738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
827

# Chapter 19. Partial Differential Equations

## 19.0 Introduction

The numerical treatment of partial differential equations is, by itself, a vast subject. Partial differential equations are at the heart of many, if not most, computer analyses or simulations of continuous physical systems, such as fluids, electromagnetic fields, the human body, and so on. The intent of this chapter is to give the briefest possible useful introduction. Ideally, there would be an entire second volume of *Numerical Recipes* dealing with partial differential equations alone. (The references [1-4] provide, of course, available alternatives.)

In most mathematics books, partial differential equations (PDEs) are classified into the three categories, *hyperbolic*, *parabolic*, and *elliptic*, on the basis of their *characteristics*, or curves of information propagation. The prototypical example of a hyperbolic equation is the one-dimensional *wave* equation

$$\frac{\partial^2 u}{\partial t^2} = v^2 \frac{\partial^2 u}{\partial x^2} \tag{19.0.1}$$

where $v =$ constant is the velocity of wave propagation. The prototypical parabolic equation is the *diffusion* equation

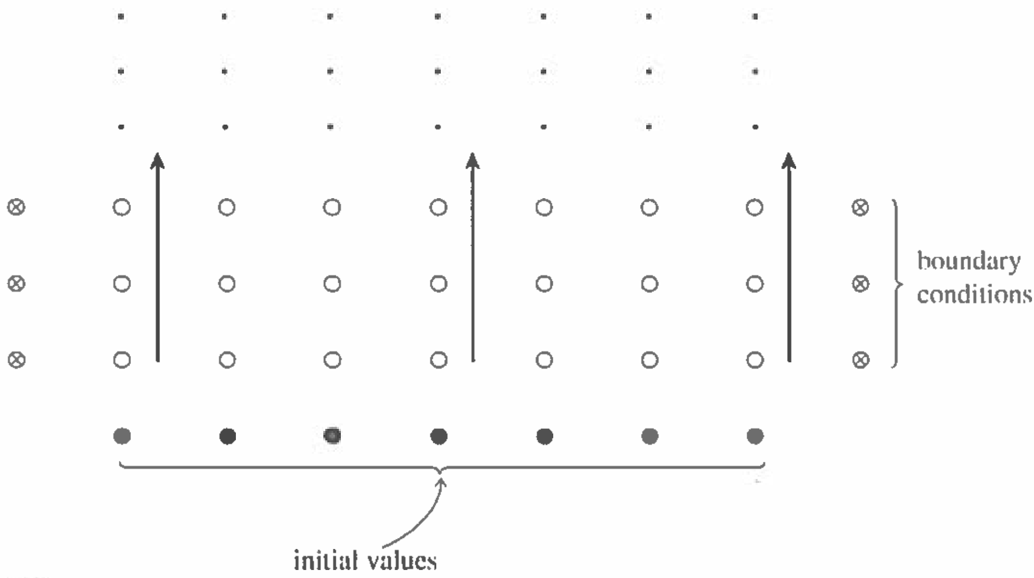
$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(D\frac{\partial u}{\partial x}\right) \tag{19.0.2}$$

where $D$ is the diffusion coefficient. The prototypical elliptic equation is the *Poisson* equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \rho(x, y) \tag{19.0.3}$$

where the source term $\rho$ is given. If the source term is equal to zero, the equation is *Laplace's equation*.

From a computational point of view, the classification into these three canonical types is not very meaningful — or at least not as important as some other essential distinctions. Equations (19.0.1) and (19.0.2) both define *initial value* or *Cauchy* problems: If information on $u$ (perhaps including time derivative information) is



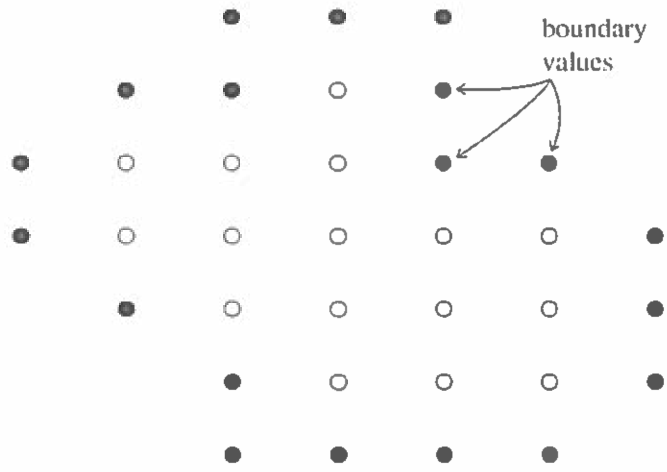




Figure 19.0.1. Initial value problem (a) and boundary value problem (b) are contrasted. In (a) initial values are given on one "time slice," and it is desired to advance the solution in time, computing successive rows of open dots in the direction shown by the arrows. Boundary conditions at the left and right edges of each row (⊗) must also be supplied, but only one row at a time. Only one, or a few, previous rows need be maintained in memory. In (b), boundary values are specified around the edge of a grid, and an iterative process is employed to find the values of all the internal points (open circles). All grid points must be maintained in memory.

given at some initial time $t_0$ for all $x$, then the equations describe how $u(x, t)$ propagates itself forward in time. In other words, equations (19.0.1) and (19.0.2) describe time evolution. The goal of a numerical code should be to track that time evolution with some desired accuracy.

By contrast, equation (19.0.3) directs us to find a single "static" function $u(x, y)$ which satisfies the equation within some $(x, y)$ region of interest, and which — one must also specify — has some desired behavior on the boundary of the region of interest. These problems are called *boundary value problems*. In general it is not

possible stably to just "integrate in from the boundary" in the same sense that an initial value problem can be "integrated forward in time." Therefore, the goal of a numerical code is somehow to converge on the correct solution everywhere at once.

This, then, is the most important classification from a computational point of view: Is the problem at hand an *initial value* (time evolution) problem? or is it a *boundary value* (static solution) problem? Figure 19.0.1 emphasizes the distinction. Notice that while the italicized terminology is standard, the terminology in parentheses is a much better description of the dichotomy from a computational perspective. The subclassification of initial value problems into parabolic and hyperbolic is much less important because (i) many actual problems are of a mixed type, and (ii) as we will see, most hyperbolic problems get parabolic pieces mixed into them by the time one is discussing practical computational schemes.

## Initial Value Problems

An initial value problem is defined by answers to the following questions:

- What are the dependent variables to be propagated forward in time?
- What is the evolution equation for each variable? Usually the evolution equations will all be coupled, with more than one dependent variable appearing on the right-hand side of each equation.
- What is the highest time derivative that occurs in each variable's evolution equation? If possible, this time derivative should be put alone on the equation's left-hand side. Not only the value of a variable, but also the value of all its time derivatives — up to the highest one — must be specified to define the evolution.
- What special equations (boundary conditions) govern the evolution in time of points on the boundary of the spatial region of interest? Examples: *Dirichlet conditions* specify the values of the boundary points as a function of time; *Neumann conditions* specify the values of the normal gradients on the boundary; *outgoing-wave boundary conditions* are just what they say.

Sections 19.1–19.3 of this chapter deal with initial value problems of several different forms. We make no pretence of completeness, but rather hope to convey a certain amount of generalizable information through a few carefully chosen model examples. These examples will illustrate an important point: One's principal *computational* concern must be the *stability* of the algorithm. Many reasonable-looking algorithms for initial value problems just don't work — they are numerically unstable.

## Boundary Value Problems

The questions that define a boundary value problem are:

- What are the variables?
- What equations are satisfied in the interior of the region of interest?
- What equations are satisfied by points on the boundary of the region of interest? (Here Dirichlet and Neumann conditions are possible choices for elliptic second-order equations, but more complicated boundary conditions can also be encountered.)

740
741
742
743
744
745
746
747
748
749
750
751
752
827
828
829

In contrast to initial value problems, stability is relatively easy to achieve for boundary value problems. Thus, the *efficiency* of the algorithms, both in computational load and storage requirements, becomes the principal concern.

Because all the conditions on a boundary value problem must be satisfied "simultaneously," these problems usually boil down, at least conceptually, to the solution of large numbers of simultaneous algebraic equations. When such equations are nonlinear, they are usually solved by linearization and iteration; so without much loss of generality we can view the problem as being the solution of special, large linear sets of equations.

As an example, one which we will refer to in §§19.4–19.6 as our "model problem," let us consider the solution of equation (19.0.3) by the *finite-difference method*. We represent the function $u(x, y)$ by its values at the discrete set of points

$$\begin{aligned} x_j &= x_0 + j\Delta, \qquad j = 0, 1, \ldots, J \\ y_l &= y_0 + l\Delta, \qquad l = 0, 1, \ldots, L \end{aligned} \tag{19.0.4}$$

where $\Delta$ is the *grid spacing*. From now on, we will write $u_{j,l}$ for $u(x_j, y_l)$, and $\rho_{j,l}$ for $\rho(x_j, y_l)$. For (19.0.3) we substitute a finite-difference representation (see Figure 19.0.2),

$$\frac{u_{j+1,l} - 2u_{j,l} + u_{j-1,l}}{\Delta^2} + \frac{u_{j,l+1} - 2u_{j,l} + u_{j,l-1}}{\Delta^2} = \rho_{j,l} \tag{19.0.5}$$

or equivalently

$$u_{j+1,l} + u_{j-1,l} + u_{j,l+1} + u_{j,l-1} - 4u_{j,l} = \Delta^2 \rho_{j,l} \tag{19.0.6}$$

To write this system of linear equations in matrix form we need to make a vector out of $u$. Let us number the two dimensions of grid points in a single one-dimensional sequence by defining

$$i \equiv j(L+1) + l \qquad \text{for} \qquad j = 0, 1, \ldots, J, \qquad l = 0, 1, \ldots, L \tag{19.0.7}$$

In other words, $i$ increases most rapidly along the columns representing $y$ values. Equation (19.0.6) now becomes

$$u_{i+L+1} + u_{i-(L+1)} + u_{i+1} + u_{i-1} - 4u_i = \Delta^2 \rho_i \tag{19.0.8}$$

This equation holds only at the interior points $j = 1, 2, \ldots, J-1$; $l = 1, 2, \ldots, L-1$.

The points where

$$\begin{aligned} j &= 0 \qquad [\text{i.e., } i = 0, \ldots, L] \\ j &= J \qquad [\text{i.e., } i = J(L+1), \ldots, J(L+1) + L] \\ l &= 0 \qquad [\text{i.e., } i = 0, L+1, \ldots, J(L+1)] \\ l &= L \qquad [\text{i.e., } i = L, L+1+L, \ldots, J(L+1)+L] \end{aligned} \tag{19.0.9}$$

742
743
744
745
746
747
748
749
750
751
752
827
828
829
830
831

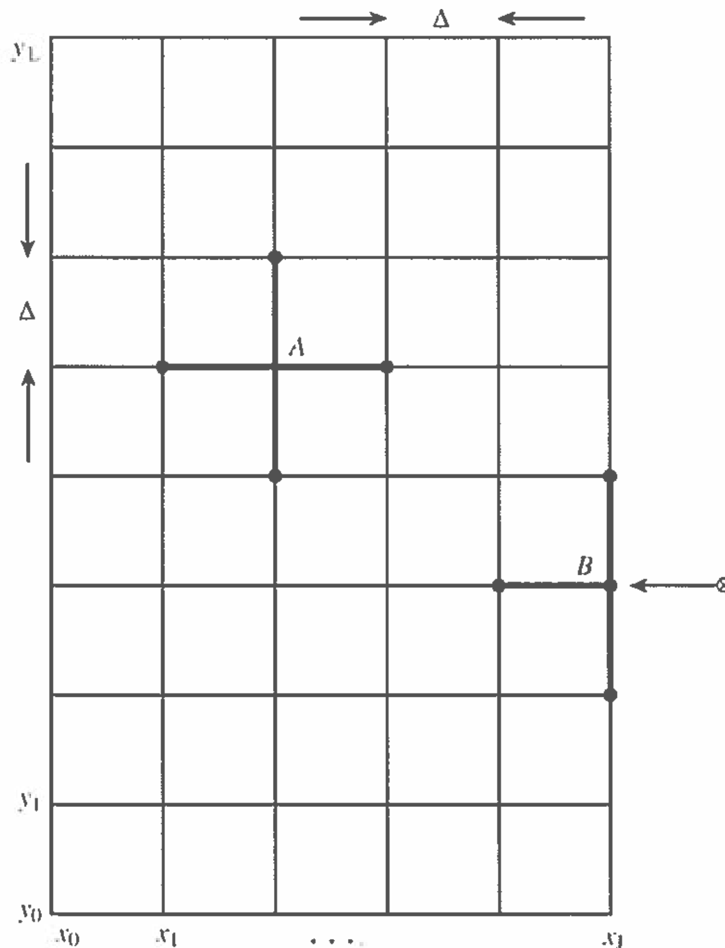



Figure 19.0.2. Finite-difference representation of a second-order elliptic equation on a two-dimensional grid. The second derivatives at the point $A$ are evaluated using the points to which $A$ is shown connected. The second derivatives at point $B$ are evaluated using the connected points and also using "right-hand side" boundary information, shown schematically as ⊗.

are boundary points where either $u$ or its derivative has been specified. If we pull all this "known" information over to the right-hand side of equation (19.0.8), then the equation takes the form

$$\mathbf{A} \cdot \mathbf{u} = \mathbf{b} \tag{19.0.10}$$

where $\mathbf{A}$ has the form shown in Figure 19.0.3. The matrix $\mathbf{A}$ is called "tridiagonal with fringes." A general linear second-order elliptic equation

$$\begin{aligned} a(x,y)\frac{\partial^2 u}{\partial x^2} + b(x,y)\frac{\partial u}{\partial x} + c(x,y)\frac{\partial^2 u}{\partial y^2} + d(x,y)\frac{\partial u}{\partial y} \\ + e(x,y)\frac{\partial^2 u}{\partial x \partial y} + f(x,y)u = g(x,y) \end{aligned} \tag{19.0.11}$$

will lead to a matrix of similar structure except that the nonzero entries will not be constants.

As a rough classification, there are three different approaches to the solution of equation (19.0.10), not all applicable in all cases: relaxation methods, "rapid" methods (e.g., Fourier methods), and direct matrix methods.

743
744
745
746
747
748
749
750
751
752
827
828
829
830
831
832

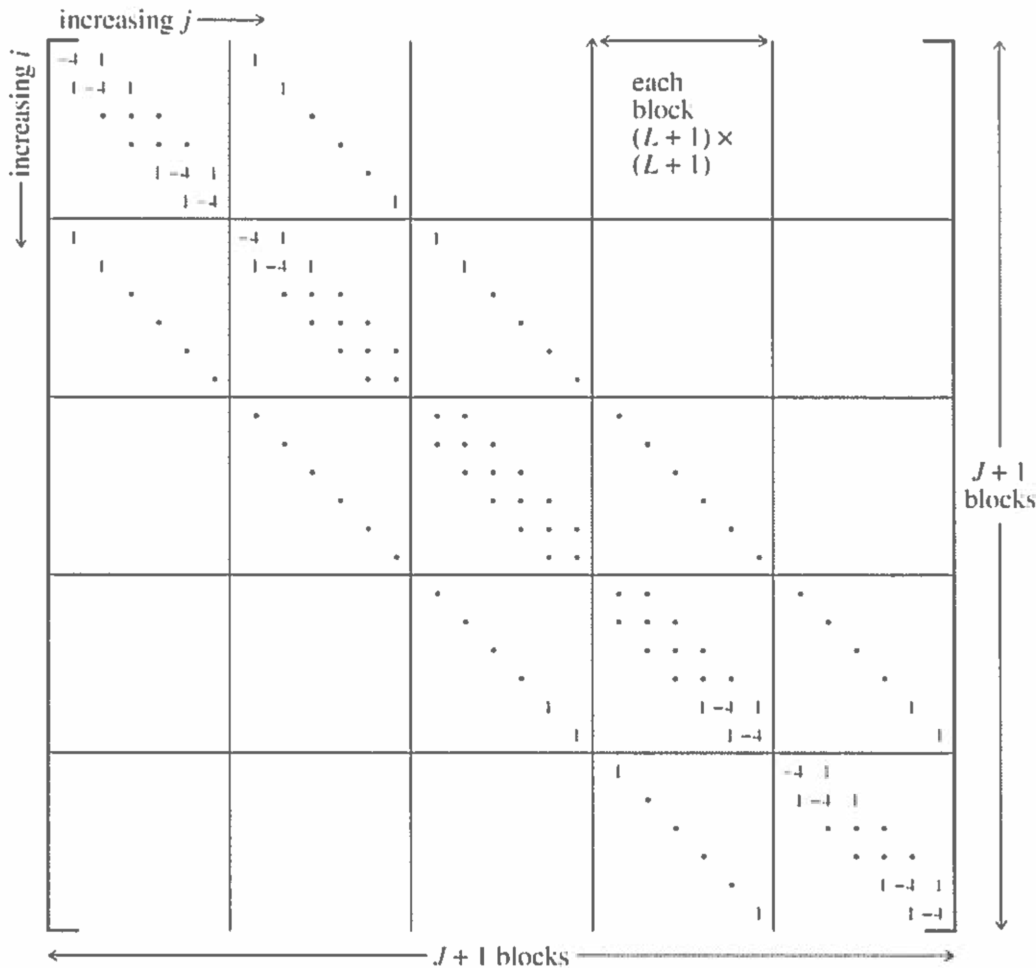



Figure 19.0.3. Matrix structure derived from a second-order elliptic equation (here equation 19.0.6). All elements not shown are zero. The matrix has diagonal blocks that are themselves tridiagonal, and sub- and super-diagonal blocks that are diagonal. This form is called "tridiagonal with fringes." A matrix this sparse would never be stored in its full form as shown here.

Relaxation methods make immediate use of the structure of the sparse matrix $\mathbf{A}$. The matrix is split into two parts

$$\mathbf{A} = \mathbf{E} - \mathbf{F} \tag{19.0.12}$$

where $\mathbf{E}$ is easily invertible and $\mathbf{F}$ is the remainder. Then (19.0.10) becomes

$$\mathbf{E} \cdot \mathbf{u} = \mathbf{F} \cdot \mathbf{u} + \mathbf{b} \tag{19.0.13}$$

The relaxation method involves choosing an initial guess $\mathbf{u}^{(0)}$ and then solving successively for iterates $\mathbf{u}^{(r)}$ from

$$\mathbf{E} \cdot \mathbf{u}^{(r)} = \mathbf{F} \cdot \mathbf{u}^{(r-1)} + \mathbf{b} \tag{19.0.14}$$

Since $\mathbf{E}$ is chosen to be easily invertible, each iteration is fast. We will discuss relaxation methods in some detail in §19.5 and §19.6.

So-called rapid methods [5] apply for only a rather special class of equations: those with constant coefficients, or, more generally, those that are separable in the chosen coordinates. In addition, the boundaries must coincide with coordinate lines. This special class of equations is met quite often in practice. We defer detailed discussion to §19.4. Note, however, that the multigrid relaxation methods discussed in §19.6 can be faster than "rapid" methods.

Matrix methods attempt to solve the equation

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{19.0.15}$$

directly. The degree to which this is practical depends very strongly on the exact structure of the matrix $\mathbf{A}$ for the problem at hand, so our discussion can go no farther than a few remarks and references at this point.

Sparseness of the matrix *must* be the guiding force. Otherwise the matrix problem is prohibitively large. For example, the simplest problem on a $100 \times 100$ spatial grid would involve 10000 unknown $u_{j,l}$'s, implying a $10000 \times 10000$ matrix $\mathbf{A}$, containing $10^8$ elements!

As we discussed at the end of §2.7, if $\mathbf{A}$ is symmetric and positive definite (as it usually is in elliptic problems), the conjugate-gradient algorithm can be used. In practice, rounding error often spoils the effectiveness of the conjugate gradient algorithm for solving finite-difference equations. However, it is useful when incorporated in methods that first rewrite the equations so that $\mathbf{A}$ is transformed to a matrix $\mathbf{A}'$ that is close to the identity matrix. The quadratic surface defined by the equations then has almost spherical contours, and the conjugate gradient algorithm works very well. In §2.7, in the routine `linbcg`, an analogous *preconditioner* was exploited for non-positive definite problems with the more general biconjugate gradient method. For the positive definite case that arises in PDEs, an example of a successful implementation is the *incomplete Cholesky conjugate gradient method (ICCG)* (see [6-8]).

Another method that relies on a transformation approach is the *strongly implicit procedure* of Stone [9]. A program called SIPSOL that implements this routine has been published [10].

A third class of matrix methods is the Analyze-Factorize-Operate approach as described in §2.7.

Generally speaking, when you have the storage available to implement these methods — not nearly as much as the $10^8$ above, but usually much more than is required by relaxation methods — then you should consider doing so. Only multigrid relaxation methods (§19.6) are competitive with the best matrix methods. For grids larger than, say, $300 \times 300$, however, it is generally found that only relaxation methods, or "rapid" methods when they are applicable, are possible.

## There Is More to Life than Finite Differencing

Besides finite differencing, there are other methods for solving PDEs. Most important are finite element, Monte Carlo, spectral, and variational methods. Unfortunately, we shall barely be able to do justice to finite differencing in this chapter, and so shall not be able to discuss these other methods in this book. Finite element methods [11-12] are often preferred by practitioners in solid mechanics and structural

engineering: these methods allow considerable freedom in putting computational elements where you want them, important when dealing with highly irregular geometries. Spectral methods [13-15] are preferred for very regular geometries and smooth functions; they converge more rapidly than finite-difference methods (cf. §19.4), but they do not work well for problems with discontinuities.

