solution at the node (x_j, t_{n+1}) in terms of approximations at three earlier nodes. Now we see how to fill up the lattice with approximate values. We know the values at $t = 0$ from the initial condition (6.4). That is, we know

$$U_j^0 = f(x_j), \quad j = 0, 1, \ldots, J.$$

From the boundary conditions (6.3) we also know

$$U_0^n = 0, \;\; U_J^n = 0, \quad n = 1, 2, \ldots, N.$$

The difference formula (6.5) can now be applied at all the *interior* lattice points, beginning with the values at the $t = 0$ level, to compute the values at the $t = t_1$ level, then using those to compute the values at the $t = t_2$ level, and so on. Therefore, using the difference equation, we can march forward in time, continually updating the previous temperature profile. Think of filling out the array U_j^n, row by row, i.e., time after time, in a marching type scheme. In physical terms, the numerical solution carries the initial temperature distribution forward in time; along each horizontal time line n, we can think of the j values, U_j^n, $j = 0, 1, 2, \ldots, J$, as an approximate time profile.

A finite difference scheme, or algorithm, like (6.5) is called an **explicit scheme** because it permits the explicit calculation of an approximation at the next time step in terms of values at a previous time step. Because of the error (called the truncation error) that is present in (6.5) due to replacing derivatives by differences, the scheme is not always accurate. As it turns out, if the time step k is too large, then highly inaccurate results will be obtained. It can be shown that we must have the **stability condition**

$$r = \frac{kD}{h^2} \leq \frac{1}{2}$$

for the scheme to converge. This condition constrains the time step according to $k < 0.5h^2/D$. Experiments that illustrate this behavior are suggested in the exercises.

It is straightforward to code the explicit algorithm (6.5) described above to calculate approximate temperatures in a rod. A simple MATLAB program listing entitled *ExplicitHeat* is given at the end of the chapter. The solution surface is plotted in Figure 6.2.

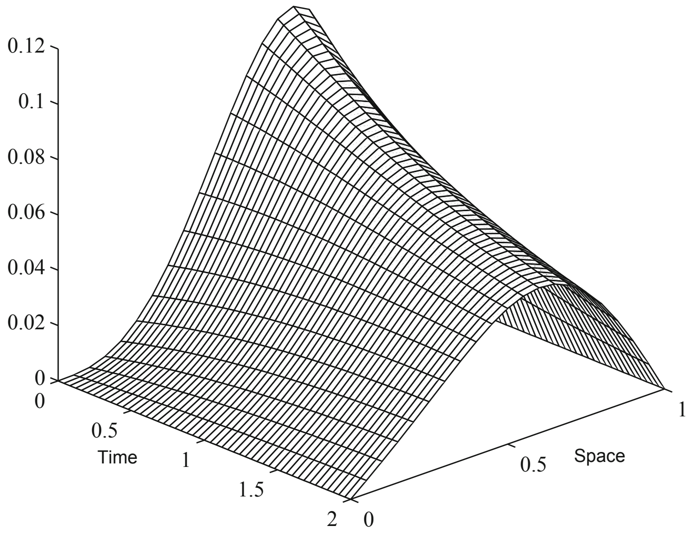

Figure 6.2 Numerical solution to the problem (6.2), (6.3), (6.4) with initial temperature $f(x) = x^3(1-x)$, diffusivity $D = 0.02$, on $0 \leq x \leq 1$, $0 \leq t \leq 2$

von Neumann Stability Analysis

There are three sources of errors in applying finite difference schemes to differential equations: (1) *truncation errors*, which measure the errors in replacing the differential equation by a difference equation; (2) errors propagated in a computation using the difference scheme itself; (3) *roundoff errors* accumulated from the computation because of the finite-digit arithmetic, or representation of real data.

We have seen that truncation errors are proportional to the step sizes taken to approximate derivative, e.g., $\mathrm{O}(h^2)$, etc. The origin of roundoff error is a topic we leave to other sources, for example, computer science. We focus here on (2), that is, how does a difference equation itself propagate incorrect information? Even if the difference approximation has a very high accuracy measured by the truncation error, we show that the difference scheme can still lead to horrendous results. The analysis, which is one of several, is called **von Neumann stability analysis**. It deals with the issue of stability, which was discussed earlier in the text; namely, if an initial error is small, does the error at later times remain small when propagated by the difference scheme?

