6

Iterative Methods for Linear Systems

6.1. Introduction

Iterative algorithms for solving $Ax = b$ are used when methods such as Gaussian elimination require too much time or too much space. Methods such as Gaussian elimination, which compute the exact answers after a finite number of steps (in the absence of roundoff!), are called *direct methods.* In contrast to direct methods, iterative methods generally do not produce the exact answer after a finite number of steps but decrease the error by some fraction after each step. Iteration ceases when the error is less than a user-supplied threshold. The final error depends on how many iterations one does as well as on properties of the method and the linear system. Our overall goal is to develop methods that decrease the error by a large amount at each iteration and do as little work per iteration as possible.

Much of the activity in this field involves exploiting the underlying mathematical or physical problem that gives rise to the linear system in order to design better iterative methods. The underlying problems are often finite difference or finite element models of physical systems, usually involving a differential equation. There are many kinds of physical systems, differential equations, and finite difference and finite element models, and so many methods. We cannot hope to cover all or even most interesting situations, so we will limit ourselves to a *model problem*, the standard finite difference approximation to Poisson's equation on a square. Poisson's equation and its close relation, Laplace's equation, arise in many applications, including electromagnetics, fluid mechanics, heat flow, diffusion, and quantum mechanics, to name a few. In addition to describing how each method works on Poisson's equation, we will indicate how generally applicable it is, and describe common variations.

The rest of this chapter is organized as follows. Section 6.2 describes on-line help and software for iterative methods discussed in this chapter. Section 6.3 describes the formulation of the model problem in detail. Section 6.4 summarizes and compares the performance of (nearly) all the iterative methods in this chapter for solving the model problem.

The next five sections describe methods in roughly increasing order of their effectiveness on the model problem. Section 6.5 describes the most basic iterative methods: Jacobi's, Gauss–Seidel, successive overrelaxation, and their variations. Section 6.6 describes Krylov subspace methods, concentrating on the conjugate gradient method. Section 6.7 describes the fast Fourier transform and how to use it to solve the model problem. Section 6.8 describes block cyclic reduction. Finally, section 6.9 discusses multigrid, our fastest algorithm for the model problem. Multigrid requires only $O(1)$ work per unknown, which is optimal.

Section 6.10 describes domain decomposition, a family of techniques for combining the simpler methods described in earlier sections to solve more complicated problems than the model problem.

6.2. On-line Help for Iterative Methods

For Poisson's equation, there will be a short list of numerical methods that are clearly superior to all the others we discuss. But for other linear systems it is not always clear which method is best (which is why we talk about so many!). To help users select the best method for solving their linear systems among the many available, on-line help is available at NETLIB/templates. This directory contains a short book [24] and software for most of the iterative methods discussed in this chapter. The book is available in both PostScript (NETLIB/templates/templates.ps) and Hypertext Markup Language (NETLIB/templates/Templates.html). The software is available in Matlab, Fortran, and C++.

The word *template* is used to describe this book and the software, because the implementations separate the details of matrix representations from the algorithm itself. In particular, the *Krylov subspace methods* (see section 6.6) require only the ability to multiply the matrix A by an arbitrary vector z. The best way to do this depends on how A is represented but does not otherwise affect the organization of the algorithm. In other words, matrix-vector multiplication is a "black-box" called by the template. It is the user's responsibility to supply an implementation of this black-box.

An analogous templates project for eigenvalue problems is underway. Other recent textbooks on iterative methods are [15, 136, 214].

For the most challenging practical problems arising from differential equations more challenging than our model problem, the linear system $Ax = b$ must be "preconditioned," or replaced with the equivalent systems $M^{-1}Ax = M^{-1}b$, which is somehow easier to solve. This is discussed at length in sections 6.6.5 and 6.10. Implementations, including parallel ones, of many of these techniques are available on-line in the package PETSc, or Portable Extensible Toolkit for Scientific computing, at http://www.mcs.anl.gov/petsc/petsc.html [232].

6.3. Poisson's Equation

6.3.1. Poisson's Equation in One Dimension

We begin with a one-dimensional version of Poisson's equation,

$$-\frac{d^2v(x)}{dx^2} = f(x), \quad 0 < x < 1, \tag{6.1}$$

where $f(x)$ is a given function and $v(x)$ is the unknown function that we want to compute. $v(x)$ must also satisfy the boundary conditions[24] $v(0) = v(1) = 0$. We *discretize* the problem by trying to compute an approximate solution at $N+2$ evenly spaced points x_i between 0 and 1: $x_i = ih$, where $h = \frac{1}{N+1}$ and $0 \le i \le N+1$. We abbreviate $v_i = v(x_i)$ and $f_i = f(x_i)$. To convert differential equation (6.1) into a linear equation for the unknowns $v_1, \ldots, v_N$, we use *finite differences* to approximate

$$\begin{aligned} \left.\frac{dv(x)}{dx}\right|_{x=(i-.5)h} &\approx \frac{v_i - v_{i-1}}{h}, \\ \left.\frac{dv(x)}{dx}\right|_{x=(i+.5)h} &\approx \frac{v_{i+1} - v_i}{h}. \end{aligned}$$

Subtracting these approximations and dividing by h yield the *centered difference approximation*

$$-\left.\frac{d^2v(x)}{dx^2}\right|_{x=x_i} = \frac{2v_i - v_{i-1} - v_{i+1}}{h^2} - \tau_i, \tag{6.2}$$

where τ_i, the so-called *truncation error*, can be shown to be $O(h^2 \cdot \|\frac{d^4v}{dx^4}\|_\infty)$. We may now rewrite equation (6.1) at $x = x_i$ as

$$-v_{i-1} + 2v_i - v_{i+1} = h^2 f_i + h^2 \tau_i,$$

where $0 < i < N+1$. Since the boundary conditions imply that $v_0 = v_{N+1} = 0$, we have N equations in N unknowns $v_1, \ldots, v_N$:

$$\begin{aligned} T_N \cdot \begin{bmatrix} v_1 \\ \vdots \\ \vdots \\ v_N \end{bmatrix} &\equiv \begin{bmatrix} 2 & -1 & & 0 \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ 0 & & -1 & 2 \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ \vdots \\ \vdots \\ v_N \end{bmatrix} \\ &= h^2 \begin{bmatrix} f_1 \\ \vdots \\ \vdots \\ f_N \end{bmatrix} + h^2 \begin{bmatrix} \tau_1 \\ \vdots \\ \vdots \\ \tau_N \end{bmatrix} \end{aligned} \tag{6.3}$$

[24] These are called *Dirichlet boundary conditions*. Other kinds of boundary conditions are also possible.

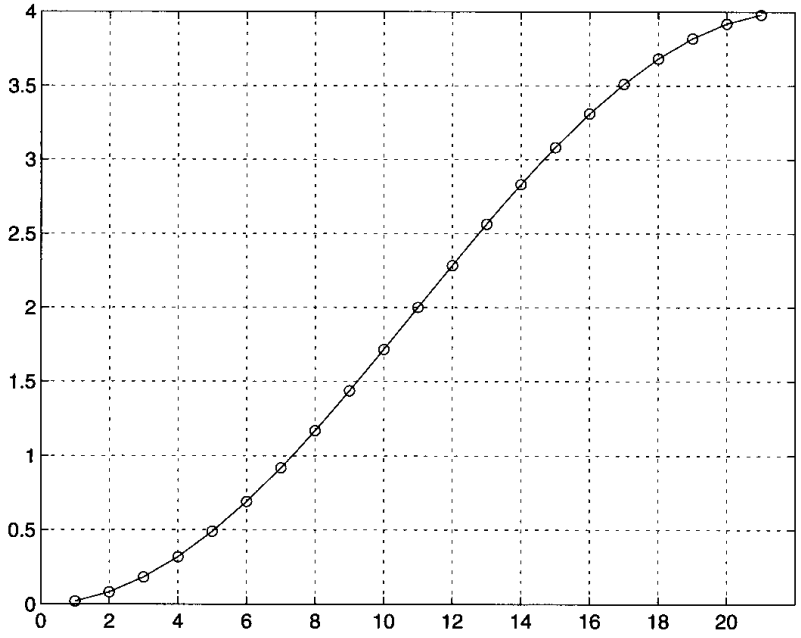


Fig. 6.1. *Eigenvalues of* T_{21}.

or

$$T_N v = h^2 f + h^2 \bar{\tau}. \tag{6.4}$$

To solve this equation, we will ignore $\bar{\tau}$, since it is small compared to f, to get

$$T_N \hat{v} = h^2 f. \tag{6.5}$$

(We bound the error $v - \hat{v}$ later.)

The coefficient matrix T_N plays a central role in all that follows, so we will examine it in some detail. First, we will compute its eigenvalues and eigenvectors. One can easily use trigonometric identities to confirm the following lemma (see Question 6.1).

LEMMA 6.1. *The eigenvalues of* T_N *are* $\lambda_j = 2(1 - \cos\frac{\pi j}{N+1})$. *The eigenvectors are* z_j, *where* $z_j(k) = \sqrt{\frac{2}{N+1}} \sin(jk\pi/(N+1))$. z_j *has unit two-norm. Let* $Z = [z_1, \ldots, z_n]$ *be the orthogonal matrix whose columns are the eigenvectors, and* $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$, *so we can write* $T_N = Z\Lambda Z^T$.

Figure 6.1 is a plot of the eigenvalues of T_N for $N = 21$.

The largest eigenvalue is $\lambda_N = 2(1 - \cos\pi\frac{N}{N+1}) \approx 4$. The smallest eigenvalue[25] is λ_1, where for small i

$$\lambda_i = 2\left(1 - \cos\frac{i\pi}{N+1}\right) \approx 2\left(1 - \left(1 - \frac{i^2\pi^2}{2(N+1)^2}\right)\right) = \left(\frac{i\pi}{N+1}\right)^2.$$

[25]Note that λ_N is the largest eigenvalue and λ_1 is the smallest eigenvalue, the opposite of the convention of Chapter 5.

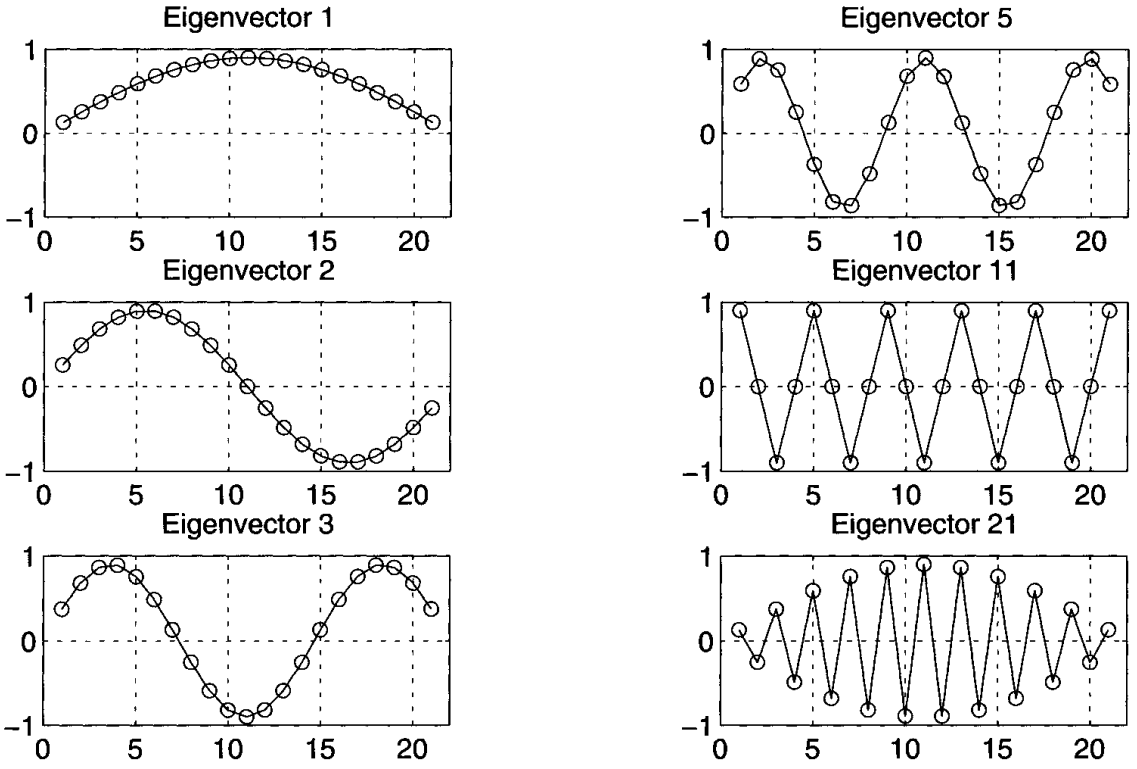


Fig. 6.2. *Eigenvectors of* T_{21}.

Thus T_N is positive definite with condition number $\lambda_N/\lambda_1 \approx 4(N+1)^2/\pi^2$ for large N. The eigenvectors are sinusoids with lowest frequency at $j = 1$ and highest at $j = N$, shown in Figure 6.2 for $N = 21$.

Now we know enough to bound the error, i.e., the difference between the solution of $T_N\hat{v} = h^2 f$ and the true solution v of the differential equation: Subtract equation (6.5) from equation (6.4) to get $v - \hat{v} = h^2 T_N^{-1}\bar{\tau}$. Taking norms yields

$$\|v - \hat{v}\|_2 \leq h^2 \|T_N^{-1}\|_2 \|\bar{\tau}\|_2 \approx h^2 \frac{(N+1)^2}{\pi^2} \|\bar{\tau}\|_2 = O(\|\bar{\tau}\|_2) = O\left(h^2 \left\| \frac{d^4 v}{dx^4} \right\|_\infty\right),$$

so the error $v - \hat{v}$ goes to zero proportionally to h^2, provided that the solution is smooth enough. ($\|\frac{d^4 v}{dx^4}\|_\infty$ is bounded.)

From now on we will not distinguish between v and its approximation $\hat{v}$ and so will simplify notation by letting $T_N v = h^2 f$.

In addition to the solution of the linear system $h^{-2}T_N v = f$ approximating the solution of the differential equation (6.1), it turns out that the eigenvalues and eigenvectors of $h^{-2}T_N$ also approximate the eigenvalues and *eigenfunctions* of the differential equation: We say that $\hat{\lambda}_i$ is an eigenvalue and $\hat{z}_i(x)$ is an eigenfunction of the differential equation if

$$-\frac{d^2 \hat{z}_i(x)}{dx^2} = \hat{\lambda}_i \hat{z}_i(x) \quad \text{with} \quad \hat{z}_i(0) = \hat{z}_i(1) = 0.$$

Let us solve for $\hat{\lambda}_i$ and $\hat{z}_i(x)$: It is easy to see that $\hat{z}_i(x)$ must equal $\alpha \sin(\sqrt{\hat{\lambda}_i}x) + \beta \cos(\sqrt{\hat{\lambda}_i}x)$ for some constants α and β. The boundary condition $\hat{z}_i(0) = 0$ implies $\beta = 0$, and the boundary condition $\hat{z}_i(1) = 0$ implies that $\sqrt{\hat{\lambda}_i}$ is an integer multiple of π, which we can take to be $i\pi$. Thus $\hat{\lambda}_i = i^2\pi^2$ and $\hat{z}_i(x) = \alpha \sin(i\pi x)$ for any nonzero constant α (which we can set to 1). Thus the eigenvector z_i is *precisely* equal to the eigenfunction $\hat{z}_i(x)$ evaluated at the sample points $x_j = jh$ (when scaled by $\sqrt{\frac{2}{N+1}}$). And when i is small, $\hat{\lambda}_i = i^2\pi^2$ is well approximated by $h^{-2} \cdot \lambda_i = (N+1)^2 \cdot 2(1 - \cos\frac{i\pi}{N+1}) = i^2\pi^2 + O((N+1)^{-2})$.

Thus we see there is a close correspondence between T_N (or $h^{-2}T_N$) and the second derivative operator $-\frac{d^2}{dx^2}$. This correspondence will be the motivation for the design and analysis of later algorithms.

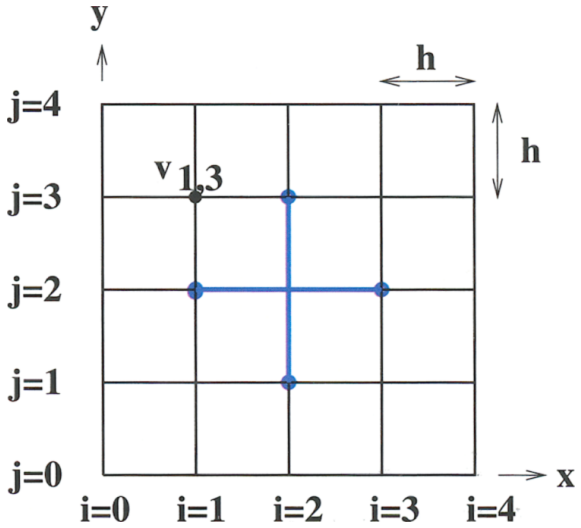
It is also possible to write down simple formulas for the Cholesky and LU factors of T_N; see Question 6.2 for details.

6.3.2. Poisson's Equation in Two Dimensions

Now we turn to Poisson's equation in two dimensions:

$$-\frac{\partial^2 v(x,y)}{\partial x^2} - \frac{\partial^2 v(x,y)}{\partial y^2} = f(x,y) \tag{6.6}$$

on the unit square $\{(x,y) : 0 < x, y < 1\}$, with boundary condition $v = 0$ on the boundary of the square. We discretize at the grid points in the square which are at (x_i, y_j) with $x_i = ih$ and $y_j = jh$, with $h = \frac{1}{N+1}$. We abbreviate $v_{ij} = v(ih, jh)$ and $f_{ij} = f(ih, jh)$, as shown below for $N = 3$:



From equation (6.2), we know that we can approximate

$$-\left.\frac{\partial^2 v(x,y)}{\partial x^2}\right|_{x=x_i, y=y_j} \approx \frac{2v_{i,j} - v_{i-1,j} - v_{i+1,j}}{h^2} \quad \text{and} \tag{6.7}$$

$$-\left.\frac{\partial^2 v(x,y)}{\partial y^2}\right|_{x=x_i, y=y_j} \approx \frac{2v_{i,j} - v_{i,j-1} - v_{i,j+1}}{h^2}. \tag{6.8}$$

Adding these approximations lets us write

$$-\frac{\partial^2 v(x,y)}{\partial x^2} - \frac{\partial^2 v(x,y)}{\partial y^2}\bigg|_{x=x_i, y=y_j} = \frac{4v_{ij} - v_{i-1,j} - v_{i+1,j} - v_{i,j-1} - v_{i,j+1}}{h^2} - \tau_{ij}, \tag{6.9}$$

where τ_{ij} is again a truncation error bounded by $O(h^2)$. The heavy (blue) cross in the middle of the above figure is called the (5-*point*) *stencil* of this equation, because it connects all (5) values of v present in equation (6.9). From the boundary conditions we know $v_{0j} = v_{N+1,j} = v_{i,0} = v_{i,N+1} = 0$ so that equation (6.9) defines a set of $n = N^2$ linear equations in the n unknowns v_{ij} for $1 \le i, j \le N$:

$$4v_{ij} - v_{i-1,j} - v_{i+1,j} - v_{i,j-1} - v_{i,j+1} = h^2 f_{ij}. \tag{6.10}$$

There are two ways to rewrite the n equations represented by (6.10) as a single matrix equation, both of which we will use later.