CITED REFERENCES AND FURTHER READING:


Ames, W.F. 1977, *Numerical Methods for Partial Differential Equations*, 2nd ed. (New York: Academic Press). [1]

Richtmyer, R.D., and Morton, K.W. 1967, *Difference Methods for Initial Value Problems*, 2nd ed. (New York: Wiley-Interscience). [2]

Roache, P.J. 1976, *Computational Fluid Dynamics* (Albuquerque: Hermosa). [3]

Mitchell, A.R., and Griffiths, D.F. 1980, *The Finite Difference Method in Partial Differential Equations* (New York: Wiley) [includes discussion of finite element methods]. [4]

Dorr, F.W. 1970, *SIAM Review*, vol. 12, pp. 248–263. [5]

Meijerink, J.A., and van der Vorst, H.A. 1977, *Mathematics of Computation*, vol. 31, pp. 148–162. [6]

van der Vorst, H.A. 1981, *Journal of Computational Physics*, vol. 44, pp. 1–19 [review of sparse iterative methods]. [7]

Kershaw, D.S. 1970, *Journal of Computational Physics*, vol. 26, pp. 43–65. [8]

Stone, H.J. 1968, *SIAM Journal on Numerical Analysis*, vol. 5, pp. 530–558. [9]

Jesshope, C.R. 1979, *Computer Physics Communications*, vol. 17, pp. 383–391. [10]

Strang, G., and Fix, G. 1973, *An Analysis of the Finite Element Method* (Englewood Cliffs, NJ: Prentice-Hall). [11]

Burnett, D.S. 1987, *Finite Element Analysis: From Concepts to Applications* (Reading, MA: Addison-Wesley). [12]

Gottlieb, D. and Orszag, S.A. 1977, *Numerical Analysis of Spectral Methods: Theory and Applications* (Philadelphia: S.I.A.M.). [13]

Canuto, C., Hussaini, M.Y., Quarteroni, A., and Zang, T.A. 1988, *Spectral Methods in Fluid Dynamics* (New York: Springer-Verlag). [14]

Boyd, J.P. 1989, *Chebyshev and Fourier Spectral Methods* (New York: Springer-Verlag). [15]


# 19.1 Flux-Conservative Initial Value Problems

A large class of initial value (time-evolution) PDEs in one space dimension can be cast into the form of a *flux-conservative equation*,

$$\frac{\partial \mathbf{u}}{\partial t} = -\frac{\partial \mathbf{F}(\mathbf{u})}{\partial x} \tag{19.1.1}$$

where $\mathbf{u}$ and $\mathbf{F}$ are vectors, and where (in some cases) $\mathbf{F}$ may depend not only on $\mathbf{u}$ but also on spatial derivatives of $\mathbf{u}$. The vector $\mathbf{F}$ is called the *conserved flux*.

For example, the prototypical hyperbolic equation, the one-dimensional wave equation with constant velocity of propagation $v$

$$\frac{\partial^2 u}{\partial t^2} = v^2 \frac{\partial^2 u}{\partial x^2} \tag{19.1.2}$$

can be rewritten as a set of two first-order equations

$$\begin{aligned}\frac{\partial r}{\partial t} &= v\frac{\partial s}{\partial x}\\ \frac{\partial s}{\partial t} &= v\frac{\partial r}{\partial x}\end{aligned} \tag{19.1.3}$$

where

$$\begin{aligned}r &\equiv v\frac{\partial u}{\partial x}\\ s &\equiv \frac{\partial u}{\partial t}\end{aligned} \tag{19.1.4}$$

In this case $r$ and $s$ become the two components of $\mathbf{u}$, and the flux is given by the linear matrix relation

$$\mathbf{F}(\mathbf{u}) = \begin{pmatrix} 0 & -v \\ -v & 0 \end{pmatrix} \cdot \mathbf{u} \tag{19.1.5}$$

(The physicist-reader may recognize equations (19.1.3) as analogous to Maxwell's equations for one-dimensional propagation of electromagnetic waves.)

We will consider, in this section, a prototypical example of the general flux-conservative equation (19.1.1), namely the equation for a scalar $u$,

$$\frac{\partial u}{\partial t} = -v\frac{\partial u}{\partial x} \tag{19.1.6}$$

with $v$ a constant. As it happens, we already know analytically that the general solution of this equation is a wave propagating in the positive $x$-direction,

$$u = f(x - vt) \tag{19.1.7}$$

where $f$ is an arbitrary function. However, the numerical strategies that we develop will be equally applicable to the more general equations represented by (19.1.1). In some contexts, equation (19.1.6) is called an *advective* equation, because the quantity $u$ is transported by a "fluid flow" with a velocity $v$.

How do we go about finite differencing equation (19.1.6) (or, analogously, 19.1.1)? The straightforward approach is to choose equally spaced points along both the $t$- and $x$-axes. Thus denote

$$\begin{aligned}x_j &= x_0 + j\Delta x, \qquad j = 0, 1, \dots, J\\ t_n &= t_0 + n\Delta t, \qquad n = 0, 1, \dots, N\end{aligned} \tag{19.1.8}$$

Let $u_j^n$ denote $u(t_n, x_j)$. We have several choices for representing the time derivative term. The obvious way is to set

$$\left.\frac{\partial u}{\partial t}\right|_{j,n} = \frac{u_j^{n+1} - u_j^n}{\Delta t} + O(\Delta t) \tag{19.1.9}$$

This is called *forward Euler* differencing (cf. equation 16.1.1). While forward Euler is only first-order accurate in $\Delta t$, it has the advantage that one is able to calculate

746
747
748
749
750
751
752
827
828
829
830
831
832
833
834
835

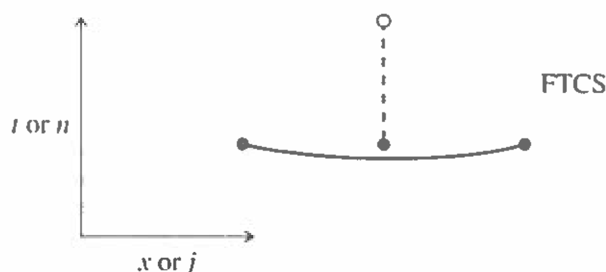



Figure 19.1.1. Representation of the Forward Time Centered Space (FTCS) differencing scheme. In this and subsequent figures, the open circle is the new point at which the solution is desired; filled circles are known points whose function values are used in calculating the new point; the solid lines connect points that are used to calculate spatial derivatives; the dashed lines connect points that are used to calculate time derivatives. The FTCS scheme is generally unstable for hyperbolic problems and cannot usually be used.

quantities at timestep $n+1$ in terms of only quantities known at timestep $n$. For the space derivative, we can use a second-order representation still using only quantities known at timestep $n$:

$$\left.\frac{\partial u}{\partial x}\right|_{j,n} = \frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x} + O(\Delta x^2) \tag{19.1.10}$$

The resulting finite-difference approximation to equation (19.1.6) is called the FTCS representation (Forward Time Centered Space),

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = -v\left(\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}\right) \tag{19.1.11}$$

which can easily be rearranged to be a formula for $u_j^{n+1}$ in terms of the other quantities. The FTCS scheme is illustrated in Figure 19.1.1. It's a fine example of an algorithm that is easy to derive, takes little storage, and executes quickly. Too bad it doesn't work! (See below.)

The FTCS representation is an *explicit* scheme. This means that $u_j^{n+1}$ for each $j$ can be calculated explicitly from the quantities that are already known. Later we shall meet *implicit* schemes, which require us to solve implicit equations coupling the $u_j^{n+1}$ for various $j$. (Explicit and implicit methods for ordinary differential equations were discussed in §16.6.) The FTCS algorithm is also an example of a *single-level* scheme, since only values at time level $n$ have to be stored to find values at time level $n+1$.

## von Neumann Stability Analysis

Unfortunately, equation (19.1.11) is of very limited usefulness. It is an *unstable* method, which can be used only (if at all) to study waves for a short fraction of one oscillation period. To find alternative methods with more general applicability, we must introduce the *von Neumann stability analysis*.

The von Neumann analysis is local: We imagine that the coefficients of the difference equations are so slowly varying as to be considered constant in space and time. In that case, the independent solutions, or *eigenmodes*, of the difference equations are all of the form

$$u_j^n = \xi^n e^{ikj\Delta x} \tag{19.1.12}$$

748
749
750
751
752
827
828
829
830
831
832
833
834
835
836
837

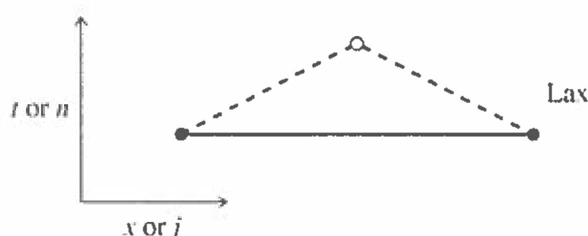



Figure 19.1.2. Representation of the Lax differencing scheme, as in the previous figure. The stability criterion for this scheme is the Courant condition.

where $k$ is a real spatial wave number (which can have any value) and $\xi = \xi(k)$ is a complex number that depends on $k$. The key fact is that the time dependence of a single eigenmode is nothing more than successive integer powers of the complex number $\xi$. Therefore, the difference equations are unstable (have exponentially growing modes) if $|\xi(k)| > 1$ for *some* $k$. The number $\xi$ is called the *amplification factor* at a given wave number $k$.

To find $\xi(k)$, we simply substitute (19.1.12) back into (19.1.11). Dividing by $\xi^n$, we get

$$\xi(k) = 1 - i\frac{v\Delta t}{\Delta x}\sin k\Delta x \tag{19.1.13}$$

whose modulus is $> 1$ for *all* $k$; so the FTCS scheme is unconditionally unstable.

If the velocity $v$ were a function of $t$ and $x$, then we would write $v_j^n$ in equation (19.1.11). In the von Neumann stability analysis we would still treat $v$ as a constant, the idea being that for $v$ slowly varying the analysis is local. In fact, even in the case of strictly constant $v$, the von Neumann analysis does not rigorously treat the end effects at $j = 0$ and $j = N$.

More generally, if the equation's right-hand side were nonlinear in $u$, then a von Neumann analysis would linearize by writing $u = u_0 + \delta u$, expanding to linear order in $\delta u$. Assuming that the $u_0$ quantities already satisfy the difference equation exactly, the analysis would look for an unstable eigenmode of $\delta u$.

Despite its lack of rigor, the von Neumann method generally gives valid answers and is much easier to apply than more careful methods. We accordingly adopt it exclusively. (See, for example, [1] for a discussion of other methods of stability analysis.)

## Lax Method

The instability in the FTCS method can be cured by a simple change due to Lax. One replaces the term $u_j^n$ in the time derivative term by its average (Figure 19.1.2):

$$u_j^n \to \frac{1}{2}\left(u_{j+1}^n + u_{j-1}^n\right) \tag{19.1.14}$$

This turns (19.1.11) into

$$u_j^{n+1} = \frac{1}{2}\left(u_{j+1}^n + u_{j-1}^n\right) - \frac{v\Delta t}{2\Delta x}\left(u_{j+1}^n - u_{j-1}^n\right) \tag{19.1.15}$$

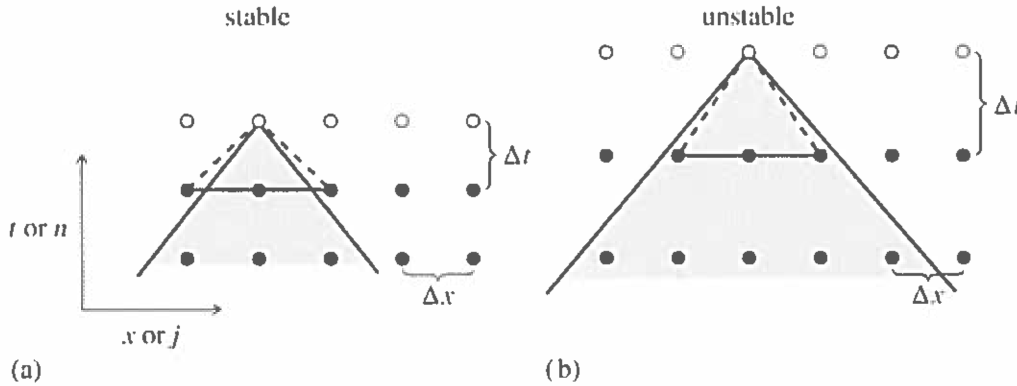



Figure 19.1.3. Courant condition for stability of a differencing scheme. The solution of a hyperbolic problem at a point depends on information within some domain of dependency to the past, shown here shaded. The differencing scheme (19.1.15) has its own domain of dependency determined by the choice of points on one time slice (shown as connected solid dots) whose values are used in determining a new point (shown connected by dashed lines). A differencing scheme is Courant stable if the differencing domain of dependency is larger than that of the PDEs, as in (a), and unstable if the relationship is the reverse, as in (b). For more complicated differencing schemes, the domain of dependency might not be determined simply by the outermost points.

Substituting equation (19.1.12), we find for the amplification factor

$$\xi = \cos k\Delta x - i\frac{v\Delta t}{\Delta x}\sin k\Delta x \tag{19.1.16}$$

The stability condition $|\xi|^2 \le 1$ leads to the requirement

$$\frac{|v|\Delta t}{\Delta x} \le 1 \tag{19.1.17}$$

This is the famous Courant-Friedrichs-Lewy stability criterion, often called simply the *Courant condition*. Intuitively, the stability condition can be understood as follows (Figure 19.1.3): The quantity $u_j^{n+1}$ in equation (19.1.15) is computed from information at points $j-1$ and $j+1$ at time $n$. In other words, $x_{j-1}$ and $x_{j+1}$ are the boundaries of the spatial region that is allowed to communicate information to $u_j^{n+1}$. Now recall that in the continuum wave equation, information actually propagates with a maximum velocity $v$. If the point $u_j^{n+1}$ is outside of the shaded region in Figure 19.1.3, then it requires information from points more distant than the differencing scheme allows. Lack of that information gives rise to an instability. Therefore, $\Delta t$ cannot be made too large.

The surprising result, that the simple replacement (19.1.14) stabilizes the FTCS scheme, is our first encounter with the fact that differencing PDEs is an art as much as a science. To see if we can demystify the art somewhat, let us compare the FTCS and Lax schemes by rewriting equation (19.1.15) so that it is in the form of equation (19.1.11) with a remainder term:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = -v\left(\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}\right) + \frac{1}{2}\left(\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{\Delta t}\right) \tag{19.1.18}$$

But this is exactly the FTCS representation of the equation

$$\frac{\partial u}{\partial t} = -v\frac{\partial u}{\partial x} + \frac{(\Delta x)^2}{2\Delta t}\nabla^2 u \tag{19.1.19}$$

where $\nabla^2 = \partial^2/\partial x^2$ in one dimension. We have, in effect, added a diffusion term to the equation, or, if you recall the form of the Navier-Stokes equation for viscous fluid flow, a dissipative term. The Lax scheme is thus said to have *numerical dissipation*, or *numerical viscosity*. We can see this also in the amplification factor. Unless $|v|\Delta t$ is exactly equal to $\Delta x$, $|\xi| < 1$ and the amplitude of the wave decreases spuriously.

Isn't a spurious decrease as bad as a spurious increase? No. The scales that we hope to study accurately are those that encompass many grid points, so that they have $k\Delta x \ll 1$. (The spatial wave number $k$ is defined by equation 19.1.12.) For these scales, the amplification factor can be seen to be very close to one, in both the stable and unstable schemes. The stable and unstable schemes are therefore about equally accurate. For the unstable scheme, however, short scales with $k\Delta x \sim 1$, *which we are not interested in*, will blow up and swamp the interesting part of the solution. Much better to have a stable scheme in which these short wavelengths die away innocuously. Both the stable and the unstable schemes are *inaccurate* for these short wavelengths, but the inaccuracy is of a tolerable character when the scheme is stable.

When the independent variable $\mathbf{u}$ is a vector, then the von Neumann analysis is slightly more complicated. For example, we can consider equation (19.1.3), rewritten as

$$\frac{\partial}{\partial t}\begin{bmatrix} r \\ s \end{bmatrix} = \frac{\partial}{\partial x}\begin{bmatrix} vs \\ vr \end{bmatrix} \tag{19.1.20}$$

The Lax method for this equation is

$$\begin{aligned} r_j^{n+1} &= \frac{1}{2}(r_{j+1}^n + r_{j-1}^n) + \frac{v\Delta t}{2\Delta x}(s_{j+1}^n - s_{j-1}^n) \\ s_j^{n+1} &= \frac{1}{2}(s_{j+1}^n + s_{j-1}^n) + \frac{v\Delta t}{2\Delta x}(r_{j+1}^n - r_{j-1}^n) \end{aligned} \tag{19.1.21}$$

The von Neumann stability analysis now proceeds by assuming that the eigenmode is of the following (vector) form,

$$\begin{bmatrix} r_j^n \\ s_j^n \end{bmatrix} = \xi^n e^{ikj\Delta x}\begin{bmatrix} r^0 \\ s^0 \end{bmatrix} \tag{19.1.22}$$

Here the vector on the right-hand side is a constant (both in space and in time) eigenvector, and $\xi$ is a complex number, as before. Substituting (19.1.22) into (19.1.21), and dividing by the power $\xi^n$, gives the homogeneous vector equation

$$\begin{bmatrix} (\cos k\Delta x) - \xi & i\dfrac{v\Delta t}{\Delta x}\sin k\Delta x \\ i\dfrac{v\Delta t}{\Delta x}\sin k\Delta x & (\cos k\Delta x) - \xi \end{bmatrix} \cdot \begin{bmatrix} r^0 \\ s^0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{19.1.23}$$

This admits a solution only if the determinant of the matrix on the left vanishes, a condition easily shown to yield the two roots $\xi$

$$\xi = \cos k\Delta x \pm i\frac{v\Delta t}{\Delta x}\sin k\Delta x \tag{19.1.24}$$

The stability condition is that both roots satisfy $|\xi| \le 1$. This again turns out to be simply the Courant condition (19.1.17).

751
752
827
828
829
830
831
832
833
834
835
836
837
838
839
840



### Other Varieties of Error

Thus far we have been concerned with *amplitude error*, because of its intimate connection with the stability or instability of a differencing scheme. Other varieties of error are relevant when we shift our concern to accuracy, rather than stability.

Finite-difference schemes for hyperbolic equations can exhibit dispersion, or *phase errors*. For example, equation (19.1.16) can be rewritten as

$$\xi = e^{-ik\Delta x} + i\left(1 - \frac{v\Delta t}{\Delta x}\right)\sin k\Delta x \tag{19.1.25}$$

An arbitrary initial wave packet is a superposition of modes with different $k$'s. At each timestep the modes get multiplied by different phase factors (19.1.25), depending on their value of $k$. If $\Delta t = \Delta x/v$, then the exact solution for each mode of a wave packet $f(x - vt)$ is obtained if each mode gets multiplied by $\exp(-ik\Delta x)$. For this value of $\Delta t$, equation (19.1.25) shows that the finite-difference solution gives the exact analytic result. However, if $v\Delta t/\Delta x$ is not exactly 1, the phase relations of the modes can become hopelessly garbled and the wave packet disperses. Note from (19.1.25) that the dispersion becomes large as soon as the wavelength becomes comparable to the grid spacing $\Delta x$.

A third type of error is one associated with nonlinear hyperbolic equations and is therefore sometimes called *nonlinear instability*. For example, a piece of the Euler or Navier-Stokes equations for fluid flow looks like

$$\frac{\partial v}{\partial t} = -v\frac{\partial v}{\partial x} + \ldots \tag{19.1.26}$$

The nonlinear term in $v$ can cause a transfer of energy in Fourier space from long wavelengths to short wavelengths. This results in a wave profile steepening until a vertical profile or "shock" develops. Since the von Neumann analysis suggests that the stability can depend on $k\Delta x$, a scheme that was stable for shallow profiles can become unstable for steep profiles. This kind of difficulty arises in a differencing scheme where the cascade in Fourier space is halted at the shortest wavelength representable on the grid, that is, at $k \sim 1/\Delta x$. If energy simply accumulates in these modes, it eventually swamps the energy in the long wavelength modes of interest.

Nonlinear instability and shock formation is thus somewhat controlled by numerical viscosity such as that discussed in connection with equation (19.1.18) above. In some fluid problems, however, shock formation is not merely an annoyance, but an actual physical behavior of the fluid whose detailed study is a goal. Then, numerical viscosity alone may not be adequate or sufficiently controllable. This is a complicated subject which we discuss further in the subsection on fluid dynamics, below.

For wave equations, propagation errors (amplitude or phase) are usually most worrisome. For advective equations, on the other hand, *transport errors* are usually of greater concern. In the Lax scheme, equation (19.1.15), a disturbance in the advected quantity $u$ at mesh point $j$ propagates to mesh points $j + 1$ and $j - 1$ at the next timestep. In reality, however, if the velocity $v$ is positive then only mesh point $j + 1$ should be affected.

752
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841

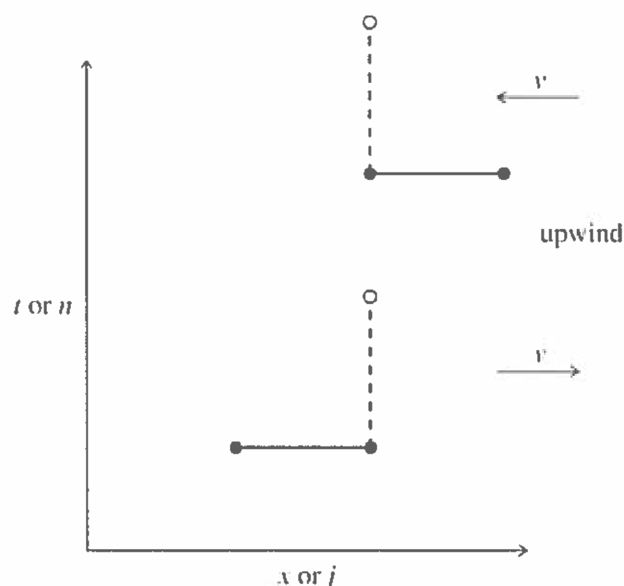



Figure 19.1.4. Representation of upwind differencing schemes. The upper scheme is stable when the advection constant $v$ is negative, as shown; the lower scheme is stable when the advection constant $v$ is positive, also as shown. The Courant condition must, of course, also be satisfied.

The simplest way to model the transport properties "better" is to use *upwind differencing* (see Figure 19.1.4):

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = -v_j^n \begin{cases} \dfrac{u_j^n - u_{j-1}^n}{\Delta x}, & v_j^n > 0 \\ \dfrac{u_{j+1}^n - u_j^n}{\Delta x}, & v_j^n < 0 \end{cases} \tag{19.1.27}$$

Note that this scheme is only first-order, not second-order, accurate in the calculation of the spatial derivatives. How can it be "better"? The answer is one that annoys the mathematicians: The goal of numerical simulations is not always "accuracy" in a strictly mathematical sense, but sometimes "fidelity" to the underlying physics in a sense that is looser and more pragmatic. In such contexts, some kinds of error are much more tolerable than others. Upwind differencing generally adds fidelity to problems where the advected variables are liable to undergo sudden changes of state, e.g., as they pass through shocks or other discontinuities. You will have to be guided by the specific nature of your own problem.

For the differencing scheme (19.1.27), the amplification factor (for constant $v$) is

$$\xi = 1 - \left|\frac{v\Delta t}{\Delta x}\right| (1 - \cos k\Delta x) - i\frac{v\Delta t}{\Delta x} \sin k\Delta x \tag{19.1.28}$$

$$|\xi|^2 = 1 - 2\left|\frac{v\Delta t}{\Delta x}\right| \left(1 - \left|\frac{v\Delta t}{\Delta x}\right|\right) (1 - \cos k\Delta x) \tag{19.1.29}$$

So the stability criterion $|\xi|^2 \leq 1$ is (again) simply the Courant condition (19.1.17).