Consider the problem

$$u_t - u_{xx} = 0, \;\; x \in \mathbb{R}, \; t > 0, \quad u(x,0) = \cos \alpha x, \;\; x \in \mathbb{R},$$

where $\alpha > 0$ can regarded as a wave number, or the number of oscillations per 2π. This problem can be solved exactly, and the solution is

$$u(x,t) = e^{-\alpha^2 t} \cos \alpha x.$$

Clearly u remains bounded as $t \to \infty$. Next, let us set up this problem numerically using the explicit scheme. For the infinite domain $t > 0$ take $x_j = \pm jh$, $j = 0, 1, 2, \ldots$, $t_n = nk$, $n = 0, 1, 2, \ldots$. The PDE is approximated by the difference equation (6.5),

$$U_j^{n+1} = U_j^n + r\left(U_{j-1}^n - 2U_j^n + U_{j+1}^n\right).$$

The initial condition gives

$$U_j^0 = \cos \alpha x_j, \;\; j = 0, \pm 1, \pm 2, \ldots.$$

Think of the initial condition as defining a distribution of errors at time $t = 0$. We look for a complex solution of the discrete problem in the form

$$U_j^n = M^n e^{i\alpha x_j},$$

where the amplitude factor M is to be determined. (Note: we can recover the real solution by taking the real part of the complex solution.) Substituting this into the difference equation gives

$$\begin{aligned} M &= r - 1 + 2r\cos\alpha h \\ &= 1 - 4r\sin^2\frac{\alpha h}{2}. \end{aligned}$$

Therefore, the real solution is

$$U_j^n = \left(1 - 4r\sin^2\frac{\alpha h}{2}\right)^n \cos\alpha x_j.$$

It is clear that U will be bounded whenever

$$\mid 1 - 4r\sin^2\frac{\alpha h}{2} \mid < 1 \quad \text{or} \quad r \le \frac{1}{2\sin^2\frac{\alpha h}{2}}.$$

If $r \le \frac{1}{2}$ this is guaranteed. If on the other hand $r > \frac{1}{2}$, then there is always a value of α for which $\mid M \mid > 1$. It follows that the explicit scheme with $r > \frac{1}{2}$ is not stable because bounded initial conditions are propagated by the scheme so that they become unbounded. For $r \le \frac{1}{2}$ we say the scheme is von Neumann stable.

Example 6.2

The following diagram shows the result of applying the explicit scheme (6.5) in the simple case $r = 1$, which is an unstable case. In this case, the difference equation becomes

$$U_j^{n+1} = -U_j^n + U_{j-1}^n + U_{j+1}^n.$$

At time $t_0 = 0$ we assume there is an initial error ϵ at $x_0 = 0$. Using the scheme, the values of U_j^n, $j = 0, \pm1, \pm2, \pm3, \ldots$ were computed (by hand) for subsequent time lines t_1, t_2, t_3, etc. One can observe that the error propagates, spreading and growing in time. It would make a numerical approximation completely invalid. □

t_3	ϵ	-3ϵ	6ϵ	-7ϵ	6ϵ	-3ϵ	$-\epsilon$
t_2	0	ϵ	-2ϵ	3ϵ	-2ϵ	ϵ	0
t_1	0	0	ϵ	$-\epsilon$	ϵ	0	0
t_0	0	0	0	ϵ	0	0	0
	x_{-3}	x_{-2}	x_{-1}	x_0	x_1	x_2	x_3

Example 6.3

(**Explicit Scheme for the Wave Equation**) This same type of explicit marching procedure that worked for the heat equation works for initial value problems associated with the wave equation. Consider the problem

$$u_{tt} = c^2 u_{xx}, \;\; 0 < x < 1, \; t > 0, \tag{6.6}$$

$$u(0,t) = u(1,t) = 0, \;\; t > 0, \tag{6.7}$$

$$u(x,0) = f(x), \quad u_t(x,0) = g(x), \;\; 0 \le x \le 1. \tag{6.8}$$

First we discretize the space-time domain as before and form a lattice $x_j = jh, \;\; t_n = nk, \quad j = 0, 1, \ldots, J, \quad n = 0, 1, \ldots, N$. We can approximate the PDE by

$$\frac{U_j^{n-1} - 2U_j^n + U_j^{n+1}}{k^2} = c^2 \frac{U_{j-1}^n - 2U_j^n + U_{j+1}^n}{h^2},$$

which leads to

$$U_j^{n+1} = 2(1 - s^2)U_j^n + s^2(U_{j-1}^n + U_{j+1}^n) - U_j^{n-1},$$

where

$$s = \frac{ck}{h}.$$

The computational atom for this explicit scheme is shown in Figure 6.3. To

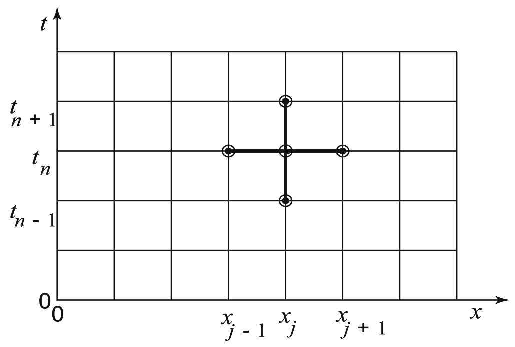

Figure 6.3 Computational atom for the wave equation. The values at the $(n + 1)$st time level depend on the two previous times, n and $n - 1$