The first way is to think of the unknowns v_{ij} as occupying an N-by-N matrix V with entries v_{ij} and the right-hand sides $h^2 f_{ij}$ as similarly occupying an N-by-N matrix $h^2 F$. The trick is to write the matrix with i, j entry $4v_{ij} - v_{i-1,j} - v_{i+1,j} - v_{i,j-1} - v_{i,j+1}$ in a simple way in terms of V and T_N: Simply note that

$$\begin{aligned} 2v_{ij} - v_{i-1,j} - v_{i+1,j} &= (T_N \cdot V)_{ij}, \\ 2v_{ij} - v_{i,j-1} - v_{i,j+1} &= (V \cdot T_N)_{ij}, \end{aligned}$$

so adding these two equations yields

$$(T_N \cdot V + V \cdot T_N)_{ij} = 4v_{ij} - v_{i-1,j} - v_{i+1,j} - v_{i,j-1} - v_{i,j+1} = h^2 f_{ij} = (h^2 F)_{ij}$$

or

$$T_N \cdot V + V \cdot T_N = h^2 F. \tag{6.11}$$

This is a linear system of equations for the unknown entries of the matrix V, even though it is not written in the usual "$Ax = b$" format, with the unknowns forming a vector x. (We will write the "$Ax = b$" format below.) Still, it is enough to tell us what the eigenvalues and eigenvectors of the underlying matrix A are, because "$Ax = \lambda x$" is the same as "$T_N V + V T_N = \lambda V$." Now suppose that $T_N z_i = \lambda_i z_i$ and $T_N z_j = \lambda_j z_j$ are any two eigenpairs of T_N, and let $V = z_i z_j^T$. Then

$$\begin{aligned} T_N V + V T_N &= (T_N z_i) z_j^T + z_i (z_j^T T_N) \\ &= (\lambda_i z_i) z_j^T + z_i (z_j^T \lambda_j) \\ &= (\lambda_i + \lambda_j) z_i z_j^T \\ &= (\lambda_i + \lambda_j) V, \end{aligned} \tag{6.12}$$

so $V = z_i z_j^T$ is an "eigenvector" and $\lambda_i + \lambda_j$ is an eigenvalue. Since V has N^2 entries, we expect N^2 eigenvalues and eigenvectors, one for each pair of eigenvalues λ_i and λ_j of T_N. In particular, the smallest eigenvalue is $2\lambda_1$ and the largest eigenvalue is $2\lambda_N$, so the condition number is the same as in the one-dimensional case. We rederive this result below using the "$Ax = b$" format. See Figure 6.3 for plots of some eigenvectors, represented as surfaces defined by the matrix entries of $z_i z_j^T$.

Just as the eigenvalues and eigenvectors of $h^{-2}T_N$ were good approximations to the eigenvalues and eigenfunctions of one-dimensional Poisson's equation, the same is true of two-dimensional Poisson's equation, whose eigenvalues and eigenfunctions are as follows (see Question 6.3):

$$\left(-\frac{\partial^2}{\partial x^2} - \frac{\partial^2}{\partial y^2}\right) \sin(i\pi x)\sin(j\pi y) = (i^2\pi^2 + j^2\pi^2)\sin(i\pi x)\sin(j\pi y). \tag{6.13}$$

The second way to write the n equations represented by equation (6.10) as a single matrix equation is to write the unknowns v_{ij} in a single long N^2-by-1 vector. This requires us to choose an order for them, and we (somewhat arbitrarily) choose to number them as shown in Figure 6.4, columnwise from the upper left to the lower right.

For example, when $N = 3$ one gets a column vector $v \equiv [v_1, \ldots, v_9]^T$. If we number f accordingly, we can transform equation (6.10) to get

$$T_{3\times 3} \cdot \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ \vdots \\ \vdots \\ v_9 \end{bmatrix} \equiv \left[\begin{array}{ccc|ccc|ccc} 4 & -1 & & -1 & & & & & \\ -1 & 4 & -1 & & -1 & & & & \\ & -1 & 4 & & & -1 & & & \\ \hline -1 & & & 4 & -1 & & -1 & & \\ & -1 & & -1 & 4 & -1 & & -1 & \\ & & -1 & & -1 & 4 & & & -1 \\ \hline & & & -1 & & & 4 & -1 & \\ & & & & -1 & & -1 & 4 & -1 \\ & & & & & -1 & & -1 & 4 \end{array}\right] \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ \vdots \\ \vdots \\ v_9 \end{bmatrix}$$

$$= h^2 \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ \vdots \\ \vdots \\ f_9 \end{bmatrix}. \tag{6.14}$$

The -1's immediately next to the diagonal correspond to subtracting the top and bottom neighbors $-v_{i,j-1} - v_{i,j+1}$. The -1's farther away away from the diagonal correspond to subtracting the left and right neighbors $-v_{i-1,j} - v_{i+1,j}$. For general N, we confirm in the next section that we get an N^2-by-N^2 linear system

$$T_{N\times N} \cdot v = h^2 f, \tag{6.15}$$

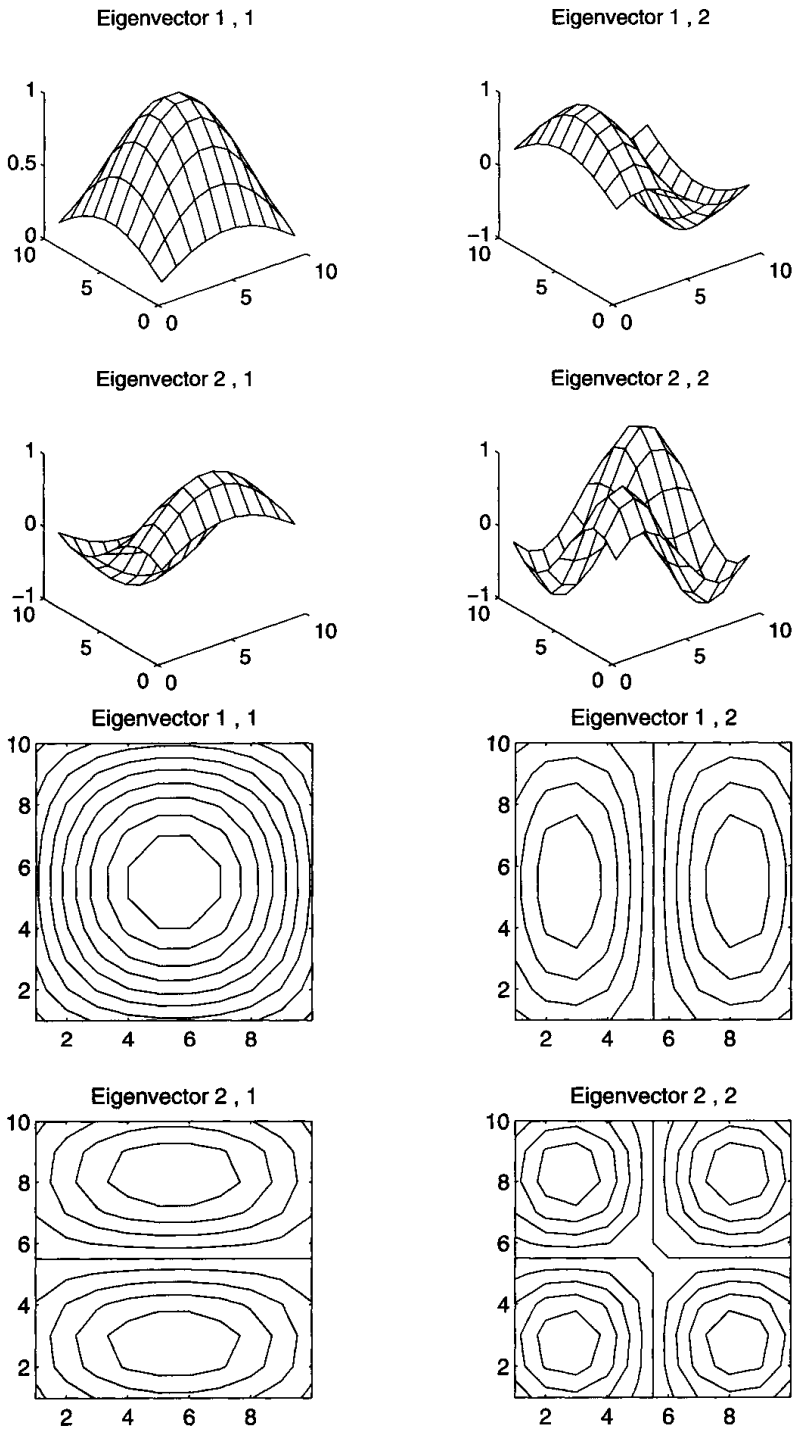


Fig. 6.3. *Three-dimensional and contour plots of first four eigenvectors of the* 10-*by*-10 *Poisson equation.*

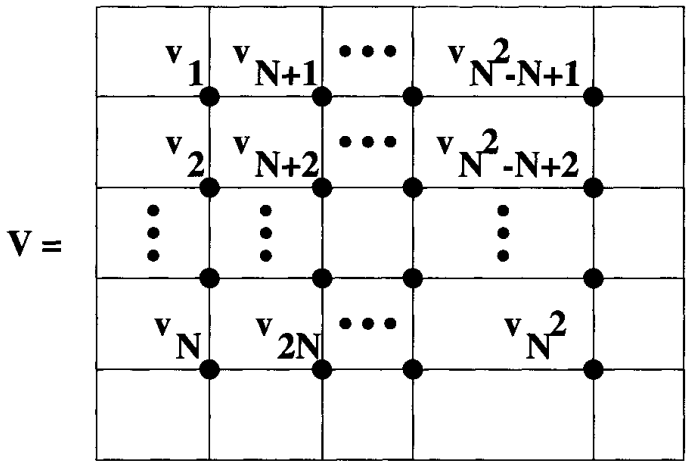


Fig. 6.4. *Numbering the unknowns in Poisson's equation.*

where $T_{N\times N}$ has N N-by-N blocks of the form $T_N + 2I_N$ on its diagonal and $-I_N$ blocks on its offdiagonals:

$$T_{N\times N} = \begin{bmatrix} T_N + 2I_N & -I_N & & \\ -I_N & \ddots & \ddots & \\ & \ddots & \ddots & -I_N \\ & & -I_N & T_N + 2I_N \end{bmatrix}. \tag{6.16}$$

6.3.3. Expressing Poisson's Equation with Kronecker Products

Here is a systematic way to derive equations (6.15) and (6.16) as well as to compute the eigenvalues and eigenvectors of $T_{N\times N}$. The method works equally well for Poisson's equation in three or more dimensions.

DEFINITION 6.1. *Let X be m-by-n. Then* vec(X) *is defined to be a column vector of size $m \cdot n$ made of the columns of X stacked atop one another from left to right.*

Note that N^2-by-1 vector v defined in Figure 6.4 can also be written $v =$ vec(V).

To express $T_{N\times N}$ as well as compute its eigenvalues and eigenvectors, we need to introduce *Kronecker products.*

DEFINITION 6.2. *Let A be an m-by-n matrix and B be a p-by-q matrix. Then $A \otimes B$, the* Kronecker product *of A and B, is the $(m \cdot p)$-by-$(n \cdot q)$ matrix*

$$\begin{bmatrix} a_{1,1} \cdot B & \dots & a_{1,n} \cdot B \\ \vdots & & \vdots \\ a_{m,1} \cdot B & \dots & a_{m,n} \cdot B \end{bmatrix}.$$

The following lemma tells us how to rewrite the Poisson equation in terms of Kronecker products and the vec($\cdot$) operator.

LEMMA 6.2. *Let A be m-by-m, B be n-by-n, and X and C be m-by-n. Then the following properties hold:*

1. $\text{vec}(AX) = (I_n \otimes A) \cdot \text{vec}(X)$.
2. $\text{vec}(XB) = (B^T \otimes I_m) \cdot \text{vec}(X)$.
3. *The Poisson equation $T_N V + V T_N = h^2 F$ is equivalent to*

$$T_{N\times N} \cdot \text{vec}(V) \equiv (I_N \otimes T_N + T_N \otimes I_N) \cdot \text{vec}(V) = h^2 \text{vec}(F). \quad (6.17)$$

Proof. We prove only part 3, leaving the other parts to Question 6.4. We start with the Poisson equation $T_N V + V T_N = h^2 F$ as expressed in equation (6.11), which is clearly equivalent to

$$\text{vec}(T_N V + V T_N) = \text{vec}(T_N V) + \text{vec}(V T_N) = \text{vec}(h^2 F).$$

By part 1 of the lemma

$$\text{vec}(T_N V) = (I_N \otimes T_N)\text{vec}(V).$$

By part 2 of the lemma and the symmetry of T_N,

$$\text{vec}(V T_N) = (T_N^T \otimes I_N)\text{vec}(V) = (T_N \otimes I_N)\text{vec}(V).$$

Adding the last two expressions completes the proof of part 3. □

The reader can confirm that the expression

$$\begin{aligned} T_{N\times N} &= I_N \otimes T_N + T_N \otimes I_N \\ &= \begin{bmatrix} T_N & & \\ & \ddots & \\ & & T_N \end{bmatrix} + \begin{bmatrix} 2I_N & -I_N & & \\ -I_N & \ddots & \ddots & \\ & \ddots & \ddots & -I_N \\ & & -I_N & 2I_N \end{bmatrix} \end{aligned}$$

from equation (6.17) agrees with equation (6.16).[26]

To compute the eigenvalues of matrices defined by Kronecker products, like $T_{N\times N}$, we need the following lemma, whose proof is also part of Question 6.4.

LEMMA 6.3. *The following facts about Kronecker products hold:*

1. *Assume that the products $A \cdot C$ and $B \cdot D$ are well defined. Then $(A \otimes B) \cdot (C \otimes D) = (A \cdot C) \otimes (B \cdot D)$.*

[26] We can use this formula to compute $T_{N\times N}$ in two lines of Matlab:

```
TN = 2*eye(N) - diag(ones(N-1,1),1) - diag(ones(N-1,1),-1);
TNxN = kron(eye(N),TN) + kron(TN,eye(N));
```

2. *If A and B are invertible, then* $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$.

3. $(A \otimes B)^T = A^T \otimes B^T$.

PROPOSITION 6.1. *Let $T_N = Z\Lambda Z^T$ be the eigendecomposition of T_N, with $Z = [z_1, \ldots, z_N]$ the orthogonal matrix whose columns are eigenvectors, and $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_N)$. Then the eigendecomposition of $T_{N\times N} = I \otimes T_N + T_N \otimes I$ is*

$$I \otimes T_N + T_N \otimes I = (Z \otimes Z) \cdot (I \otimes \Lambda + \Lambda \otimes I) \cdot (Z \otimes Z)^T. \tag{6.18}$$

$I \otimes \Lambda + \Lambda \otimes I$ is a diagonal matrix whose $(iN + j)$th diagonal entry, the (i, j)th eigenvalue of $T_{N\times N}$, is $\lambda_{i,j} = \lambda_i + \lambda_j$. $Z \otimes Z$ is an orthogonal matrix whose $(iN + j)$th column, the corresponding eigenvector, is $z_i \otimes z_j$.

Proof. From parts 1 and 3 of Lemma 6.3, it is easy to verify that $Z \otimes Z$ is orthogonal, since $(Z \otimes Z)(Z \otimes Z)^T = (Z \otimes Z)(Z^T \otimes Z^T) = (Z \cdot Z^T) \otimes (Z \cdot Z^T) = I \otimes I = I$. We can now verify equation (6.18):

$$\begin{aligned}
&(Z \otimes Z) \cdot (I \otimes \Lambda + \Lambda \otimes I) \cdot (Z \otimes Z)^T \\
&\quad = (Z \otimes Z) \cdot (I \otimes \Lambda + \Lambda \otimes I) \cdot (Z^T \otimes Z^T) \\
&\qquad\qquad \text{by part 3 of Lemma 6.3} \\
&\quad = (Z \cdot I \cdot Z^T) \otimes (Z \cdot \Lambda \cdot Z^T) + (Z \cdot \Lambda \cdot Z^T) \otimes (Z \cdot I \cdot Z^T) \\
&\qquad\qquad \text{by part 1 of Lemma 6.3} \\
&\quad = (I) \otimes (T_N) + (T_N) \otimes (I) \\
&\quad = T_{N\times N}.
\end{aligned}$$

Also, it is easy to verify that $I \otimes \Lambda + \Lambda \otimes I$ is diagonal, with diagonal entry $(iN + j)$ given by $\lambda_j + \lambda_i$, so that equation (6.18) really is the eigendecomposition of $T_{N\times N}$. Finally, from the definition of Kronecker product, one can see that column $iN + j$ of $Z \otimes Z$ is $z_i \otimes \dot{z}_j$. □

The reader can confirm that the eigenvector $z_i \otimes z_j = \mathrm{vec}(z_j z_i^T)$, thus matching the expression for an eigenvector in equation (6.12).

For a generalization of Proposition 6.1 to the matrix $A \otimes I + B^T \otimes I$, which arises when solving the Sylvester equation $AX - XB = C$, see Question 6.5 (and Question 4.6).

Similarly, Poisson's equation in three dimensions leads to

$$T_{N\times N\times N} \equiv T_N \otimes I_N \otimes I_N + I_N \otimes T_N \otimes I_N + I_N \otimes I_N \otimes T_N,$$

with eigenvalues all possible triple sums of eigenvalues of T_N, and eigenvector matrix $Z \otimes Z \otimes Z$. Poisson's equation in higher dimensions is represented analogously.

Method	Serial Time	Space	Direct or Iterative	Section
Dense Cholesky	n^3	n^2	D	2.7.1
Explicit inverse	n^2	n^2	D	
Band Cholesky	n^2	$n^{3/2}$	D	2.7.3
Jacobi's	n^2	n	I	6.5
Gauss–Seidel	n^2	n	I	6.5
Sparse Cholesky	$n^{3/2}$	$n \cdot \log n$	D	2.7.4
Conjugate gradients	$n^{3/2}$	n	I	6.6
Successive overrelaxation	$n^{3/2}$	n	I	6.5
SSOR with Chebyshev accel.	$n^{5/4}$	n	I	6.5
Fast Fourier transform	$n \cdot \log n$	n	D	6.7
Block cyclic reduction	$n \cdot \log n$	n	D	6.8
Multigrid	n	n	I	6.9
Lower bound	n	n		

Table 6.1. *Order of complexity of solving Poisson's equation on an N-by-N grid* ($n = N^2$).

6.4. Summary of Methods for Solving Poisson's Equation

Table 6.1 lists the costs of various direct and iterative methods for solving the model problem on an N-by-N grid. The variable $n = N^2$, the number of unknowns. Since direct methods provide the exact answer (in the absence of roundoff), whereas iterative methods provide only approximate answers, we must be careful when comparing their costs, since a low-accuracy answer can be computed more cheaply by an iterative method than a high-accuracy answer. Therefore, we compare costs, assuming that the iterative methods iterate often enough to make the error at most some fixed small value[27] (say, 10^{-6}).

The second and third columns of Table 6.1 give the number of arithmetic operations (or time) and space required on a serial machine. Column 4 indicates whether the method is direct (D) or iterative (I). All entries are meant in the $O(\cdot)$ sense; the constants depend on implementation details and the stopping criterion for the iterative methods (say, 10^{-6}). For example, the entry for Cholesky also applies to Gaussian elimination, since this changes the constant only by a factor of two. The last column indicates where the algorithm is discussed in the text.

The methods are listed in increasing order of speed, from slowest (dense

[27] Alternatively, we could iterate until the error is $O(h^2) = O((N+1)^{-2})$, the size of the truncation error. One can show that this would increase the costs of the iterative methods in Table 6.1 by a factor of $O(\log n)$.

Cholesky) to fastest (multigrid), ending with a lower bound applying to any method. The lower bound is n because at least one operation is required per solution component, since otherwise they could not all be different and also depend on the input. The methods are also, roughly speaking, in order of decreasing generality, with dense Cholesky applicable to any symmetric positive definite matrix and later algorithms applicable (or at least provably convergent) only for limited classes of matrices. In later sections we will describe the applicability of various methods in more detail.

The "explicit inverse" algorithm refers to precomputing the explicit inverse of $T_{N\times N}$, and computing $v = T_{N\times N}^{-1} f$ by a single matrix-vector multiplication (and not counting the flops to precompute $T_{N\times N}^{-1}$). Along with dense Cholesky, it uses n^2 space, vastly more than the other methods. It is not a good method. Band Cholesky was discussed in section 2.7.3; this is just Cholesky taking advantage of the fact that there are no entries to compute or store outside a band of $2N + 1$ diagonals.

Jacobi's and Gauss–Seidel are classical iterative methods and not particularly fast, but they form the basis for other faster methods: successive overrelaxation, symmetric successive overrelaxation, and multigrid, our fastest algorithm. So we will study them in some detail in section 6.5.

Sparse Cholesky refers to the algorithm discussed in section 2.7.4: it is an implementation of Cholesky that avoids storing or operating on the zero entries of $T_{N\times N}$ or its Cholesky factor. Furthermore, we are assuming the rows and columns of $T_{N\times N}$ have been "optimally ordered" to minimize work and storage (using nested dissection [112, 113]). While sparse Cholesky is reasonably fast on Poisson's equation in two dimensions, it it significantly worse in three dimensions (using $O(N^6) = O(n^2)$ time and $O(N^4) = O(n^{4/3})$ space), because there is more "fill-in" of zero entries during the algorithm.