There are various ways of improving the accuracy of first-order upwind differencing. In the continuum equation, material originally a distance $v\Delta t$ away

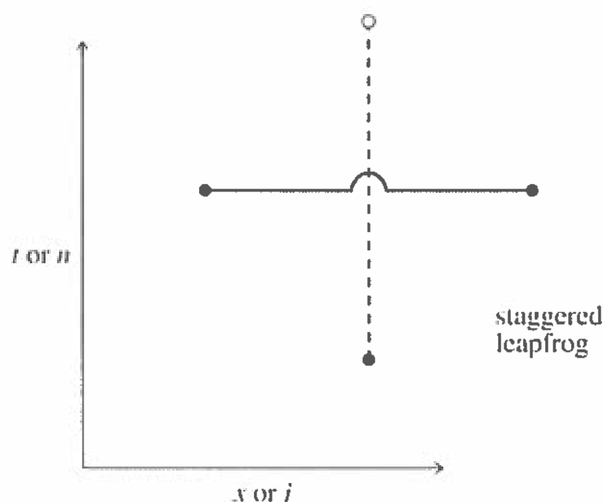



Figure 19.1.5. Representation of the staggered leapfrog differencing scheme. Note that information from two previous time slices is used in obtaining the desired point. This scheme is second-order accurate in both space and time.

arrives at a given point after a time interval $\Delta t$. In the first-order method, the material always arrives from $\Delta x$ away. If $v\Delta t \ll \Delta x$ (to insure accuracy), this can cause a large error. One way of reducing this error is to interpolate $u$ between $j-1$ and $j$ before transporting it. This gives effectively a second-order method. Various schemes for second-order upwind differencing are discussed and compared in [2-3].

## Second-Order Accuracy in Time

When using a method that is first-order accurate in time but second-order accurate in space, one generally has to take $v\Delta t$ significantly smaller than $\Delta x$ to achieve desired accuracy, say, by at least a factor of 5. Thus the Courant condition is not actually the limiting factor with such schemes in practice. However, there are schemes that are second-order accurate in both space and time, and these can often be pushed right to their stability limit, with correspondingly smaller computation times.

For example, the *staggered leapfrog* method for the conservation equation (19.1.1) is defined as follows (Figure 19.1.5): Using the values of $u^n$ at time $t^n$, compute the fluxes $F_j^n$. Then compute new values $u^{n+1}$ using the time-centered values of the fluxes:

$$u_j^{n+1} - u_j^{n-1} = -\frac{\Delta t}{\Delta x}(F_{j+1}^n - F_{j-1}^n) \tag{19.1.30}$$

The name comes from the fact that the time levels in the time derivative term "leapfrog" over the time levels in the space derivative term. The method requires that $u^{n-1}$ and $u^n$ be stored to compute $u^{n+1}$.

For our simple model equation (19.1.6), staggered leapfrog takes the form

$$u_j^{n+1} - u_j^{n-1} = -\frac{v\Delta t}{\Delta x}(u_{j+1}^n - u_{j-1}^n) \tag{19.1.31}$$

The von Neumann stability analysis now gives a quadratic equation for $\xi$, rather than a linear one, because of the occurrence of three consecutive powers of $\xi$ when the

form (19.1.12) for an eigenmode is substituted into equation (19.1.31).

$$\xi^2 - 1 = -2i\xi \frac{v\Delta t}{\Delta x} \sin k\Delta x \tag{19.1.32}$$

whose solution is

$$\xi = -i\frac{v\Delta t}{\Delta x} \sin k\Delta x \pm \sqrt{1 - \left(\frac{v\Delta t}{\Delta x} \sin k\Delta x\right)^2} \tag{19.1.33}$$

Thus the Courant condition is again required for stability. In fact, in equation (19.1.33), $|\xi|^2 = 1$ for any $v\Delta t \le \Delta x$. This is the great advantage of the staggered leapfrog method: There is no amplitude dissipation.

Staggered leapfrog differencing of equations like (19.1.20) is most transparent if the variables are centered on appropriate half-mesh points:

$$\begin{aligned} r^n_{j+1/2} &\equiv v \left.\frac{\partial u}{\partial x}\right|^n_{j+1/2} = v\frac{u^n_{j+1} - u^n_j}{\Delta x} \\ s^{n+1/2}_j &\equiv \left.\frac{\partial u}{\partial t}\right|^{n+1/2}_j = \frac{u^{n+1}_j - u^n_j}{\Delta t} \end{aligned} \tag{19.1.34}$$

This is purely a notational convenience: we can think of the mesh on which $r$ and $s$ are defined as being twice as fine as the mesh on which the original variable $u$ is defined. The leapfrog differencing of equation (19.1.20) is

$$\begin{aligned} \frac{r^{n+1}_{j+1/2} - r^n_{j+1/2}}{\Delta t} &= \frac{s^{n+1/2}_{j+1} - s^{n+1/2}_j}{\Delta x} \\ \frac{s^{n+1/2}_j - s^{n-1/2}_j}{\Delta t} &= v\frac{r^n_{j+1/2} - r^n_{j-1/2}}{\Delta x} \end{aligned} \tag{19.1.35}$$

If you substitute equation (19.1.22) in equation (19.1.35), you will find that once again the Courant condition is required for stability, and that there is no amplitude dissipation when it is satisfied.

If we substitute equation (19.1.34) in equation (19.1.35), we find that equation (19.1.35) is equivalent to

$$\frac{u^{n+1}_j - 2u^n_j + u^{n-1}_j}{(\Delta t)^2} = v^2\frac{u^n_{j+1} - 2u^n_j + u^n_{j-1}}{(\Delta x)^2} \tag{19.1.36}$$

This is just the "usual" second-order differencing of the wave equation (19.1.2). We see that it is a two-level scheme, requiring both $u^n$ and $u^{n-1}$ to obtain $u^{n+1}$. In equation (19.1.35) this shows up as both $s^{n-1/2}$ and $r^n$ being needed to advance the solution.

For equations more complicated than our simple model equation, especially nonlinear equations, the leapfrog method usually becomes unstable when the gradients get large. The instability is related to the fact that odd and even mesh points are completely decoupled, like the black and white squares of a chess board, as shown

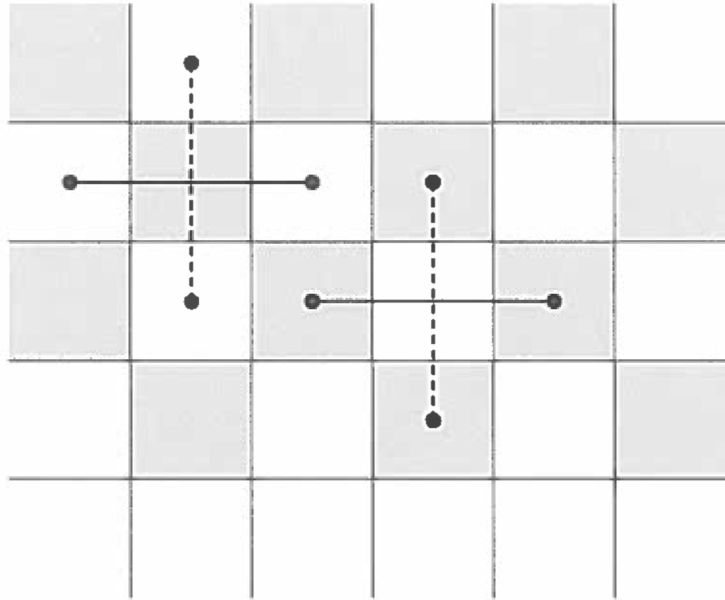


Figure 19.1.6. Origin of mesh-drift instabilities in a staggered leapfrog scheme. If the mesh points are imagined to lie in the squares of a chess board, then white squares couple to themselves, black to themselves, but there is no coupling between white and black. The fix is to introduce a small diffusive mesh-coupling piece.

in Figure 19.1.6. This mesh drifting instability is cured by coupling the two meshes through a numerical viscosity term, e.g., adding to the right side of (19.1.31) a small coefficient ($\ll 1$) times $u_{j+1}^n - 2u_j^n + u_{j-1}^n$. For more on stabilizing difference schemes by adding numerical dissipation, see, e.g., [4].

The *Two-Step Lax-Wendroff* scheme is a second-order in time method that avoids large numerical dissipation and mesh drifting. One defines intermediate values $u_{j+1/2}$ at the half timesteps $t_{n+1/2}$ and the half mesh points $x_{j+1/2}$. These are calculated by the Lax scheme:

$$u_{j+1/2}^{n+1/2} = \frac{1}{2}(u_{j+1}^n + u_j^n) - \frac{\Delta t}{2\Delta x}(F_{j+1}^n - F_j^n) \qquad (19.1.37)$$

Using these variables, one calculates the fluxes $F_{j+1/2}^{n+1/2}$. Then the updated values $u_j^{n+1}$ are calculated by the properly centered expression

$$u_j^{n+1} = u_j^n - \frac{\Delta t}{\Delta x}\left(F_{j+1/2}^{n+1/2} - F_{j-1/2}^{n+1/2}\right) \qquad (19.1.38)$$

The provisional values $u_{j+1/2}^{n+1/2}$ are now discarded. (See Figure 19.1.7.)

Let us investigate the stability of this method for our model advective equation, where $F = vu$. Substitute (19.1.37) in (19.1.38) to get

$$\begin{aligned} u_j^{n+1} = u_j^n - \alpha \Big[ &\frac{1}{2}(u_{j+1}^n + u_j^n) - \frac{1}{2}\alpha(u_{j+1}^n - u_j^n) \\ &- \frac{1}{2}(u_j^n + u_{j-1}^n) + \frac{1}{2}\alpha(u_j^n - u_{j-1}^n)\Big] \end{aligned} \qquad (19.1.39)$$

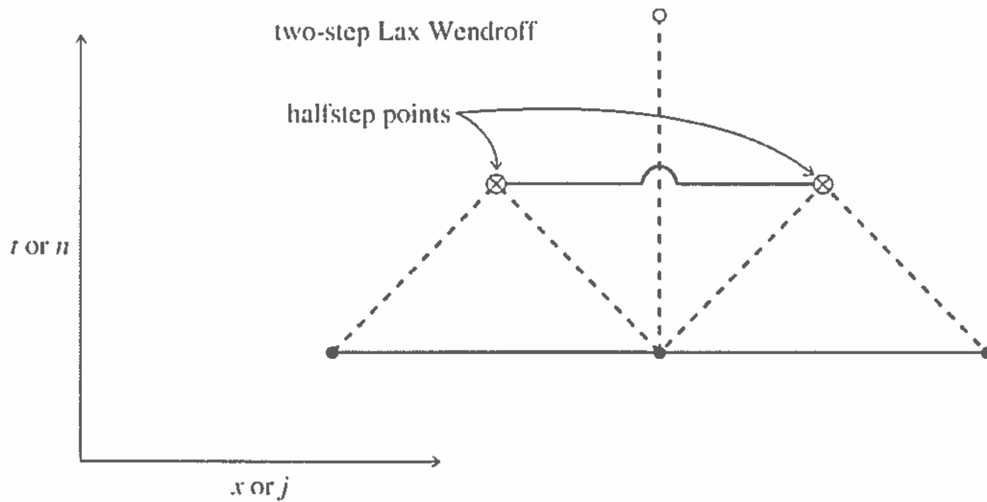



Figure 19.1.7. Representation of the two-step Lax-Wendroff differencing scheme. Two halfstep points (⊗) are calculated by the Lax method. These, plus one of the original points, produce the new point via staggered leapfrog. Halfstep points are used only temporarily and do not require storage allocation on the grid. This scheme is second-order accurate in both space and time.

where

$$\alpha \equiv \frac{v\Delta t}{\Delta x} \tag{19.1.40}$$

Then

$$\xi = 1 - i\alpha \sin k\Delta x - \alpha^2(1 - \cos k\Delta x) \tag{19.1.41}$$

so

$$|\xi|^2 = 1 - \alpha^2(1 - \alpha^2)(1 - \cos k\Delta x)^2 \tag{19.1.42}$$

The stability criterion $|\xi|^2 \leq 1$ is therefore $\alpha^2 \leq 1$, or $v\Delta t \leq \Delta x$ as usual. Incidentally, you should not think that the Courant condition is the only stability requirement that ever turns up in PDEs. It keeps doing so in our model examples just because those examples are so simple in form. The method of analysis is, however, general.

Except when $\alpha = 1$, $|\xi|^2 < 1$ in (19.1.42), so some amplitude damping does occur. The effect is relatively small, however, for wavelengths large compared with the mesh size $\Delta x$. If we expand (19.1.42) for small $k\Delta x$, we find

$$|\xi|^2 = 1 - \alpha^2(1 - \alpha^2)\frac{(k\Delta x)^4}{4} + \dots \tag{19.1.43}$$

The departure from unity occurs only at fourth order in $k$. This should be contrasted with equation (19.1.16) for the Lax method, which shows that

$$|\xi|^2 = 1 - (1 - \alpha^2)(k\Delta x)^2 + \dots \tag{19.1.44}$$

for small $k\Delta x$.

In summary, our recommendation for initial value problems that can be cast in flux-conservative form, and especially problems related to the wave equation, is to use the staggered leapfrog method when possible. We have personally had better success with it than with the Two-Step Lax-Wendroff method. For problems sensitive to transport errors, upwind differencing or one of its refinements should be considered.

## Fluid Dynamics with Shocks

As we alluded to earlier, the treatment of fluid dynamics problems with shocks has become a very complicated and very sophisticated subject. All we can attempt to do here is to guide you to some starting points in the literature.

There are basically three important general methods for handling shocks. The oldest and simplest method, invented by von Neumann and Richtmyer, is to add *artificial viscosity* to the equations, modeling the way Nature uses real viscosity to smooth discontinuities. A good starting point for trying out this method is the differencing scheme in §12.11 of [1]. This scheme is excellent for nearly all problems in one spatial dimension.

The second method combines a high-order differencing scheme that is accurate for smooth flows with a low order scheme that is very dissipative and can smooth the shocks. Typically, various upwind differencing schemes are combined using weights chosen to zero the low order scheme unless steep gradients are present, and also chosen to enforce various "monotonicity" constraints that prevent nonphysical oscillations from appearing in the numerical solution. References [2-3,5] are a good place to start with these methods.

The third, and potentially most powerful method, is Godunov's approach. Here one gives up the simple linearization inherent in finite differencing based on Taylor series and includes the nonlinearity of the equations explicitly. There is an analytic solution for the evolution of two uniform states of a fluid separated by a discontinuity, the Riemann shock problem. Godunov's idea was to approximate the fluid by a large number of cells of uniform states, and piece them together using the Riemann solution. There have been many generalizations of Godunov's approach, of which the most powerful is probably the PPM method [6].

Readable reviews of all these methods, discussing the difficulties arising when one-dimensional methods are generalized to multidimensions, are given in [7-9].

CITED REFERENCES AND FURTHER READING:

Ames, W.F. 1977, *Numerical Methods for Partial Differential Equations*, 2nd ed. (New York: Academic Press), Chapter 4.

Richtmyer, R.D., and Morton, K.W. 1967, *Difference Methods for Initial Value Problems*, 2nd ed. (New York: Wiley-Interscience). [1]

Centrella, J., and Wilson, J.R. 1984, *Astrophysical Journal Supplement*, vol. 54, pp. 229–249, Appendix B. [2]

Hawley, J.F., Smarr, L.L., and Wilson, J.R. 1984, *Astrophysical Journal Supplement*, vol. 55, pp. 211–246, §2c. [3]

Kreiss, H.-O. 1978, *Numerical Methods for Solving Time-Dependent Problems for Partial Differential Equations* (Montreal: University of Montreal Press), pp. 66ff. [4]

Harten, A., Lax, P.D., and Van Leer, B. 1983, *SIAM Review*, vol. 25, pp. 36–61. [5]

Woodward, P., and Colella, P. 1984, *Journal of Computational Physics*, vol. 54, pp. 174–201. [6]

Roache, P.J. 1976, *Computational Fluid Dynamics* (Albuquerque: Hermosa). [7]

Woodward, P., and Colella, P. 1984, *Journal of Computational Physics*, vol. 54, pp. 115–173. [8]
Rizzi, A., and Engquist, B. 1987, *Journal of Computational Physics*, vol. 72, pp. 1–69. [9]

## 19.2 Diffusive Initial Value Problems

Recall the model parabolic equation, the diffusion equation in one space dimension,

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(D\frac{\partial u}{\partial x}\right) \tag{19.2.1}$$

where $D$ is the diffusion coefficient. Actually, this equation is a flux-conservative equation of the form considered in the previous section, with

$$F = -D\frac{\partial u}{\partial x} \tag{19.2.2}$$

the flux in the $x$-direction. We will assume $D \geq 0$, otherwise equation (19.2.1) has physically unstable solutions: A small disturbance evolves to become more and more concentrated instead of dispersing. (Don't make the mistake of trying to find a stable differencing scheme for a problem whose underlying PDEs are themselves unstable!)

Even though (19.2.1) is of the form already considered, it is useful to consider it as a model in its own right. The particular form of flux (19.2.2), and its direct generalizations, occur quite frequently in practice. Moreover, we have already seen that numerical viscosity and artificial viscosity can introduce diffusive pieces like the right-hand side of (19.2.1) in many other situations.

Consider first the case when $D$ is a constant. Then the equation

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2} \tag{19.2.3}$$

can be differenced in the obvious way:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = D\left[\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta x)^2}\right] \tag{19.2.4}$$

This is the FTCS scheme again, except that it is a second derivative that has been differenced on the right-hand side. But this makes a world of difference! The FTCS scheme was unstable for the hyperbolic equation; however, a quick calculation shows that the amplification factor for equation (19.2.4) is

$$\xi = 1 - \frac{4D\Delta t}{(\Delta x)^2}\sin^2\left(\frac{k\Delta x}{2}\right) \tag{19.2.5}$$

The requirement $|\xi| \leq 1$ leads to the stability criterion

$$\frac{2D\Delta t}{(\Delta x)^2} \leq 1 \tag{19.2.6}$$

The physical interpretation of the restriction (19.2.6) is that the maximum allowed timestep is, up to a numerical factor, the diffusion time across a cell of width $\Delta x$.

More generally, the diffusion time $\tau$ across a spatial scale of size $\lambda$ is of order

$$\tau \sim \frac{\lambda^2}{D} \tag{19.2.7}$$

Usually we are interested in modeling accurately the evolution of features with spatial scales $\lambda \gg \Delta x$. If we are limited to timesteps satisfying (19.2.6), we will need to evolve through of order $\lambda^2/(\Delta x)^2$ steps before things start to happen on the scale of interest. This number of steps is usually prohibitive. We must therefore find a stable way of taking timesteps comparable to, or perhaps — for accuracy — somewhat smaller than, the time scale of (19.2.7).

This goal poses an immediate "philosophical" question. Obviously the large timesteps that we propose to take are going to be woefully inaccurate for the small scales that we have decided not to be interested in. We want those scales to do something stable, "innocuous," and perhaps not too physically unreasonable. We want to build this innocuous behavior into our differencing scheme. What should it be?

There are two different answers, each of which has its pros and cons. The first answer is to seek a differencing scheme that drives small-scale features to their *equilibrium* forms, e.g., satisfying equation (19.2.3) with the left-hand side set to zero. This answer generally makes the best physical sense; but, as we will see, it leads to a differencing scheme ("fully implicit") that is only *first-order* accurate in time for the scales that we are interested in. The second answer is to let small-scale features *maintain* their initial amplitudes, so that the evolution of the larger-scale features of interest takes place superposed with a kind of "frozen in" (though fluctuating) background of small-scale stuff. This answer gives a differencing scheme ("Crank-Nicolson") that is *second-order* accurate in time. Toward the end of an evolution calculation, however, one might want to switch over to some steps of the other kind, to drive the small-scale stuff into equilibrium. Let us now see where these distinct differencing schemes come from:

Consider the following differencing of (19.2.3),

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = D\left[\frac{u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}}{(\Delta x)^2}\right] \tag{19.2.8}$$

This is exactly like the FTCS scheme (19.2.4), except that the spatial derivatives on the right-hand side are evaluated at timestep $n+1$. Schemes with this character are called *fully implicit* or *backward time*, by contrast with FTCS (which is called *fully explicit*). To solve equation (19.2.8) one has to solve a set of simultaneous linear equations at each timestep for the $u_j^{n+1}$. Fortunately, this is a simple problem because the system is tridiagonal: Just group the terms in equation (19.2.8) appropriately:

$$-\alpha u_{j-1}^{n+1} + (1+2\alpha)u_j^{n+1} - \alpha u_{j+1}^{n+1} = u_j^n, \qquad j = 1, 2 \ldots J-1 \tag{19.2.9}$$

where

$$\alpha \equiv \frac{D\Delta t}{(\Delta x)^2} \tag{19.2.10}$$

Supplemented by Dirichlet or Neumann boundary conditions at $j = 0$ and $j = J$, equation (19.2.9) is clearly a tridiagonal system, which can easily be solved at each timestep by the method of §2.4.

What is the behavior of (19.2.8) for very large timesteps? The answer is seen most clearly in (19.2.9), in the limit $\alpha \to \infty$ ($\Delta t \to \infty$). Dividing by $\alpha$, we see that the difference equations are just the finite-difference form of the equilibrium equation

$$\frac{\partial^2 u}{\partial x^2} = 0 \tag{19.2.11}$$

What about stability? The amplification factor for equation (19.2.8) is

$$\xi = \frac{1}{1 + 4\alpha \sin^2\left(\frac{k\Delta x}{2}\right)} \tag{19.2.12}$$

Clearly $|\xi| < 1$ for any stepsize $\Delta t$. The scheme is unconditionally stable. The details of the small-scale evolution from the initial conditions are obviously inaccurate for large $\Delta t$. But, as advertised, the correct equilibrium solution is obtained. This is the characteristic feature of implicit methods.

Here, on the other hand, is how one gets to the second of our above philosophical answers, combining the stability of an implicit method with the accuracy of a method that is second-order in both space and time. Simply form the average of the explicit and implicit FTCS schemes:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = \frac{D}{2}\left[\frac{(u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}) + (u_{j+1}^n - 2u_j^n + u_{j-1}^n)}{(\Delta x)^2}\right] \tag{19.2.13}$$

Here both the left- and right-hand sides are centered at timestep $n + \frac{1}{2}$, so the method is second-order accurate in time as claimed. The amplification factor is

$$\xi = \frac{1 - 2\alpha \sin^2\left(\frac{k\Delta x}{2}\right)}{1 + 2\alpha \sin^2\left(\frac{k\Delta x}{2}\right)} \tag{19.2.14}$$

so the method is stable for any size $\Delta t$. This scheme is called the *Crank-Nicolson* scheme, and is our recommended method for any simple diffusion problem (perhaps supplemented by a few fully implicit steps at the end). (See Figure 19.2.1.)