compute the value at the $(n + 1)$st time step we now need values from the previous *two* time steps. Therefore, to start the marching scheme we require values from the first two ($t_0 = 0$ and $t_1 = k$) time rows. The $t = 0$ row is given by the initial condition $u(x, 0) = f(x)$; so we have

$$U_j^0 = f(x_j), \quad j = 0, 1, \ldots, J.$$

The $t = t_1$ time row can be computed using the initial velocity condition $u_t(x, 0) = g(x)$, which we approximate by a forward difference formula,

$$\frac{U_j^1 - U_j^0}{k} = g(x_j), \quad j = 1, 2, \ldots, J - 1.$$

Thus, the values U_j^1 at the second row are

$$U_j^1 = U_j^0 + kg(x_j).$$

Now we have the ingredients to start the scheme and march forward in time to fill up the entire grid with values, marching forward from time row to time row. Again there is a stability condition, namely, $s \le 1$, or

$$c \le \frac{h}{k}.$$

This is the **CFL** condition, or the Courant–Friedrichs–Lax condition. Physically, it states that the speed h/k at which lattice points are being calculated must exceed the speed c that waves are propagated in the system. Thus the calculated values at the lattice points at a given time level will contain all of the information about the wave at the previous time level. We ask the reader to write a code for the explicit scheme to solve the wave equation. □

Example 6.4

(**Other boundary conditions**) In the explicit marching scheme for the heat equation we considered fixed, Dirichlet boundary conditions. For example, this forces the values U_0^n along the grid points $(0, n)$ to be fixed along the left boundary. If, however, the left boundary has a Neuman condition, then we do not know U_0^n *a priori*, and the values must be calculated. For illustration, assume a boundary condition of the form

$$u_x(0, t) = b(t), \ \ t > 0.$$

Using the spatial grid $x_j = jh, \ j = 0, 1, 2, \ldots$, as before, we discretize this condition by

$$\frac{u(x_1, t_n) - u(0, t_n)}{h} = b(t_n) + \mathrm{O}(h),$$

for each n. In terms of the grid function U, this is

$$\frac{U_1^n - U_0^n}{h} = b(t_n), \ \text{or} \ U_0^n = U_1^n - hb(t_n), \quad n = 1, 2, 3, \ldots.$$

Therefore, given the values along the nth row, the calculation of the values along the $(n+1)$st row is

$$\text{for } j = 1 \text{ to } J-1;\ U_j^{n+1} = U_j^n + r(U_{j-1}^n - 2U_j^n + U_{j+1}^n);\ \text{end}$$
$$U_0^{n+1} = U_1^{n+1} - hb(t_n);$$

Similar conditions can be derived for radiation boundary conditions. The wave equation can be handled in a similar way. □

EXERCISES

1. Use Euler's method with step size $h = 0.1$ to numerically solve the initial value problem $y' = -2ty + y^2, \quad y(0) = 1$ on the interval $0 \leq t \leq 2$. Compare your approximations with the exact solution.

2. Use the notation in this section, derive the centered difference approximation to the first derivative,
$$u'(x) = \frac{u(x+h) - u(x-h)}{2h} + \mathrm{O}(h^2).$$

3. Develop an explicit scheme to solve the heat conduction problem
$$\begin{aligned} u_t &= u_{xx}, \quad 0 < x < 1, \quad t > 0, \\ u(0,t) &= 1, \quad u_x(1,t) = -(u(1,t) - g(t)), \quad t > 0, \\ u(x,0) &= 0, \quad 0 < x < 1. \end{aligned}$$
Hint: Approximate the radiation condition $u_x(1,t) = -(u(1,t) - g(t))$ by
$$\frac{u(x_J, t_n) - u(x_{J-1}, t_n)}{h} = -(u(x_J, t_n) - g(t_n)).$$
Pick $g(t) = 1$ and compute an approximate solution surface. Finally, describe the physical context of this problem.

4. Write a program in some programming language or in a computer algebra package to solve the hyperbolic problem (6.6)–(6.8) when $f(x) = 0$ and $g(x) = x(1-x)$ with $c = 0.25$. Take $h = 0.125$ and experiment with various time step sizes k and illustrate the validity of the CFL condition. Compare with the exact solution.