Conjugate gradients are a representative of a much larger class of methods, called *Krylov subspace* methods, which are very widely applicable both for linear system solving and finding eigenvalues of sparse matrices. We will discuss these methods in more detail in section 6.6.

The fastest methods are block cyclic reduction, the fast Fourier transform (FFT), and multigrid. In particular, multigrid does only $O(1)$ operations per solution component, which is asymptotically optimal.

A final warning is that this table does not give a complete picture, since the constants are missing. For a particular size problem on a particular machine, one cannot immediately deduce which method is fastest. Still, it is clear that iterative methods such as Jacobi's, Gauss–Seidel, conjugate gradients, and successive overrelaxation are inferior to the FFT, block cyclic reduction, and multigrid for large enough n. But they remain of interest because they are building blocks for some of the faster methods and because they apply to larger classes of problems than the faster methods.

All of these algorithms can be implemented in parallel; see the lectures on PARALLEL_HOMEPAGE for details. It is interesting that, depending on

the parallel machine, multigrid may no longer be fastest. This is because on a parallel machine the time required for separate processors to communicate data to one another may be as costly as the floating point operations, and other algorithms may communicate less than multigrid.

6.5. Basic Iterative Methods

In this section we will talk about the most basic iterative methods:

Jacobi's,
Gauss–Seidel,
successive overrelaxation (SOR(ω)),
Chebyshev acceleration with symmetric successive overrelaxation (SSOR(ω)).

These methods are also discussed and their implementations are provided at NETLIB/templates.

Given x_0, these methods generate a sequence x_m converging to the solution $A^{-1}b$ of $Ax = b$, where x_{m+1} is cheap to compute from x_m.

DEFINITION 6.3. *A* splitting *of A is a decomposition $A = M - K$, with M nonsingular.*

A splitting yields an iterative method as follows: $Ax = Mx - Kx = b$ implies $Mx = Kx + b$ or $x = M^{-1}Kx + M^{-1}b \equiv Rx + c$. So we can take $x_{m+1} = Rx_m + c$ as our iterative method. Let us see when it converges.

LEMMA 6.4. *Let $\|\cdot\|$ be any operator norm ($\|R\| \equiv \max_{x\neq 0} \frac{\|Rx\|}{\|x\|}$). If $\|R\| < 1$, then $x_{m+1} = Rx_m + c$ converges for any x_0.*

Proof. Subtract $x = Rx+c$ from $x_{m+1} = Rx_m+c$ to get $x_{m+1}-x = R(x_m-x)$. Thus $\|x_{m+1} - x\| \leq \|R\| \cdot \|x_m - x\| \leq \|R\|^{m+1} \cdot \|x_0 - x\|$, which converges to 0 since $\|R\| < 1$. □

Our ultimate convergence criterion will depend on the following property of R.

DEFINITION 6.4. *The* spectral radius *of R is $\rho(R) \equiv \max|\lambda|$, where the maximum is taken over all eigenvalues λ of R.*

LEMMA 6.5. *For all operator norms $\rho(R) \leq \|R\|$. For all R and for all $\epsilon > 0$ there is an operator norm $\|\cdot\|_\star$ such that $\|R\|_\star \leq \rho(R) + \epsilon$. The norm $\|\cdot\|_\star$ depends on both R and ϵ.*

Proof. To show $\rho(R) \leq \|R\|$ for any operator norm, let x be an eigenvector for λ, where $\rho(R) = |\lambda|$ and so $\|R\| = \max_{y\neq 0} \frac{\|Ry\|}{\|y\|} \geq \frac{\|Rx\|}{\|x\|} = \frac{\|\lambda x\|}{\|x\|} = |\lambda|$.

To construct an operator norm $\|\cdot\|_\star$ such that $\|R\|_\star \leq \rho(R)+\epsilon$, let $S^{-1}RS = J$ be in Jordan form. Let $D_\epsilon = \text{diag}(1, \epsilon, \epsilon^2, \ldots, \epsilon^{n-1})$. Then

$$
\begin{aligned}
(SD_\epsilon)^{-1}R(SD_\epsilon) &= D_\epsilon{}^{-1}JD_\epsilon \\
&= \left[\begin{array}{ccccc|ccc|c}
\lambda_1 & \epsilon & & & & & & & \\
& \ddots & \ddots & & & & & & \\
& & \ddots & \epsilon & & & & & \\
& & & \lambda_1 & & & & & \\
\hline
& & & & & \lambda_2 & \epsilon & & \\
& & & & & & \ddots & \ddots & \\
& & & & & & & \epsilon & \\
& & & & & & & \lambda_2 & \\
\hline
& & & & & & & & \ddots
\end{array}\right],
\end{aligned}
$$

i.e., a "Jordan form" with ϵ's above the diagonal. Now use the vector norm $\|x\|_\star \equiv \|(SD_\epsilon)^{-1}x\|_\infty$ to generate the operator norm

$$
\begin{aligned}
\|R\|_\star &\equiv \max_{x\neq 0} \frac{\|Rx\|_\star}{\|x\|_\star} \\
&= \max_{x\neq 0} \frac{\|(SD_\epsilon)^{-1}Rx\|_\infty}{\|(SD_\epsilon)^{-1}x\|_\infty} \\
&= \max_{y\neq 0} \frac{\|(SD_\epsilon)^{-1}R(SD_\epsilon)y\|_\infty}{\|y\|_\infty} \\
&= \|(SD_\epsilon)^{-1}R(SD_\epsilon)\|_\infty \\
&= \max_i |\lambda_i| + \epsilon \\
&= \rho(R) + \epsilon. \quad \square
\end{aligned}
$$

THEOREM 6.1. *The iteration $x_{m+1} = Rx_m + c$ converges to the solution of $Ax = b$ for all starting vectors x_0 and for all b if and only if $\rho(R) < 1$.*

Proof. If $\rho(R) \geq 1$, choose $x_0 - x$ to be an eigenvector of R with eigenvalue λ where $|\lambda| = \rho(R)$. Then

$$(x_{m+1} - x) = R(x_m - x) = \cdots = R^{m+1}(x_0 - x) = \lambda^{m+1}(x_0 - x)$$

will not approach 0. If $\rho(R) < 1$, use Lemma 6.5 to choose an operator norm so $\|R\|_\star < 1$ and then apply Lemma 6.4 to conclude that the method converges. $\square$

DEFINITION 6.5. *The* rate of convergence *of $x_{m+1} = Rx_m + c$ is $r(R) \equiv -\log_{10}\rho(R)$.*

$r(R)$ is the increase in the number of correct decimal places in the solution per iteration, since $\log_{10} \|x_m - x\|_\star - \log_{10} \|x_{m+1} - x\|_\star \geq r(R) + O(\epsilon)$. The smaller is $\rho(R)$, the higher is the rate of convergence, i.e., the greater is the number of correct decimal places computed per iteration.

Our goal is now to choose a splitting $A = M - K$ so that both

(1) $Rx = M^{-1}Kx$ and $c = M^{-1}b$ are easy to evaluate,

(2) $\rho(R)$ is small.

We will need to balance these conflicting goals. For example, choosing $M = I$ is good for goal (1) but may not make $\rho(R) < 1$. On the other hand, choosing $M = A$ and $K = 0$ is good for goal (2) but probably bad for goal (1).

The splittings for the methods discussed in this section all share the following notation. When A has no zeros on its diagonal, we write

$$A = D - \tilde{L} - \tilde{U} = D(I - L - U), \tag{6.19}$$

where D is the diagonal of A, $-\tilde{L}$ is the strictly lower triangular part of A, $DL = \tilde{L}$, $-\tilde{U}$ is the strictly upper triangular part of A, and $DU = \tilde{U}$.

6.5.1. Jacobi's Method

Jacobi's method can be described as repeatedly looping through the equations, changing variable j so that equation j is satisfied exactly. Using the notation of equation (6.19), the splitting for Jacobi's method is $A = D-(\tilde{L}+\tilde{U})$; we denote $R_J \equiv D^{-1}(\tilde{L}+\tilde{U}) = L+U$ and $c_J \equiv D^{-1}b$, so we can write one step of Jacobi's method as $x_{m+1} = R_J x_m + c_J$. To see that this formula corresponds to our first description of Jacobi's method, note that it implies $Dx_{m+1} = (\tilde{L} + \tilde{U})x_m + b$, or $a_{jj}x_{m+1,j} = -\sum_{k \neq j} a_{jk}x_{m,k} + b_j$, or $a_{jj}x_{m+1,j} + \sum_{k \neq j} a_{jk}x_{m,k} = b_j$.

ALGORITHM 6.1. *One step of Jacobi's method:*

for $j = 1$ *to* n
  $x_{m+1,j} = \frac{1}{a_{jj}}(b_j - \sum_{k \neq j} a_{jk}x_{m,k})$
end for

In the special case of the model problem, the implementation of Jacobi's algorithm simplifies as follows. Working directly from equation (6.10) and letting $v_{m,i,j}$ denote the mth value of the solution at grid point i, j, Jacobi's method becomes the following.

ALGORITHM 6.2. *One step of Jacobi's method for two-dimensional Poisson's equation:*

for $i = 1$ *to* N
  for $j = 1$ *to* N
    $v_{m+1,i,j} = (v_{m,i-1,j} + v_{m,i+1,j} + v_{m,i,j-1} + v_{m,i,j+1} + h^2 f_{ij})/4$

end for
end for

In other words, at each step the new value of v_{ij} is obtained by "averaging" its neighbors with $h^2 f_{ij}$. Note that all new values $v_{m+1,i,j}$ may be computed independently of one another. Indeed, Algorithm 6.2 can be implemented in one line of Matlab if the $v_{m+1,i,j}$ are stored in a square array $\hat{V}$ that includes an extra first and last row of zeros and first and last column of zeros (see Question 6.6).

6.5.2. Gauss–Seidel Method

The motivation for this method is that at the jth step of the loop for Jacobi's method, we have improved values of the first $j-1$ components of the solution, so we should use them in the sum.

ALGORITHM 6.3. *One step of the Gauss–Seidel method:*

for $j = 1$ *to* n

$$x_{m+1,j} = \frac{1}{a_{jj}} \left(b_j - \underbrace{\sum_{k=1}^{j-1} a_{jk} x_{m+1,k}}_{\text{updated } x\text{'s}} - \underbrace{\sum_{k=j+1}^{n} a_{jk} x_{m,k}}_{\text{older } x\text{'s}} \right)$$

end for

For the purpose of later analysis, we want to write this algorithm in the form $x_{m+1} = R_{GS} x_m + c_{GS}$. To this end, note that it can first be rewritten as

$$\sum_{k=1}^{j} a_{jk} x_{m+1,k} = -\sum_{k=j+1}^{n} a_{jk} x_{m,k} + b_j. \tag{6.20}$$

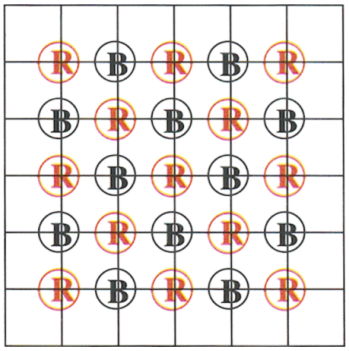
Then using the notation of equation (6.19), we can rewrite equation (6.20) as $(D - \tilde{L}) x_{m+1} = \tilde{U} x_m + b$ or

$$\begin{aligned} x_{m+1} &= (D - \tilde{L})^{-1} \tilde{U} x_m + (D - \tilde{L})^{-1} b \\ &= (I - L)^{-1} U x_m + (I - L)^{-1} D^{-1} b \\ &\equiv R_{GS} x_m + c_{GS}. \end{aligned}$$

As with Jacobi's method, we consider how to implement the Gauss–Seidel method for our model problem. In principle it is quite similar, except that we have to keep track of which variables are new (numbered $m+1$) and which are old (numbered m). But depending on the order in which we loop through the grid points i, j, we will get different (and valid) implementations of the

Gauss–Seidel method. This is unlike Jacobi's method, in which the order in which we update the variables is irrelevant. For example, if we update $v_{m,1,1}$ first (before any other $v_{m,i,j}$), then all its neighboring values are necessarily old. But if we update $v_{m,1,1}$ last, then all its neighboring values are necessarily new, so we get a different value for $v_{m,1,1}$. Indeed, there are as many possible implementations of the Gauss–Seidel method as there are ways to order N^2 variables (namely, $N^2!$). But of all these orderings, two are of most interest. The first is the ordering shown in Figure 6.4; this is called the *natural ordering.*

The second ordering is called *red-black ordering.* It is important because our best convergence results in sections 6.5.4 and 6.5.5 depend on it. To explain red-black ordering, consider the chessboard-like coloring of the grid of unknowns below; the Ⓑ nodes correspond to the black squares on a chessboard, and the Ⓡ nodes correspond to the red squares.



The red-black ordering is to order the red nodes before the black nodes. Note that red nodes are adjacent to only black nodes. So if we update all the red nodes first, they will use only old data from the black nodes. Then when we update the black nodes, which are only adjacent to red nodes, they will use only new data from the red nodes. Thus the algorithm becomes the following.

ALGORITHM 6.4. *One step of the Gauss–Seidel method on two-dimensional Poisson's equation with red-black ordering:*

for all nodes i, j *that are red* (Ⓡ)
$\quad v_{m+1,i,j} = (v_{m,i-1,j} + v_{m,i+1,j} + v_{m,i,j-1} + v_{m,i,j+1} + h^2 f_{ij})/4$
end for
for all nodes i, j *that are black* (Ⓑ)
$\quad v_{m+1,i,j} = (v_{m+1,i-1,j} + v_{m+1,i+1,j} + v_{m+1,i,j-1} + v_{m+1,i,j+1} + h^2 f_{ij})/4$
end for

6.5.3. Successive Overrelaxation

We refer to this method as SOR(ω), where ω is the *relaxation parameter.* The motivation is to improve the Gauss–Seidel loop by taking an appropriate

weighted average of the $x_{m+1,j}$ and $x_{m,j}$:

$$\text{SOR's} \quad x_{m+1,j} = (1-\omega)x_{m,j} + \omega x_{m+1,j},$$

yielding the following algorithm.

ALGORITHM 6.5. *SOR:*

for $j = 1$ *to* n

$x_{m+1,j} = (1-\omega)x_{m,j} + \frac{\omega}{a_{jj}}\left[b_j - \sum_{k=1}^{j-1} a_{jk}x_{m+1,k} - \sum_{k=j+1}^{n} a_{jk}x_{m,k}\right]$

end for

We may rearrange this to get, for $j = 1$ to n,

$$a_{jj}x_{m+1,j} + \omega\sum_{k=1}^{j-1} a_{jk}x_{m+1,k} = (1-\omega)a_{jj}x_{m,j} - \omega\sum_{k=j+1}^{n} a_{jk}x_{m,k} + \omega b_j$$

or, again using the notation of equation (6.19),

$$(D - \omega\tilde{L})x_{m+1} = ((1-\omega)D + \omega\tilde{U})x_m + \omega b$$

or

$$\begin{aligned} x_{m+1} &= (D-\omega\tilde{L})^{-1}((1-\omega)D + \omega\tilde{U})x_m + \omega(D-\omega\tilde{L})^{-1}b \\ &= (I-\omega L)^{-1}((1-\omega)I + \omega U)x_m + \omega(I-\omega L)^{-1}D^{-1}b \\ &\equiv R_{SOR(\omega)}x_m + c_{SOR(\omega)}. \end{aligned} \tag{6.21}$$

We distinguish three cases, depending on the values of ω: $\omega = 1$ is equivalent to the Gauss–Seidel method, $\omega < 1$ is called *underrelaxation*, and $\omega > 1$ is called *overrelaxation*. A somewhat superficial motivation for overrelaxation is that if the direction from x_m to x_{m+1} is a good direction in which to move the solution, then moving $\omega > 1$ times as far in that direction is better.

In the next two sections, we will show how to pick the optimal ω for the model problem. This optimality depends on using red-black ordering.

ALGORITHM 6.6. *One step of SOR(ω) on two-dimensional Poisson's equation with red-black ordering:*

for all nodes i, j *that are red* (R)

$v_{m+1,i,j} = (1-\omega)v_{m,i,j} +$
$\omega(v_{m,i-1,j} + v_{m,i+1,j} + v_{m,i,j-1} + v_{m,i,j+1} + h^2 f_{ij})/4$

end for

for all nodes i, j *that are black* (B)

$v_{m+1,i,j} = (1-\omega)v_{m,i,j} +$
$\omega(v_{m+1,i-1,j} + v_{m+1,i+1,j} + v_{m+1,i,j-1} + v_{m+1,i,j+1} + h^2 f_{ij})/4$

end for

6.5.4. Convergence of Jacobi's, Gauss–Seidel, and SOR(ω) Methods on the Model Problem

It is easy to compute how fast Jacobi's method converges on the model problem, since the corresponding splitting is $T_{N\times N} = 4I - (4I - T_{N\times N})$, and so $R_J = (4I)^{-1}(4I - T_{N\times N}) = I - T_{N\times N}/4$. Thus the eigenvalues of R_J are $1 - \lambda_{i,j}/4$, where the $\lambda_{i,j}$ are the eigenvalues of $T_{N\times N}$:

$$\lambda_{i,j} = \lambda_i + \lambda_j = 4 - 2\left(\cos\frac{\pi i}{N+1} + \cos\frac{\pi j}{N+1}\right).$$

$\rho(R_J)$ is the largest of $|1 - \lambda_{i,j}/4|$, namely,

$$\rho(R_J) = |1 - \lambda_{1,1}/4| = |1 - \lambda_{N,N}/4| = \cos\frac{\pi}{N+1} \approx 1 - \frac{\pi^2}{2(N+1)^2}.$$

Note that as N grows and T becomes more ill-conditioned, the spectral radius $\rho(R_J)$ approaches 1. Since the error is multiplied by the spectral radius at each step, convergence slows down. To estimate the speed of convergence more precisely, let us compute the number m of Jacobi iterations required to decrease the error by $e^{-1} = \exp(-1)$. Then m must satisfy $(\rho(R_J))^m = e^{-1}$, or $(1 - \frac{\pi^2}{2(N+1)^2})^m = e^{-1}$, or $m \approx \frac{2(N+1)^2}{\pi^2} = O(N^2) = O(n)$. Thus the number of iterations is proportional to the number of unknowns. Since one step of Jacobi costs $O(1)$ to update each solution component or $O(n)$ to update all of them, it costs $O(n^2)$ to decrease the error by e^{-1} (or by any constant factor less than 1). This explains the entry for Jacobi's method in Table 6.1.

This is a common phenomenon: the more ill-conditioned the original problem, the more slowly most iterative methods converge. There are important exceptions, such as multigrid and domain decomposition, which we discuss later.

In the next section we will show, provided that the variables in Poisson's equation are updated in red-black order (see Algorithm 6.4 and Corollary 6.1), that $\rho(R_{GS}) = \rho(R_J)^2 = \cos^2\frac{\pi}{N+1}$. In other words, one Gauss–Seidel step decreases the error as much as two Jacobi steps. This is a general phenomenon for matrices arising from approximating differential equations with certain finite difference approximations. This also explains the entry for the Gauss–Seidel method in Table 6.1; since it is only twice as fast as Jacobi, it still has the same complexity in the $O(\cdot)$ sense.

For the same red-black update order (see Algorithm 6.6 and Theorem 6.7), we will also show that for the relaxation parameter $1 < \omega = 2/(1+\sin\frac{\pi}{N+1}) < 2$

$$\rho(R_{SOR(\omega)}) = \frac{\cos^2\frac{\pi}{N+1}}{(1+\sin\frac{\pi}{N+1})^2} \approx 1 - \frac{2\pi}{N+1} \text{ for large } N.$$

This is in contrast to $\rho(R) = 1 - O(\frac{1}{N^2})$ for R_J and R_{GS}. This is the optimal value for ω; i.e., it minimizes $R_{SOR(\omega)}$. With this choice of ω, SOR(ω) is

approximately N times faster than Jacobi's or the Gauss–Seidel method, since if SOR(ω) takes j steps to decrease the error as much as k steps of Jacobi's or the Gauss–Seidel method, then $(1-\frac{1}{N^2})^k \approx (1-\frac{1}{N})^j$, implying $1-\frac{k}{N^2} \approx 1-\frac{j}{N}$ or $k \approx j \cdot N$. This lowers the complexity of SOR(ω) from $O(n^2)$ to $O(n^{3/2})$, as shown in Table 6.1.