Now turn to some generalizations of the simple diffusion equation (19.2.3). Suppose first that the diffusion coefficient $D$ is not constant, say $D = D(x)$. We can adopt either of two strategies. First, we can make an analytic change of variable

$$y = \int \frac{dx}{D(x)} \tag{19.2.15}$$

Copyright (C) Numerical Recipes Software 1988-1992

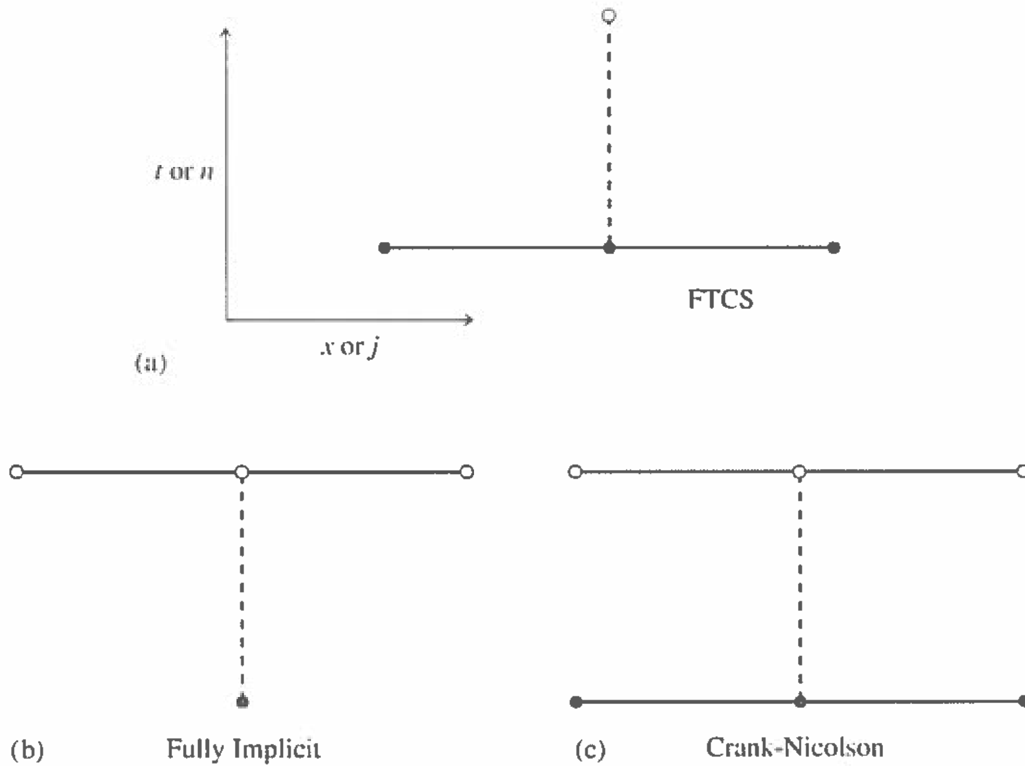



Figure 19.2.1. Three differencing schemes for diffusive problems (shown as in Figure 19.1.2). (a) Forward Time Center Space is first-order accurate, but stable only for sufficiently small timesteps. (b) Fully Implicit is stable for arbitrarily large timesteps, but is still only first-order accurate. (c) Crank-Nicolson is second-order accurate, and is usually stable for large timesteps.

Then

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x} D(x) \frac{\partial u}{\partial x} \tag{19.2.16}$$

becomes

$$\frac{\partial u}{\partial t} = \frac{1}{D(y)} \frac{\partial^2 u}{\partial y^2} \tag{19.2.17}$$

and we evaluate $D$ at the appropriate $y_j$. Heuristically, the stability criterion (19.2.6) in an explicit scheme becomes

$$\Delta t \le \min_j \left[ \frac{(\Delta y)^2}{2D_j^{-1}} \right] \tag{19.2.18}$$

Note that constant spacing $\Delta y$ in $y$ does not imply constant spacing in $x$.

An alternative method that does not require analytically tractable forms for $D$ is simply to difference equation (19.2.16) as it stands, centering everything appropriately. Thus the FTCS method becomes

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = \frac{D_{j+1/2}(u_{j+1}^n - u_j^n) - D_{j-1/2}(u_j^n - u_{j-1}^n)}{(\Delta x)^2} \tag{19.2.19}$$

where

$$D_{j+1/2} \equiv D(x_{j+1/2}) \tag{19.2.20}$$

and the heuristic stability criterion is

$$\Delta t \leq \min_j \left[ \frac{(\Delta x)^2}{2D_{j+1/2}} \right] \tag{19.2.21}$$

The Crank-Nicolson method can be generalized similarly.

The second complication one can consider is a nonlinear diffusion problem, for example where $D = D(u)$. Explicit schemes can be generalized in the obvious way. For example, in equation (19.2.19) write

$$D_{j+1/2} = \frac{1}{2} \left[ D(u^n_{j+1}) + D(u^n_j) \right] \tag{19.2.22}$$

Implicit schemes are not as easy. The replacement (19.2.22) with $n \to n+1$ leaves us with a nasty set of coupled nonlinear equations to solve at each timestep. Often there is an easier way: If the form of $D(u)$ allows us to integrate

$$dz = D(u)du \tag{19.2.23}$$

analytically for $z(u)$, then the right-hand side of (19.2.1) becomes $\partial^2 z / \partial x^2$, which we difference implicitly as

$$\frac{z^{n+1}_{j+1} - 2z^{n+1}_j + z^{n+1}_{j-1}}{(\Delta x)^2} \tag{19.2.24}$$

Now linearize each term on the right-hand side of equation (19.2.24), for example

$$\begin{aligned} z^{n+1}_j \equiv z(u^{n+1}_j) &= z(u^n_j) + (u^{n+1}_j - u^n_j) \left. \frac{\partial z}{\partial u} \right|_{j,n} \\ &= z(u^n_j) + (u^{n+1}_j - u^n_j) D(u^n_j) \end{aligned} \tag{19.2.25}$$

This reduces the problem to tridiagonal form again and in practice usually retains the stability advantages of fully implicit differencing.

## Schrödinger Equation

Sometimes the physical problem being solved imposes constraints on the differencing scheme that we have not yet taken into account. For example, consider the time-dependent Schrödinger equation of quantum mechanics. This is basically a parabolic equation for the evolution of a complex quantity $\psi$. For the scattering of a wavepacket by a one-dimensional potential $V(x)$, the equation has the form

$$i \frac{\partial \psi}{\partial t} = -\frac{\partial^2 \psi}{\partial x^2} + V(x)\psi \tag{19.2.26}$$

(Here we have chosen units so that Planck's constant $\hbar = 1$ and the particle mass $m = 1/2$.) One is given the initial wavepacket, $\psi(x, t = 0)$, together with boundary

836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851

conditions that $\psi \to 0$ at $x \to \pm\infty$. Suppose we content ourselves with first-order accuracy in time, but want to use an implicit scheme, for stability. A slight generalization of (19.2.8) leads to

$$i\left[\frac{\psi_j^{n+1}-\psi_j^n}{\Delta t}\right] = -\left[\frac{\psi_{j+1}^{n+1}-2\psi_j^{n+1}+\psi_{j-1}^{n+1}}{(\Delta x)^2}\right] + V_j\psi_j^{n+1} \tag{19.2.27}$$

for which

$$\xi = \frac{1}{1+i\left[\frac{4\Delta t}{(\Delta x)^2}\sin^2\left(\frac{k\Delta x}{2}\right)+V_j\Delta t\right]} \tag{19.2.28}$$

This is unconditionally stable, but unfortunately is not *unitary*. The underlying physical problem requires that the total probability of finding the particle somewhere remains unity. This is represented formally by the modulus-square norm of $\psi$ remaining unity:

$$\int_{-\infty}^{\infty} |\psi|^2 dx = 1 \tag{19.2.29}$$

The initial wave function $\psi(x, 0)$ is normalized to satisfy (19.2.29). The Schrödinger equation (19.2.26) then guarantees that this condition is satisfied at all later times.

Let us write equation (19.2.26) in the form

$$i\frac{\partial\psi}{\partial t} = H\psi \tag{19.2.30}$$

where the operator $H$ is

$$H = -\frac{\partial^2}{\partial x^2} + V(x) \tag{19.2.31}$$

The formal solution of equation (19.2.30) is

$$\psi(x,t) = e^{-iHt}\psi(x,0) \tag{19.2.32}$$

where the exponential of the operator is defined by its power series expansion.

The unstable explicit FTCS scheme approximates (19.2.32) as

$$\psi_j^{n+1} = (1 - iH\Delta t)\psi_j^n \tag{19.2.33}$$

where $H$ is represented by a centered finite-difference approximation in $x$. The stable implicit scheme (19.2.27) is, by contrast,

$$\psi_j^{n+1} = (1 + iH\Delta t)^{-1}\psi_j^n \tag{19.2.34}$$

These are both first-order accurate in time, as can be seen by expanding equation (19.2.32). However, neither operator in (19.2.33) or (19.2.34) is unitary.

The correct way to difference Schrödinger's equation [1,2] is to use *Cayley's form* for the finite-difference representation of $e^{-iHt}$, which is second-order accurate *and* unitary:

$$e^{-iHt} \simeq \frac{1 - \frac{1}{2} i H \Delta t}{1 + \frac{1}{2} i H \Delta t} \tag{19.2.35}$$

In other words,

$$\left(1 + \tfrac{1}{2} i H \Delta t\right) \psi_j^{n+1} = \left(1 - \tfrac{1}{2} i H \Delta t\right) \psi_j^n \tag{19.2.36}$$

On replacing $H$ by its finite-difference approximation in $x$, we have a complex tridiagonal system to solve. The method is stable, unitary, and second-order accurate in space and time. In fact, it is simply the Crank-Nicolson method once again!

CITED REFERENCES AND FURTHER READING:

Ames, W.F. 1977, *Numerical Methods for Partial Differential Equations*, 2nd ed. (New York: Academic Press), Chapter 2.

Goldberg, A., Schey, H.M., and Schwartz, J.L. 1967, *American Journal of Physics*, vol. 35, pp. 177–186. [1]

Galbraith, I., Ching, Y.S., and Abraham, E. 1984, *American Journal of Physics*, vol. 52, pp. 60–68. [2]

# 19.3 Initial Value Problems in Multidimensions

The methods described in §19.1 and §19.2 for problems in 1 + 1 dimension (one space and one time dimension) can easily be generalized to $N + 1$ dimensions. However, the computing power necessary to solve the resulting equations is enormous. If you have solved a one-dimensional problem with 100 spatial grid points, solving the two-dimensional version with $100 \times 100$ mesh points requires *at least* 100 times as much computing. You generally have to be content with very modest spatial resolution in multidimensional problems.

Indulge us in offering a bit of advice about the development and testing of multidimensional PDE codes: You should always first run your programs on *very small* grids, e.g., $8 \times 8$, even though the resulting accuracy is so poor as to be useless. When your program is all debugged and demonstrably stable, *then* you can increase the grid size to a reasonable one and start looking at the results. We have actually heard someone protest, "my program would be unstable for a crude grid, but I am sure the instability will go away on a larger grid." That is nonsense of a most pernicious sort, evidencing total confusion between accuracy and stability. In fact, new instabilities sometimes do show up on *larger* grids; but old instabilities never (in our experience) just go away.

Forced to live with modest grid sizes, some people recommend going to higher-order methods in an attempt to improve accuracy. This is very dangerous. Unless the solution you are looking for is known to be smooth, and the high-order method you

are using is known to be extremely stable, we do not recommend anything higher than second-order in time (for sets of first-order equations). For spatial differencing, we recommend the order of the underlying PDEs, perhaps allowing second-order spatial differencing for first-order-in-space PDEs. When you increase the order of a differencing method to greater than the order of the original PDEs, you introduce spurious solutions to the difference equations. This does not create a problem if they all happen to decay exponentially; otherwise you are going to see all hell break loose!

### Lax Method for a Flux-Conservative Equation

As an example, we show how to generalize the Lax method (19.1.15) to two dimensions for the conservation equation

$$\frac{\partial u}{\partial t} = -\nabla \cdot \mathbf{F} = -\left(\frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y}\right) \tag{19.3.1}$$

Use a spatial grid with

$$\begin{aligned} x_j &= x_0 + j\Delta \\ y_l &= y_0 + l\Delta \end{aligned} \tag{19.3.2}$$

We have chosen $\Delta x = \Delta y \equiv \Delta$ for simplicity. Then the Lax scheme is

$$\begin{aligned} u_{j,l}^{n+1} = {} & \frac{1}{4}(u_{j+1,l}^n + u_{j-1,l}^n + u_{j,l+1}^n + u_{j,l-1}^n) \\ & - \frac{\Delta t}{2\Delta}(F_{j+1,l}^n - F_{j-1,l}^n + F_{j,l+1}^n - F_{j,l-1}^n) \end{aligned} \tag{19.3.3}$$

Note that as an abbreviated notation $F_{j+1}$ and $F_{j-1}$ refer to $F_x$, while $F_{l+1}$ and $F_{l-1}$ refer to $F_y$.

Let us carry out a stability analysis for the model advective equation (analog of 19.1.6) with

$$F_x = v_x u, \qquad F_y = v_y u \tag{19.3.4}$$

This requires an eigenmode with two dimensions in space, though still only a simple dependence on powers of $\xi$ in time,

$$u_{j,l}^n = \xi^n e^{ik_x j\Delta} e^{ik_y l\Delta} \tag{19.3.5}$$

Substituting in equation (19.3.3), we find

$$\xi = \frac{1}{2}(\cos k_x\Delta + \cos k_y\Delta) - i\alpha_x \sin k_x\Delta - i\alpha_y \sin k_y\Delta \tag{19.3.6}$$

where

$$\alpha_x = \frac{v_x \Delta t}{\Delta}, \qquad \alpha_y = \frac{v_y \Delta t}{\Delta} \tag{19.3.7}$$

The expression for $|\xi|^2$ can be manipulated into the form

$$\begin{aligned}|\xi|^2 = 1 - (\sin^2 k_x\Delta + \sin^2 k_y\Delta)\left[\frac{1}{2} - (\alpha_x^2 + \alpha_y^2)\right] \\ - \frac{1}{4}(\cos k_x\Delta - \cos k_y\Delta)^2 - (\alpha_y \sin k_x\Delta - \alpha_x \sin k_y\Delta)^2\end{aligned} \tag{19.3.8}$$

The last two terms are negative, and so the stability requirement $|\xi|^2 \leq 1$ becomes

$$\frac{1}{2} - (\alpha_x^2 + \alpha_y^2) \geq 0 \tag{19.3.9}$$

or

$$\Delta t \leq \frac{\Delta}{\sqrt{2}(v_x^2 + v_y^2)^{1/2}} \tag{19.3.10}$$

This is an example of the general result for the $N$-dimensional Courant condition: If $|v|$ is the maximum propagation velocity in the problem, then

$$\Delta t \leq \frac{\Delta}{\sqrt{N}|v|} \tag{19.3.11}$$

is the Courant condition.

## Diffusion Equation in Multidimensions

Let us consider the two-dimensional diffusion equation,

$$\frac{\partial u}{\partial t} = D\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \tag{19.3.12}$$

An explicit method, such as FTCS, can be generalized from the one-dimensional case in the obvious way. However, we have seen that diffusive problems are usually best treated implicitly. Suppose we try to implement the Crank-Nicolson scheme in two dimensions. This would give us

$$u_{j,l}^{n+1} = u_{j,l}^n + \frac{1}{2}\alpha\left(\delta_x^2 u_{j,l}^{n+1} + \delta_x^2 u_{j,l}^n + \delta_y^2 u_{j,l}^{n+1} + \delta_y^2 u_{j,l}^n\right) \tag{19.3.13}$$

Here

$$\alpha \equiv \frac{D\Delta t}{\Delta^2} \qquad \Delta \equiv \Delta x = \Delta y \tag{19.3.14}$$

$$\delta_x^2 u_{j,l}^n \equiv u_{j+1,l}^n - 2u_{j,l}^n + u_{j-1,l}^n \tag{19.3.15}$$

and similarly for $\delta_y^2 u_{j,l}^n$. This is certainly a viable scheme; the problem arises in solving the coupled linear equations. Whereas in one space dimension the system was tridiagonal, that is no longer true, though the matrix is still very sparse. One possibility is to use a suitable sparse matrix technique (see §2.7 and §19.0).

Another possibility, which we generally prefer, is a slightly different way of generalizing the Crank-Nicolson algorithm. It is still second-order accurate in time and space, and unconditionally stable, but the equations are easier to solve than

(19.3.13). Called the *alternating-direction implicit method (ADI)*, this embodies the powerful concept of *operator splitting* or *time splitting*, about which we will say more below. Here, the idea is to divide each timestep into two steps of size $\Delta t/2$. In each substep, a different dimension is treated implicitly:

$$
\begin{aligned}
u_{j,l}^{n+1/2} &= u_{j,l}^{n} + \frac{1}{2}\alpha\left(\delta_x^2 u_{j,l}^{n+1/2} + \delta_y^2 u_{j,l}^{n}\right)\\
u_{j,l}^{n+1} &= u_{j,l}^{n+1/2} + \frac{1}{2}\alpha\left(\delta_x^2 u_{j,l}^{n+1/2} + \delta_y^2 u_{j,l}^{n+1}\right)
\end{aligned}
\tag{19.3.16}
$$

The advantage of this method is that each substep requires only the solution of a simple tridiagonal system.

## Operator Splitting Methods Generally

The basic idea of operator splitting, which is also called *time splitting* or *the method of fractional steps*, is this: Suppose you have an initial value equation of the form

$$\frac{\partial u}{\partial t} = \mathcal{L}u \tag{19.3.17}$$

where $\mathcal{L}$ is some operator. While $\mathcal{L}$ is not necessarily linear, suppose that it can at least be written as a linear sum of $m$ pieces, which act additively on $u$,

$$\mathcal{L}u = \mathcal{L}_1 u + \mathcal{L}_2 u + \cdots + \mathcal{L}_m u \tag{19.3.18}$$

Finally, suppose that for *each* of the pieces, you already know a differencing scheme for updating the variable $u$ from timestep $n$ to timestep $n+1$, valid if that piece of the operator were the *only* one on the right-hand side. We will write these updatings symbolically as

$$
\begin{aligned}
u^{n+1} &= \mathcal{U}_1(u^n, \Delta t)\\
u^{n+1} &= \mathcal{U}_2(u^n, \Delta t)\\
&\cdots\\
u^{n+1} &= \mathcal{U}_m(u^n, \Delta t)
\end{aligned}
\tag{19.3.19}
$$

Now, one form of operator splitting would be to get from $n$ to $n+1$ by the following sequence of updatings:

$$
\begin{aligned}
u^{n+(1/m)} &= \mathcal{U}_1(u^n, \Delta t)\\
u^{n+(2/m)} &= \mathcal{U}_2(u^{n+(1/m)}, \Delta t)\\
&\cdots\\
u^{n+1} &= \mathcal{U}_m(u^{n+(m-1)/m}, \Delta t)
\end{aligned}
\tag{19.3.20}
$$

For example, a combined advective-diffusion equation, such as

$$\frac{\partial u}{\partial t} = -v\frac{\partial u}{\partial x} + D\frac{\partial^2 u}{\partial x^2} \tag{19.3.21}$$

might profitably use an explicit scheme for the advective term combined with a Crank-Nicolson or other implicit scheme for the diffusion term.

The alternating-direction implicit (ADI) method, equation (19.3.16), is an example of operator splitting with a slightly different twist. Let us reinterpret (19.3.19) to have a different meaning: Let $\mathcal{U}_1$ now denote an updating method that includes algebraically *all* the pieces of the total operator $\mathcal{L}$, but which is desirably *stable* only for the $\mathcal{L}_1$ piece; likewise $\mathcal{U}_2, \dots \mathcal{U}_m$. Then a method of getting from $u^n$ to $u^{n+1}$ is

$$\begin{aligned} u^{n+1/m} &= \mathcal{U}_1(u^n, \Delta t/m) \\ u^{n+2/m} &= \mathcal{U}_2(u^{n+1/m}, \Delta t/m) \\ &\cdots \\ u^{n+1} &= \mathcal{U}_m(u^{n+(m-1)/m}, \Delta t/m) \end{aligned} \tag{19.3.22}$$

The timestep for each fractional step in (19.3.22) is now only $1/m$ of the full timestep, because each partial operation acts with all the terms of the original operator.

Equation (19.3.22) is usually, though not always, stable as a differencing scheme for the operator $\mathcal{L}$. In fact, as a rule of thumb, it is often sufficient to have stable $\mathcal{U}_i$'s only for the operator pieces having the highest number of spatial derivatives — the other $\mathcal{U}_i$'s can be *unstable* — to make the overall scheme stable!

It is at this point that we turn our attention from initial value problems to boundary value problems. These will occupy us for the remainder of the chapter.

CITED REFERENCES AND FURTHER READING:

Ames, W.F. 1977, *Numerical Methods for Partial Differential Equations*, 2nd ed. (New York: Academic Press).

# 19.4 Fourier and Cyclic Reduction Methods for Boundary Value Problems

As discussed in §19.0, most boundary value problems (elliptic equations, for example) reduce to solving large sparse linear systems of the form

$$\mathbf{A} \cdot \mathbf{u} = \mathbf{b} \tag{19.4.1}$$

either once, for boundary value equations that are linear, or iteratively, for boundary value equations that are nonlinear.

842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857

Two important techniques lead to "rapid" solution of equation (19.4.1) when the sparse matrix is of certain frequently occurring forms. The *Fourier transform method* is directly applicable when the equations have coefficients that are constant in space. The *cyclic reduction* method is somewhat more general; its applicability is related to the question of whether the equations are separable (in the sense of "separation of variables"). Both methods require the boundaries to coincide with the coordinate lines. Finally, for some problems, there is a powerful combination of these two methods called *FACR (Fourier Analysis and Cyclic Reduction).* We now consider each method in turn, using equation (19.0.3), with finite-difference representation (19.0.6), as a model example. Generally speaking, the methods in this section are faster, when they apply, than the simpler relaxation methods discussed in §19.5; but they are not necessarily faster than the more complicated multigrid methods discussed in §19.6.