5. Consider the explicit method for the wave equation wave equation in $x \in \mathbb{R}$, $t > 0$ with CFL parameter $s = 2$. Take the initial condition $f(x)$ at the discrete grid points x_j to be $\ldots, 0, 0, 0, 1, 2, 1, 0, 0, 0, \ldots$ and assume $u_t(x, 0) = 0$ for all x. By hand, compute the values of U_j^n for the four time levels $n = 1, 2, 3, 4$. Next make the same calculation with $s = 1$. Discuss your observations.

6. Consider the Cauchy problem for the advection equation, $u_t + cu_x = 0$, where $c > 0$.

   a) Expand $u(x, t+k)$ in a Taylor series up to $\mathrm{O}(k^3)$ terms. Then use the advection equation to obtain

   $$u(x, t+k) = u(x,t) - cku_(x,t) + \frac{c^2k^2}{2}u_{xx}(x,t) + \mathrm{O}(k^3).$$

   b) Replace u_x and u_xx by centered difference approximations to obtain the explicit scheme

   $$U_j^{n+1} = (1-s^2)U_j^n + \frac{1}{2}(1+s)U_{j-1}^n - \frac{1}{2}(1-s)U_{j+1}^n, \quad s = \frac{ck}{h}.$$

   This is the **Lax–Wendroff method**. It is von Neumann stable for $0 < s \leq 1$ and it is widely used to solve both linear and nonlinear first-order hyperbolic systems.

   c) Use the Lax–Wendroff method to numerically solve $u_t + 3u_x = 0$ in the upper half-plane where $u(x,0) = 2x(2-x)$ if $0 \leq x \leq 2$ and $u(x,0) = 0$ otherwise. Take $h = 0.2$, $k = 0.05$. Compare your answer to the exact solution.

6.3 Laplace's Equation

Next we solve a Dirichlet problem for Laplace's equation in the unit square with prescribed values on the boundary of the square. This Dirichlet problem is a pure boundary value problem, and we should not expect an explicit marching-type scheme to work; there is no guarantee that when we march from one boundary in the problem to another we will reach the already specified values at that boundary. This means that we must solve for the approximate values at all of the interior lattice points simultaneously. To illustrate the procedure we consider the Dirichlet problem

$$u_{xx} + u_{yy} = 0, \quad 0 < x < 1, 0 < y < 1, \tag{6.9}$$

$$u(0,y) = u(1,y) = 0, \quad 0 < y < 1, \tag{6.10}$$

$$u(x,0) = f(x), \quad u(x,1) = 0, \quad 0 \leq x \leq 1. \tag{6.11}$$

We discretize the unit square by defining lattice points

$$x_i = ih, \quad y_j = jk, \quad i = 0, 1, \ldots, I; \quad j = 0, 1, \ldots, J,$$

where $h = 1/I$ and $k = 1/J$ are the fixed step sizes in the x and y directions, respectively. We let $U_{i,j}$, with two lower subscripts, denote the approximate value of $u(x_i, y_j)$. Then Laplace's equation can be approximated at the lattice point (x_i, y_j) by the difference equation

$$\frac{U_{i-1,j} - 2U_{i,j} + U_{i+1,j}}{h^2} + \frac{U_{i,j-1} - 2U_{i,j} + U_{i,j+1}}{k^2} = 0.$$

The reader may wish to review Section 1.8, where a similar approximation is made when $h = k$. The difference equation can be rewritten as

$$U_{i,j} = \frac{k^2}{2k^2 + 2h^2}(U_{i-1,j} + U_{i+1,j}) + \frac{h^2}{2k^2 + 2h^2}(U_{i,j-1} + U_{i,j+1}). \qquad (6.12)$$

The computational atom is shown in Figure 6.4. The difference equation relates the node at the center of the atom to four adjacent nodes. As we observed in Section 1.8, if $h = k$, then this difference equation states that the value at the center node is approximately the average of the values at the four adjacent nodes; if $h \neq k$, the approximate value at the center is a weighted average of the four surrounding values.