In the next section we will show generally for certain finite difference matrices how to choose ω to minimize $\rho(R_{SOR(\omega)})$.

6.5.5. Detailed Convergence Criteria for Jacobi's, Gauss–Seidel, and SOR(ω) Methods

We will give a sequence of criteria that guarantee the convergence of these methods. The first criterion is simple to evaluate but is not always applicable, in particular not to the model problem. Then we give several more complicated criteria, which place stronger conditions on the matrix A but in return give more information about convergence. These more complicated criteria are tailored to fit the matrices arising from discretizing certain kinds of partial differential equations such as Poisson's equation.

Here is a summary of the results of this section:

1. If A is strictly row diagonally dominant (Definition 6.6), then Jacobi's and Gauss–Seidel methods both converge, and the Gauss–Seidel method is faster (Theorem 6.2). Strict row diagonal dominance means that each diagonal entry of A is larger in magnitude than the sum of the magnitudes of the other entries in its row.

2. Since our model problem is not strictly row diagonally dominant, the last result does not apply. So we ask for a weaker form of diagonal dominance (Definition 6.11) but impose a condition called *irreducibility* on the pattern of nonzero entries of A (Definition 6.7) to prove convergence of Jacobi's and Gauss–Seidel methods. The Gauss–Seidel method again converges faster than Jacobi's method (Theorem 6.3). This result applies to the model problem.

3. Turning to SOR(ω), we show that $0 < \omega < 2$ is necessary for convergence (Theorem 6.4). If A is also positive definite (like the model problem), $0 < \omega < 2$ is also sufficient for convergence (Theorem 6.5).

4. To quantitatively compare Jacobi's, Gauss–Seidel, and SOR(ω) methods, we make one more assumption about the pattern of nonzero entries of A. This property is called *property A* (Definition 6.12) and is equivalent to saying that the *graph of the matrix* is *bipartite*. Property A essentially says that we can update the variables using red-black ordering. Given property A there is a simple algebraic formula relating the eigenvalues of R_J, R_{GS}, and $R_{SOR(\omega)}$ (Theorem 6.6), which lets us compare their rates

of convergence. This formula also lets us compute the optimal ω that makes SOR(ω) converge as fast as possible (Theorem 6.7).

DEFINITION 6.6. *A is* strictly row diagonally dominant *if* $|a_{ii}| > \sum_{j \neq i} |a_{ij}|$ *for all* i.

THEOREM 6.2. *If A is strictly row diagonally dominant, both Jacobi's and Gauss–Seidel methods converge. In fact* $\|R_{GS}\|_\infty \leq \|R_J\|_\infty < 1$.

The inequality $\|R_{GS}\|_\infty \leq \|R_J\|_\infty$ implies that one step of the worst problem for the Gauss–Seidel method converges at least as fast as one step of the worst problem for Jacobi's method. It does *not* guarantee that for any particular $Ax = b$, the Gauss–Seidel method will be faster than Jacobi's method; Jacobi's method could "accidentally" have a smaller error at some step.

Proof. Again using the notation of equation (6.19), we write $R_J = L + U$ and $R_{GS} = (I - L)^{-1}U$. We want to prove

$$\|R_{GS}\|_\infty = \||R_{GS}|e\|_\infty \leq \||R_J|e\|_\infty = \|R_J\|_\infty, \tag{6.22}$$

where $e = [1, \ldots, 1]^T$ is the vector of all ones. Inequality (6.22) will be true if can prove the stronger componentwise inequality

$$|(I - L)^{-1}U| \cdot e = |R_{GS}| \cdot e \leq |R_J| \cdot e = (|L| + |U|) \cdot e. \tag{6.23}$$

Since

$$\begin{aligned}
|(I - L)^{-1}U| \cdot e &\leq |(I - L)^{-1}| \cdot |U| \cdot e && \text{by the triangle inequality} \\
&= \left| \sum_{i=0}^{n-1} L^i \right| \cdot |U| \cdot e && \text{since } L^n = 0 \\
&\leq \sum_{i=0}^{n-1} |L|^i \cdot |U| \cdot e && \text{by the triangle inequality} \\
&= (I - |L|)^{-1} \cdot |U| \cdot e && \text{since } |L|^n = 0,
\end{aligned}$$

inequality (6.23) will be true if can prove the even stronger componentwise inequality

$$(I - |L|)^{-1} \cdot |U| \cdot e \leq (|L| + |U|) \cdot e. \tag{6.24}$$

Since all entries of $(I - |L|)^{-1} = \sum_{i=0}^{n-1} |L|^i$ are nonnegative, inequality (6.24) will be true if we can prove

$$|U| \cdot e \leq (I - |L|) \cdot (|L| + |U|) \cdot e = (|L| + |U| - |L|^2 - |L| \cdot |U|) \cdot e$$

or

$$0 \leq (|L| - |L|^2 - |L| \cdot |U|) \cdot e = |L| \cdot (I - |L| - |U|) \cdot e. \tag{6.25}$$

Since all entries of $|L|$ are nonnegative, inequality (6.25) will be true if we can prove

$$0 \leq (I - |L| - |U|) \cdot e \quad \text{or} \quad |R_J| \cdot e = (|L| + |U|)e \leq e. \tag{6.26}$$

Finally, inequality (6.26) is true because by assumption $\||R_J| \cdot e\|_\infty = \|R_J\|_\infty = \rho < 1$. □

An analogous result holds when A is strictly column diagonally dominant (i.e., A^T is strictly row diagonally dominant).

The reader may easily confirm that this simple criterion does not apply to the model problem, so we need to weaken the assumption of strict diagonal dominance. Doing so requires looking at the *graph properties* of a matrix.

DEFINITION 6.7. *A is an* irreducible matrix *if there is no permutation matrix P such that*

$$PAP^T = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline 0 & A_{22} \end{array}\right].$$

We connect this definition to *graph theory* as follows.

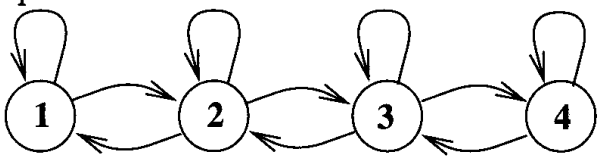
DEFINITION 6.8. *A* directed graph *is a finite collection of* nodes *connected by a finite collection of* directed edges, *i.e., arrows from one node to another. A* path *in a directed graph is a sequence of nodes $n_1, \ldots, n_m$ with an edge from each n_i to n_{i+1}. A* self edge *is an edge from a node to itself.*

DEFINITION 6.9. *The* directed graph of A, $G(A)$, *is a graph with nodes $1, 2, \ldots, n$ and an edge from node i to node j if and only if $a_{ij} \neq 0$.*

EXAMPLE 6.1. The matrix

$$A = \begin{bmatrix} 2 & -1 & & \\ -1 & 2 & -1 & \\ & -1 & 2 & -1 \\ & & -1 & 2 \end{bmatrix}$$

has the directed graph



◇

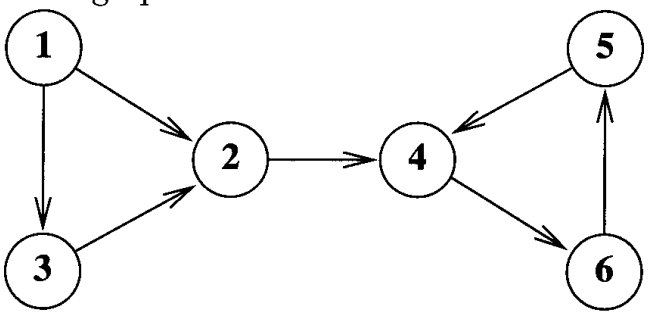
DEFINITION 6.10. *A directed graph is called* strongly connected *if there exists a path from every node i to every node j. A* strongly connected component *of a directed graph is a subgraph (a subset of the nodes with all edges connecting them) which is strongly connected and cannot be made larger yet still be strongly connected.*

EXAMPLE 6.2. The graph in Example 6.1 is strongly connected. ◇

EXAMPLE 6.3. Let

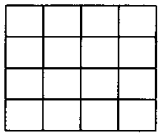
$$A = \left[\begin{array}{cc|cc} 1 & 1 & & \\ & & 1 & \\ 1 & & & \\ \hline & & & 1 \\ & & 1 & \\ & & & 1 \end{array}\right],$$

which has the directed graph



This graph is not strongly connected, since there is no path to node 1 from anywhere else. Nodes 4, 5, and 6 form a strongly connected component, since there is a path from any one of them to any other. ◇

EXAMPLE 6.4. The graph of the model problem is strongly connected. The graph is essentially



except that each edge in the grid represents two edges (one in each direction), and the self edges are not shown. ◇

LEMMA 6.6. *A is irreducible if and only if $G(A)$ is strongly connected.*

Proof. If $A = \begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{bmatrix}$ is reducible, then there is clearly no way to get from the nodes corresponding to A_{22} back to the ones corresponding to A_{11}; i.e., $G(A)$ is not strongly connected. Similarly, if $G(A)$ is not strongly connected, renumber the rows (and columns) so that all the nodes in a particular strongly connected component come first; then the matrix PAP^T will be block upper triangular. □

EXAMPLE 6.5. The matrix A in Example 6.3 is reducible.

DEFINITION 6.11. *A is* weakly row diagonally dominant *if for all i, $|a_{ii}| \geq \sum_{k \neq i} |a_{ik}|$ with strict inequality at least once.*

THEOREM 6.3. *If A is irreducible and weakly row diagonally dominant, then both Jacobi's and Gauss–Seidel methods converge, and $\rho(R_{GS}) < \rho(R_J) < 1$.*

For a proof of this theorem, see [249].

EXAMPLE 6.6. The model problem is weakly diagonally dominant and irreducible but not strongly diagonally dominant. (The diagonal is 4, and the offdiagonal sums are either 2, 3, or 4.) So Jacobi's and Gauss–Seidel methods converge on the model problem. ◇

Despite the above results showing that under certain conditions the Gauss–Seidel method is faster than Jacobi's method, no such general result holds. This is because there are nonsymmetric matrices for which Jacobi's method converges and the Gauss–Seidel method diverges, as well as matrices for which the Gauss–Seidel method converges and Jacobi's method diverges [249].

Now we consider the convergence of $SOR(\omega)$ [249]. Recall its definition:

$$R_{SOR(\omega)} = (I - \omega L)^{-1}((1-\omega)I + \omega U).$$

THEOREM 6.4. $\rho(R_{SOR(\omega)}) \geq |\omega - 1|$. *Therefore $0 < \omega < 2$ is required for convergence.*

Proof. Write the characteristic polynomial of $R_{SOR(\omega)}$ as $\varphi(\lambda) = \det(\lambda I - R_{SOR(\omega)}) = \det((I - \omega L)(\lambda I - R_{SOR(\omega)})) = \det((\lambda + \omega - 1)I - \omega\lambda L - \omega U)$ so that

$$\varphi(0) = \pm \prod_{i=1}^{n} \lambda_i(R_{SOR(\omega)}) = \pm \det((\omega - 1)I) = \pm(\omega - 1)^n,$$

implying $\max_i |\lambda_i(R_{SOR(\omega)})| \geq |\omega - 1|$. □

THEOREM 6.5. *If A is symmetric positive definite, then $\rho(R_{SOR(\omega)}) < 1$ for all $0 < \omega < 2$, so $SOR(\omega)$ converges for all $0 < \omega < 2$. Taking $\omega = 1$, we see that the Gauss–Seidel method also converges.*

Proof. There are two steps. We abbreviate $R_{SOR(\omega)} = R$. Using the notation of equation (6.19), let $M = \omega^{-1}(D - \omega\tilde{L})$. Then we

(1) define $Q = A^{-1}(2M - A)$ and show $\Re\lambda_i(Q) > 0$ for all i,

(2) show that $R = (Q - I)(Q + I)^{-1}$, implying $|\lambda_i(R)| < 1$ for all i.

For (1), note that $Qx = \lambda x$ implies $(2M - A)x = \lambda Ax$ or $x^*(2M - A)x = \lambda x^*Ax$. Add this last equation to its conjugate transpose to get $x^*(M + M^* - A)x = (\Re\lambda)(x^*Ax)$. So $\Re\lambda = x^*(M + M^* - A)x/x^*Ax = x^*(\frac{2}{\omega} - 1)Dx/x^*Ax > 0$ since A and $(\frac{2}{\omega} - 1)D$ are positive definite.

To prove (2), note that $(Q-I)(Q+I)^{-1} = (2A^{-1}M - 2I)(2A^{-1}M)^{-1} = I - M^{-1}A = R$, so by the spectral mapping theorem (Question 4.5)

$$|\lambda(R)| = \left|\frac{\lambda(Q)-1}{\lambda(Q)+1}\right| = \left|\frac{(\Re\lambda(Q)-1)^2 + (\Im\lambda(Q))^2}{(\Re\lambda(Q)+1)^2 + (\Im\lambda(Q))^2}\right|^{1/2} < 1. \quad \square$$

Together, Theorems 6.4 and 6.5 imply that if A is symmetric positive definite, then SOR(ω) converges if and only if $0 < \omega < 2$.

EXAMPLE 6.7. The model problem is symmetric positive definite, so SOR(ω) converges for $0 < \omega < 2$. $\diamond$

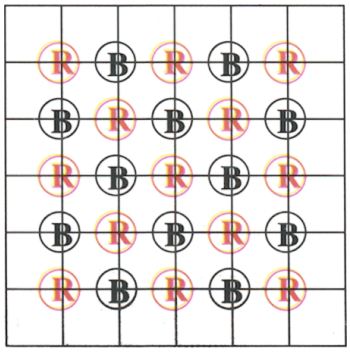
For the final comparison of the costs of Jacobi's, Gauss–Seidel, and SOR(ω) methods on the model problem we impose another graph theoretic condition on A that often arises from certain discretized partial differential equations, such as Poisson's equation. This condition will let us compute $\rho(R_{GS})$ and $\rho(R_{SOR(\omega)})$ explicitly in terms of $\rho(R_J)$.

DEFINITION 6.12. *A matrix T has* property A *if there exists a permutation P such that*

$$PTP^T = \left[\begin{array}{c|c} T_{11} & T_{12} \\ \hline T_{21} & T_{22} \end{array}\right],$$

where T_{11} and T_{22} are diagonal. In other words in the graph $G(A)$ the nodes divide into two sets $S_1 \cup S_2$, where there are no edges between two nodes both in S_1 or both in S_2 (ignoring self edges); such a graph is called bipartite.

EXAMPLE 6.8. *Red-black ordering* for the model problem. This was introduced in section 6.5.2, using the following chessboard-like depiction of the graph of the model problem: The black Ⓑ nodes are in S_1, and the red Ⓡ nodes are in S_2.



As described in section 6.5.2, each equation in the model problem relates the value at a grid point to the values at its left, right, top, and bottom neighbors, which are colored differently from the grid point in the middle. In other words, there is no direct connection from an Ⓡ node to an Ⓡ node or from a Ⓑ node to a Ⓑ node. So if we number the red nodes before the

black nodes, the matrix will be in the form demanded by Definition 6.12. For example, in the case of a 3-by-3 grid, we get the following:

$$P\left[\begin{array}{ccc|ccc|ccc} 4 & -1 & & -1 & & & & & \\ -1 & 4 & -1 & & -1 & & & & \\ & -1 & 4 & & & -1 & & & \\ \hline -1 & & & 4 & -1 & & -1 & & \\ & -1 & & -1 & 4 & -1 & & -1 & \\ & & -1 & & -1 & 4 & & & -1 \\ \hline & & & -1 & & & 4 & -1 & \\ & & & & -1 & & -1 & 4 & -1 \\ & & & & & -1 & & -1 & 4 \end{array}\right] P^T$$

$$= \left[\begin{array}{ccc|ccc|ccc} 4 & & & & & & -1 & -1 & \\ & 4 & & & & & -1 & & -1 \\ & & 4 & & & & -1 & -1 & -1 \\ & & & 4 & & & & -1 & \\ & & & & 4 & & & -1 & -1 \\ & & & & & 4 & & & -1 \\ \hline -1 & -1 & -1 & & & & 4 & & \\ -1 & & -1 & -1 & & & & 4 & \\ & -1 & -1 & & -1 & & & & 4 \\ & & -1 & -1 & -1 & & & & \end{array}\right] . \quad \diamond$$

Now suppose that T has property A, so we can write (where $D_i = T_{ii}$ is diagonal)

$$\begin{aligned} PTP^T &= \begin{bmatrix} D_1 & T_{12} \\ T_{21} & D_2 \end{bmatrix} = \begin{bmatrix} D_1 & \\ & D_2 \end{bmatrix} - \begin{bmatrix} 0 & 0 \\ -T_{21} & 0 \end{bmatrix} - \begin{bmatrix} 0 & -T_{12} \\ 0 & 0 \end{bmatrix} \\ &= D - \tilde{L} - \tilde{U}. \end{aligned}$$

DEFINITION 6.13. *Let $R_J(\alpha) = \alpha L + \frac{1}{\alpha}U$. Then $R_J(1) = R_J$ is the iteration matrix for Jacobi's method.*

PROPOSITION 6.2. *The eigenvalues of $R_J(\alpha)$ are independent of α.*

Proof.

$$R_J(\alpha) = -\begin{bmatrix} 0 & \frac{1}{\alpha}D_1^{-1}T_{12} \\ \alpha D_2^{-1}T_{21} & 0 \end{bmatrix}$$

has the same eigenvalues as the similar matrix

$$\begin{bmatrix} I & \\ & \alpha I \end{bmatrix}^{-1} R_J(\alpha) \begin{bmatrix} I & \\ & \alpha I \end{bmatrix} = -\begin{bmatrix} 0 & D_1^{-1}T_{12} \\ D_2^{-1}T_{21} & 0 \end{bmatrix} = R_J(1). \quad \square$$

DEFINITION 6.14. *Let T be any matrix, with $T = D - \tilde{L} - \tilde{U}$ and $R_J(\alpha) = \alpha D^{-1}\tilde{L} + \frac{1}{\alpha}D^{-1}\tilde{U}$. If $R_J(\alpha)$'s eigenvalues are independent of α, then T is called* consistent ordering.

It is an easy fact that if T has property A, such as the model problem, then PTP^T is consistently ordered for the permutation P that makes $PTP^T = \begin{bmatrix} T_{11} & T_{12} \\ T_{21} & T_{22} \end{bmatrix}$ have diagonal T_{11} and T_{22}. It is not true that consistent ordering implies a matrix has property A.

EXAMPLE 6.9. Any block tridiagonal matrix

$$\begin{bmatrix} D_1 & A_1 & & \\ B_1 & \ddots & \ddots & \\ & \ddots & \ddots & A_{n-1} \\ & & B_{n-1} & D_n \end{bmatrix}$$

is consistently ordered when the D_i are diagonal. $\diamond$

Consistent ordering implies that there are simple formulas relating the eigenvalues of R_J, R_{GS}, and $R_{SOR(\omega)}$ [249].

THEOREM 6.6. *If A is consistently ordered and $\omega \neq 0$, then the following are true:*

1) *The eigenvalues of R_J appear in $\pm$ pairs.*

2) *If μ is an eigenvalue of R_J and*

$$(\lambda + \omega - 1)^2 = \lambda\omega^2\mu^2, \tag{6.27}$$

then λ is an eigenvalue of $R_{SOR(\omega)}$.

3) *Conversely, if $\lambda \neq 0$ is an eigenvalue of $R_{SOR(\omega)}$, then μ in equation (6.27) is an eigenvalue of R_J.*

Proof.

1) Consistent ordering implies that the eigenvalues of $R_J(\alpha)$ are independent of α, so $R_J = R_J(1)$ and $R_J(-1) = -R_J(1)$ have same eigenvalues; hence they appear in $\pm$ pairs.

2) If $\lambda = 0$ and equation (6.27) holds, then $\omega = 1$ and 0 is indeed an eigenvalue of $R_{SOR(1)} = R_{GS} = (I - L)^{-1}U$ since R_{GS} is singular. Otherwise

$$\begin{aligned} 0 &= \det(\lambda I - R_{SOR(\omega)}) \\ &= \det((I - \omega L)(\lambda I - R_{SOR(\omega)})) \\ &= \det((\lambda + \omega - 1)I - \omega\lambda L - \omega U) \\ &= \det\left(\sqrt{\lambda}\omega\left(\left(\frac{\lambda + \omega - 1}{\sqrt{\lambda}\omega}\right)I - \sqrt{\lambda}L - \frac{1}{\sqrt{\lambda}}U\right)\right) \\ &= \det\left(\left(\frac{\lambda + \omega - 1}{\sqrt{\lambda}\omega}\right)I - L - U\right)(\sqrt{\lambda}\omega)^n, \end{aligned}$$

where the last equality is true because of Proposition 6.2. Therefore $\frac{\lambda+\omega-1}{\sqrt{\lambda}\omega} = \mu$, an eigenvalue of $L + U = R_J$, and $(\lambda + \omega - 1)^2 = \mu^2\omega^2\lambda$.