## Fourier Transform Method

The discrete inverse Fourier transform in both $x$ and $y$ is

$$u_{jl} = \frac{1}{JL} \sum_{m=0}^{J-1} \sum_{n=0}^{L-1} \widehat{u}_{mn} e^{-2\pi i jm/J} e^{-2\pi i ln/L} \tag{19.4.2}$$

This can be computed using the FFT independently in each dimension, or else all at once via the routine `fourn` of §12.4 or the routine `rlft3` of §12.5. Similarly,

$$\rho_{jl} = \frac{1}{JL} \sum_{m=0}^{J-1} \sum_{n=0}^{L-1} \widehat{\rho}_{mn} e^{-2\pi i jm/J} e^{-2\pi i ln/L} \tag{19.4.3}$$

If we substitute expressions (19.4.2) and (19.4.3) in our model problem (19.0.6), we find

$$\widehat{u}_{mn} \left( e^{2\pi i m/J} + e^{-2\pi i m/J} + e^{2\pi i n/L} + e^{-2\pi i n/L} - 4 \right) = \widehat{\rho}_{mn} \Delta^2 \tag{19.4.4}$$

or

$$\widehat{u}_{mn} = \frac{\widehat{\rho}_{mn} \Delta^2}{2 \left( \cos \dfrac{2\pi m}{J} + \cos \dfrac{2\pi n}{L} - 2 \right)} \tag{19.4.5}$$

Thus the strategy for solving equation (19.0.6) by FFT techniques is:

- Compute $\widehat{\rho}_{mn}$ as the Fourier transform

$$\widehat{\rho}_{mn} = \sum_{j=0}^{J-1} \sum_{l=0}^{L-1} \rho_{jl}\, e^{2\pi i mj/J} e^{2\pi i nl/L} \tag{19.4.6}$$

- Compute $\widehat{u}_{mn}$ from equation (19.4.5).

843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858

• Compute $u_{jl}$ by the inverse Fourier transform (19.4.2).

The above procedure is valid for periodic boundary conditions. In other words, the solution satisfies

$$u_{jl} = u_{j+J,l} = u_{j,l+L} \tag{19.4.7}$$

Next consider a Dirichlet boundary condition $u = 0$ on the rectangular boundary. Instead of the expansion (19.4.2), we now need an expansion in sine waves:

$$u_{jl} = \frac{2}{J}\frac{2}{L}\sum_{m=1}^{J-1}\sum_{n=1}^{L-1}\widehat{u}_{mn}\sin\frac{\pi jm}{J}\sin\frac{\pi ln}{L} \tag{19.4.8}$$

This satisfies the boundary conditions that $u = 0$ at $j = 0, J$ and at $l = 0, L$. If we substitute this expansion and the analogous one for $\rho_{jl}$ into equation (19.0.6), we find that the solution procedure parallels that for periodic boundary conditions:

• Compute $\widehat{\rho}_{mn}$ by the sine transform

$$\widehat{\rho}_{mn} = \sum_{j=1}^{J-1}\sum_{l=1}^{L-1}\rho_{jl}\sin\frac{\pi jm}{J}\sin\frac{\pi ln}{L} \tag{19.4.9}$$

(A fast sine transform algorithm was given in §12.3.)

• Compute $\widehat{u}_{mn}$ from the expression analogous to (19.4.5),

$$\widehat{u}_{mn} = \frac{\Delta^2\widehat{\rho}_{mn}}{2\left(\cos\frac{\pi m}{J} + \cos\frac{\pi n}{L} - 2\right)} \tag{19.4.10}$$

• Compute $u_{jl}$ by the inverse sine transform (19.4.8).

If we have inhomogeneous boundary conditions, for example $u = 0$ on all boundaries except $u = f(y)$ on the boundary $x = J\Delta$, we have to add to the above solution a solution $u^H$ of the homogeneous equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{19.4.11}$$

that satisfies the required boundary conditions. In the continuum case, this would be an expression of the form

$$u^H = \sum_n A_n \sinh\frac{n\pi x}{L\Delta}\sin\frac{n\pi y}{L\Delta} \tag{19.4.12}$$

where $A_n$ would be found by requiring that $u = f(y)$ at $x = J\Delta$. In the discrete case, we have

$$u^H_{jl} = \frac{2}{L}\sum_{n=1}^{L-1} A_n \sinh\frac{\pi nj}{L}\sin\frac{\pi nl}{L} \tag{19.4.13}$$

845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860



If $f(y = l\Delta) \equiv f_l$, then we get $A_n$ from the inverse formula

$$A_n = \frac{1}{\sinh(\pi n J/L)} \sum_{l=1}^{L-1} f_l \sin\frac{\pi n l}{L} \tag{19.4.14}$$

The complete solution to the problem is

$$u = u_{jl} + u_{jl}^{H} \tag{19.4.15}$$

By adding appropriate terms of the form (19.4.12), we can handle inhomogeneous terms on any boundary surface.

A much simpler procedure for handling inhomogeneous terms is to note that whenever boundary terms appear on the left-hand side of (19.0.6), they can be taken over to the right-hand side since they are known. The effective source term is therefore $\rho_{jl}$ plus a contribution from the boundary terms. To implement this idea formally, write the solution as

$$u = u' + u^B \tag{19.4.16}$$

where $u' = 0$ on the boundary, while $u^B$ vanishes everywhere *except* on the boundary. There it takes on the given boundary value. In the above example, the only nonzero values of $u^B$ would be

$$u_{J,l}^B = f_l \tag{19.4.17}$$

The model equation (19.0.3) becomes

$$\nabla^2 u' = -\nabla^2 u^B + \rho \tag{19.4.18}$$

or, in finite-difference form,

$$\begin{aligned} &u'_{j+1,l} + u'_{j-1,l} + u'_{j,l+1} + u'_{j,l-1} - 4u'_{j,l} = \\ &\quad -\left(u^B_{j+1,l} + u^B_{j-1,l} + u^B_{j,l+1} + u^B_{j,l-1} - 4u^B_{j,l}\right) + \Delta^2 \rho_{j,l} \end{aligned} \tag{19.4.19}$$

All the $u^B$ terms in equation (19.4.19) vanish except when the equation is evaluated at $j = J - 1$, where

$$u'_{J,l} + u'_{J-2,l} + u'_{J-1,l+1} + u'_{J-1,l-1} - 4u'_{J-1,l} = -f_l + \Delta^2 \rho_{J-1,l} \tag{19.4.20}$$

Thus the problem is now equivalent to the case of zero boundary conditions, except that one row of the source term is modified by the replacement

$$\Delta^2 \rho_{J-1,l} \rightarrow \Delta^2 \rho_{J-1,l} - f_l \tag{19.4.21}$$

The case of Neumann boundary conditions $\nabla u = 0$ is handled by the cosine expansion (12.3.17):

$$u_{jl} = \frac{2}{J}\frac{2}{L} \sum_{m=0}^{J}{}'' \sum_{n=0}^{L}{}'' \widehat{u}_{mn} \cos\frac{\pi j m}{J} \cos\frac{\pi l n}{L} \tag{19.4.22}$$

Here the double prime notation means that the terms for $m = 0$ and $m = J$ should be multiplied by $\frac{1}{2}$, and similarly for $n = 0$ and $n = L$. Inhomogeneous terms $\nabla u = g$ can be again included by adding a suitable solution of the homogeneous equation, or more simply by taking boundary terms over to the right-hand side. For example, the condition

$$\frac{\partial u}{\partial x} = g(y) \qquad \text{at} \quad x = 0 \tag{19.4.23}$$

becomes

$$\frac{u_{1,l} - u_{-1,l}}{2\Delta} = g_l \tag{19.4.24}$$

where $g_l \equiv g(y = l\Delta)$. Once again we write the solution in the form (19.4.16), where now $\nabla u' = 0$ on the boundary. This time $\nabla u^B$ takes on the prescribed value on the boundary, but $u^B$ vanishes everywhere except just *outside* the boundary. Thus equation (19.4.24) gives

$$u^B_{-1,l} = -2\Delta g_l \tag{19.4.25}$$

All the $u^B$ terms in equation (19.4.19) vanish except when $j = 0$:

$$u'_{1,l} + u'_{-1,l} + u'_{0,l+1} + u'_{0,l-1} - 4u'_{0,l} = 2\Delta g_l + \Delta^2 \rho_{0,l} \tag{19.4.26}$$

Thus $u'$ is the solution of a zero-gradient problem, with the source term modified by the replacement

$$\Delta^2 \rho_{0,l} \rightarrow \Delta^2 \rho_{0,l} + 2\Delta g_l \tag{19.4.27}$$

Sometimes Neumann boundary conditions are handled by using a staggered grid, with the $u$'s defined midway between zone boundaries so that first derivatives are centered on the mesh points. You can solve such problems using similar techniques to those described above if you use the alternative form of the cosine transform, equation (12.3.23).

## Cyclic Reduction

Evidently the FFT method works only when the original PDE has constant coefficients, and boundaries that coincide with the coordinate lines. An alternative algorithm, which can be used on somewhat more general equations, is called *cyclic reduction (CR)*.

We illustrate cyclic reduction on the equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + b(y)\frac{\partial u}{\partial y} + c(y)u = g(x, y) \tag{19.4.28}$$

This form arises very often in practice from the Helmholtz or Poisson equations in polar, cylindrical, or spherical coordinate systems. More general separable equations are treated in [1].

The finite-difference form of equation (19.4.28) can be written as a set of vector equations

$$\mathbf{u}_{j-1} + \mathbf{T}\cdot\mathbf{u}_j + \mathbf{u}_{j+1} = \mathbf{g}_j\Delta^2 \tag{19.4.29}$$

Here the index $j$ comes from differencing in the $x$-direction, while the $y$-differencing (denoted by the index $l$ previously) has been left in vector form. The matrix $\mathbf{T}$ has the form

$$\mathbf{T} = \mathbf{B} - 2\mathbf{1} \tag{19.4.30}$$

where the $2\mathbf{1}$ comes from the $x$-differencing and the matrix $\mathbf{B}$ from the $y$-differencing. The matrix $\mathbf{B}$, and hence $\mathbf{T}$, is tridiagonal with variable coefficients.

The CR method is derived by writing down three successive equations like (19.4.29):

$$\begin{aligned}
\mathbf{u}_{j-2} + \mathbf{T}\cdot\mathbf{u}_{j-1} + \mathbf{u}_j &= \mathbf{g}_{j-1}\Delta^2\\
\mathbf{u}_{j-1} + \mathbf{T}\cdot\mathbf{u}_j + \mathbf{u}_{j+1} &= \mathbf{g}_j\Delta^2\\
\mathbf{u}_j + \mathbf{T}\cdot\mathbf{u}_{j+1} + \mathbf{u}_{j+2} &= \mathbf{g}_{j+1}\Delta^2
\end{aligned} \tag{19.4.31}$$

Matrix-multiplying the middle equation by $-\mathbf{T}$ and then adding the three equations, we get

$$\mathbf{u}_{j-2} + \mathbf{T}^{(1)}\cdot\mathbf{u}_j + \mathbf{u}_{j+2} = \mathbf{g}_j^{(1)}\Delta^2 \tag{19.4.32}$$

This is an equation of the same form as (19.4.29), with

$$\begin{aligned}
\mathbf{T}^{(1)} &= 2\mathbf{1} - \mathbf{T}^2\\
\mathbf{g}_j^{(1)} &= \Delta^2(\mathbf{g}_{j-1} - \mathbf{T}\cdot\mathbf{g}_j + \mathbf{g}_{j+1})
\end{aligned} \tag{19.4.33}$$

After one level of CR, we have reduced the number of equations by a factor of two. Since the resulting equations are of the same form as the original equation, we can repeat the process. Taking the number of mesh points to be a power of 2 for simplicity, we finally end up with a single equation for the central line of variables:

$$\mathbf{T}^{(f)}\cdot\mathbf{u}_{J/2} = \Delta^2\mathbf{g}_{J/2}^{(f)} - \mathbf{u}_0 - \mathbf{u}_J \tag{19.4.34}$$

Here we have moved $\mathbf{u}_0$ and $\mathbf{u}_J$ to the right-hand side because they are known boundary values. Equation (19.4.34) can be solved for $\mathbf{u}_{J/2}$ by the standard tridiagonal algorithm. The two equations at level $f-1$ involve $\mathbf{u}_{J/4}$ and $\mathbf{u}_{3J/4}$. The equation for $\mathbf{u}_{J/4}$ involves $\mathbf{u}_0$ and $\mathbf{u}_{J/2}$, both of which are known, and hence can be solved by the usual tridiagonal routine. A similar result holds true at every stage, so we end up solving $J-1$ tridiagonal systems.

In practice, equations (19.4.33) should be rewritten to avoid numerical instability. For these and other practical details, refer to [2].

## FACR Method

The *best* way to solve equations of the form (19.4.28), including the constant coefficient problem (19.0.3), is a combination of Fourier analysis and cyclic reduction, the FACR method [3-6]. If at the $r$th stage of CR we Fourier analyze the equations of the form (19.4.32) along $y$, that is, with respect to the suppressed vector index, we will have a tridiagonal system in the $x$-direction for each $y$-Fourier mode:

$$\widehat{u}^k_{j-2^r} + \lambda^{(r)}_k \widehat{u}^k_j + \widehat{u}^k_{j+2^r} = \Delta^2 g^{(r)k}_j \qquad (19.4.35)$$

Here $\lambda^{(r)}_k$ is the eigenvalue of $\mathbf{T}^{(r)}$ corresponding to the $k$th Fourier mode. For the equation (19.0.3), equation (19.4.5) shows that $\lambda^{(r)}_k$ will involve terms like $\cos(2\pi k/L) - 2$ raised to a power. Solve the tridiagonal systems for $\widehat{u}^k_j$ at the levels $j = 2^r, 2 \times 2^r, 4 \times 2^r, ..., J - 2^r$. Fourier synthesize to get the $y$-values on these $x$-lines. Then fill in the intermediate $x$-lines as in the original CR algorithm.

The trick is to choose the number of levels of CR so as to minimize the total number of arithmetic operations. One can show that for a typical case of a $128 \times 128$ mesh, the optimal level is $r = 2$; asymptotically, $r \to \log_2(\log_2 J)$.

A rough estimate of running times for these algorithms for equation (19.0.3) is as follows: The FFT method (in both $x$ and $y$) and the CR method are roughly comparable. FACR with $r = 0$ (that is, FFT in one dimension and solve the tridiagonal equations by the usual algorithm in the other dimension) gives about a factor of two gain in speed. The optimal FACR with $r = 2$ gives another factor of two gain in speed.

CITED REFERENCES AND FURTHER READING:

Swartzrauber, P.N. 1977, *SIAM Review*, vol. 19, pp. 490–501. [1]

Buzbee, B.L, Golub, G.H., and Nielson, C.W. 1970, *SIAM Journal on Numerical Analysis*, vol. 7, pp. 627–656; see also *op. cit.* vol. 11, pp. 753–763. [2]

Hockney, R.W. 1965, *Journal of the Association for Computing Machinery*, vol. 12, pp. 95–113. [3]

Hockney, R.W. 1970, in *Methods of Computational Physics*, vol. 9 (New York: Academic Press), pp. 135–211. [4]

Hockney, R.W., and Eastwood, J.W. 1981, *Computer Simulation Using Particles* (New York: McGraw-Hill), Chapter 6. [5]

Temperton, C. 1980, *Journal of Computational Physics*, vol. 34, pp. 314–329. [6]

# 19.5 Relaxation Methods for Boundary Value Problems

As we mentioned in §19.0, relaxation methods involve splitting the sparse matrix that arises from finite differencing and then iterating until a solution is found.

There is another way of thinking about relaxation methods that is somewhat more physical. Suppose we wish to solve the elliptic equation

$$\mathcal{L}u = \rho \qquad (19.5.1)$$

where $\mathcal{L}$ represents some elliptic operator and $\rho$ is the source term. Rewrite the equation as a diffusion equation,

$$\frac{\partial u}{\partial t} = \mathcal{L}u - \rho \tag{19.5.2}$$

An initial distribution $u$ *relaxes* to an equilibrium solution as $t \to \infty$. This equilibrium has all time derivatives vanishing. Therefore it is the solution of the original elliptic problem (19.5.1). We see that all the machinery of §19.2, on diffusive initial value equations, can be brought to bear on the solution of boundary value problems by relaxation methods.

Let us apply this idea to our model problem (19.0.3). The diffusion equation is

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} - \rho \tag{19.5.3}$$

If we use FTCS differencing (cf. equation 19.2.4), we get

$$u^{n+1}_{j,l} = u^n_{j,l} + \frac{\Delta t}{\Delta^2}\left(u^n_{j+1,l} + u^n_{j-1,l} + u^n_{j,l+1} + u^n_{j,l-1} - 4u^n_{j,l}\right) - \rho_{j,l}\Delta t \tag{19.5.4}$$

Recall from (19.2.6) that FTCS differencing is stable in one spatial dimension only if $\Delta t/\Delta^2 \leq \frac{1}{2}$. In two dimensions this becomes $\Delta t/\Delta^2 \leq \frac{1}{4}$. Suppose we try to take the largest possible timestep, and set $\Delta t = \Delta^2/4$. Then equation (19.5.4) becomes

$$u^{n+1}_{j,l} = \frac{1}{4}\left(u^n_{j+1,l} + u^n_{j-1,l} + u^n_{j,l+1} + u^n_{j,l-1}\right) - \frac{\Delta^2}{4}\rho_{j,l} \tag{19.5.5}$$

Thus the algorithm consists of using the average of $u$ at its four nearest-neighbor points on the grid (plus the contribution from the source). This procedure is then iterated until convergence.

This method is in fact a classical method with origins dating back to the last century, called *Jacobi's method* (not to be confused with the Jacobi method for eigenvalues). The method is not practical because it converges too slowly. However, it is the basis for understanding the modern methods, which are always compared with it.

Another classical method is the *Gauss-Seidel* method, which turns out to be important in multigrid methods (§19.6). Here we make use of updated values of $u$ on the right-hand side of (19.5.5) as soon as they become available. In other words, the averaging is done "in place" instead of being "copied" from an earlier timestep to a later one. If we are proceeding along the rows, incrementing $j$ for fixed $l$, we have

$$u^{n+1}_{j,l} = \frac{1}{4}\left(u^n_{j+1,l} + u^{n+1}_{j-1,l} + u^n_{j,l+1} + u^{n+1}_{j,l-1}\right) - \frac{\Delta^2}{4}\rho_{j,l} \tag{19.5.6}$$

This method is also slowly converging and only of theoretical interest when used by itself, but some analysis of it will be instructive.

Let us look at the Jacobi and Gauss-Seidel methods in terms of the matrix splitting concept. We change notation and call **u** "**x**," to conform to standard matrix notation. To solve

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{19.5.7}$$

we can consider splitting $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U} \tag{19.5.8}$$

where $\mathbf{D}$ is the diagonal part of $\mathbf{A}$, $\mathbf{L}$ is the lower triangle of $\mathbf{A}$ with zeros on the diagonal, and $\mathbf{U}$ is the upper triangle of $\mathbf{A}$ with zeros on the diagonal.

In the Jacobi method we write for the $r$th step of iteration

$$\mathbf{D} \cdot \mathbf{x}^{(r)} = -(\mathbf{L} + \mathbf{U}) \cdot \mathbf{x}^{(r-1)} + \mathbf{b} \tag{19.5.9}$$

For our model problem (19.5.5), $\mathbf{D}$ is simply the identity matrix. The Jacobi method converges for matrices $\mathbf{A}$ that are "diagonally dominant" in a sense that can be made mathematically precise. For matrices arising from finite differencing, this condition is usually met.

What is the rate of convergence of the Jacobi method? A detailed analysis is beyond our scope, but here is some of the flavor: The matrix $-\mathbf{D}^{-1} \cdot (\mathbf{L} + \mathbf{U})$ is the *iteration matrix* which, apart from an additive term, maps one set of $\mathbf{x}$'s into the next. The iteration matrix has eigenvalues, each one of which reflects the factor by which the amplitude of a particular eigenmode of undesired residual is suppressed during one iteration. Evidently those factors had better all have modulus $< 1$ for the relaxation to work at all! The rate of convergence of the method is set by the rate for the slowest-decaying eigenmode, i.e., the factor with largest modulus. The modulus of this largest factor, therefore lying between 0 and 1, is called the *spectral radius* of the relaxation operator, denoted $\rho_s$.

The number of iterations $r$ required to reduce the overall error by a factor $10^{-p}$ is thus estimated by

$$r \approx \frac{p \ln 10}{(-\ln \rho_s)} \tag{19.5.10}$$

In general, the spectral radius $\rho_s$ goes asymptotically to the value 1 as the grid size $J$ is increased, so that more iterations are required. For any given equation, grid geometry, *and boundary condition*, the spectral radius can, in principle, be computed analytically. For example, for equation (19.5.5) on a $J \times J$ grid with Dirichlet boundary conditions on all four sides, the asymptotic formula for large $J$ turns out to be

$$\rho_s \simeq 1 - \frac{\pi^2}{2J^2} \tag{19.5.11}$$

The number of iterations $r$ required to reduce the error by a factor of $10^{-p}$ is thus

$$r \simeq \frac{2pJ^2 \ln 10}{\pi^2} \simeq \frac{1}{2} p J^2 \tag{19.5.12}$$

In other words, the number of iterations is proportional to the number of mesh points, $J^2$. Since $100 \times 100$ and larger problems are common, it is clear that the Jacobi method is only of theoretical interest.

The Gauss-Seidel method, equation (19.5.6), corresponds to the matrix decomposition

$$(\mathbf{L}+\mathbf{D})\cdot\mathbf{x}^{(r)} = -\mathbf{U}\cdot\mathbf{x}^{(r-1)}+\mathbf{b} \tag{19.5.13}$$

The fact that $\mathbf{L}$ is on the left-hand side of the equation follows from the updating in place, as you can easily check if you write out (19.5.13) in components. One can show [1-3] that the spectral radius is just the square of the spectral radius of the Jacobi method. For our model problem, therefore,

$$\rho_s \simeq 1-\frac{\pi^2}{J^2} \tag{19.5.14}$$

$$r \simeq \frac{pJ^2 \ln 10}{\pi^2} \simeq \frac{1}{4}pJ^2 \tag{19.5.15}$$

The factor of two improvement in the number of iterations over the Jacobi method still leaves the method impractical.