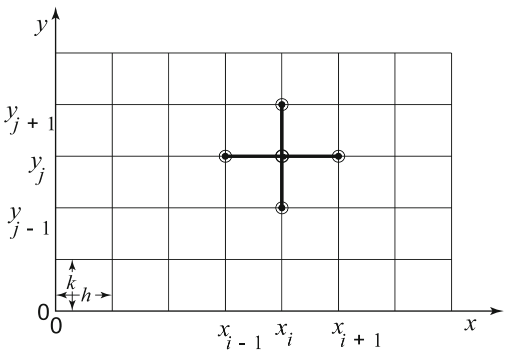

Figure 6.4 Computational atom for Laplace's equation

The strategy is this. The values of $U_{i,j}$ are known at the nodes on the boundary of the square. We want to find the values at the $(I-1) \times (J-1)$ interior lattice points. Therefore, we apply the difference equation (6.12) to each interior lattice point, i.e., for $i = 1, \ldots, I-1$; $j = 1, \ldots, J-1$. The result is a linear-algebraic system consisting of $(I-1) \times (J-1)$ equations in the $(I-1) \times (J-1)$ unknowns $U_{i,j}$. Therefore, we have reduced the Dirichlet problem to the linear algebra problem of solving for grid function $U_{i,j}$ at the interior nodes.

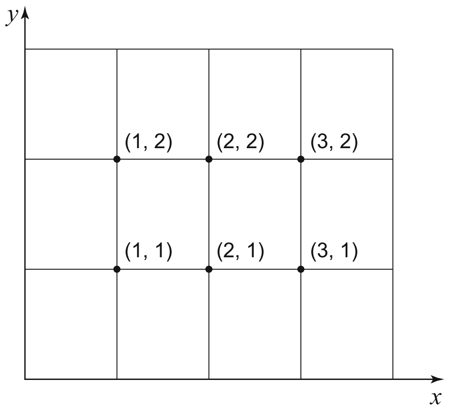

Figure 6.5 Discrete lattice

Example 6.5

In (6.11) take $f(x) = x^3(1-x)$. To illustrate the procedure take $I = 4$ and $J = 3$; thus $j = 1/4$ and $k = 1/3$. See Figure 6.5. Then

$$\frac{h^2}{2k^2+2h^2} = 0.18, \quad \frac{k^2}{2k^2+2h^2} = 0.32.$$

There are six interior lattice points where the solution is to be found. By the boundary conditions, the values along the top and two sides are zero, and the values along the lower boundary are $U_{0,0} = 0, U_{1,0} = 0.0117, U_{2,0} = 0.0625, U_{3,0} = 0.1054, U_{4,0} = 0$. Now we apply the difference equation (6.12) successively at the six points $(i,j) = (1,1), (1,2), (2,1), (2,2), (3,1), (3,2)$ to obtain the six coupled algebraic equations

$$\begin{aligned}
U_{1,1} &= 0.32(0 + U_{2,1}) + 0.18(0.0117 + U_{1,2}),\\
U_{1,2} &= 0.32(0 + U_{2,2}) + 0.18(U_{1,1} + 0),\\
U_{2,1} &= 0.32(U_{1,1} + U_{3,1}) + 0.18(0.0625 + U_{2,2}),\\
U_{2,2} &= 0.32(U_{1,2} + U_{3,2}) + 0.18(U_{2,1} + 0),\\
U_{3,1} &= 0.32(U_{2,1} + 0) + 0.18(0.1054 + U_{3,2}),\\
U_{3,2} &= 0.32(U_{2,2} + 0) + 0.18(U_{3,1} + 0).
\end{aligned}$$

We can solve these six linear equations to obtain the approximate solution at the six interior nodal points:

$$\begin{aligned}
&U_{1,1} = 0.007,\ U_{1,2} = 0.003,\ U_{2,1} = 0.015,\\
&U_{2,2} = 0.006,\ U_{3,1} = 0.024,\ U_{3,2} = 0.006. \quad \square
\end{aligned}$$

In the previous example we solved the linear system easily on a calculator. But if the lattice has a large number of nodes, which is the usual case, then direct solution is inefficient, even though the coefficient matrix for the system is tridiagonal (nonzero only on the sub-, super- and main diagonals). Another way to solve a linear system is by an iterative process known as the **Gauss–Seidel method**. In this method we fix the values of the array $U_{i,j}$ at the known boundary points and then initialize the array at the interior lattice points by setting them to zero, or setting them to values that might approximate the solution. Then we systematically cycle through all the *interior* lattice points, replacing the old estimates with new ones calculated by the difference equation (6.12). We can cycle along rows or columns, but it is common to cycle row by row, from bottom to top. After a number of cycles, the estimates will converge to the solution of the linear system. One can terminate the iteration process when the change from one complete iteration to the next is small, measured in some manner. This process is extremely simple to program, and the cycling is accomplished in a few lines of code. Actually, there are several ways the iteration procedure can be accelerated, and we refer the reader to the references and the key words *successive overrelaxation* (SOR).