3) If $\lambda \neq 0$, the last set of equalities works in the opposite direction. □

COROLLARY 6.1. *If A is consistently ordered, then $\rho(R_{GS}) = (\rho(R_J))^2$. This means that the Gauss–Seidel method is twice as fast as Jacobi's method.*

Proof. The choice $\omega = 1$ is equivalent to the Gauss–Seidel method, so $\lambda^2 = \lambda\mu^2$ or $\lambda = \mu^2$. □

To get the most benefit from overrelaxation, we would like to find ω_{opt} minimizing $\rho(R_{SOR(\omega)})$ [249].

THEOREM 6.7. *Suppose that A is consistently ordered, R_J has real eigenvalues, and $\mu = \rho(R_J) < 1$. Then*

$$\omega_{opt} = \frac{2}{1+\sqrt{1-\mu^2}},$$

$$\rho(R_{SOR(\omega_{opt})}) = \omega_{opt} - 1 = \frac{\mu^2}{[1+\sqrt{1-\mu^2}]^2},$$

$$\rho(R_{SOR(\omega)}) = \begin{cases} \omega - 1, & \omega_{opt} \leq \omega \leq 2, \\ 1 - \omega + \frac{1}{2}\omega^2\mu^2 + \omega\mu\sqrt{1-\omega+\frac{1}{4}\omega^2\mu^2}, & 0 < \omega \leq \omega_{opt}. \end{cases}$$

Proof. Solve $(\lambda + \omega - 1)^2 = \lambda\omega^2\mu^2$ for λ. □

EXAMPLE 6.10. The model problem is an example: R_J is symmetric, so it has real eigenvalues. Figure 6.5 shows a plot of $\rho(R_{SOR(\omega)})$ versus ω, along with $\rho(R_{GS})$ and $\rho(R_J)$, for the model problem on an N-by-N grid with $N = 16$ and $N = 64$. The plots on the left are of $\rho(R)$, and the plots on the right are semilogarithmic plots of $1 - \rho(R)$. The main conclusion that we can draw is that the graph of $\rho(R_{SOR(\omega)})$ has a vary narrow minimum, so if ω is even slightly different from ω_{opt}, the convergence will slow down significantly. The second conclusion is that if you have to guess ω_{opt}, a large value (near 2) is a better guess than a small value. ◇

6.5.6. Chebyshev Acceleration and Symmetric SOR (SSOR)

Of the methods we have discussed so far, Jacobi's and Gauss–Seidel methods require no information about the matrix to execute them (although proving that they converge requires some information). SOR(ω) depends on a parameter ω, which can be chosen depending on $\rho(R_J)$ to accelerate convergence. Chebyshev acceleration is useful when we know even more about the spectrum of R_J than just $\rho(R_J)$ and lets us further accelerate convergence.

Suppose that we convert $Ax = b$ to the iteration $x_{i+1} = Rx_i + c$, using some method (Jacobi's, Gauss–Seidel, or SOR(ω)). Then we get a sequence $\{x_i\}$ where $x_i \to x$ as $i \to \infty$ if $\rho(R) < 1$.

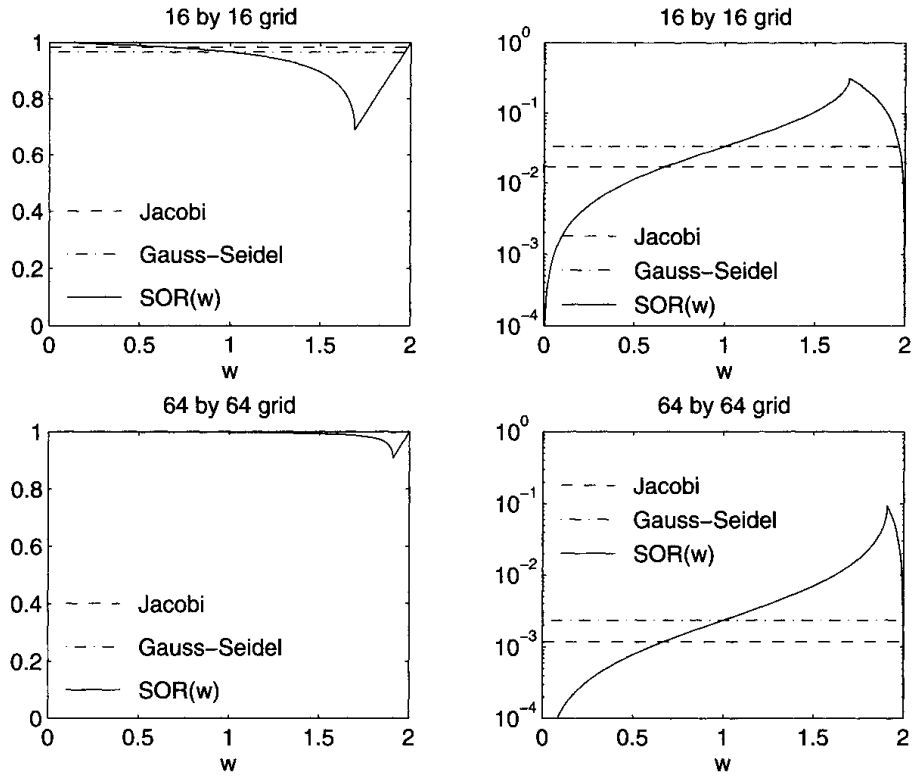


Fig. 6.5. *Convergence of Jacobi's, Gauss-Seidel, and SOR(ω) methods versus ω on the model problem on a 16-by-16 grid and a 64-by-64 grid. The spectral radius ρ(R) of each method (ρ(R_J), ρ(R_GS), and ρ(R_SOR(ω))) is plotted on the left, and 1 − ρ(R) on the right.*

Given all these approximations x_i, it is natural to ask whether some linear combination of them, $y_m = \sum_{i=1}^{m} \gamma_{mi} x_i$, is an even better approximation of the solution x. Note that the scalars γ_{mi} must satisfy $\sum_{i=0}^{m} \gamma_{mi} = 1$, since if $x_0 = x_1 = \cdots = x$, we want $y_m = x$, too. So we can write the error in y_m as

$$\begin{aligned} y_m - x &= \sum_{i=0}^{m} \gamma_{mi} x_i - x \\ &= \sum_{i=0}^{m} \gamma_{mi} (x_i - x) \\ &= \sum_{i=0}^{m} \gamma_{mi} R^i (x_0 - x) \\ &= p_m(R)(x_0 - x), \end{aligned} \tag{6.28}$$

where $p_m(R) = \sum_{i=0}^{m} \gamma_{mi} R^i$ is a polynomial of degree m with $p_m(1) = \sum_{i=0}^{m} \gamma_{mi} = 1$.

EXAMPLE 6.11. If we could choose p_m to be the characteristic polynomial of R, then $p_m(R) = 0$ by the Cayley–Hamilton theorem, and we would converge in m steps. But this is not practical, because we seldom know the eigenvalues

of R and we want to converge much faster than in $m = \dim(R)$ steps anyway. $\diamond$

Instead of seeking a polynomial such that $p_m(R)$ is zero, we will settle for making the spectral radius of $p_m(R)$ as small as we can. Suppose that we knew

- the eigenvalues of R were real, and
- the eigenvalues of R lay in an interval $[-\rho, \rho]$ not containing 1.

Then we could try to choose a polynomial p_m where

1) $p_m(1) = 1$, and

2) $\max_{-\rho \le x \le \rho} |p_m(x)|$ is as small as possible.

Since the eigenvalues of $p_m(R)$ are $p_m(\lambda(R))$ (see Problem 4.5), these eigenvalues would be small and so the spectral radius (the largest eigenvalue in absolute value) would be small.

Finding a polynomial p_m to satisfy conditions 1) and 2) above is a classical problem in approximation theory whose solution is based on *Chebyshev polynomials.*

DEFINITION 6.15. *The mth Chebyshev polynomial is defined by the recurrence* $T_m(x) \equiv 2xT_{m-1}(x) - T_{m-2}(x)$, *where* $T_0(x) = 1$ *and* $T_1(x) = x$.

Chebyshev polynomials have many interesting properties [240]. Here are a few, which are easy to prove from the definition (see Question 6.7).

LEMMA 6.7. *Chebyshev polynomials have the following properties:*

- $T_m(1) = 1$.
- $T_m(x) = 2^{m-1}x^m + O(x^{m-1})$.
- $T_m(x) = \begin{cases} \cos(m \cdot \arccos x) & \text{if } |x| \le 1, \\ \cosh(m \cdot \operatorname{arccosh} x) & \text{if } |x| \ge 1. \end{cases}$
- $|T_m(x)| \le 1$ *if* $|x| \le 1$.
- *The zeros of* $T_m(x)$ *are* $x_i = \cos((2i-1)\pi/(2m))$ *for* $i = 1, \ldots, m$.
- $T_m(x) = \frac{1}{2}[(x + \sqrt{x^2-1})^m + (x + \sqrt{x^2-1})^{-m}]$ *if* $|x| > 1$.
- $T_m(1+\epsilon) \ge .5(1 + m\sqrt{2\epsilon})$ *if* $\epsilon > 0$.

Here is a table of values of $T_m(1+\epsilon)$. Note how fast it grows as m grows, even when ϵ is tiny (see Figure 6.6).

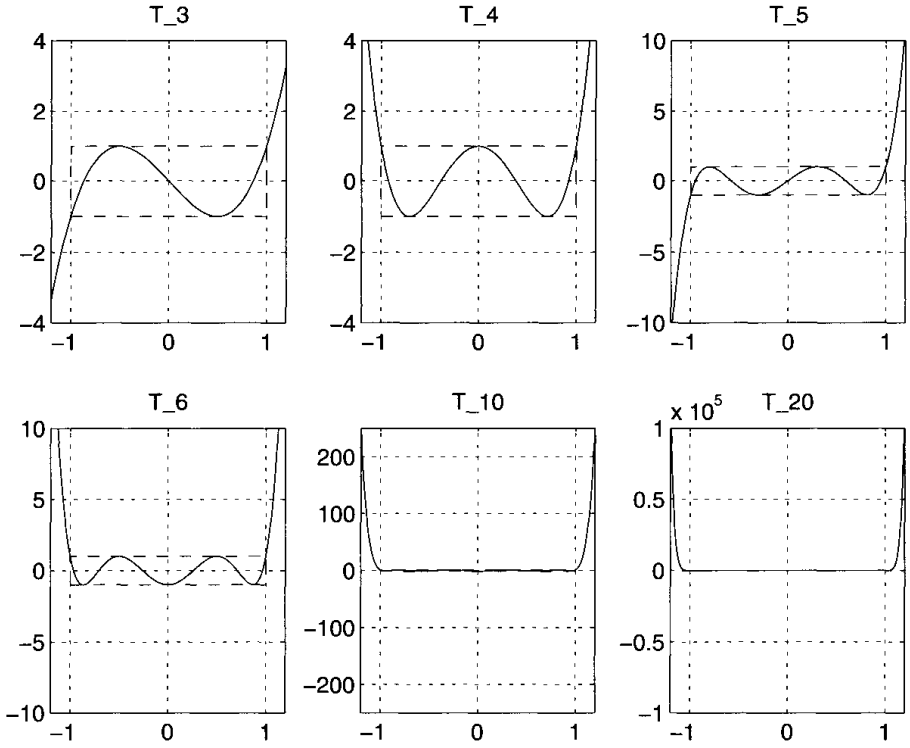


Fig. 6.6. *Graph of* $T_m(x)$ *versus* x. *The dotted lines indicate that* $|T_m(x)| \leq 1$ *for* $|x| \leq 1$.

m	ϵ		
	10^{-4}	10^{-3}	10^{-2}
10	1.0	1.1	2.2
100	2.2	44	$6.9 \cdot 10^5$
200	8.5	$3.8 \cdot 10^3$	$9.4 \cdot 10^{11}$
1000	$6.9 \cdot 10^5$	$1.3 \cdot 10^{19}$	$1.2 \cdot 10^{61}$

A polynomial with the properties we want is $p_m(x) = T_m(x/\rho)/T_m(1/\rho)$. To see why, note that $p_m(1) = 1$ and that if $x \in [-\rho, \rho]$, then $|p_m(x)| \leq 1/T_m(1/\rho)$. For example, if $\rho = 1/(1+\epsilon)$, then $|p_m(x)| \leq 1/T_m(1+\epsilon)$. As we have just seen, this bound is tiny for small ϵ and modest m.

To implement this cheaply, we use the three-term recurrence $T_m(x) = 2xT_{m-1}(x) - T_{m-2}(x)$ used to define Chebyshev polynomials. This means that we need to save and combine only three vectors y_m, y_{m-1}, and y_{m-2}, not all the previous x_m. To see how this works, let $\mu_m \equiv 1/T_m(1/\rho)$, so $p_m(R) = \mu_m T_m(R/\rho)$ and $\frac{1}{\mu_m} = \frac{2}{\rho\mu_{m-1}} - \frac{1}{\mu_{m-2}}$ by the three-term recurrence in Definition 6.15. Then

$$\begin{aligned} y_m - x &= p_m(R)(x_0 - x) \quad \text{by equation (6.28)} \\ &= \mu_m T_m\left(\frac{R}{\rho}\right)(x_0 - x) \\ &= \mu_m \left[2 \cdot \frac{R}{\rho} \cdot T_{m-1}\left(\frac{R}{\rho}\right)(x_0 - x) - T_{m-2}\left(\frac{R}{\rho}\right)(x_0 - x)\right] \end{aligned}$$

$$
\begin{aligned}
& \text{by Definition 6.15} \\
= \ & \mu_m \left[2 \cdot \frac{R}{\rho} \cdot \frac{p_{m-1}(\frac{R}{\rho})(x_0 - x)}{\mu_{m-1}} - \frac{p_{m-2}(\frac{R}{\rho})(x_0 - x)}{\mu_{m-2}} \right] \\
= \ & \mu_m \left[2 \cdot \frac{R}{\rho} \cdot \frac{y_{m-1} - x}{\mu_{m-1}} - \frac{y_{m-2} - x}{\mu_{m-2}} \right] \quad \text{by equation (6.28)}
\end{aligned}
$$

or

$$
y_m = \frac{2\mu_m}{\mu_{m-1}} \frac{R}{\rho} y_{m-1} - \frac{\mu_m}{\mu_{m-2}} y_{m-2} + d_m,
$$

where

$$
\begin{aligned}
d_m &= x - \frac{2\mu_m}{\mu_{m-1}} \left(\frac{R}{\rho} \right) x + \frac{\mu_m}{\mu_{m-2}} x \\
&= x - \frac{2\mu_m}{\mu_{m-1}} \left(\frac{x - c}{\rho} \right) + \frac{\mu_m}{\mu_{m-2}} x \qquad \text{since } x = Rx + c \\
&= \mu_m \left(\frac{1}{\mu_m} - \frac{2}{\rho\mu_{m-1}} + \frac{1}{\mu_{m-2}} \right) x + \frac{2\mu_m}{\rho\mu_{m-1}} c \\
&= \frac{2\mu_m}{\rho\mu_{m-1}} c \quad \text{by the definition of } \mu_m.
\end{aligned}
$$

This yields the algorithm.

ALGORITHM 6.7. *Chebyshev acceleration of* $x_{i+1} = Rx_i + c$*:*

$\mu_0 = 1$; $\mu_1 = \rho$; $y_0 = x_0$; $y_1 = Rx_0 + c$
for $m = 2, 3, \ldots$
$\quad \mu_m = 1/\left(\frac{2}{\rho\mu_{m-1}} - \frac{1}{\mu_{m-2}} \right)$
$\quad y_m = \frac{2\mu_m}{\rho\mu_{m-1}} R y_{m-1} - \frac{\mu_m}{\mu_{m-2}} y_{m-2} + \frac{2\mu_m}{\rho\mu_{m-1}} c$
end for

Note that each iteration takes just one application of R, so if this is significantly more expensive than the other scalar and vector operations, this algorithm is no more expensive per step than the original iteration $x_{m+1} = Rx_m + c$.

Unfortunately, we cannot apply this directly to SOR(ω) for solving $Ax = b$, because $R_{SOR(\omega)}$ generally has complex eigenvalues, and Chebyshev acceleration requires that R have real eigenvalues in the interval $[-\rho, \rho]$. But we can fix this by using the following algorithm.

ALGORITHM 6.8. *Symmetric SOR (SSOR):*

1. *Take one step of SOR(ω) computing the components of x in the usual increasing order:* $x_{i,1}, x_{i,2}, \ldots, x_{i,n}$,
2. *Take one step of SOR(ω) computing backwards:* $x_{i,n}, x_{i,n-1}, \ldots, x_{i,1}$.

We will reexpress this algorithm as $x_{i+1} = E_\omega x_i + c_\omega$ and show that E_ω has real eigenvalues, so we can use Chebyschev acceleration.

Suppose A is symmetric as in the model problem and again write $A = D - \tilde{L} - \tilde{U} = D(I - L - U)$ as in equation (6.19). Since $A = A^T$, $U = L^T$. Use equation (6.21) to rewrite the two steps of SSOR as

$$\begin{array}{lrcl} 1. & x_{i+1/2} & = & (I-\omega L)^{-1}((1-\omega)I + \omega U)x_i + c_{1/2} \equiv L_\omega x_i + c_{1/2}, \\ 2. & x_i & = & (I-\omega U)^{-1}((1-\omega)I + \omega L)x_{i+1/2} + c_1 \equiv U_\omega x_{i+1/2} + c_1. \end{array}$$

Eliminating $x_{i+1/2}$ yields $x_{i+1} = E_\omega x_i + \hat{c}$, where

$$\begin{aligned} E_\omega &= U_\omega L_\omega \\ &= I + (\omega-2)^2(I-\omega U)^{-1}(I-\omega L)^{-1} + (\omega-2)(I-\omega U)^{-1} \\ &\quad +(\omega-2)(I-\omega U)^{-1}(I-\omega L)^{-1}(I-\omega U). \end{aligned}$$

We claim that E_ω has real eigenvalues, since it has the same eigenvalues as the similar matrix

$$\begin{aligned} &(I-\omega U)E_\omega(I-\omega U)^{-1} \\ &= I + (2-\omega)^2(I-\omega L)^{-1}(I-\omega U)^{-1} + (\omega-2)(I-\omega U)^{-1} \\ &\quad +(\omega-2)(I-\omega L)^{-1} \\ &= I + (2-\omega)^2(I-\omega L)^{-1}(I-\omega L^T)^{-1} + (\omega-2)(I-\omega L^T)^{-1} \\ &\quad +(\omega-2)(I-\omega L)^{-1}, \end{aligned}$$

which is clearly symmetric and so must have real eigenvalues.

EXAMPLE 6.12. Let us apply SSOR(ω) with Chebyshev acceleration to the model problem. We need to both choose ω and estimate the spectral radius $\rho = \rho(E_\omega)$. The optimal ω that minimizes ρ is not known but Young [267, 137] has shown that the choice $\omega = \frac{2}{1+[2(1-\rho(R_J))]^{1/2}}$ is a good one, yielding $\rho(E_\omega) \approx 1 - \frac{\pi}{2N}$. With Chebyshev acceleration the error is multiplied by $\mu_m \approx \frac{1}{T_m(1+\frac{\pi}{2N})} \leq 2/(1 + m\sqrt{\frac{\pi}{N}})$ at step m. Therefore, to decrease the error by a fixed factor < 1 requires $m = O(N^{1/2}) = O(n^{1/4})$ iterations. Since each iteration has the same cost as an iteration of SOR(ω), $O(n)$, the overall cost is $O(n^{5/4})$. This explains the entry for SSOR with Chebyshev acceleration in Table 6.1.