## Successive Overrelaxation (SOR)

We get a better algorithm — one that was the standard algorithm until the 1970s — if we make an *overcorrection* to the value of $\mathbf{x}^{(r)}$ at the $r$th stage of Gauss-Seidel iteration, thus anticipating future corrections. Solve (19.5.13) for $\mathbf{x}^{(r)}$, add and subtract $\mathbf{x}^{(r-1)}$ on the right-hand side, and hence write the Gauss-Seidel method as

$$\mathbf{x}^{(r)} = \mathbf{x}^{(r-1)} - (\mathbf{L}+\mathbf{D})^{-1}\cdot[(\mathbf{L}+\mathbf{D}+\mathbf{U})\cdot\mathbf{x}^{(r-1)}-\mathbf{b}] \tag{19.5.16}$$

The term in square brackets is just the residual vector $\xi^{(r-1)}$, so

$$\mathbf{x}^{(r)} = \mathbf{x}^{(r-1)} - (\mathbf{L}+\mathbf{D})^{-1}\cdot\xi^{(r-1)} \tag{19.5.17}$$

Now *overcorrect*, defining

$$\mathbf{x}^{(r)} = \mathbf{x}^{(r-1)} - \omega(\mathbf{L}+\mathbf{D})^{-1}\cdot\xi^{(r-1)} \tag{19.5.18}$$

Here $\omega$ is called the *overrelaxation parameter*, and the method is called *successive overrelaxation* (SOR).

The following theorems can be proved [1-3]:

- The method is convergent only for $0<\omega<2$. If $0<\omega<1$, we speak of *underrelaxation*.
- Under certain mathematical restrictions generally satisfied by matrices arising from finite differencing, only overrelaxation ($1<\omega<2$) can give faster convergence than the Gauss-Seidel method.
- If $\rho_{\text{Jacobi}}$ is the spectral radius of the Jacobi iteration (so that the square of it is the spectral radius of the Gauss-Seidel iteration), then the *optimal* choice for $\omega$ is given by

$$\omega = \frac{2}{1+\sqrt{1-\rho_{\text{Jacobi}}^2}} \tag{19.5.19}$$

- For this optimal choice, the spectral radius for SOR is

$$\rho_{\text{SOR}} = \left( \frac{\rho_{\text{Jacobi}}}{1 + \sqrt{1 - \rho_{\text{Jacobi}}^2}} \right)^2 \tag{19.5.20}$$

As an application of the above results, consider our model problem for which $\rho_{\text{Jacobi}}$ is given by equation (19.5.11). Then equations (19.5.19) and (19.5.20) give

$$\omega \simeq \frac{2}{1 + \pi/J} \tag{19.5.21}$$

$$\rho_{\text{SOR}} \simeq 1 - \frac{2\pi}{J} \qquad \text{for large} \quad J \tag{19.5.22}$$

Equation (19.5.10) gives for the number of iterations to reduce the initial error by a factor of $10^{-p}$,

$$r \simeq \frac{pJ \ln 10}{2\pi} \simeq \frac{1}{3} pJ \tag{19.5.23}$$

Comparing with equation (19.5.12) or (19.5.15), we see that optimal SOR requires of order $J$ iterations, as opposed to of order $J^2$. Since $J$ is typically 100 or larger, this makes a tremendous difference! Equation (19.5.23) leads to the mnemonic that 3-figure accuracy ($p = 3$) requires a number of iterations equal to the number of mesh points along a side of the grid. For 6-figure accuracy, we require about twice as many iterations.

How do we choose $\omega$ for a problem for which the answer is not known analytically? That is just the weak point of SOR! The advantages of SOR obtain only in a fairly narrow window around the correct value of $\omega$. It is better to take $\omega$ slightly too large, rather than slightly too small, but best to get it right.

One way to choose $\omega$ is to map your problem approximately onto a known problem, replacing the coefficients in the equation by average values. Note, however, that the known problem must have the same grid size and boundary conditions as the actual problem. We give for reference purposes the value of $\rho_{\text{Jacobi}}$ for our model problem on a rectangular $J \times L$ grid, allowing for the possibility that $\Delta x \neq \Delta y$:

$$\rho_{\text{Jacobi}} = \frac{\cos \dfrac{\pi}{J} + \left( \dfrac{\Delta x}{\Delta y} \right)^2 \cos \dfrac{\pi}{L}}{1 + \left( \dfrac{\Delta x}{\Delta y} \right)^2} \tag{19.5.24}$$

Equation (19.5.24) holds for homogeneous Dirichlet or Neumann boundary conditions. For periodic boundary conditions, make the replacement $\pi \to 2\pi$.

A second way, which is especially useful if you plan to solve many similar elliptic equations each time with slightly different coefficients, is to determine the optimum value $\omega$ empirically on the first equation and then use that value for the remaining equations. Various automated schemes for doing this and for "seeking out" the best values of $\omega$ are described in the literature.

While the matrix notation introduced earlier is useful for theoretical analyses, for practical implementation of the SOR algorithm we need explicit formulas.

Consider a general second-order elliptic equation in $x$ and $y$, finite differenced on a square as for our model equation. Corresponding to each row of the matrix $\mathbf{A}$ is an equation of the form

$$a_{j,l}u_{j+1,l} + b_{j,l}u_{j-1,l} + c_{j,l}u_{j,l+1} + d_{j,l}u_{j,l-1} + e_{j,l}u_{j,l} = f_{j,l} \qquad (19.5.25)$$

For our model equation, we had $a = b = c = d = 1$, $e = -4$. The quantity $f$ is proportional to the source term. The iterative procedure is defined by solving (19.5.25) for $u_{j,l}$:

$$u^*{}_{j,l} = \frac{1}{e_{j,l}}\left(f_{j,l} - a_{j,l}u_{j+1,l} - b_{j,l}u_{j-1,l} - c_{j,l}u_{j,l+1} - d_{j,l}u_{j,l-1}\right) \qquad (19.5.26)$$

Then $u_{j,l}^{\text{new}}$ is a weighted average

$$u_{j,l}^{\text{new}} = \omega u^*{}_{j,l} + (1-\omega)u_{j,l}^{\text{old}} \qquad (19.5.27)$$

We calculate it as follows: The residual at any stage is

$$\xi_{j,l} = a_{j,l}u_{j+1,l} + b_{j,l}u_{j-1,l} + c_{j,l}u_{j,l+1} + d_{j,l}u_{j,l-1} + e_{j,l}u_{j,l} - f_{j,l} \qquad (19.5.28)$$

and the SOR algorithm (19.5.18) or (19.5.27) is

$$u_{j,l}^{\text{new}} = u_{j,l}^{\text{old}} - \omega\frac{\xi_{j,l}}{e_{j,l}} \qquad (19.5.29)$$

This formulation is very easy to program, and the norm of the residual vector $\xi_{j,l}$ can be used as a criterion for terminating the iteration.

Another practical point concerns the order in which mesh points are processed. The obvious strategy is simply to proceed in order down the rows (or columns). Alternatively, suppose we divide the mesh into "odd" and "even" meshes, like the red and black squares of a checkerboard. Then equation (19.5.26) shows that the odd points depend only on the even mesh values and vice versa. Accordingly, we can carry out one half-sweep updating the odd points, say, and then another half-sweep updating the even points with the new odd values. For the version of SOR implemented below, we shall adopt odd-even ordering.

The last practical point is that in practice the asymptotic rate of convergence in SOR is not attained until of order $J$ iterations. The error often grows by a factor of 20 before convergence sets in. A trivial modification to SOR resolves this problem. It is based on the observation that, while $\omega$ is the optimum *asymptotic* relaxation parameter, it is not necessarily a good initial choice. In SOR with *Chebyshev acceleration*, one uses odd-even ordering and changes $\omega$ at each half-sweep according to the following prescription:

$$\begin{aligned}
\omega^{(0)} &= 1 \\
\omega^{(1/2)} &= 1/(1-\rho_{\text{Jacobi}}^2/2) \\
\omega^{(n+1/2)} &= 1/(1-\rho_{\text{Jacobi}}^2\omega^{(n)}/4), \qquad n = 1/2, 1, \ldots, \infty \\
\omega^{(\infty)} &\to \omega_{\text{optimal}}
\end{aligned} \qquad (19.5.30)$$

The beauty of Chebyshev acceleration is that the norm of the error always decreases with each iteration. (This is the norm of the actual error in $u_{j,l}$. The norm of the residual $\xi_{j,l}$ need not decrease monotonically.) While the asymptotic rate of convergence is the same as ordinary SOR, there is never any excuse for not using Chebyshev acceleration to reduce the total number of iterations required.

Here we give a routine for SOR with Chebyshev acceleration.

```
#include <math.h>
#define MAXITS 1000
#define EPS 1.0e-5

void sor(double **a, double **b, double **c, double **d, double **e,
    double **f, double **u, int jmax, double rjac)
Successive overrelaxation solution of equation (19.5.25) with Chebyshev acceleration. a, b, c,
d, e, and f are input as the coefficients of the equation, each dimensioned to the grid size
[1..jmax][1..jmax]. u is input as the initial guess to the solution, usually zero, and returns
with the final value. rjac is input as the spectral radius of the Jacobi iteration, or an estimate
of it.
{
    void nrerror(char error_text[]);
    int ipass,j,jsw,l,lsw,n;
    double anorm,anormf=0.0,omega=1.0,resid;
    Double precision is a good idea for jmax bigger than about 25.

    for (j=2;j<jmax;j++)
    Compute initial norm of residual and terminate iteration when norm has been reduced by
    a factor EPS.
        for (l=2;l<jmax;l++)
            anormf += fabs(f[j][l]);            Assumes initial u is zero.
    for (n=1;n<=MAXITS;n++) {
        anorm=0.0;
        jsw=1;
        for (ipass=1;ipass<=2;ipass++) {       Odd-even ordering.
            lsw=jsw;
            for (j=2;j<jmax;j++) {
                for (l=lsw+1;l<jmax;l+=2) {
                    resid=a[j][l]*u[j+1][l]
                        +b[j][l]*u[j-1][l]
                        +c[j][l]*u[j][l+1]
                        +d[j][l]*u[j][l-1]
                        +e[j][l]*u[j][l]
                        -f[j][l];
                    anorm += fabs(resid);
                    u[j][l] -= omega*resid/e[j][l];
                }
                lsw=3-lsw;
            }
            jsw=3-jsw;
            omega=(n == 1 && ipass == 1 ? 1.0/(1.0-0.5*rjac*rjac) :
                1.0/(1.0-0.25*rjac*rjac*omega));
        }
        if (anorm < EPS*anormf) return;
    }
    nrerror("MAXITS exceeded");
}
```

The main advantage of SOR is that it is very easy to program. Its main disadvantage is that it is still very inefficient on large problems.

## ADI (Alternating-Direction Implicit) Method

The ADI method of §19.3 for diffusion equations can be turned into a relaxation method for elliptic equations [1-4]. In §19.3, we discussed ADI as a method for solving the time-dependent heat-flow equation

$$\frac{\partial u}{\partial t} = \nabla^2 u - \rho \tag{19.5.31}$$

By letting $t \to \infty$ one also gets an iterative method for solving the elliptic equation

$$\nabla^2 u = \rho \tag{19.5.32}$$

In either case, the operator splitting is of the form

$$\mathcal{L} = \mathcal{L}_x + \mathcal{L}_y \tag{19.5.33}$$

where $\mathcal{L}_x$ represents the differencing in $x$ and $\mathcal{L}_y$ that in $y$.

For example, in our model problem (19.0.6) with $\Delta x = \Delta y = \Delta$, we have

$$\begin{aligned} \mathcal{L}_x u &= 2u_{j,l} - u_{j+1,l} - u_{j-1,l} \\ \mathcal{L}_y u &= 2u_{j,l} - u_{j,l+1} - u_{j,l-1} \end{aligned} \tag{19.5.34}$$

More complicated operators may be similarly split, but there is some art involved. A bad choice of splitting can lead to an algorithm that fails to converge. Usually one tries to base the splitting on the physical nature of the problem. We know for our model problem that an initial transient diffuses away, and we set up the $x$ and $y$ splitting to mimic diffusion in each dimension.

Having chosen a splitting, we difference the time-dependent equation (19.5.31) implicitly in two half-steps:

$$\begin{aligned} \frac{u^{n+1/2} - u^n}{\Delta t/2} &= -\frac{\mathcal{L}_x u^{n+1/2} + \mathcal{L}_y u^n}{\Delta^2} - \rho \\ \frac{u^{n+1} - u^{n+1/2}}{\Delta t/2} &= -\frac{\mathcal{L}_x u^{n+1/2} + \mathcal{L}_y u^{n+1}}{\Delta^2} - \rho \end{aligned} \tag{19.5.35}$$

(cf. equation 19.3.16). Here we have suppressed the spatial indices $(j, l)$. In matrix notation, equations (19.5.35) are

$$(\mathbf{L}_x + r\mathbf{1}) \cdot \mathbf{u}^{n+1/2} = (r\mathbf{1} - \mathbf{L}_y) \cdot \mathbf{u}^n - \Delta^2 \rho \tag{19.5.36}$$

$$(\mathbf{L}_y + r\mathbf{1}) \cdot \mathbf{u}^{n+1} = (r\mathbf{1} - \mathbf{L}_x) \cdot \mathbf{u}^{n+1/2} - \Delta^2 \rho \tag{19.5.37}$$

where

$$r \equiv \frac{2\Delta^2}{\Delta t} \tag{19.5.38}$$

The matrices on the left-hand sides of equations (19.5.36) and (19.5.37) are tridiagonal (and usually positive definite), so the equations can be solved by the

standard tridiagonal algorithm. Given $\mathbf{u}^n$, one solves (19.5.36) for $\mathbf{u}^{n+1/2}$, substitutes on the right-hand side of (19.5.37), and then solves for $\mathbf{u}^{n+1}$. The key question is how to choose the iteration parameter $r$, the analog of a choice of timestep for an initial value problem.

As usual, the goal is to minimize the spectral radius of the iteration matrix. Although it is beyond our scope to go into details here, it turns out that, for the optimal choice of $r$, the ADI method has the same rate of convergence as SOR. The individual iteration steps in the ADI method are much more complicated than in SOR, so the ADI method would appear to be inferior. This is in fact true if we choose the same parameter $r$ for every iteration step. However, it is possible to choose a *different* $r$ for each step. If this is done optimally, then ADI is generally more efficient than SOR. We refer you to the literature [1-4] for details.

Our reason for not fully implementing ADI here is that, in most applications, it has been superseded by the multigrid methods described in the next section. Our advice is to use SOR for trivial problems (e.g., $20 \times 20$), or for solving a larger problem once only, where ease of programming outweighs expense of computer time. Occasionally, the sparse matrix methods of §2.7 are useful for solving a set of difference equations directly. For production solution of large elliptic problems, however, multigrid is now almost always the method of choice.

CITED REFERENCES AND FURTHER READING:

Hockney, R.W., and Eastwood, J.W. 1981, *Computer Simulation Using Particles* (New York: McGraw-Hill), Chapter 6.

Young, D.M. 1971, *Iterative Solution of Large Linear Systems* (New York: Academic Press). [1]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §§8.3–8.6. [2]

Varga, R.S. 1962, *Matrix Iterative Analysis* (Englewood Cliffs, NJ: Prentice-Hall). [3]

Spanier, J. 1967, in *Mathematical Methods for Digital Computers, Volume 2* (New York: Wiley), Chapter 11. [4]

## 19.6 Multigrid Methods for Boundary Value Problems

Practical multigrid methods were first introduced in the 1970s by Brandt. These methods can solve elliptic PDEs discretized on $N$ grid points in $O(N)$ operations. The "rapid" direct elliptic solvers discussed in §19.4 solve special kinds of elliptic equations in $O(N \log N)$ operations. The numerical coefficients in these estimates are such that multigrid methods are comparable to the rapid methods in execution speed. Unlike the rapid methods, however, the multigrid methods can solve general elliptic equations with nonconstant coefficients with hardly any loss in efficiency. Even nonlinear equations can be solved with comparable speed.

Unfortunately there is not a single multigrid algorithm that solves all elliptic problems. Rather there is a multigrid technique that provides the framework for solving these problems. You have to adjust the various components of the algorithm within this framework to solve your specific problem. We can only give a brief

introduction to the subject here. In particular, we will give two sample multigrid routines, one linear and one nonlinear. By following these prototypes and by perusing the references [1-4], you should be able to develop routines to solve your own problems.

There are two related, but distinct, approaches to the use of multigrid techniques. The first, termed "the multigrid method," is a means for speeding up the convergence of a traditional relaxation method, as defined by you on a grid of pre-specified fineness. In this case, you need define your problem (e.g., evaluate its source terms) only on this grid. Other, coarser, grids defined by the method can be viewed as temporary computational adjuncts.

The second approach, termed (perhaps confusingly) "the full multigrid (FMG) method," requires you to be able to define your problem on grids of various sizes (generally by discretizing the same underlying PDE into different-sized sets of finite-difference equations). In this approach, the method obtains successive solutions on finer and finer grids. You can stop the solution either at a pre-specified fineness, or you can monitor the truncation error due to the discretization, quitting only when it is tolerably small.

In this section we will first discuss the "multigrid method," then use the concepts developed to introduce the FMG method. The latter algorithm is the one that we implement in the accompanying programs.

## From One-Grid, through Two-Grid, to Multigrid

The key idea of the multigrid method can be understood by considering the simplest case of a two-grid method. Suppose we are trying to solve the linear elliptic problem

$$\mathcal{L}u = f \tag{19.6.1}$$

where $\mathcal{L}$ is some linear elliptic operator and $f$ is the source term. Discretize equation (19.6.1) on a uniform grid with mesh size $h$. Write the resulting set of linear algebraic equations as

$$\mathcal{L}_h u_h = f_h \tag{19.6.2}$$

Let $\widetilde{u}_h$ denote some approximate solution to equation (19.6.2). We will use the symbol $u_h$ to denote the exact solution to the difference equations (19.6.2). Then the *error* in $\widetilde{u}_h$ or the *correction* is

$$v_h = u_h - \widetilde{u}_h \tag{19.6.3}$$

The *residual* or *defect* is

$$d_h = \mathcal{L}_h \widetilde{u}_h - f_h \tag{19.6.4}$$

(Beware: some authors define residual as minus the defect, and there is not universal agreement about which of these two quantities 19.6.4 defines.) Since $\mathcal{L}_h$ is linear, the error satisfies

$$\mathcal{L}_h v_h = -d_h \tag{19.6.5}$$

At this point we need to make an approximation to $\mathcal{L}_h$ in order to find $v_h$. The classical iteration methods, such as Jacobi or Gauss-Seidel, do this by finding, at each stage, an approximate solution of the equation

$$\widehat{\mathcal{L}}_h \widehat{v}_h = -d_h \tag{19.6.6}$$

where $\widehat{\mathcal{L}}_h$ is a "simpler" operator than $\mathcal{L}_h$. For example, $\widehat{\mathcal{L}}_h$ is the diagonal part of $\mathcal{L}_h$ for Jacobi iteration, or the lower triangle for Gauss-Seidel iteration. The next approximation is generated by

$$\widetilde{u}_h^{\text{new}} = \widetilde{u}_h + \widehat{v}_h \tag{19.6.7}$$

Now consider, as an alternative, a completely different type of approximation for $\mathcal{L}_h$, one in which we "coarsify" rather than "simplify." That is, we form some appropriate approximation $\mathcal{L}_H$ of $\mathcal{L}_h$ on a coarser grid with mesh size $H$ (we will always take $H = 2h$, but other choices are possible). The residual equation (19.6.5) is now approximated by

$$\mathcal{L}_H v_H = -d_H \tag{19.6.8}$$

Since $\mathcal{L}_H$ has smaller dimension, this equation will be easier to solve than equation (19.6.5). To define the defect $d_H$ on the coarse grid, we need a *restriction operator* $\mathcal{R}$ that restricts $d_h$ to the coarse grid:

$$d_H = \mathcal{R} d_h \tag{19.6.9}$$

The restriction operator is also called the *fine-to-coarse operator* or the *injection operator*. Once we have a solution $\widetilde{v}_H$ to equation (19.6.8), we need a *prolongation operator* $\mathcal{P}$ that prolongates or interpolates the correction to the fine grid:

$$\widetilde{v}_h = \mathcal{P} \widetilde{v}_H \tag{19.6.10}$$

The prolongation operator is also called the *coarse-to-fine operator* or the *interpolation operator*. Both $\mathcal{R}$ and $\mathcal{P}$ are chosen to be linear operators. Finally the approximation $\widetilde{u}_h$ can be updated:

$$\widetilde{u}_h^{\text{new}} = \widetilde{u}_h + \widetilde{v}_h \tag{19.6.11}$$

One step of this *coarse-grid correction scheme* is thus:

*Coarse-Grid Correction*

- Compute the defect on the fine grid from (19.6.4).
- Restrict the defect by (19.6.9).
- Solve (19.6.8) exactly on the coarse grid for the correction.
- Interpolate the correction to the fine grid by (19.6.10).

- Compute the next approximation by (19.6.11).

Let's contrast the advantages and disadvantages of relaxation and the coarse-grid correction scheme. Consider the error $v_h$ expanded into a discrete Fourier series. Call the components in the lower half of the frequency spectrum the *smooth components* and the high-frequency components the *nonsmooth components*. We have seen that relaxation becomes very slowly convergent in the limit $h \to 0$, i.e., when there are a large number of mesh points. The reason turns out to be that the smooth components are only slightly reduced in amplitude on each iteration. However, many relaxation methods reduce the amplitude of the nonsmooth components by large factors on each iteration: They are good *smoothing operators*.

For the two-grid iteration, on the other hand, components of the error with wavelengths $\lesssim 2H$ are not even representable on the coarse grid and so cannot be reduced to zero on this grid. But it is exactly these high-frequency components that can be reduced by relaxation on the fine grid! This leads us to combine the ideas of relaxation and coarse-grid correction:

*Two-Grid Iteration*

- Pre-smoothing: Compute $\bar{u}_h$ by applying $\nu_1 \geq 0$ steps of a relaxation method to $\tilde{u}_h$.
- Coarse-grid correction: As above, using $\bar{u}_h$ to give $\bar{u}_h^{\text{new}}$.
- Post-smoothing: Compute $\tilde{u}_h^{\text{new}}$ by applying $\nu_2 \geq 0$ steps of the relaxation method to $\bar{u}_h^{\text{new}}$.

It is only a short step from the above two-grid method to a multigrid method. Instead of solving the coarse-grid defect equation (19.6.8) exactly, we can get an approximate solution of it by introducing an even coarser grid and using the two-grid iteration method. If the convergence factor of the two-grid method is small enough, we will need only a few steps of this iteration to get a good enough approximate solution. We denote the number of such iterations by $\gamma$. Obviously we can apply this idea recursively down to some coarsest grid. There the solution is found easily, for example by direct matrix inversion or by iterating the relaxation scheme to convergence.

One iteration of a multigrid method, from finest grid to coarser grids and back to finest grid again, is called a *cycle*. The exact structure of a cycle depends on the value of $\gamma$, the number of two-grid iterations at each intermediate stage. The case $\gamma = 1$ is called a V-cycle, while $\gamma = 2$ is called a W-cycle (see Figure 19.6.1). These are the most important cases in practice.