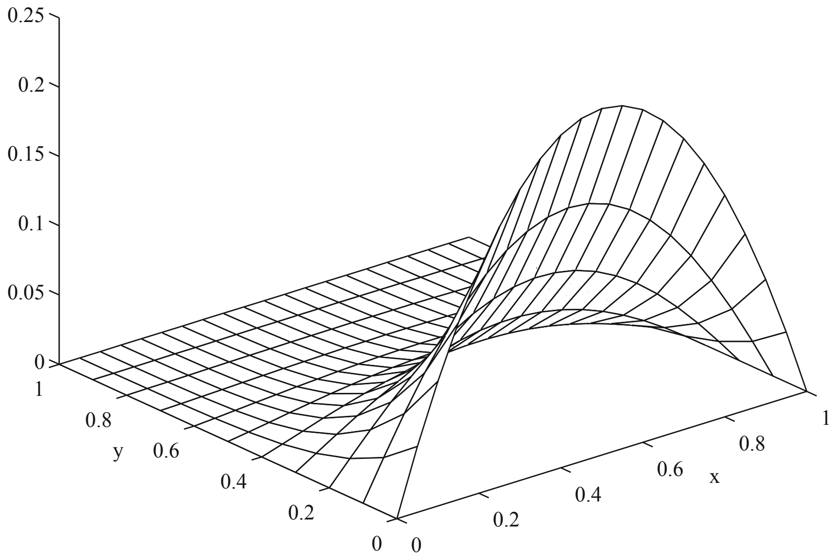

Figure 6.6 Solution to the BVP (6.9), (6.10), (6.11) for Laplace's equation when $f(x) = x(1-x)$

Here is a basic outline of the Gauss–Seidel method. After putting in the prescribed values on the boundary and initializing the the array $U_{i,j}$ (to zero, say) at the interior grid points, we perform the following row-by-row iteration using three nested loops:

```
for n from 1 to number of iterations;
for i from 1 to I-1;
for j from 1 to J-1;
U[i,j] =k∧2/(2*k∧2+2*h∧2)*(U[i-1,j]+U[i+1,j])
+h∧2/(2*k∧2+2*h∧2)*(U[i,j-1]+U[i,j+1]);
end; end; end;
```

The MATLAB code *Laplace* is at the end of the chapter. We take $I = 20$ and $J = 10$, giving $19 \times 9 = 171$ interior lattice points. Figure 6.6 shows the solution surface after 20 iterations through the grid.

Neumann and radiation boundary conditions can be handled similarly as in the heat and wave equations. See the Exercises.

EXERCISES

1. In Example 6.5 write the linear system for the six unknowns $U_{i,j}$ in standard matrix form, $A\mathbf{x} = \mathbf{b}$, where $\mathbf{x}$ is the column vector of unknowns, and A is the coefficient matrix. Note that A is a tridiagonal matrix.

2. Write a finite difference code to solve the following steady-state problems:

   a)

$$\begin{aligned} u_{xx} + u_{yy} &= 0, \quad 0 < x < 2,\ 0 < y < 1, \\ u(0,y) &= 2\sin \pi y, \quad u(2,y) = 0, \ \ 0 < y < 1, \\ u(x,0) &= x(2-x), \ \ u(x,1) = 0, \quad 0 < x < 2. \end{aligned}$$

   b)

$$\begin{aligned} u_{xx} + u_{yy} &= 0, \quad 0 < x < 2,\ 0 < y < 1, \\ u_x(0,y) &= 0, \quad u(2,y) = 0, \ \ 0 < y < 1, \\ u(x,0) &= x(2-x)^2, \ \ u(x,1) = 0, \quad 0 < x < 2. \end{aligned}$$

   To handle the insulated boundary condition at $x = 0$ in (b), use a forward difference approximation; note that you must calculate the solution along that boundary.

3. Consider Laplace's equation on the triangle with vertices $(0,0)$, $(1,0)$, $(0,1)$. Use a lattice with spacing $h = k = \frac{1}{6}$. If the oblique edge is insulated, find a difference approximation at a point (x_j, y_k) on that boundary. If the left

edge is held at 1 and the lower edge is held at 0, find the matrix equation for the unknown grid points. (Recall, the oblique boundary points must be calculated.)

6.4 Implicit Scheme for the Heat Equation

In an *explicit* difference scheme we compute a new time values of the grid function U_j^{n+1} explicitly only in terms of the previous time values U_j^n. As we found in the explicit method for the heat equation, we were forced to take very small time to insure stability. There are other difference schemes that are *implicit* in nature. In these we are not able to solve for a new time value explicitly, but rather we obtain a *system of algebraic equations* at the new time for all the values simultaneously. These methods typically have a strong advantage: they are stable without putting a constraint on the size of time step. A disadvantage is that coding such schemes is a little more difficult.