In contrast, after m steps of SOR(ω_{opt}), the error would decrease only by $(1 - \frac{\pi}{N})^m$. For example, consider $N = 1000$. Then SOR(ω_{opt}) requires $m = 220$ iterations to cut the error in half, whereas SSOR(ω_{opt}) with Chebyshev acceleration requires only $m = 17$ iterations. $\diamond$

6.6. Krylov Subspace Methods

These methods are used both to solve $Ax = b$ and to find eigenvalues of A. They assume that A is accessible only via a "black-box" subroutine that returns $y = Az$ given any z (and perhaps $y = A^T z$ if A is nonsymmetric). In

other words, no direct access or manipulation of matrix entries is used. This is a reasonable assumption for several reasons. First, the cheapest nontrivial operation that one can perform on a (sparse) matrix is to multiply it by a vector; if A has m nonzero entries, matrix-vector multiplication costs m multiplications and (at most) m additions. Second, A may not be represented explicitly as a matrix but may be available only as a subroutine for computing Ax.

EXAMPLE 6.13. Suppose that we have a physical device whose behavior is modeled by a program that takes a vector x of input parameters and produces a vector y of output parameters describing the device's behavior. The output y may be an arbitrarily complicated function $y = f(x)$, perhaps requiring the solution of nonlinear differential equations. For example, x could be parameters describing the shape of a wing and $f(x)$ could be the drag on the wing, computed by solving the Navier–Stokes equations for the airflow over the wing. A common engineering design problem is to pick the input x to optimize the device behavior $f(x)$, where for concreteness we assume that this means making $f(x)$ as small as possible. Our problem is then to try to solve $f(x) = 0$ as nearly as we can. Assume for illustration that x and y are vectors of equal dimension. Then Newton's method is an obvious candidate, yielding the iteration $x^{(m+1)} = x^{(m)} - (\nabla f(x^{(m)}))^{-1} f(x^{(m)})$, where $\nabla f(x^{(m)})$ is the Jacobian of f at $x^{(m)}$. We can rewrite this as solving the linear system $(\nabla f(x^{(m)})) \cdot \delta^{(m)} = f(x^{(m)})$ for $\delta^{(m)}$ and then computing $x^{(m+1)} = x^{(m)} - \delta^{(m)}$. But how do we solve this linear system with coefficient matrix $\nabla f(x^{(m)})$ when computing $f(x^{(m)})$ is already complicated? It turns out that we can compute the matrix-vector product $(\nabla f(x)) \cdot z$ for an arbitrary vector z so that we can use Krylov subspace methods to solve the linear system. One way to compute $(\nabla f(x)) \cdot z$ is with *divided differences* or by using a Taylor expansion to see that $[f(x + hz) - f(x)]/h \approx (\nabla f(x)) \cdot z$. Thus, computing $(\nabla f(x)) \cdot z$ requires two calls to the subroutine that computes $f(\cdot)$, once with argument x and once with $x + hz$. However, sometimes it is difficult to choose h to get an accurate approximation of the derivative (choosing h too small results in a loss of accuracy due to roundoff). Another way to compute $(\nabla f(x)) \cdot z$ is to actually differentiate the function f. If f is simple enough, this can be done by hand. For complicated f, compiler tools can take a (nearly) arbitrary subroutine for computing $f(x)$ and automatically produce another subroutine for computing $(\nabla f(x)) \cdot z$ [29]. This can also be done by using the operator overloading facilities of C++ or Fortran 90, although this is less efficient. ◇

A variety of different Krylov subspace methods exist. Some are suitable for nonsymmetric matrices, and others assume symmetry or positive-definiteness. Some methods for nonsymmetric matrices assume that $A^T z$ can be computed as well as Az; depending on how A is represented, $A^T z$ may or may not be available (see Example 6.13). The most efficient and best understood method, the conjugate gradient method (CG), is suitable only for symmetric positive

definite matrices, including the model problem. We will concentrate on CG in this chapter.

Given a matrix that is not symmetric positive definite, it can be difficult to pick the best method from the many available. In section 6.6.6 we will give a short summary of the other methods available, besides CG, along with advice on which method to use in which situation. We also refer the reader to the more comprehensive on-line help at NETLIB/templates, which includes a book [24] and implementations in Matlab, Fortran, and C++. For a survey of current research in Krylov subspace methods, see [15, 107, 136, 214].

In Chapter 7, we will also discuss Krylov subspace methods for finding eigenvalues.

6.6.1. Extracting Information about A via Matrix-Vector Multiplication

Given a vector b and a subroutine for computing $A \cdot x$, what can we deduce about A? The most obvious thing that we can do is compute the sequence of matrix-vector products $y_1 = b$, $y_2 = Ay_1$, $y_3 = Ay_2 = A^2y_1$, $\ldots$, $y_n = Ay_{n-1} = A^{n-1}y_1$, where A is n-by-n. Let $K = [y_1, y_2, \ldots, y_n]$. Then we can write

$$A \cdot K = [Ay_1, \ldots, Ay_{n-1}, Ay_n] = [y_2, \ldots, y_n, A^n y_1]. \tag{6.29}$$

Note that the leading $n-1$ columns of $A \cdot K$ are the same as the trailing $n-1$ columns of K, shifted left by one. Assume for the moment that K is nonsingular, so we can compute $c = -K^{-1}A^n y_1$. Then

$$A \cdot K = K \cdot [e_2, e_3, \ldots, e_n, -c] \equiv K \cdot C,$$

where e_i is the ith column of the identity matrix, or

$$K^{-1}AK = C = \begin{bmatrix} 0 & 0 & \cdots & 0 & -c_1 \\ 1 & 0 & \cdots & 0 & -c_2 \\ 0 & 1 & \cdots & \vdots & \vdots \\ \vdots & 0 & \cdots & \vdots & \vdots \\ \vdots & \vdots & \cdots & 0 & \vdots \\ \vdots & \vdots & \cdots & 1 & -c_n \end{bmatrix}.$$

Note that C is upper Hessenberg. In fact, it is a *companion matrix* (see section 4.5.3), which means that its characteristic polynomial is $p(x) = x^n + \sum_{i=1}^{n} c_i x^{i-1}$. Thus, just by matrix-vector multiplication, we have reduced A to a very simple form, and in principle we could now find the eigenvalues of A by finding the zeros of $p(x)$.

However, this simple form is not useful in practice, for the following reasons:

1. Finding c requires $n-1$ matrix-vector multiplications by A and then solving a linear system with K. Even if A is sparse, K is likely to be dense, so there is no reason to expect solving a linear system with K will be any easier than solving the original problem $Ax = b$.

2. K is likely to be very ill-conditioned, so c would be very inaccurately computed. This is because the algorithm is performing the power method (Algorithm 4.1) to get the columns y_i of K, so that y_i is converging to an eigenvector corresponding to the largest eigenvalue of A. Thus, the columns of K tend to get more and more parallel.

We will overcome these problems as follows: We will replace K with an orthogonal matrix Q such that for all k, the leading k columns of K and Q span the same the same space. This space is called a *Krylov subspace.* In contrast to K, Q is well conditioned and easy to invert. Furthermore, we will compute only as many leading columns of Q as needed to get an accurate solution (for $Ax = b$ or $Ax = \lambda x$). In practice we usually need very few columns compared to the matrix dimension n.

We proceed by writing $K = QR$, the QR decomposition of K. Then

$$K^{-1}AK = (R^{-1}Q^T)A(QR) = C,$$

implying

$$Q^TAQ = RCR^{-1} \equiv H.$$

Since R and R^{-1} are both upper triangular and C is upper Hessenberg, it is easy to confirm that $H = RCR^{-1}$ is also upper Hessenberg (see Question 6.11). In other words, we have reduced A to upper Hessenberg form by an orthogonal transformation Q. (This is the first step of the algorithm for finding eigenvalues of nonsymmetric matrices discussed in section 4.4.6.) Note that if A is symmetric, so is $Q^TAQ = H$, and a symmetric matrix which is upper Hessenberg must also be lower Hessenberg, i.e., tridiagonal. In this case we write $Q^TAQ = T$.

We still need to show how to compute the columns of Q one at a time, rather than all of them: Let $Q = [q_1, \ldots, q_n]$. Since $Q^TAQ = H$ implies $AQ = QH$, we can equate column j on both sides of $AQ = QH$, yielding

$$Aq_j = \sum_{i=1}^{j+1} h_{i,j}q_i.$$

Since the q_i are orthonormal, we can multiply both sides of this last equality by q_m^T to get

$$q_m^TAq_j = \sum_{i=1}^{j+1} h_{i,j}q_m^Tq_i = h_{m,j} \quad \text{for } 1 \le m \le j$$

and so

$$h_{j+1,j}q_{j+1} = Aq_j - \sum_{i=1}^{j} h_{i,j}q_i.$$

This justifies the following algorithm.

ALGORITHM 6.9. *The Arnoldi algorithm for (partial) reduction to Hessenberg form:*

$q_1 = b/\|b\|_2$
/ k is the number of columns of Q and H to compute */*
for $j = 1$ *to* k
  $z = Aq_j$
  for $i = 1$ *to* j
    $h_{i,j} = q_i^T z$
    $z = z - h_{i,j}q_i$
  end for
  $h_{j+1,j} = \|z\|_2$
  if $h_{j+1,j} = 0$, *quit*
  $q_{j+1} = z/h_{j+1,j}$
end for

The q_j computed by Arnoldi's algorithm are often called *Arnoldi vectors.* The loop over i updating z can be also be described as applying the *modified Gram–Schmidt algorithm* (Algorithm 3.1) to subtract the components in the directions q_1 through q_j away from z, leaving z orthogonal to them. Computing q_1 through q_k costs k matrix-vector multiplications by A, plus $O(k^2n)$ other work. If we stop the algorithm here, what have we learned about A? Let us write $Q = [Q_k, Q_u]$, where $Q_k = [q_1, \ldots, q_k]$ and $Q_u = [q_{k+1}, \ldots, q_n]$. Note that we have computed only Q_k and q_{k+1}; the other columns of Q_u are unknown. Then

$$H = Q^TAQ = [Q_k, Q_u]^T A[Q_k, Q_u] = \begin{bmatrix} Q_k^TAQ_k & Q_k^TAQ_u \\ Q_u^TAQ_k & Q_u^TAQ_u \end{bmatrix}$$

$$\equiv \begin{array}{c} \\ k \\ n-k \end{array} \begin{array}{c} \begin{array}{cc} k & n-k \end{array} \\ \begin{pmatrix} H_k & H_{uk} \\ H_{ku} & H_u \end{pmatrix} \end{array}. \tag{6.30}$$

Note that H_k is upper Hessenberg, because H has the same property. For the same reason, H_{ku} has a single (possibly) nonzero entry in its upper right corner, namely, $h_{k+1,k}$. Thus, H_u and H_{uk} are unknown; we know only H_k and H_{ku}.

When A is symmetric, $H = T$ is symmetric and tridiagonal, and the Arnoldi algorithm simplifies considerably, because most of the $h_{i,j}$ are zero: Write

$$T = \begin{bmatrix} \alpha_1 & \beta_1 & & \\ \beta_1 & \ddots & \ddots & \\ & \ddots & \ddots & \beta_{n-1} \\ & & \beta_{n-1} & \alpha_n \end{bmatrix}.$$

Equating column j on both sides of $AQ = QT$ yields

$$Aq_j = \beta_{j-1}q_{j-1} + \alpha_j q_j + \beta_j q_{j+1}.$$

Since the columns of Q are orthonormal, multiplying both sides of this equation by q_j yields $q_j A q_j = \alpha_j$. This justifies the following version of the Arnoldi algorithm, called the *Lanczos algorithm.*

ALGORITHM 6.10. *The Lanczos algorithm for (partial) reduction to symmetric tridiagonal form.*

$q_1 = b/\|b\|_2$, $\beta_0 = 0$, $q_0 = 0$
for $j = 1$ *to* k
  $z = Aq_j$
  $\alpha_j = q_j^T z$
  $z = z - \alpha_j q_j - \beta_{j-1} q_{j-1}$
  $\beta_j = \|z\|_2$
  if $\beta_j = 0$, *quit*
  $q_{j+1} = z/\beta_j$
end for

The q_j computed by the Lanczos algorithm are often called *Lanczos vectors.* After k steps of the Lanczos algorithm, here is what we have learned about A:

$$\begin{aligned} T &= Q^T A Q = [Q_k, Q_u]^T A [Q_k, Q_u]^T \\ &= \begin{bmatrix} Q_k^T A Q_k & Q_k^T A Q_u \\ Q_u^T A Q_k & Q_u^T A Q_u \end{bmatrix} \\ &\equiv \begin{array}{c} \\ k \\ n-k \end{array} \begin{array}{c} \begin{array}{cc} k & n-k \end{array} \\ \begin{pmatrix} T_k & T_{uk} \\ T_{ku} & T_u \end{pmatrix} \end{array} \\ &= \begin{bmatrix} T_k & T_{ku}^T \\ T_{ku} & T_u \end{bmatrix}. \end{aligned} \tag{6.31}$$

Because A is symmetric, we know T_k and $T_{ku} = T_{uk}^T$ but not T_u. T_{ku} has a single (possibly) nonzero entry in its upper right corner, namely, β_k. Note that β_k is nonnegative, because it is computed as the norm of z.

We define some standard notation associated with the partial factorization of A computed by the Arnoldi and Lanczos algorithms.

DEFINITION 6.16. *The* Krylov subspace $\mathcal{K}_k(A,b)$ *is* $\text{span}[b, Ab, A^2b, \ldots, A^{k-1}b]$.

We will write $\mathcal{K}_k$ instead of $\mathcal{K}_k(A,b)$ if A and b are implicit from the context. Provided that the algorithm does not quit because $z = 0$, the vectors Q_k computed by the Arnoldi or Lanczos algorithms form an orthonormal basis of the Krylov subspace $\mathcal{K}_k$. (One can show that $\mathcal{K}_k$ has dimension k if and only if the Arnoldi or Lanczos algorithm can compute q_k without quitting first; see Question 6.12.) We also call H_k (or T_k) the *projection* of A onto the Krylov subpace $\mathcal{K}_k$.

Our goal is to design algorithms to solve $Ax = b$ using only the information computed by k steps of the Arnoldi or the Lanczos algorithm. We hope that k can be much smaller than n, so the algorithms are efficient.

(In Chapter 7 we will use this same information for find eigenvalues of A. We can already sketch how we will do this: Note that if $h_{k+1,k}$ happens to be zero, then H (or T) is block upper triangular and so all the eigenvalues of H_k are also eigenvalues of H, and therefore also of A, since A and H are similar. The (right) eigenvectors of H_k are eigenvectors of H, and if we multiply them by Q_k, we get eigenvectors of A. When $h_{k+1,k}$ is nonzero but small, we expect the eigenvalues and eigenvectors of H_k to provide good approximations to the eigenvalues and eigenvectors of A.)

We finish this introduction by noting that roundoff error causes a number of the algorithms that we discuss to behave *entirely differently* from how they would in exact arithmetic. In particular, the vectors q_i computed by the Lanczos algorithm can quickly lose orthogonality and in fact often become linearly dependent. This apparently disastrous numerical instability led researchers to abandon these algorithms for several years after their discovery. But eventually researchers learned either how to stabilize the algorithms or that convergence occurred despite instability! We return to these points in section 6.6.4, where we analyze the convergence of the conjugate gradient method for solving $Ax = b$ (which is "unstable" but converges anyway), and in Chapter 7, especially in sections 7.4 and 7.5, where we show how to compute eigenvalues (and the basic algorithm is modified to ensure stability).

6.6.2. Solving $Ax = b$ Using the Krylov Subspace $\mathcal{K}_k$

How do we solve $Ax = b$, given only the information available from k steps of either the Arnoldi or the Lanczos algorithm?

Since the only vectors we know are the columns of Q_k, the only place to "look" for an approximate solution is in the Krylov subspace $\mathcal{K}_k$ spanned by these vectors. In other words, we see the "best" approximate solution of the form

$$x_k = \sum_{j=1}^{k} z_k q_k = Q_k \cdot z, \quad \text{where} \quad z = [z_1, \ldots, z_k]^T.$$

Now we have to define "best." There are several natural but different

definitions, leading to different algorithms. We let $x = A^{-1}b$ denote the true solution and $r_k = b - Ax_k$ denote the residual.

1. The "best" x_k minimizes $\|x_k - x\|_2$. Unfortunately, we do not have enough information in our Krylov subspace to compute this x_k.

2. The "best" x_k minimizes $\|r_k\|_2$. This is implementable, and the corresponding algorithms are called MINRES (for *minimum residual*) when A is symmetric [194] and GMRES (for *generalized minimum residual*) when A is nonsymmetric [215].

3. The "best" x_k makes $r_k \perp \mathcal{K}_k$, i.e., $Q_k^T r_k = 0$. This is sometimes called the *orthogonal residual* property, or a *Galerkin condition*, by analogy to a similar condition in the theory of finite elements. When A is symmetric, the corresponding algorithm is called SYMMLQ [194]. When A is nonsymmetric, a variation of GMRES works [211].

4. When A is symmetric and positive definite, it defines a norm $\|r\|_{A^{-1}} = (r^T A^{-1} r)^{1/2}$ (see Lemma 1.3). We say the "best" x_k minimizes $\|r_k\|_{A^{-1}}$. This norm is the same as $\|x_k - x\|_A$. The algorithm is called the conjugate gradient algorithm [145].

When A is symmetric positive definite, the last two definitions of "best" also turn out to be equivalent.

THEOREM 6.8. *Let A be symmetric, $T_k = Q_k^T A Q_k$, and $r_k = b - Ax_k$, where $x_k \in \mathcal{K}_k$. If T_k is nonsingular and $x_k = Q_k T_k^{-1} e_1 \|b\|_2$, where $e_1^{k\times 1} = [1, 0, \ldots, 0]^T$, then $Q_k^T r_k = 0$. If A is also positive definite, then T_k must be nonsingular, and this choice of x_k also minimizes $\|r_k\|_{A^{-1}}$ over all $x_k \in \mathcal{K}_k$. We also have that $r_k = \pm \|r_k\|_2 q_{k+1}$.*

Proof. We drop the subscripts k for ease of notation. Let $x = QT^{-1}e_1\|b\|_2$ and $r = b - Ax$, and assume that $T = Q^T AQ$ is nonsingular. We confirm that $Q^T r = 0$ by computing

$$\begin{aligned} Q^T r = Q^T(b - Ax) &= Q^T b - Q^T Ax \\ &= e_1\|b\|_2 - Q^T A(QT^{-1}e_1\|b\|_2) \\ &\quad \text{because the first column of } Q \text{ is } b/\|b\|_2 \\ &\quad \text{and its other columns are orthogonal to } b \\ &= e_1\|b\|_2 - (Q^T AQ)T^{-1}e_1\|b\|_2 \\ &= e_1\|b\|_2 - (T)T^{-1}e_1\|b\|_2 \quad \text{because } Q^T AQ = T \\ &= 0. \end{aligned}$$

Now assume that A is also positive definite. Then T must be positive definite and thus nonsingular too (see Question 6.13). Let $\hat{x} = x + Qz$ be

another candidate solution in $\mathcal{K}$, and let $\hat{r} = b - A\hat{x}$. We need to show that $\|\hat{r}\|_{A^{-1}}$ is minimized when $z = 0$. But

$$\begin{aligned}
\|\hat{r}\|_{A^{-1}}^2 &= \hat{r}^T A^{-1} \hat{r} \qquad \text{by definition} \\
&= (r - AQz)^T A^{-1} (r - AQz) \\
&\qquad \text{since } \hat{r} = b - A\hat{x} = b - A(x + Qz) = r - AQz \\
&= r^T A^{-1} r^T - 2(AQz)^T A^{-1} r + (AQz)^T A^{-1} (AQz) \\
&= \|r\|_{A^{-1}}^2 - 2z^T Q^T r + \|AQz\|_{A^{-1}}^2 \\
&\qquad \text{since } (AQz)^T A^{-1} r = z^T Q^T A A^{-1} r = z^T Q^T r \\
&= \|r\|_{A^{-1}}^2 + \|AQz\|_{A^{-1}}^2 \qquad \text{since } Q^T r = 0,
\end{aligned}$$

so $\|\hat{r}\|_{A^{-1}}$ is minimized if and only if $AQz = 0$. But $AQz = 0$ if and only if $z = 0$ since A is nonsingular and Q has full column rank.