Note that once more than two grids are involved, the pre-smoothing steps after the first one on the finest grid need an initial approximation for the error $v$. This should be taken to be zero.

## Smoothing, Restriction, and Prolongation Operators

The most popular smoothing method, and the one you should try first, is Gauss-Seidel, since it usually leads to a good convergence rate. If we order the mesh points from 1 to $N$, then the Gauss-Seidel scheme is

$$u_i = -\Big(\sum_{\substack{j=1 \\ j\neq i}}^{N} L_{ij} u_j - f_i\Big)\frac{1}{L_{ii}} \qquad i = 1, \ldots, N \tag{19.6.12}$$

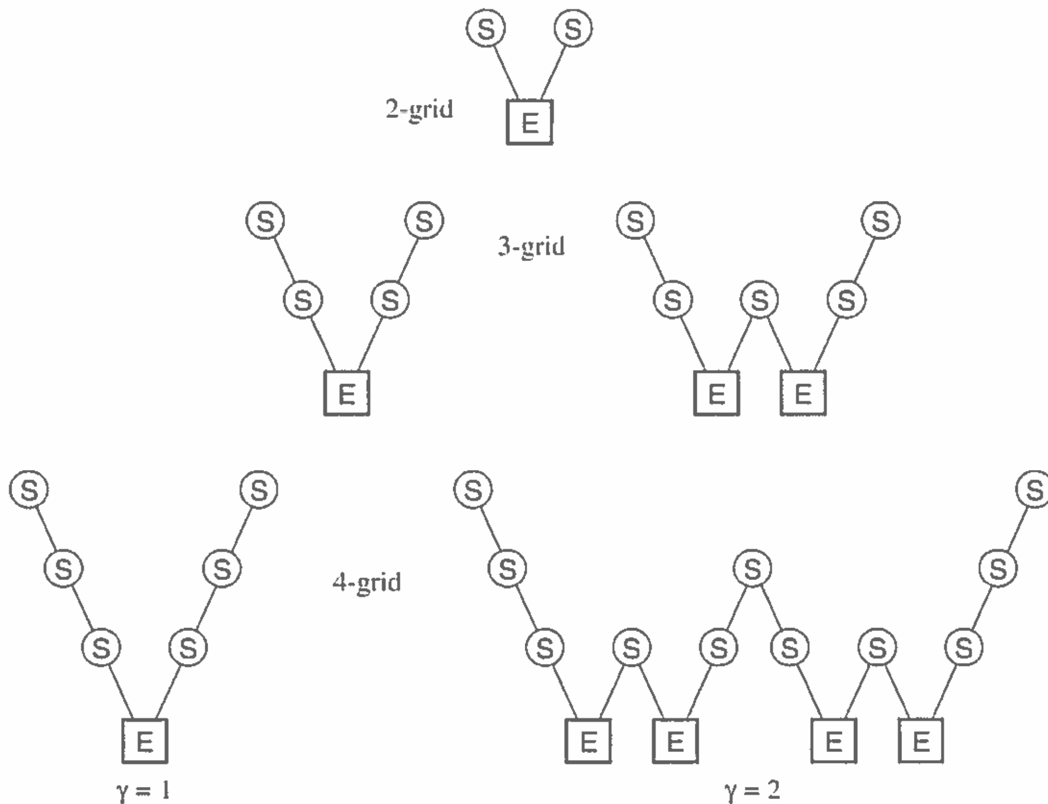



Figure 19.6.1. Structure of multigrid cycles. S denotes smoothing, while E denotes exact solution on the coarsest grid. Each descending line \ denotes restriction ($\mathcal{R}$) and each ascending line / denotes prolongation ($\mathcal{P}$). The finest grid is at the top level of each diagram. For the V-cycles ($\gamma = 1$) the E step is replaced by one 2-grid iteration each time the number of grid levels is increased by one. For the W-cycles ($\gamma = 2$), each E step gets replaced by two 2-grid iterations.

where new values of $u$ are used on the right-hand side as they become available. The exact form of the Gauss-Seidel method depends on the ordering chosen for the mesh points. For typical second-order elliptic equations like our model problem equation (19.0.3), as differenced in equation (19.0.8), it is usually best to use red-black ordering, making one pass through the mesh updating the "even" points (like the red squares of a checkerboard) and another pass updating the "odd" points (the black squares). When quantities are more strongly coupled along one dimension than another, one should relax a whole line along that dimension simultaneously. Line relaxation for nearest-neighbor coupling involves solving a tridiagonal system, and so is still efficient. Relaxing odd and even lines on successive passes is called zebra relaxation and is usually preferred over simple line relaxation.

Note that SOR should *not* be used as a smoothing operator. The overrelaxation destroys the high-frequency smoothing that is so crucial for the multigrid method.

A succint notation for the prolongation and restriction operators is to give their *symbol*. The symbol of $\mathcal{P}$ is found by considering $v_H$ to be 1 at some mesh point $(x, y)$, zero elsewhere, and then asking for the values of $\mathcal{P}v_H$. The most popular prolongation operator is simple bilinear interpolation. It gives nonzero values at the 9 points $(x, y)$, $(x + h, y), \ldots, (x - h, y - h)$, where the values are $1, \frac{1}{2}, \ldots, \frac{1}{4}$.

Its symbol is therefore

$$\begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix} \tag{19.6.13}$$

The symbol of $\mathcal{R}$ is defined by considering $v_h$ to be defined everywhere on the fine grid, and then asking what is $\mathcal{R}v_h$ at $(x, y)$ as a linear combination of these values. The simplest possible choice for $\mathcal{R}$ is *straight injection*, which means simply filling each coarse-grid point with the value from the corresponding fine-grid point. Its symbol is "[1]." However, difficulties can arise in practice with this choice. It turns out that a safe choice for $\mathcal{R}$ is to make it the adjoint operator to $\mathcal{P}$. To define the adjoint, define the scalar product of two grid functions $u_h$ and $v_h$ for mesh size $h$ as

$$\langle u_h | v_h \rangle_h \equiv h^2 \sum_{x,y} u_h(x, y) v_h(x, y) \tag{19.6.14}$$

Then the adjoint of $\mathcal{P}$, denoted $\mathcal{P}^\dagger$, is defined by

$$\langle u_H | \mathcal{P}^\dagger v_h \rangle_H = \langle \mathcal{P} u_H | v_h \rangle_h \tag{19.6.15}$$

Now take $\mathcal{P}$ to be bilinear interpolation, and choose $u_H = 1$ at $(x, y)$, zero elsewhere. Set $\mathcal{P}^\dagger = \mathcal{R}$ in (19.6.15) and $H = 2h$. You will find that

$$(\mathcal{R}v_h)_{(x,y)} = \tfrac{1}{4} v_h(x, y) + \tfrac{1}{8} v_h(x + h, y) + \tfrac{1}{16} v_h(x + h, y + h) + \cdots \tag{19.6.16}$$

so that the symbol of $\mathcal{R}$ is

$$\begin{bmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{bmatrix} \tag{19.6.17}$$

Note the simple rule: The symbol of $\mathcal{R}$ is $\frac{1}{4}$ the transpose of the matrix defining the symbol of $\mathcal{P}$, equation (19.6.13). This rule is general whenever $\mathcal{R} = \mathcal{P}^\dagger$ and $H = 2h$.

The particular choice of $\mathcal{R}$ in (19.6.17) is called *full weighting*. Another popular choice for $\mathcal{R}$ is *half weighting*, "halfway" between full weighting and straight injection. Its symbol is

$$\begin{bmatrix} 0 & \frac{1}{8} & 0 \\ \frac{1}{8} & \frac{1}{2} & \frac{1}{8} \\ 0 & \frac{1}{8} & 0 \end{bmatrix} \tag{19.6.18}$$

A similar notation can be used to describe the difference operator $\mathcal{L}_h$. For example, the standard differencing of the model problem, equation (19.0.6), is represented by the *five-point difference star*

$$\mathcal{L}_h = \frac{1}{h^2} \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{19.6.19}$$

(Obsolete 2nd Edition. Copyright (C) Numerical Recipes Software 1988-2003. Try the 3rd Edition at ...)

If you are confronted with a new problem and you are not sure what $\mathcal{P}$ and $\mathcal{R}$ choices are likely to work well, here is a safe rule: Suppose $m_p$ is the order of the interpolation $\mathcal{P}$ (i.e., it interpolates polynomials of degree $m_p - 1$ exactly). Suppose $m_r$ is the order of $\mathcal{R}$, and that $\mathcal{R}$ is the adjoint of some $\mathcal{P}$ (not necessarily the $\mathcal{P}$ you intend to use). Then if $m$ is the order of the differential operator $\mathcal{L}_h$, you should satisfy the inequality $m_p + m_r > m$. For example, bilinear interpolation and its adjoint, full weighting, for Poisson's equation satisfy $m_p + m_r = 4 > m = 2$.

Of course the $\mathcal{P}$ and $\mathcal{R}$ operators should enforce the boundary conditions for your problem. The easiest way to do this is to rewrite the difference equation to have homogeneous boundary conditions by modifying the source term if necessary (cf. §19.4). Enforcing homogeneous boundary conditions simply requires the $\mathcal{P}$ operator to produce zeros at the appropriate boundary points. The corresponding $\mathcal{R}$ is then found by $\mathcal{R} = \mathcal{P}^{\dagger}$.

## Full Multigrid Algorithm

So far we have described multigrid as an iterative scheme, where one starts with some initial guess on the finest grid and carries out enough cycles (V-cycles, W-cycles,...) to achieve convergence. This is the simplest way to use multigrid: Simply apply enough cycles until some appropriate convergence criterion is met. However, efficiency can be improved by using the *Full Multigrid Algorithm* (FMG), also known as *nested iteration*.

Instead of starting with an arbitrary approximation on the finest grid (e.g., $u_h = 0$), the first approximation is obtained by interpolating from a coarse-grid solution:

$$u_h = \mathcal{P} u_H \tag{19.6.20}$$

The coarse-grid solution itself is found by a similar FMG process from even coarser grids. At the coarsest level, you start with the exact solution. Rather than proceed as in Figure 19.6.1, then, FMG gets to its solution by a series of increasingly tall "N's," each taller one probing a finer grid (see Figure 19.6.2).

Note that $\mathcal{P}$ in (19.6.20) need not be the same $\mathcal{P}$ used in the multigrid cycles. It should be at least of the same order as the discretization $\mathcal{L}_h$, but sometimes a higher-order operator leads to greater efficiency.

It turns out that you usually need one or at most two multigrid cycles at each level before proceeding down to the next finer grid. While there is theoretical guidance on the required number of cycles (e.g., [2]), you can easily determine it empirically. Fix the finest level and study the solution values as you increase the number of cycles per level. The asymptotic value of the solution is the exact solution of the difference equations. The difference between this exact solution and the solution for a small number of cycles is the iteration error. Now fix the number of cycles to be large, and vary the number of levels, i.e., the smallest value of $h$ used. In this way you can estimate the truncation error for a given $h$. In your final production code, there is no point in using more cycles than you need to get the iteration error down to the size of the truncation error.

The simple multigrid iteration (cycle) needs the right-hand side $f$ only at the finest level. FMG needs $f$ at all levels. If the boundary conditions are homogeneous,

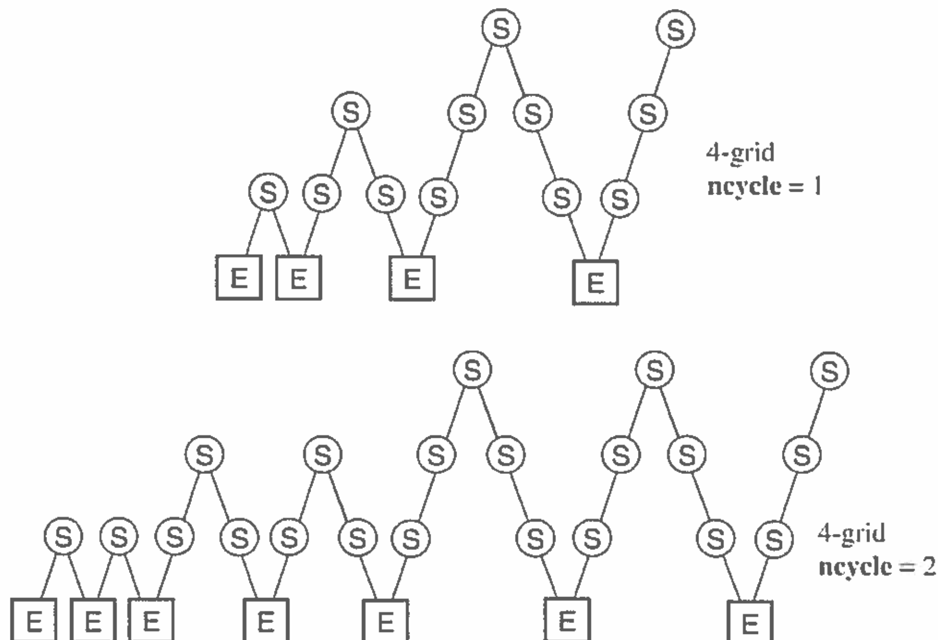



Figure 19.6.2. Structure of cycles for the full multigrid (FMG) method. This method starts on the coarsest grid, interpolates, and then refines (by "V's"), the solution onto grids of increasing fineness.

you can use $f_H = \mathcal{R} f_h$. This prescription is not always safe for inhomogeneous boundary conditions. In that case it is better to discretize $f$ on each coarse grid.

Note that the FMG algorithm produces the solution on all levels. It can therefore be combined with techniques like Richardson extrapolation.

We now give a routine `mglin` that implements the Full Multigrid Algorithm for a linear equation, the model problem (19.0.6). It uses red-black Gauss-Seidel as the smoothing operator, bilinear interpolation for $\mathcal{P}$, and half-weighting for $\mathcal{R}$. To change the routine to handle another linear problem, all you need do is modify the functions `relax`, `resid`, and `slvsml` appropriately. A feature of the routine is the dynamical allocation of storage for variables defined on the various grids.

```
#include "nrutil.h"
#define NPRE 1                    Number of relaxation sweeps before ...
#define NPOST 1                   ... and after the coarse-grid correction is com-
#define NGMAX 15                      puted.

void mglin(double **u, int n, int ncycle)
```
Full Multigrid Algorithm for solution of linear elliptic equation, here the model problem (19.0.6). On input `u[1..n][1..n]` contains the right-hand side $\rho$, while on output it returns the solution. The dimension `n` must be of the form $2^j + 1$ for some integer $j$. ($j$ is actually the number of grid levels used in the solution, called `ng` below.) `ncycle` is the number of V-cycles to be used at each level.
```
{
    void addint(double **uf, double **uc, double **res, int nf);
    void copy(double **aout, double **ain, int n);
    void fill0(double **u, int n);
    void interp(double **uf, double **uc, int nf);
    void relax(double **u, double **rhs, int n);
    void resid(double **res, double **u, double **rhs, int n);
    void rstrct(double **uc, double **uf, int nc);
    void slvsml(double **u, double **rhs);
```

```
    unsigned int j,jcycle,jj,jpost,jpre,nf,ng=0,ngrid,nn;
    double **ires[NGMAX+1],**irho[NGMAX+1],**irhs[NGMAX+1],**iu[NGMAX+1];

    nn=n;
    while (nn >>= 1) ng++;
    if (n != 1+(1L << ng)) nrerror("n-1 must be a power of 2 in mglin.");
    if (ng > NGMAX) nrerror("increase NGMAX in mglin.");
    nn=n/2+1;
    ngrid=ng-1;
    irho[ngrid]=dmatrix(1,nn,1,nn);          Allocate storage for r.h.s. on grid ng − 1,
    rstrct(irho[ngrid],u,nn);                and fill it by restricting from the fine grid.
    while (nn > 3) {                         Similarly allocate storage and fill r.h.s. on all
        nn=nn/2+1;                               coarse grids.
        irho[--ngrid]=dmatrix(1,nn,1,nn);
        rstrct(irho[ngrid],irho[ngrid+1],nn);
    }
    nn=3;
    iu[1]=dmatrix(1,nn,1,nn);
    irhs[1]=dmatrix(1,nn,1,nn);
    slvsml(iu[1],irho[1]);                   Initial solution on coarsest grid.
    free_dmatrix(irho[1],1,nn,1,nn);
    ngrid=ng;
    for (j=2;j<=ngrid;j++) {                 Nested iteration loop.
        nn=2*nn-1;
        iu[j]=dmatrix(1,nn,1,nn);
        irhs[j]=dmatrix(1,nn,1,nn);
        ires[j]=dmatrix(1,nn,1,nn);
        interp(iu[j],iu[j-1],nn);
        Interpolate from coarse grid to next finer grid.
        copy(irhs[j],(j != ngrid ? irho[j] : u),nn);      Set up r.h.s.
        for (jcycle=1;jcycle<=ncycle;jcycle++) {          V-cycle loop.
            nf=nn;
            for (jj=j;jj>=2;jj--) {                       Downward stoke of the V.
            for (jpre=1;jpre<=NPRE;jpre++)                Pre-smoothing.
                relax(iu[jj],irhs[jj],nf);
            resid(ires[jj],iu[jj],irhs[jj],nf);
            nf=nf/2+1;
            rstrct(irhs[jj-1],ires[jj],nf);
            Restriction of the residual is the next r.h.s.
            fill0(iu[jj-1],nf);                           Zero for initial guess in next
            }                                                 relaxation.
            slvsml(iu[1],irhs[1]);                        Bottom of V: solve on coars-
            nf=3;                                             est grid.
            for (jj=2;jj<=j;jj++) {                       Upward stroke of V.
            nf=2*nf-1;
            addint(iu[jj],iu[jj-1],ires[jj],nf);
            Use res for temporary storage inside addint.
            for (jpost=1;jpost<=NPOST;jpost++)            Post-smoothing.
                relax(iu[jj],irhs[jj],nf);
            }
        }
    }
    copy(u,iu[ngrid],n);                                  Return solution in u.
    for (nn=n,j=ng;j>=2;j--,nn=nn/2+1) {
        free_dmatrix(ires[j],1,nn,1,nn);
        free_dmatrix(irhs[j],1,nn,1,nn);
        free_dmatrix(iu[j],1,nn,1,nn);
        if (j != ng) free_dmatrix(irho[j],1,nn,1,nn);
    }
    free_dmatrix(irhs[1],1,3,1,3);
    free_dmatrix(iu[1],1,3,1,3);
}
```

```
void rstrct(double **uc, double **uf, int nc)
```

Half-weighting restriction. nc is the coarse-grid dimension. The fine-grid solution is input in uf[1..2*nc-1][1..2*nc-1], the coarse-grid solution is returned in uc[1..nc][1..nc].

```
{
    int ic,iif,jc,jf,ncc=2*nc-1;

    for (jf=3,jc=2;jc<nc;jc++,jf+=2) {          Interior points.
        for (iif=3,ic=2;ic<nc;ic++,iif+=2) {
            uc[ic][jc]=0.5*uf[iif][jf]+0.125*(uf[iif+1][jf]+uf[iif-1][jf]
                    +uf[iif][jf+1]+uf[iif][jf-1]);
        }
    }
    for (jc=1,ic=1;ic<=nc;ic++,jc+=2) {         Boundary points.
        uc[ic][1]=uf[jc][1];
        uc[ic][nc]=uf[jc][ncc];
    }
    for (jc=1,ic=1;ic<=nc;ic++,jc+=2) {
        uc[1][ic]=uf[1][jc];
        uc[nc][ic]=uf[ncc][jc];
    }
}
```

```
void interp(double **uf, double **uc, int nf)
```

Coarse-to-fine prolongation by bilinear interpolation. nf is the fine-grid dimension. The coarse-grid solution is input as uc[1..nc][1..nc], where $nc = nf/2 + 1$. The fine-grid solution is returned in uf[1..nf][1..nf].

```
{
    int ic,iif,jc,jf,nc;
    nc=nf/2+1;
    for (jc=1,jf=1;jc<=nc;jc++,jf+=2)           Do elements that are copies.
        for (ic=1;ic<=nc;ic++) uf[2*ic-1][jf]=uc[ic][jc];
    for (jf=1;jf<=nf;jf+=2)                     Do odd-numbered columns, interpolat-
        for (iif=2;iif<nf;iif+=2)                   ing vertically.
            uf[iif][jf]=0.5*(uf[iif+1][jf]+uf[iif-1][jf]);

    for (jf=2;jf<nf;jf+=2)                      Do even-numbered columns, interpolat-
        for (iif=1;iif <= nf;iif++)                 ing horizontally.
            uf[iif][jf]=0.5*(uf[iif][jf+1]+uf[iif][jf-1]);
}
```

```
void addint(double **uf, double **uc, double **res, int nf)
```

Does coarse-to-fine interpolation and adds result to uf. nf is the fine-grid dimension. The coarse-grid solution is input as uc[1..nc][1..nc], where $nc = nf/2 + 1$. The fine-grid solution is returned in uf[1..nf][1..nf]. res[1..nf][1..nf] is used for temporary storage.

```
{
    void interp(double **uf, double **uc, int nf);
    int i,j;

    interp(res,uc,nf);
    for (j=1;j<=nf;j++)
        for (i=1;i<=nf;i++)
            uf[i][j] += res[i][j];
}
```

```
void slvsml(double **u, double **rhs)
```

Solution of the model problem on the coarsest grid, where $h = \frac{1}{2}$. The right-hand side is input in rhs[1..3][1..3] and the solution is returned in u[1..3][1..3].

```
{
    void fill0(double **u, int n);
    double h=0.5;

    fill0(u,3);
```

866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881

```
    u[2][2] = -h*h*rhs[2][2]/4.0;
}


void relax(double **u, double **rhs, int n)
```

Red-black Gauss-Seidel relaxation for model problem. Updates the current value of the solution u[1..n][1..n], using the right-hand side function rhs[1..n][1..n].

```
{
    int i,ipass,isw,j,jsw=1;
    double h,h2;

    h=1.0/(n-1);
    h2=h*h;
    for (ipass=1;ipass<=2;ipass++,jsw=3-jsw) {        Red and black sweeps.
        isw=jsw;
        for (j=2;j<n;j++,isw=3-isw)
            for (i=isw+1;i<n;i+=2)                    Gauss-Seidel formula.
                u[i][j]=0.25*(u[i+1][j]+u[i-1][j]+u[i][j+1]
                    +u[i][j-1]-h2*rhs[i][j]);
    }
}


void resid(double **res, double **u, double **rhs, int n)
```

Returns *minus* the residual for the model problem. Input quantities are u[1..n][1..n] and rhs[1..n][1..n], while res[1..n][1..n] is returned.

```
{
    int i,j;
    double h,h2i;

    h=1.0/(n-1);
    h2i=1.0/(h*h);
    for (j=2;j<n;j++)            Interior points.
        for (i=2;i<n;i++)
            res[i][j] = -h2i*(u[i+1][j]+u[i-1][j]+u[i][j+1]+u[i][j-1]-
                    4.0*u[i][j])+rhs[i][j];
    for (i=1;i<=n;i++)           Boundary points.
        res[i][1]=res[i][n]=res[1][i]=res[n][i]=0.0;
}


void copy(double **aout, double **ain, int n)
```

Copies ain[1..n][1..n] to aout[1..n][1..n].

```
{
    int i,j;
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++)
            aout[j][i]=ain[j][i];

}


void fill0(double **u, int n)
```

Fills u[1..n][1..n] with zeros.

```
{
    int i,j;
    for (j=1;j<=n;j++)
        for (i=1;i<=n;i++)
            u[i][j]=0.0;
}
```

The routine `mglin` is written for clarity, not maximum efficiency, so that it is easy to modify. Several simple changes will speed up the execution time:

- The defect $d_h$ vanishes identically at all black mesh points after a red-black Gauss-Seidel step. Thus $d_H = \mathcal{R} d_h$ for half-weighting reduces to simply copying half the defect from the fine grid to the corresponding coarse-grid point. The calls to `resid` followed by `rstrct` in the first part of the V-cycle can be replaced by a routine that loops only over the coarse grid, filling it with half the defect.
- Similarly, the quantity $\widetilde{u}_h^{\text{new}} = \widetilde{u}_h + \mathcal{P} \widetilde{v}_H$ need not be computed at red mesh points, since they will immediately be redefined in the subsequent Gauss-Seidel sweep. This means that `addint` need only loop over black points.
- You can speed up `relax` in several ways. First, you can have a special form when the initial guess is zero, and omit the routine `fill0`. Next, you can store $h^2 f_h$ on the various grids and save a multiplication. Finally, it is possible to save an addition in the Gauss-Seidel formula by rewriting it with intermediate variables.
- On typical problems, `mglin` with `ncycle` $= 1$ will return a solution with the iteration error bigger than the truncation error for the given size of $h$. To knock the error down to the size of the truncation error, you have to set `ncycle` $= 2$ or, more cheaply, `npre` $= 2$. A more efficient way turns out to be to use a higher-order $\mathcal{P}$ in (19.6.20) than the linear interpolation used in the V-cycle.