In this section we set up such a scheme for the heat equation. Consider the initial BVP problem for the heat, or diffusion, equation presented in Section 6.2:

$$u_t = Du_{xx}, \quad 0 < x < 1, \quad 0 < t < T, \tag{6.13}$$

$$u(0,t) = u(1,t) = 0, \quad t > 0, \tag{6.14}$$

$$u(x,0) = f(x), \quad 0 < x < 1. \tag{6.15}$$

The first step to discretize, as before, the region of space-time where we want to obtain a solution. In this case the region is $0 \le x \le 1$, $0 \le t \le T$. Therefore we define a **lattice** of points in this space-time region by

$$x_j = jh, \quad t_n = nk, \quad j = 0, 1, \ldots, J; \quad n = 0, 1, \ldots, N,$$

where the fixed numbers h and k are the spatial and temporal step sizes given by $h = 1/J$ and $k = 1/N$. For an implicit scheme we approximate the time derivative as before by a forward difference

$$u_t(x_j, t_n) = \frac{u(x_j, t_{n+1} - u(x_j, t_n)}{k} + \mathrm{O}(k).$$

But for the second spatial derivative u_{xx} at (x_j, t_n), rather than a centered difference at time t_n, we take a *weighted average* of two centered difference approximations, one at t_n and one at t_{n+1}. Therefore,

$$\begin{aligned} u_{xx}(x_j, t_n) = (1-\theta)&\frac{u(x_{j-1}, t_{n+1}) - 2u(x_j, t_{n+1}) + u(x_{j+1}, t_{n+1})}{h^2} \\ &+ \theta\frac{u(x_{j-1}, t_n) - 2u(x_j, t_n) + u(x_{j+1}, t_n)}{h^2} + \mathrm{O}(h^2), \end{aligned}$$

where $0 \leq \theta \leq 1$. Therefore, the computational molecule is shown in Figure 6.7. Substituting these approximations into the PDE (6.13) using U_j^n as the

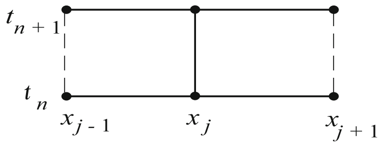

Figure 6.7 A six-node computational molecule for the implicit difference scheme

approximation for $u(x_j, t_n)$, we obtain the finite difference equation

$$\frac{U_j^{n+1} - U_j^n}{k} = D\theta \frac{U_{j-1}^{n+1} - 2U_j^{n+1} + U_{j+1}^{n+1}}{h^2} + D(1-\theta)\frac{U_{j-1}^n - 2U_j^n + U_{j+1}^n}{h^2}.$$

We have dropped the truncation error terms. We rearrange the terms in this equation to obtain

$$\begin{aligned} -\theta r U_{j-1}^{n+1} + &(1+2r\theta)U_j^{n+1} - \theta r U_{j+1}^{n+1} \\ &= (1-\theta) r U_{j-1}^n + [1 - 2(1-\theta)r]U_j^n + (1-\theta) r U_{j+1}^n, \end{aligned} \tag{6.16}$$

where

$$r = \frac{kD}{h^2}.$$

Notice that when $\theta = 0$ this scheme reduces to the classical explicit scheme; when $\theta = \frac{1}{2}$, the scheme is called the **Crank–Nicolson scheme**. We have written the three unknown values U_{j-1}^{n+1}, U_j^{n+1}, U_{j+1}^{n+1} at the $(n+1)$st time level in terms of the three known values at the nth time level. For each fixed n, we have to solve this equation for $j = 1, 2, \ldots, J-1$. Therefore equation (6.16) represents a system of $J-1$ equations for the $J-1$ values $U_1^{n+1}, U_2^{n+1}, \ldots, U_{J-1}^{n+1}$. Each equation has three unknowns. In matrix form (6.16) can be written as a tridiagonal system

$$\begin{pmatrix} 1+2r\theta & -r\theta & & & & \\ -r\theta & 1+2r\theta & -r\theta & & & \\ & -r\theta & 1+2r\theta & -r\theta & & \\ & & \cdots & \cdots & & \\ & & & \cdots & -r\theta \\ & & & -r\theta & 1+2r\theta \end{pmatrix} \begin{pmatrix} U_1^{n+1} \\ U_2^{n+1} \\ U_3^{n+1} \\ \cdots \\ U_{J-2}^{n+1} \\ U_{J-1}^{n+1} \end{pmatrix} = \begin{pmatrix} F_1^n \\ F_2^n \\ F_3^n \\ \cdots \\ F_{J-2}^n \\ F_{J-1}^n \end{pmatrix}$$

where the right side involves the terms on the right of (6.16) and the boundary conditions. This system must be solved at each time step, but there are no constraints on the time step as in explicit methods. This scheme is stable unconditionally. As it turns out, there are simple direct algorithms to solve tridiagonal systems (see the references, MATLAB manuals, or other computer algebra systems). In addition, there are iterative methods similar to the Gauss–Seidel method that can be applied at each time step to solve the system.