To show that $r_k = \pm \|r_k\|_2 q_{k+1}$, we reintroduce subscripts. Since $x_k \in \mathcal{K}_k$, we must have $r_k = b - Ax_k \in \mathcal{K}_{k+1}$, so r_k is a linear combination of the columns of Q_{k+1}, since these columns span $\mathcal{K}_{k+1}$. But since $Q_k^T r_k = 0$, the only column of Q_{k+1} to which r_k is not orthogonal is q_{k+1}. □

6.6.3. Conjugate Gradient Method

The algorithm of choice for symmetric positive definite matrices is CG. Theorem 6.8 characterizes the solution x_k computed by CG. While MINRES might seem more natural than CG because it minimizes $\|r_k\|_2$ instead of $\|r_k\|_{A^{-1}}$, it turns out that MINRES requires more work to implement, is more susceptible to numerical instabilities, and thus often produces less accurate answers than CG. We will see that CG has the particularly attractive property that it can be implemented by keeping only four vectors in memory at one time, and not k (q_1 through q_k). Furthermore, the work in the inner loop, beyond the matrix-vector product, is limited to two dot products, three "saxpy" operations (adding a multiple of one vector to another), and a handful of scalar operations. This is a very small amount of work and storage.

Now we derive CG. There are several ways to do this. We will start with the Lanczos algorithm (Algorithm 6.10), which computes the columns of the orthogonal matrix Q_k and the entries of the tridiagonal matrix T_k, along with the formula $x_k = Q_k T_k^{-1} e_1 \|b\|_2$ from Theorem 6.8. We will show how to compute x_k directly via recurrences for three sets of vectors. We will keep only the most recent vector from each set in memory at one time, overwriting the old ones. The first set of vectors are the approximate solutions x_k. The second set of vectors are the residuals $r_k = b - Ax_k$, which Theorem 6.8 showed were parallel to the Lanczos vectors q_{k+1}. The third set of vectors are the *conjugate gradients* p_k. The p_k are called *gradients* because a single step of CG can be interpreted as choosing a scalar ν so that the new solution $x_k = x_{k-1} + \nu p_k$ minimizes the residual norm $\|r_k\|_{A^{-1}} = (r_k^T A^{-1} r_k)^{1/2}$. In other words, the p_k are used as *gradient search directions*. The p_k are called *conjugate*, or more

precisely *A-conjugate*, because $p_k^T A p_j = 0$ if $j \neq k$. In other words, the p_k are orthogonal with respect to the inner product defined by A (see Lemma 1.3).

Since A is symmetric positive definite, so is $T_k = Q_k^T A Q_k$ (see Question 6.13). This means we can perform Cholesky on T_k to get $T_k = \hat{L}_k \hat{L}_k^T = L_k D_k L_k^T$, where L_k is unit lower bidiagonal and D_k is diagonal. Then using the formula for x_k from Theorem 6.8, we get

$$\begin{aligned} x_k &= Q_k T_k^{-1} e_1 \|b\|_2 \\ &= Q_k (L_k^{-T} D_k^{-1} L_k^{-1}) e_1 \|b\|_2 \\ &= (Q_k L_k^{-T})(D_k^{-1} L_k^{-1} e_1 \|b\|_2) \\ &\equiv (\tilde{P}_k)(y_k), \end{aligned}$$

where $\tilde{P}_k \equiv Q_k L_k^{-T}$ and $y_k \equiv D_k^{-1} L_k^{-1} e_1 \|b\|_2$. Write $\tilde{P}_k = [\tilde{p}_1, \ldots, \tilde{p}_k]$. The conjugate gradients p_i will turn out to be parallel to the columns $\tilde{p}_i$ of $\tilde{P}_k$. We know enough to prove the following lemma.

LEMMA 6.8. *The columns $\tilde{p}_i$ of $\tilde{P}_k$ are A-conjugate. In other words, $\tilde{P}_k^T A \tilde{P}_k$ is diagonal.*

Proof. We compute

$$\begin{aligned} \tilde{P}_k^T A \tilde{P}_k &= (Q_k L_k^{-T})^T A (Q_k L_k^{-T}) = L_k^{-1} (Q_k^T A Q_k) L_k^{-T} = L_k^{-1} (T_k) L_k^{-T} \\ &= L_k^{-1} (L_k D_k L_k^T) L_k^{-T} = D_k. \quad \square \end{aligned}$$

Now we derive simple recurrences for the columns of $\tilde{P}_k$ and entries of y_k. We will show that $y_{k-1} \equiv [\eta_1, \ldots, \eta_{k-1}]^T$ is identical to the leading $k-1$ entries of $y_k = [\eta_1, \ldots, \eta_{k-1}, \eta_k]^T$ and that $\tilde{P}_{k-1}$ is identical to the leading $k-1$ columns of $\tilde{P}_k$. Therefore we can let

$$x_k = \tilde{P}_k \cdot y_k = [\tilde{P}_{k-1}, \tilde{p}_k] \cdot \begin{bmatrix} y_{k-1} \\ \eta_k \end{bmatrix} = \tilde{P}_{k-1} y_{k-1} + \tilde{p}_k \eta_k = x_{k-1} + \tilde{p}_k \eta_k \quad (6.32)$$

be our recurrence for x_k.

The recurrence for the η_k is derived as follows. Since T_{k-1} is the leading $(k-1)$-by-$(k-1)$ submatrix of T_k, L_{k-1} and D_{k-1} are also the leading $(k-1)$-by-$(k-1)$ submatrices of L_k and D_k, respectively:

$$\begin{aligned} T_k &= \begin{bmatrix} \alpha_1 & \beta_1 & & \\ \beta_1 & \ddots & \ddots & \\ & \ddots & \ddots & \beta_{k-1} \\ & & \beta_{k-1} & \alpha_k \end{bmatrix} \\ &= L_k D_k L_k^T \end{aligned}$$

$$= \begin{bmatrix} 1 & & \\ l_1 & \ddots & \\ & \ddots & \ddots \\ & & l_{k-1} \quad 1 \end{bmatrix} \cdot \begin{bmatrix} d_1 & & & \\ & \ddots & & \\ & & d_{k-1} & \\ & & & d_k \end{bmatrix} \cdot \begin{bmatrix} 1 & & & \\ l_1 & \ddots & & \\ & \ddots & \ddots & \\ & & l_{k-1} & 1 \end{bmatrix}^T$$

$$= \begin{bmatrix} L_{k-1} & \\ l_{k-1}\hat{e}_{k-1}^T & 1 \end{bmatrix} \cdot \operatorname{diag}(D_{k-1}, d_k) \cdot \begin{bmatrix} L_{k-1} & \\ l_{k-1}\hat{e}_{k-1}^T & 1 \end{bmatrix}^T,$$

where $\hat{e}_{k-1}^T = [0, \ldots, 0, 1]$ has dimension $k-1$. Similarly, D_{k-1}^{-1} and L_{k-1}^{-1} are also the leading $(k-1)$-by-$(k-1)$ submatrices of $D_k^{-1} = \operatorname{diag}(D_{k-1}^{-1}, d_k^{-1})$ and

$$L_k^{-1} = \begin{bmatrix} L_{k-1}^{-1} & \\ \star & 1 \end{bmatrix},$$

respectively, where the details of the last row $\star$ do not concern us. This means that $y_{k-1} = D_{k-1}^{-1} L_{k-1}^{-1} \hat{e}_1 \|b\|_2$, where $\hat{e}_1$ has dimension $k-1$, is identical to the leading $k-1$ components of

$$\begin{aligned} y_k &= D_k^{-1} L_k^{-1} e_1 \|b\|_2 = \begin{bmatrix} D_{k-1}^{-1} & \\ & d_k^{-1} \end{bmatrix} \cdot \begin{bmatrix} L_{k-1}^{-1} & \\ \star & 1 \end{bmatrix} \cdot e_1 \|b\|_2 \\ &= \begin{bmatrix} D_{k-1}^{-1} L_{k-1}^{-1} \hat{e}_1 \|b\|_2 \\ \eta_k \end{bmatrix} = \begin{bmatrix} y_{k-1} \\ \eta_k \end{bmatrix}. \end{aligned}$$

Now we need a recurrence for the columns of $\tilde{P}_k = [\tilde{p}_1, \ldots, \tilde{p}_k]$. Since L_{k-1}^T is upper triangular, so is L_{k-1}^{-T}, and it forms the leading $(k-1)$-by-$(k-1)$ submatrix of L_k^{-T}. Therefore $\tilde{P}_{k-1}$ is identical to the leading $k-1$ columns of

$$\tilde{P}_k = Q_k L_k^{-T} = [Q_{k-1}, q_k] \begin{bmatrix} L_{k-1}^{-T} & \star \\ 0 & 1 \end{bmatrix} = [Q_{k-1} L_{k-1}^{-T}, \tilde{p}_k] = [\tilde{P}_{k-1}, \tilde{p}_k].$$

From $\tilde{P}_k = Q_k L_k^{-T}$, we get $\tilde{P}_k L_k^T = Q_k$ or, equating the kth column on both sides, the recurrence

$$\tilde{p}_k = q_k - l_{k-1} \tilde{p}_{k-1}. \tag{6.33}$$

Altogether, we have recursions for q_k (from the Lanczos algorithm), for $\tilde{p}_k$ (from equation (6.33)), and for the approximate solution x_k (from equation (6.32)). All these recursions are *short*; i.e., they require only the previous iterate or two to implement. Thus, they together provide the means to compute x_k while storing a small number of vectors and doing a small number of dot products, saxpys, and scalar work in the inner loop.

We still have to simplify these recursions slightly to get the ultimate CG algorithm. Since Theorem 6.8 tells us that r_k and q_{k+1} are parallel, we can replace the Lanczos recurrence for q_{k+1} with the recurrence $r_k = b - Ax_k$ or equivalently $r_k = r_{k-1} - \eta_k A \tilde{p}_k$ (gotten from multiplying the recurrence

$x_k = x_{k-1} + \eta_k \tilde{p}_k$ by A and subtracting from $b = b$). This yields the three vector recurrences

$$\begin{aligned} r_k &= r_{k-1} - \eta_k A\tilde{p}_k, && (6.34)\\ x_k &= x_{k-1} + \eta_k \tilde{p}_k \quad \text{from equation (6.32)}, && (6.35)\\ \tilde{p}_k &= q_k - l_{k-1}\tilde{p}_{k-1} \quad \text{from equation (6.33)}. && (6.36) \end{aligned}$$

In order to eliminate q_k, substitute $q_k = r_{k-1}/\|r_{k-1}\|_2$ and $p_k \equiv \|r_{k-1}\|_2 \tilde{p}_k$ into the above recurrences to get

$$\begin{aligned} r_k &= r_{k-1} - \frac{\eta_k}{\|r_{k-1}\|_2} Ap_k \\ &\equiv r_{k-1} - \nu_k Ap_k, && (6.37)\\ x_k &= x_{k-1} + \frac{\eta_k}{\|r_{k-1}\|_2} p_k \\ &\equiv x_{k-1} + \nu_k p_k, && (6.38)\\ p_k &= r_{k-1} - \frac{\|r_{k-1}\|_2 l_{k-1}}{\|r_{k-2}\|_2} \cdot p_{k-1} \\ &\equiv r_{k-1} + \mu_k \cdot p_{k-1}. && (6.39) \end{aligned}$$

We still need formulas for the scalars ν_k and μ_k. As we will see, there are several equivalent mathematical expression for them in terms of dot products of vectors computed by the algorithm. Our ultimate formulas are chosen to minimize the number of dot products needed and because they are more stable than the alternatives.

To get a formula for ν_k, first we multiply both sides of equation (6.39) on the left by $p_k^T A$ and use the fact that p_k and p_{k-1} are A-conjugate (Lemma 6.8) to get

$$p_k^T Ap_k = p_k^T Ar_{k-1} + 0 = r_{k-1}^T Ap_k. \tag{6.40}$$

Then, multiply both sides of equation (6.37) on the left by r_{k-1}^T and use the fact that $r_{k-1}^T r_k = 0$ (since the r_i are parallel to the columns of the orthogonal matrix Q) to get

$$\begin{aligned} \nu_k &= \frac{r_{k-1}^T r_{k-1}}{r_{k-1}^T Ap_k} \\ &= \frac{r_{k-1}^T r_{k-1}}{p_k^T Ap_k} \quad \text{by equation (6.40)}. && (6.41) \end{aligned}$$

(Equation (6.41) can also be derived from a property of ν_k in Theorem 6.8, namely, that it minimizes the residual norm

$$\begin{aligned} \|r_k\|_{A^{-1}}^2 &= r_k^T A^{-1} r_k \\ &= (r_{k-1} - \nu_k Ap_k)^T A^{-1}(r_{k-1} - \nu_k Ap_k) \quad \text{by equation (6.37)} \\ &= r_{k-1} A^{-1} r_{k-1}^T - 2\nu_k p_k^T r_{k-1} + \nu_k^2 p_k^T Ap_k. \end{aligned}$$

This expression is a quadratic function of ν_k, so it can be easily minimized by setting its derivative with respect to ν_k to zero and solving for ν_k. This yields

$$\begin{aligned}\nu_k &= \frac{p_k^T r_{k-1}}{p_k^T A p_k} \\ &= \frac{(r_{k-1} + \mu_k \cdot p_{k-1})^T r_{k-1}}{p_k^T A p_k} \quad \text{by equation (6.39)} \\ &= \frac{r_{k-1}^T r_{k-1}}{p_k^T A p_k},\end{aligned}$$

where we have used the fact that $p_{k-1}^T r_{k-1} = 0$, which holds since r_{k-1} is orthogonal to all vectors in $\mathcal{K}_{k-1}$, including p_{k-1}.)

To get a formula for μ_k, multiply both sides of equation (6.39) on the left by $p_{k-1}^T A$ and use the fact that p_k and p_{k-1} are A-conjugate (Lemma 6.8) to get

$$\mu_k = -\frac{p_{k-1}^T A r_{k-1}}{p_{k-1}^T A p_{k-1}}. \tag{6.42}$$

The trouble with this formula for μ_k is that it requires another dot product, $p_{k-1}^T A r_{k-1}$, besides the two required for ν_k. So we will derive another formula requiring no new dot products.

We do this by deriving an alternate formula for ν_k: Multiply both sides of equation (6.37) on the left by r_k^T, again use the fact that $r_{k-1}^T r_k = 0$, and solve for ν_k to get

$$\nu_k = -\frac{r_k^T r_k}{r_k^T A p_k}. \tag{6.43}$$

Equating the two expressions (6.41) and (6.43) for ν_{k-1} (note that we have subtracted 1 from the subscript), rearranging, and comparing to equation (6.42) yield our ultimate formula for μ_k:

$$\begin{aligned}\mu_k &= -\frac{p_{k-1}^T A r_{k-1}}{p_{k-1}^T A p_{k-1}} \\ &= \frac{r_{k-1}^T r_{k-1}}{r_{k-2}^T r_{k-2}}.\end{aligned} \tag{6.44}$$

Combining recurrences (6.37), (6.38), and (6.39) and formulas (6.41) and (6.44) yields our final implementation of the conjugate gradient algorithm.

ALGORITHM 6.11. *Conjugate gradient algorithm:*

$k = 0$; $x_0 = 0$; $r_0 = b$; $p_1 = b$;
repeat
  $k = k + 1$

$$
\begin{aligned}
&z = A \cdot p_k \\
&\nu_k = (r_{k-1}^T r_{k-1})/(p_k^T z) \\
&x_k = x_{k-1} + \nu_k p_k \\
&r_k = r_{k-1} - \nu_k z \\
&\mu_{k+1} = (r_k^T r_k)/(r_{k-1}^T r_{k-1}) \\
&p_{k+1} = r_k + \mu_{k+1} p_k \\
&\textit{until } \|r_k\|_2 \textit{ is small enough}
\end{aligned}
$$

The cost of the inner loop for CG is one matrix-vector product $z = A \cdot p_k$, two inner products (by saving the value of $r_k^T r_k$ from one loop iteration to the next), three saxpys, and a few scalar operations. The only vectors that need to be stored are the current values of r, x, p, and $z = Ap$. For more implementation details, including how to decide if "$\|r_k\|_2$ is small enough," see NETLIB/templates/Templates.html.

6.6.4. Convergence Analysis of the Conjugate Gradient Method

We begin with a convergence analysis of CG that depends only on the condition number of A. This analysis will show that the number of CG iterations needed to reduce the error by a fixed factor less than 1 is proportional to the square root of the condition number. This worst-case analysis is a good estimate for the speed of convergence on our model problem, Poisson's equation. But it severely *underestimates* the speed of convergence in many other cases. After presenting the bound based on the condition number, we describe when we can expect faster convergence.

We start with the initial approximate solution $x_0 = 0$. Recall that x_k minimizes the A^{-1}-norm of the residual $r_k = b - Ax_k$ over all possible solutions $x_k \in \mathcal{K}_k(A, b)$. This means x_k minimizes

$$\|b - Az\|_{A^{-1}}^2 \equiv f(z) = (b - Az)^T A^{-1}(b - Az) = (x - z)^T A(x - z)$$

over all $z \in \mathcal{K}_k = \text{span}[b, Ab, A^2b, \ldots, A^{k-1}b]$. Any $z \in \mathcal{K}_k(A, b)$ may be written $z = \sum_{j=0}^{k-1} \alpha_j A^j b = p_{k-1}(A)b = p_{k-1}(A)Ax$, where $p_{k-1}(\xi) = \sum_{j=0}^{k-1} \alpha_j \xi^j$ is a polynomial of degree $k - 1$. Therefore,

$$
\begin{aligned}
f(z) &= [(I - p_{k-1}(A)A)x]^T A[(I - p_{k-1}(A)A)x] \\
&\equiv (q_k(A)x)^T A(q_k(A)x) \\
&= x^T q_k(A) A q_k(A) x,
\end{aligned}
$$

where $q_k(\xi) \equiv 1 - p_{k-1}(\xi) \cdot \xi$ is a degree-k polynomial with $q_k(0) = 1$. Note that $(q_k(A))^T = q_k(A)$ because $A = A^T$. Letting $\mathcal{Q}_k$ be the set of all degree-k polynomials which take the value 1 at 0, this means

$$f(x_k) = \min_{z \in \mathcal{K}_k} f(z) = \min_{q_k \in \mathcal{Q}_k} x^T q_k(A) A q_k(A) x. \tag{6.45}$$

To simplify this expression, write the eigendecomposition $A = Q\Lambda Q^T$ and let $Q^T x = y$ so that

$$\begin{aligned}
f(x_k) = \min_{z \in \mathcal{K}_k} f(z) &= \min_{q_k \in \mathcal{Q}_k} x^T (q_k(Q\Lambda Q^T))(Q\Lambda Q^T)(q_k(Q\Lambda Q^T))x \\
&= \min_{q_k \in \mathcal{Q}_k} x^T (Q q_k(\Lambda) Q^T)(Q\Lambda Q^T)(Q q_k(\Lambda) Q^T)x \\
&= \min_{q_k \in \mathcal{Q}_k} y^T q_k(\Lambda)\Lambda q_k(\Lambda) y \\
&= \min_{q_k \in \mathcal{Q}_k} y^T \cdot \operatorname{diag}(q_k(\lambda_i)\lambda_i q_k(\lambda_i)) \cdot y \\
&= \min_{q_k \in \mathcal{Q}_k} \sum_{i=1}^{n} {y_i}^2 \lambda_i (q_k(\lambda_i))^2 \\
&\leq \min_{q_k \in \mathcal{Q}_k} \left(\max_{\lambda_i \in \lambda(A)} (q_k(\lambda_i))^2 \right) \sum_{i=1}^{n} {y_i}^2 \lambda_i \\
&= \min_{q_k \in \mathcal{Q}_k} \left(\max_{\lambda_i \in \lambda(A)} (q_k(\lambda_i))^2 \right) f(x_0)
\end{aligned}$$

since $x_0 = 0$ implies $f(x_0) = x^T A x = y^T \Lambda y = \sum_{i=1}^{n} {y_i}^2 \lambda_i$. Therefore,

$$\frac{\|r_k\|_{A^{-1}}^2}{\|r_0\|_{A^{-1}}^2} = \frac{f(x_k)}{f(x_0)} \leq \min_{q_k \in \mathcal{Q}} \max_{\lambda_i \in \lambda(A)} (q_k(\lambda_i))^2$$

or

$$\frac{\|r_k\|_{A^{-1}}}{\|r_0\|_{A^{-1}}} \leq \min_{q_k \in \mathcal{Q}} \max_{\lambda_i \in \lambda(A)} |q_k(\lambda_i)|.$$