Implementing all the above features typically gives up to a factor of two improvement in execution time and is certainly worthwhile in a production code.

## *Nonlinear Multigrid: The FAS Algorithm*

Now turn to solving a nonlinear elliptic equation, which we write symbolically as

$$\mathcal{L}(u) = 0 \tag{19.6.21}$$

Any explicit source term has been moved to the left-hand side. Suppose equation (19.6.21) is suitably discretized:

$$\mathcal{L}_h(u_h) = 0 \tag{19.6.22}$$

We will see below that in the multigrid algorithm we will have to consider equations where a nonzero right-hand side is generated during the course of the solution:

$$\mathcal{L}_h(u_h) = f_h \tag{19.6.23}$$

One way of solving nonlinear problems with multigrid is to use Newton's method, which produces linear equations for the correction term at each iteration. We can then use linear multigrid to solve these equations. A great strength of the multigrid idea, however, is that it can be applied *directly* to nonlinear problems. All we need is a suitable *nonlinear* relaxation method to smooth the errors, plus a procedure for approximating corrections on coarser grids. This direct approach is Brandt's Full Approximation Storage Algorithm (FAS). No nonlinear equations need be solved, except perhaps on the coarsest grid.

To develop the nonlinear algorithm, suppose we have a relaxation procedure that can smooth the residual vector as we did in the linear case. Then we can seek a smooth correction $v_h$ to solve (19.6.23):

$$\mathcal{L}_h(\widetilde{u}_h + v_h) = f_h \tag{19.6.24}$$

To find $v_h$, note that

$$\begin{aligned} \mathcal{L}_h(\widetilde{u}_h + v_h) - \mathcal{L}_h(\widetilde{u}_h) &= f_h - \mathcal{L}_h(\widetilde{u}_h) \\ &= -d_h \end{aligned} \tag{19.6.25}$$

The right-hand side is smooth after a few nonlinear relaxation sweeps. Thus we can transfer the left-hand side to a coarse grid:

$$\mathcal{L}_H(u_H) - \mathcal{L}_H(\mathcal{R}\widetilde{u}_h) = -\mathcal{R}d_h \tag{19.6.26}$$

that is, we solve

$$\mathcal{L}_H(u_H) = \mathcal{L}_H(\mathcal{R}\widetilde{u}_h) - \mathcal{R}d_h \tag{19.6.27}$$

on the coarse grid. (This is how nonzero right-hand sides appear.) Suppose the approximate solution is $\widetilde{u}_H$. Then the coarse-grid correction is

$$\widetilde{v}_H = \widetilde{u}_H - \mathcal{R}\widetilde{u}_h \tag{19.6.28}$$

and

$$\widetilde{u}_h^{\text{new}} = \widetilde{u}_h + \mathcal{P}(\widetilde{u}_H - \mathcal{R}\widetilde{u}_h) \tag{19.6.29}$$

Note that $\mathcal{PR} \neq 1$ in general, so $\widetilde{u}_h^{\text{new}} \neq \mathcal{P}\widetilde{u}_H$. This is a key point: In equation (19.6.29) the interpolation error comes only from the correction, not from the full solution $\widetilde{u}_H$.

Equation (19.6.27) shows that one is solving for the full approximation $u_H$, not just the error as in the linear algorithm. This is the origin of the name FAS.

The FAS multigrid algorithm thus looks very similar to the linear multigrid algorithm. The only differences are that both the defect $d_h$ and the relaxed approximation $u_h$ have to be restricted to the coarse grid, where now it is equation (19.6.27) that is solved by recursive invocation of the algorithm. However, instead of implementing the algorithm this way, we will first describe the so-called *dual viewpoint*, which leads to a powerful alternative way of looking at the multigrid idea.

The dual viewpoint considers the *local truncation error*, defined as

$$\tau \equiv \mathcal{L}_h(u) - f_h \tag{19.6.30}$$

where $u$ is the exact solution of the original continuum equation. If we rewrite this as

$$\mathcal{L}_h(u) = f_h + \tau \tag{19.6.31}$$

we see that $\tau$ can be regarded as the correction to $f_h$ so that the solution of the fine-grid equation will be the exact solution $u$.

Now consider the *relative truncation error* $\tau_h$, which is defined on the $H$-grid relative to the $h$-grid:

$$\tau_h \equiv \mathcal{L}_H(\mathcal{R}u_h) - \mathcal{R}\mathcal{L}_h(u_h) \tag{19.6.32}$$

Since $\mathcal{L}_h(u_h) = f_h$, this can be rewritten as

$$\mathcal{L}_H(u_H) = f_H + \tau_h \tag{19.6.33}$$

In other words, we can think of $\tau_h$ as the correction to $f_H$ that makes the solution of the coarse-grid equation equal to the fine-grid solution. Of course we cannot compute $\tau_h$, but we do have an approximation to it from using $\widetilde{u}_h$ in equation (19.6.32):

$$\tau_h \simeq \widetilde{\tau}_h \equiv \mathcal{L}_H(\mathcal{R}\widetilde{u}_h) - \mathcal{R}\mathcal{L}_h(\widetilde{u}_h) \tag{19.6.34}$$

Replacing $\tau_h$ by $\widetilde{\tau}_h$ in equation (19.6.33) gives

$$\mathcal{L}_H(u_H) = \mathcal{L}_H(\mathcal{R}\widetilde{u}_h) - \mathcal{R}d_h \tag{19.6.35}$$

which is just the coarse-grid equation (19.6.27)!

Thus we see that there are two complementary viewpoints for the relation between coarse and fine grids:

- Coarse grids are used to accelerate the convergence of the smooth components of the fine-grid residuals.

- Fine grids are used to compute correction terms to the coarse-grid equations, yielding fine-grid accuracy on the coarse grids.

One benefit of this new viewpoint is that it allows us to derive a natural stopping criterion for a multigrid iteration. Normally the criterion would be

$$\|d_h\| \leq \epsilon \tag{19.6.36}$$

and the question is how to choose $\epsilon$. There is clearly no benefit in iterating beyond the point when the remaining error is dominated by the local truncation error $\tau$. The computable quantity is $\tilde{\tau}_h$. What is the relation between $\tau$ and $\tilde{\tau}_h$? For the typical case of a second-order accurate differencing scheme,

$$\tau = \mathcal{L}_h(u) - \mathcal{L}_h(u_h) = h^2\tau_2(x,y) + \cdots \tag{19.6.37}$$

Assume the solution satisfies $u_h = u + h^2 u_2(x,y) + \cdots$. Then, assuming $\mathcal{R}$ is of high enough order that we can neglect its effect, equation (19.6.32) gives

$$\begin{aligned}\tau_h &\simeq \mathcal{L}_H(u + h^2 u_2) - \mathcal{L}_h(u + h^2 u_2) \\ &= \mathcal{L}_H(u) - \mathcal{L}_h(u) + h^2[\mathcal{L}'_H(u_2) - \mathcal{L}'_h(u_2)] + \cdots \\ &= (H^2 - h^2)\tau_2 + O(h^4)\end{aligned} \tag{19.6.38}$$

For the usual case of $H = 2h$ we therefore have

$$\tau \simeq \tfrac{1}{3}\tau_h \simeq \tfrac{1}{3}\tilde{\tau}_h \tag{19.6.39}$$

The stopping criterion is thus equation (19.6.36) with

$$\epsilon = \alpha\|\tilde{\tau}_h\|, \qquad \alpha \sim \tfrac{1}{3} \tag{19.6.40}$$

We have one remaining task before implementing our nonlinear multigrid algorithm: choosing a nonlinear relaxation scheme. Once again, your first choice should probably be the nonlinear Gauss-Seidel scheme. If the discretized equation (19.6.23) is written with some choice of ordering as

$$L_i(u_1,\ldots,u_N) = f_i, \qquad i = 1,\ldots,N \tag{19.6.41}$$

then the nonlinear Gauss-Seidel schemes solves

$$L_i(u_1,\ldots,u_{i-1},u_i^{\text{new}},u_{i+1},\ldots,u_N) = f_i \tag{19.6.42}$$

for $u_i^{\text{new}}$. As usual new $u$'s replace old $u$'s as soon as they have been computed. Often equation (19.6.42) is linear in $u_i^{\text{new}}$, since the nonlinear terms are discretized by means of its neighbors. If this is not the case, we replace equation (19.6.42) by one step of a Newton iteration:

$$u_i^{\text{new}} = u_i^{\text{old}} - \frac{L_i(u_i^{\text{old}}) - f_i}{\partial L_i(u_i^{\text{old}})/\partial u_i} \tag{19.6.43}$$

For example, consider the simple nonlinear equation

$$\nabla^2 u + u^2 = \rho \tag{19.6.44}$$

In two-dimensional notation, we have

$$\mathcal{L}(u_{i,j}) = (u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j})/h^2 + u_{i,j}^2 - \rho_{i,j} = 0 \tag{19.6.45}$$

Since

$$\frac{\partial\mathcal{L}}{\partial u_{i,j}} = -4/h^2 + 2u_{i,j} \tag{19.6.46}$$

the Newton Gauss-Seidel iteration is

$$u_{i,j}^{\text{new}} = u_{i,j} - \frac{\mathcal{L}(u_{i,j})}{-4/h^2 + 2u_{i,j}} \tag{19.6.47}$$

Here is a routine `mgfas` that solves equation (19.6.44) using the Full Multigrid Algorithm and the FAS scheme. Restriction and prolongation are done as in `mglin`. We have included the convergence test based on equation (19.6.40). A successful multigrid solution of a problem should aim to satisfy this condition with the maximum number of V-cycles, `maxcyc`, equal to 1 or 2. The routine `mgfas` uses the same functions `copy`, `interp`, and `rstrct` as `mglin`.

```
#include "nrutil.h"
#define NPRE 1                    Number of relaxation sweeps before ...
#define NPOST 1                   ... and after the coarse-grid correction is computed.
#define ALPHA 0.33                Relates the estimated truncation error to the norm
#define NGMAX 15                      of the residual.

void mgfas(double **u, int n, int maxcyc)
```

Full Multigrid Algorithm for FAS solution of nonlinear elliptic equation, here equation (19.6.44). On input `u[1..n][1..n]` contains the right-hand side $\rho$, while on output it returns the solution. The dimension `n` must be of the form $2^j + 1$ for some integer $j$. ($j$ is actually the number of grid levels used in the solution, called `ng` below.) `maxcyc` is the maximum number of V-cycles to be used at each level.

```
{
    double anorm2(double **a, int n);
    void copy(double **aout, double **ain, int n);
    void interp(double **uf, double **uc, int nf);
    void lop(double **out, double **u, int n);
    void matadd(double **a, double **b, double **c, int n);
    void matsub(double **a, double **b, double **c, int n);
    void relax2(double **u, double **rhs, int n);
    void rstrct(double **uc, double **uf, int nc);
    void slvsm2(double **u, double **rhs);
    unsigned int j,jcycle,jj,jm1,jpost,jpre,nf,ng=0,ngrid,nn;
    double **irho[NGMAX+1],**irhs[NGMAX+1],**itau[NGMAX+1],
        **itemp[NGMAX+1],**iu[NGMAX+1];
    double res,trerr;

    nn=n;
    while (nn >>= 1) ng++;
    if (n != 1+(1L << ng)) nrerror("n-1 must be a power of 2 in mgfas.");
    if (ng > NGMAX) nrerror("increase NGMAX in mglin.");
    nn=n/2+1;
    ngrid=ng-1;
    irho[ngrid]=dmatrix(1,nn,1,nn);        Allocate storage for r.h.s. on grid ng - 1,
    rstrct(irho[ngrid],u,nn);              and fill it by restricting from the fine grid.
    while (nn > 3) {                       Similarly allocate storage and fill r.h.s. on all
        nn=nn/2+1;                             coarse grids.
        irho[--ngrid]=dmatrix(1,nn,1,nn);
        rstrct(irho[ngrid],irho[ngrid+1],nn);
    }
    nn=3;
    iu[1]=dmatrix(1,nn,1,nn);
    irhs[1]=dmatrix(1,nn,1,nn);
    itau[1]=dmatrix(1,nn,1,nn);
    itemp[1]=dmatrix(1,nn,1,nn);
    slvsm2(iu[1],irho[1]);                 Initial solution on coarsest grid.
    free_dmatrix(irho[1],1,nn,1,nn);
    ngrid=ng;
    for (j=2;j<=ngrid;j++) {               Nested iteration loop.
        nn=2*nn-1;
        iu[j]=dmatrix(1,nn,1,nn);
        irhs[j]=dmatrix(1,nn,1,nn);
        itau[j]=dmatrix(1,nn,1,nn);
        itemp[j]=dmatrix(1,nn,1,nn);
        interp(iu[j],iu[j-1],nn);
        Interpolate from coarse grid to next finer grid.
        copy(irhs[j],(j != ngrid ? irho[j] : u),nn);    Set up r.h.s.
        for (jcycle=1;jcycle<=maxcyc;jcycle++) {        V-cycle loop.
        nf=nn;
            for (jj=j;jj>=2;jj--) {                     Downward stoke of the V.
                for (jpre=1;jpre<=NPRE;jpre++)          Pre-smoothing.
                    relax2(iu[jj],irhs[jj],nf);
                lop(itemp[jj],iu[jj],nf);               L_h(ũ_h).
                nf=nf/2+1;
```

```
            jm1=jj-1;
            rstrct(itemp[jm1],itemp[jj],nf);          RL_h(ũ_h).
            rstrct(iu[jm1],iu[jj],nf);                Rũ_h.
            lop(itau[jm1],iu[jm1],nf);
            L_H(Rũ_h) stored temporarily in τ̃_h.
            matsub(itau[jm1],itemp[jm1],itau[jm1],nf);     Form τ̃_h.
            if (jj == j)
                trerr=ALPHA*anorm2(itau[jm1],nf);     Estimate truncation error τ.
            rstrct(irhs[jm1],irhs[jj],nf);            f_H.
            matadd(irhs[jm1],itau[jm1],irhs[jm1],nf);   f_H + τ̃_h.
        }
        slvsm2(iu[1],irhs[1]);                        Bottom of V: Solve on coars-
        nf=3;                                             est grid.
        for (jj=2;jj<=j;jj++) {                       Upward stroke of V.
        jm1=jj-1;
        rstrct(itemp[jm1],iu[jj],nf);                 Rũ_h.
        matsub(iu[jm1],itemp[jm1],itemp[jm1],nf);     ũ_H − Rũ_h.
        nf=2*nf-1;
        interp(itau[jj],itemp[jm1],nf);               P(ũ_H − Rũ_h) stored in τ̃_h.
        matadd(iu[jj],itau[jj],iu[jj],nf);            Form ũ_h^new.
        for (jpost=1;jpost<=NPOST;jpost++)            Post-smoothing.
            relax2(iu[jj],irhs[jj],nf);
        }
        lop(itemp[j],iu[j],nf);                       Form residual ||d_h||.
        matsub(itemp[j],irhs[j],itemp[j],nf);
        res=anorm2(itemp[j],nf);
        if (res < trerr) break;                       No more V-cycles needed if
    }                                                     residual small enough.
  }
  copy(u,iu[ngrid],n);                                Return solution in u.
  for (nn=n,j=ng;j>=1;j--,nn=nn/2+1) {
      free_dmatrix(itemp[j],1,nn,1,nn);
      free_dmatrix(itau[j],1,nn,1,nn);
      free_dmatrix(irhs[j],1,nn,1,nn);
      free_dmatrix(iu[j],1,nn,1,nn);
      if (j != ng && j != 1) free_dmatrix(irho[j],1,nn,1,nn);
  }
}
```

```
void relax2(double **u, double **rhs, int n)
```

Red-black Gauss-Seidel relaxation for equation (19.6.44). The current value of the solution `u[1..n][1..n]` is updated, using the right-hand side function `rhs[1..n][1..n]`.

```
{
    int i,ipass,isw,j,jsw=1;
    double foh2,h,h2i,res;

    h=1.0/(n-1);
    h2i=1.0/(h*h);
    foh2 = -4.0*h2i;
    for (ipass=1;ipass<=2;ipass++,jsw=3-jsw) {        Red and black sweeps.
        isw=jsw;
        for (j=2;j<n;j++,isw=3-isw) {
            for (i=isw+1;i<n;i+=2) {
                res=h2i*(u[i+1][j]+u[i-1][j]+u[i][j+1]+u[i][j-1]-
                    4.0*u[i][j])+u[i][j]*u[i][j]-rhs[i][j];
                u[i][j] -= res/(foh2+2.0*u[i][j]);     Newton Gauss-Seidel formula.
            }
        }
    }
}
```

```
#include <math.h>

void slvsm2(double **u, double **rhs)
```
Solution of equation (19.6.44) on the coarsest grid, where $h = \frac{1}{2}$. The right-hand side is input in rhs[1..3][1..3] and the solution is returned in u[1..3][1..3].
```
{
    void fill0(double **u, int n);
    double disc,fact,h=0.5;

    fill0(u,3);
    fact=2.0/(h*h);
    disc=sqrt(fact*fact+rhs[2][2]);
    u[2][2] = -rhs[2][2]/(fact+disc);
}

void lop(double **out, double **u, int n)
```
Given u[1..n][1..n], returns $\mathcal{L}_h(\tilde{u}_h)$ for equation (19.6.44) in out[1..n][1..n].
```
{
    int i,j;
    double h,h2i;

    h=1.0/(n-1);
    h2i=1.0/(h*h);
    for (j=2;j<n;j++)                    Interior points.
        for (i=2;i<n;i++)
            out[i][j]=h2i*(u[i+1][j]+u[i-1][j]+u[i][j+1]+u[i][j-1]-
                4.0*u[i][j])+u[i][j]*u[i][j];
    for (i=1;i<=n;i++)                   Boundary points.
        out[i][1]=out[i][n]=out[1][i]=out[n][i]=0.0;
}

void matadd(double **a, double **b, double **c, int n)
```
Adds a[1..n][1..n] to b[1..n][1..n] and returns result in c[1..n][1..n].
```
{
    int i,j;

    for (j=1;j<=n;j++)
        for (i=1;i<=n;i++)
            c[i][j]=a[i][j]+b[i][j];
}

void matsub(double **a, double **b, double **c, int n)
```
Subtracts b[1..n][1..n] from a[1..n][1..n] and returns result in c[1..n][1..n].
```
{
    int i,j;

    for (j=1;j<=n;j++)
        for (i=1;i<=n;i++)
            c[i][j]=a[i][j]-b[i][j];
}

#include <math.h>

double anorm2(double **a, int n)
```
Returns the Euclidean norm of the matrix a[1..n][1..n].
```
{
    int i,j;
    double sum=0.0;

    for (j=1;j<=n;j++)
        for (i=1;i<=n;i++)
            sum += a[i][j]*a[i][j];
    return sqrt(sum)/n;
}
```

Copyright (C) Numerical Recipes Software 1988-2002. Try the 3rd Edition in C++

CITED REFERENCES AND FURTHER READING:

Brandt, A. 1977, *Mathematics of Computation*, vol. 31, pp. 333–390. [1]

Hackbusch, W. 1985, *Multi-Grid Methods and Applications* (New York: Springer-Verlag). [2]

Stuben, K., and Trottenberg, U. 1982, in *Multigrid Methods*, W. Hackbusch and U. Trottenberg, eds. (Springer Lecture Notes in Mathematics No. 960) (New York: Springer-Verlag), pp. 1–176. [3]

Brandt, A. 1982, in *Multigrid Methods*, W. Hackbusch and U. Trottenberg, eds. (Springer Lecture Notes in Mathematics No. 960) (New York: Springer-Verlag). [4]

Baker, L. 1991, *More C Tools for Scientists and Engineers* (New York: McGraw-Hill).

Briggs, W.L. 1987, *A Multigrid Tutorial* (Philadelphia: S.I.A.M.).

Jespersen, D. 1984, *Multigrid Methods for Partial Differential Equations* (Washington: Mathematical Association of America).

McCormick, S.F. (ed.) 1988, *Multigrid Methods: Theory, Applications, and Supercomputing* (New York: Marcel Dekker).

Hackbusch, W., and Trottenberg, U. (eds.) 1991, *Multigrid Methods III* (Boston: Birkhauser).

Wesseling, P. 1992, *An Introduction to Multigrid Methods* (New York: Wiley).

873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888