EXERCISES

1. Use the Crank–Nicolson scheme with $\theta = 0.5$, $D = 1$, $h = 0.2$, $k = 0.08$ and write out the scheme (6.16) for $n = 0$ when the initial condition (6.15) is given by $f(x) = x(1 - x)$. Write the tridiagonal system explicitly (there are four unknowns) and solve it to find the solution at the time line $n = 1$.

2. Use the Crank–Nicolson scheme to numerically solve the problem

$$\begin{aligned} u_t &= u_{xx} + u, \quad 0 < x < 1,\, t > 0, \\ u(0,t) &= u(1,t) = 0, \quad t > 0, \\ u(x,0) &= 1, \quad 0 \le x \le 1. \end{aligned}$$

   Take $h = k = 0.2$.

3. Consider the BVP for the advection equation

$$u_t + cu_x = 0, \quad x \in \mathbb{R}, \quad (c > 0),$$

   with $u(x,0) = f(x),\ x \ge 0,\ u(0,t) = g(t),\ t > 0$.

   a) Using the three-point molecule (x_j, t_n), (x_{j-1}, t_{n+1}), (x_j, t_{n+1}), derive the implicit difference equation approximation

$$(1 + s)U_j^{n+1} - sU_{j-1}^{n+1} = U_j^n, \quad s = \frac{ch}{k}.$$

   b) Show that the scheme is von Neumann stable.

   c) Develop an algorithm to compute U_j^n at lattice points in the first quadrant. Explain carefully your procedure.

MATLAB Program Listings

Programming note: MATLAB, like other computer algebra systems, does not recognize an index with value 0. Many of the formulas in the text require, for example, ranges such as $j = 0, 1, 2, \ldots, J$. For MATLAB expressions we must shift the indices and use $j = 1, 2, \ldots, J + 1$. This is an unfortunate nuisance in writing programs.

```
function EulerMethod
f=@(t,Y) -0.1*(Y-(15+12*cos(3.14*t/12)))
t0=0; Y0=68; T=72; N=1000; h=(T-t0)/N; Y=zeros(N+1); Y(1)=Y0;
for n=1:N
 t(n) = a+(n-1)*h;
Y(n+1) = Y(n)+h*f(t(n),Y(n));
end
t=t0:h:T; plot(t,Y), xlabel('time in hours'),
  ylabel('temperature Y in deg F'),
ylim([0 70]), title('Newton Law of Cooling')
```

```
function ExplicitHeat
xmax=1; J=20; h=xmax/J; tmax=2; N=50; k=tmax/N;
D=0.02; r=D*k/(h*h)
U=zeros(J+1,N+1);
f=@(x) x.∧3. x*(1-x);
for j=1:J+1; U(j,1)=f((j-1)*h); end
for n=1:N+1; U(1,n)=0; U(J+1,n)=0; end
for n=1:N
 for j=2:J
 U(j,n+1)=U(j,n)+r*(U(j-1,n)-2*U(j,n)+U(j+1,n));
 end
end
[T,X]=meshgrid(0:h:xmax,0:k:tmax);
mesh(X',T',U,'EdgeColor','black')
xlabel('t','FontSize',14),ylabel('x','FontSize',14)
title('Explicit Scheme for Heat Equation')
```

```
function Laplace
% Gauss--Seidel method
xmax=1; ymax=1;
I=20; J=10; h=1/I; k=1/J; Numit=20;
r=0.5*k∧2/(k∧2+h∧2); s=0.5*h∧2/(k∧2+h∧2);
U=zeros(I+1,J+1);
f=@(x) x. * (1-x);
for i=1:I+1; U(i,1)=f((i-1)*h); end
for i=1:I+1; U(i,J+1)=0; end
for j=2:J; U(1,j)=0; end
for j=2:J; U(I+1,j)=0; end
for n=1:Numit
 for i=2:I
 for j=2:J
 U(i,j)=r*(U(i-1,j)+U(i+1,j))+s*(U(i,j-1)+U(i,j+1));
 end
 end
end
[X,Y]=meshgrid(0:h:xmax,0:k:ymax);
mesh(X',Y',U,'EdgeColor','black')
xlabel('x','FontSize',14),ylabel('y','FontSize',14)
```