We have thus reduced the question of how fast CG converges to a question about polynomials: How small can a degree-k polynomial $q_k(\xi)$ be when ξ ranges over the eigenvalues of A, while simultaneously satisfying $q_k(0) = 1$? Since A is positive definite, its eigenvalues lie in the interval $[\lambda_{\min}, \lambda_{\max}]$, where $0 < \lambda_{\min} \leq \lambda_{\max}$, so to get a simple upper bound we will instead seek a degree-k polynomial $\hat{q}_k(\xi)$ that is small on the whole interval $[\lambda_{\min}, \lambda_{\max}]$ and 1 at 0. A polynomial $\hat{q}_k(\xi)$ that has this property is easily constructed from the Chebyshev polynomials $T_k(\xi)$ discussed in section 6.5.6. Recall that $|T_k(\xi)| \leq 1$ when $|\xi| \leq 1$ and increases rapidly when $|\xi| > 1$ (see Figure 6.6). Now let

$$\hat{q}_k(\xi) = T_k\left(\frac{\lambda_{\max} + \lambda_{\min} - 2\xi}{\lambda_{\max} - \lambda_{\min}}\right) \Big/ T_k\left(\frac{\lambda_{\max} + \lambda_{\min}}{\lambda_{\max} - \lambda_{\min}}\right).$$

It is easy to see that $\hat{q}(0) = 1$, and if $\xi \in [\lambda_{\min}, \lambda_{\max}]$, then

$$\left|\frac{\lambda_{\max} + \lambda_{\min} - 2\xi}{\lambda_{\max} - \lambda_{\min}}\right| \leq 1,$$

so

$$\begin{aligned}
\frac{\|r_k\|_{A^{-1}}}{\|r_0\|_{A^{-1}}} &\leq \min_{q_k \in \mathcal{Q}} \max_{\lambda_i \in \lambda(A)} |q_k(\lambda_i)| \\
&\leq \frac{1}{T_k(\frac{\lambda_{\max}+\lambda_{\min}}{\lambda_{\max}-\lambda_{\min}})} = \frac{1}{T_k(\frac{\kappa+1}{\kappa-1})} = \frac{1}{T_k(1 + \frac{2}{\kappa-1})},
\end{aligned} \tag{6.46}$$

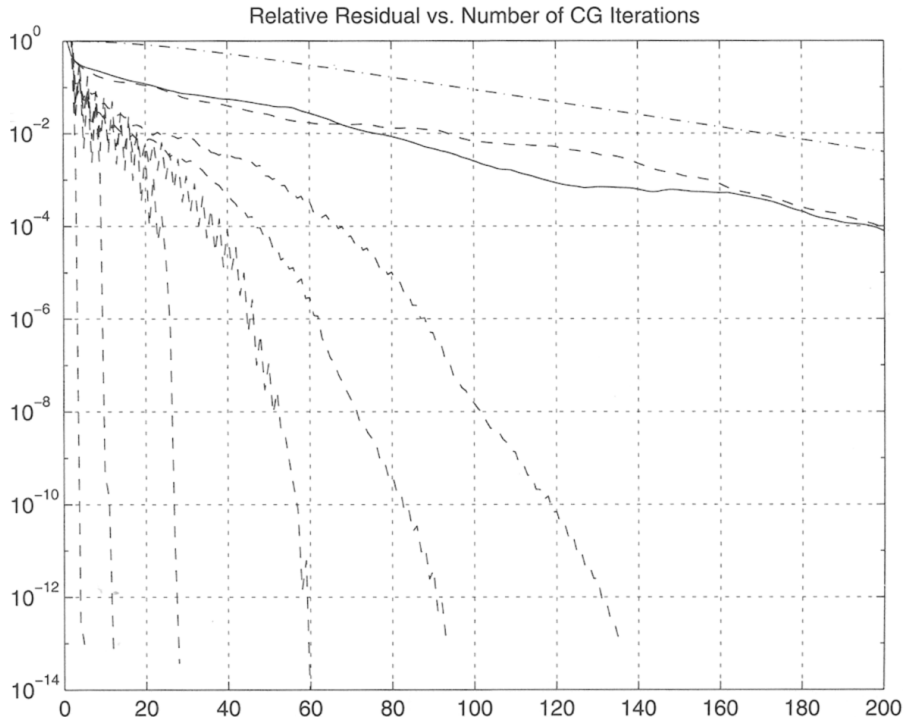


Fig. 6.7. *Graph of relative residuals computed by CG.*

where $\kappa = \lambda_{\max}/\lambda_{\min}$ is the condition number of A.

If the condition number κ is near 1, $1 + 2/(\kappa - 1)$ is large, $1/T_k(1 + \frac{2}{\kappa-1})$ is small, and convergence is rapid. If κ is large, convergence slows down, with the A^{-1}-norm of the residual r_k going to zero like

$$\frac{1}{T_k(1 + \frac{2}{\kappa-1})} \leq \frac{2}{1 + \frac{2k}{\sqrt{\kappa-1}}}.$$

EXAMPLE 6.14. For the N-by-N model problem, $\kappa = O(N^2)$, so after k steps of CG the residual is multiplied by about $(1 - O(N^{-1}))^k$, the same as SOR with optimal overrelaxation parameter ω. In other words, CG takes $O(N) = O(n^{1/2})$ iterations to converge. Since each iteration costs $O(n)$, the overall cost is $O(n^{3/2})$. This explains the entry for CG in Table 6.1. ◇

This analysis using the condition number does not explain all the important convergence behavior of CG. The next example shows that the entire distribution of eigenvalues of A is important, not just the ratio of the largest to the smallest one.

EXAMPLE 6.15. Let us consider Figure 6.7, which plots the relative residual $\|r_k\|_2/\|r_0\|_2$ at each CG step for eight different linear systems. The relative residual $\|r_k\|_2/\|r_0\|_2$ measures the speed of convergence; our implementation of CG terminates when this ratio sinks below 10^{-13}, or after $k = 200$ steps, whichever comes first.

All eight linear systems shown have the same dimension $n = 10^4$ and the same condition number $\kappa \approx 4134$, yet their convergence behaviors are radically different. The uppermost (dash-dot) line is $1/T_k(1 + \frac{2}{\kappa-1})$, which inequality (6.46) tells us is an upper bound on $\|r_k\|_{A^{-1}}/\|r_0\|_{A^{-1}}$. It turns out the graphs of $\|r_k\|_2/\|r_0\|_2$ and the graphs of $\|r_k\|_{A^{-1}}/\|r_0\|_{A^{-1}}$ are nearly the same, so we plot only the former, which are easier to interpret.

The solid line is $\|r_k\|_2/\|r_0\|_2$ for Poisson's equation on a 100-by-100 grid with a random right-hand side b. We see that the upper bound captures its general convergence behavior. The seven dashed lines are plots of $\|r_k\|_2/\|r_0\|_2$ for seven diagonal linear systems $D_i x = b$, numbered from D_1 on the left to D_7 on the right. Each D_i has the same dimension and condition number as Poisson's equation, so we need to study them more closely to understand their differing convergence behaviors.

We have constructed each D_i so that its smallest m_i and largest m_i eigenvalues are identical to those of Poisson's equation, with the remaining $n - 2m_i$ eigenvalues equal to the geometric mean of the largest and smallest eigenvalues. In other words, D_i has only $d_i = 2m_i + 1$ distinct eigenvalues. We let k_i denote the number of CG iterations it takes for the solution of $D_i x = b$ to reach $\|r_k\|_2/\|r_0\|_2 \leq 10^{-13}$. The convergence properties are summarized in the following table:

Example number	i	1	2	3	4	5	6	7
Number of distinct eigenvalues	d_i	3	11	41	81	201	401	5000
Number of steps to converge	k_i	3	11	27	59	94	134	> 200

We see that the number k_i of steps required to converge grows with the number d_i of distinct eigenvalues. D_7 has the same spectrum as Poisson's equation, and converges about as slowly.

In the absence of roundoff, we claim that CG would take *exactly* $k_i = d_i$ steps to converge. The reason is that we can find a polynomial $q_{d_i}(\xi)$ of degree d_i that is zero at the eigenvalues α_j of A, while $q_{d_i}(0) = 1$, namely,

$$q_{d_i}(\xi) = \frac{\prod_{j=1}^{d_i}(\alpha_j - \xi)}{\prod_{j=1}^{d_i}(\alpha_j)}.$$

Equation (6.45) tells us that after d_i steps, CG minimizes $\|r_{d_i}\|^2_{A^{-1}} = f(x_{d_i})$ over all possible degree-d_i polynomials equaling 1 at 0. Since q_{d_i} is one of those polynomials and $q_{d_i}(A) = 0$, we must have $\|r_{d_i}\|^2_{A^{-1}} = 0$, or $r_{d_i} = 0$. $\diamond$

One lesson of Example 6.15 is that if the largest and smallest eigenvalues of A are few in number (or clustered closely together), then CG will converge much more quickly than an analysis based just on A's condition number would indicate.

Another lesson is that the behavior of CG in floating point arithmetic can differ significantly from its behavior in exact arithmetic. We saw this because

the number d_i of distinct eigenvalues frequently differed from the number k_i of steps required to converge, although in theory we showed that they should be identical. Still, d_i and k_i were of the same order of magnitude.

Indeed, if one were to perform CG in exact arithmetic and compare the computed solutions and residuals with those computed in floating point arithmetic, they would very probably diverge and soon be quite different. Still, as long as A is not too ill-conditioned, the floating point result will eventually converge to the desired solution of $Ax = b$, and so CG is still very useful. The fact that the exact and floating point results can differ dramatically is interesting but does not prevent the practical use of CG.

When CG was discovered, it was proven that in exact arithmetic it would provide the exact answer after n steps, since then r_{n+1} would be orthogonal to n other orthogonal vectors r_1 through r_n, and so must be zero. In other words, CG was thought of as a *direct method* rather than an *iterative method.* When convergence after n steps did not occur in practice, CG was considered unstable and then abandoned for many years. Eventually it was recognized as a perfectly good iterative method, often providing quite accurate answers after $k \ll n$ steps.

Recently, a subtle backward error analysis was devised to explain the observed behavior of CG in floating point and explain how it can differ from exact arithmetic [123]. This behavior can also include long "plateaus" in the convergence, with $\|r_k\|_2$ decreasing little for many iterations, interspersed with periods of rapid convergence. This behavior can be explained by showing that CG applied to $Ax = b$ in floating point arithmetic behaves *exactly* like CG applied to $\tilde{A}\tilde{x} = \tilde{b}$ in exact arithmetic, where $\tilde{A}$ is close to A in the following sense: $\tilde{A}$ has a much larger dimension than A, but $\tilde{A}$'s eigenvalues all lie in narrow clusters around the eigenvalues of A. Thus the plateaus in convergence correspond to the polynomial q_k underlying CG developing more and more zeros near the eigenvalues of $\tilde{A}$ lying in a cluster.

6.6.5. Preconditioning

In the previous section we saw that the convergence rate of CG depended on the condition number of A, or more generally the distribution of A's eigenvalues. Other Krylov subspace methods have the same property. *Preconditioning* means replacing the system $Ax = b$ with the system $M^{-1}Ax = M^{-1}b$, where M is an approximation to A with the properties that

1. M is symmetric and positive definite,
2. $M^{-1}A$ is well conditioned or has few extreme eigenvalues,
3. $Mx = b$ is easy to solve.

A careful, problem-dependent choice of M can often make the condition number of $M^{-1}A$ much smaller than the condition number of A and thus accelerate

convergence dramatically. Indeed, a good preconditioner is often necessary for an iterative method to converge at all, and much current research in iterative methods is directed at finding better preconditioners (see also section 6.10).

We cannot apply CG directly to the system $M^{-1}Ax = M^{-1}b$, because $M^{-1}A$ is generally not symmetric. We derive the *preconditioned conjugate gradient method* as follows. Let $M = Q\Lambda Q^T$ be the eigendecomposition of M, and define $M^{1/2} \equiv Q\Lambda^{1/2}Q^T$. Note that $M^{1/2}$ is also symmetric positive definite, and $(M^{1/2})^2 = M$. Now multiply $M^{-1}Ax = M^{-1}b$ through by $M^{1/2}$ to get the new symmetric positive definite system $(M^{-1/2}AM^{-1/2})(M^{1/2}x) = M^{-1/2}b$, or $\hat{A}\hat{x} = \hat{b}$. Note that $\hat{A}$ and $M^{-1}A$ have the same eigenvalues since they are similar ($M^{-1}A = M^{-1/2}\hat{A}M^{1/2}$). We now apply CG *implicitly* to the system $\hat{A}\hat{x} = \hat{b}$ in such a way that avoids the need to multiply by $M^{-1/2}$. This yields the following algorithm.

ALGORITHM 6.12. *Preconditioned CG algorithm:*

$k = 0$; $x_0 = 0$; $r_0 = b$; $p_1 = M^{-1}b$; $y_0 = M^{-1}r_0$
repeat
  $k = k + 1$
  $z = A \cdot p_k$
  $\nu_k = (y_{k-1}^T r_{k-1})/(p_k^T z)$
  $x_k = x_{k-1} + \nu_k p_k$
  $r_k = r_{k-1} - \nu_k z$
  $y_k = M^{-1}r_k$
  $\mu_{k+1} = (y_k^T r_k)/(y_{k-1}^T r_{k-1})$
  $p_{k+1} = y_k + \mu_{k+1}p_k$
until $\|r_k\|_2$ *is small enough*

THEOREM 6.9. *Let A and M be symmetric positive definite, $\hat{A} = M^{-1/2}AM^{1/2}$, and $\hat{b} = M^{-1/2}b$. The CG algorithm applied to $\hat{A}\hat{x} = \hat{b}$,*

$k = 0$; $\hat{x}_0 = 0$; $\hat{r}_0 = \hat{b}$; $\hat{p}_1 = \hat{b}$;
repeat
  $k = k + 1$
  $\hat{z} = \hat{A} \cdot \hat{p}_k$
  $\hat{\nu}_k = (\hat{r}_{k-1}^T \hat{r}_{k-1})/(\hat{p}_k^T \hat{z})$
  $\hat{x}_k = \hat{x}_{k-1} + \hat{\nu}_k \hat{p}_k$
  $\hat{r}_k = \hat{r}_{k-1} - \hat{\nu}_k \hat{z}$
  $\hat{\mu}_{k+1} = (\hat{r}_k^T \hat{r}_k)/(\hat{r}_{k-1}^T \hat{r}_{k-1})$
  $\hat{p}_{k+1} = \hat{r}_k + \hat{\mu}_{k+1}\hat{p}_k$
until $\|\hat{r}_k\|_2$ *is small enough*

and Algorithm 6.12 *are related as follows:*

$$\begin{aligned} \hat{\mu}_k &= \mu_k, \\ \hat{\nu}_k &= \nu_k, \end{aligned}$$

$$\begin{aligned} \hat{z} &= M^{-1/2} z, \\ \hat{x}_k &= M^{1/2} x_k, \\ \hat{r}_k &= M^{-1/2} r_k, \\ \hat{p}_k &= M^{1/2} p_k. \end{aligned}$$

Therefore, x_k converges to $M^{-1/2}$ times the solution of $\hat{A}\hat{x} = \hat{b}$, i.e., to $M^{-1/2}\hat{A}^{-1}\hat{b} = A^{-1}b$.

For a proof, see Question 6.14.

Now we describe some common preconditioners. Note that our twin goals of minimizing the condition number of $M^{-1}A$ and keeping $Mx = b$ easy to solve are in conflict with one another: Choosing $M = A$ minimizes the condition number of $M^{-1}A$ but leaves $Mx = b$ as hard to solve as the original problem. Choosing $M = I$ makes solving $Mx = b$ trivial but leaves the condition number of $M^{-1}A$ unchanged. Since we need to solve $Mx = b$ in the inner loop of the algorithm, we restrict our discussion to those M for which solving $Mx = b$ is easy, and describe when they are likely to decrease the condition number of $M^{-1}A$.

- If A has widely varying diagonal entries, we may use the simple *diagonal preconditioner* $M = \text{diag}(\ a_{11},\ \ldots,\ a_{nn}\ )$. One can show that among all possible diagonal preconditioners, this choice reduces the condition number of $M^{-1}A$ to within a factor of n of its minimum value [244]. This is also called *Jacobi preconditioning.*

- As a generalization of the first preconditioner, let

$$A = \begin{bmatrix} A_{11} & \cdots & A_{1k} \\ \vdots & \ddots & \vdots \\ A_{k1} & \cdots & A_{kk} \end{bmatrix}$$

  be a block matrix, where the diagonal blocks A_{ii} are square. Then among all block diagonal preconditioners

$$M = \begin{bmatrix} M_{11} & & \\ & \ddots & \\ & & M_{kk} \end{bmatrix},$$

  where M_{ii} and A_{ii} have the same dimensions, the choice $M_{ii} = A_{ii}$ minimizes the condition number of $M^{-1/2}AM^{-1/2}$ to within a factor of k [69]. This is also called *block Jacobi preconditioning.*

- Like Jacobi, SSOR can also be used to create a (block) preconditioner.

- An *incomplete Cholesky factorization* LL^T of A is an approximation $A \approx LL^T$, where L is limited to a particular sparsity pattern, such as

the original pattern of A. In other words, no fill-in is allowed during Cholesky. Then $M = LL^T$ is used. (For nonsymmetric problems, there is a corresponding *incomplete LU preconditioner.*)

- *Domain decomposition* is used when A represents an equation (such as Poisson's equation) on a physical region Ω. So far, for Poisson's equation, we have let Ω be the unit square. More generally, the region Ω may be broken up into disjoint (or slightly overlapping) subregions $\Omega = \cup_j \Omega_j$, and the equation may be solved on each subregion independently. For example, if we are solving Poisson's equation and if the subregions are squares or rectangles, these subproblems can be solved very quickly using FFTs (see section 6.7). Solving these subproblem corresponds to a block diagonal M (if the subregions are disjoint) or a product of block diagonal M (if the subregions overlap). This is discussed in more detail in section 6.10.

A number of these preconditioners have been implemented in the software packages PETSc [232] and PARPRE (NETLIB/scalapack/parpre.tar.gz).

6.6.6. Other Krylov Subspace Algorithms for Solving $Ax = b$

So far we have concentrated on the symmetric positive definite linear systems and minimized the A^{-1}-norm of the residual. In this section we describe methods for other kinds of linear systems and offer advice on which method to use, based on simple properties of the matrix. See Figure 6.8 for a summary [15, 107, 136, 214] and NETLIB/templates for details, in particular for more comprehensive advice on choosing a method, along with software.

Any system $Ax = b$ can be changed to a symmetric positive definite system by solving the normal equations $A^TAx = A^Tb$ (or $AA^Ty = b$, $x = A^Ty$). This includes the least squares problem $\min_x \|Ax - b\|_2$. This lets us use CG, provided that we can multiply vectors both by A and A^T. Since the condition number of A^TA or AA^T is the square of the condition number of A, this method can lead to slow convergence if A is ill-conditioned but is fast if A is well-conditioned (or A^TA has a "good" distribution of eigenvalues, as discussed in section 6.6.4).

We can minimize the two-norm of the residual instead of the A^{-1}-norm when A is symmetric positive definite. This is called the minimum residual algorithm, or MINRES [194]. Since MINRES is more expensive than CG and is often less accurate because of numerical instabilities, it is not used for positive definite systems. But MINRES can be used when the matrix is symmetric indefinite, whereas CG cannot. In this case, we can also use the SYMMLQ algorithm of Paige and Saunders [194], which produces a residual $r_k \perp \mathcal{K}_k(A, b)$ at each step.

Unfortunately, there are few matrices other than symmetric matrices where algorithms like CG exist that simultaneously

1. either minimize the residual $\|r_k\|_2$ or keep it orthogonal $r_k \perp \mathcal{K}_k$,
2. require a fixed number of dot products and saxpy's in the inner loop, independent of k.

Essentially, algorithms satisfying these two properties exist only for matrices of the form $e^{i\theta}(T + \sigma I)$, where $T = T^T$ (or $TH = (HT)^T$ for some symmetric positive definite H), θ is real, and σ is complex [102, 251]. For these symmetric and special nonsymmetric A, it turns out we can find a short recurrence, as in the Lanczos algorithm, for computing an orthogonal basis $[q_1, \ldots, q_k]$ of $\mathcal{K}_k(A, b)$. The fact that there are just a few terms in the recurrence for updating q_k means that it can be computed very efficiently.

This existence of short recurrences no longer holds for general nonsymmetric A. In this case, we can use Arnoldi's algorithm. So instead of the tridiagonal matrix $T_k = Q_k^T A Q_k$, we get a fully upper Hessenberg matrix $H_k = Q_k^T A Q_k$. The *GMRES algorithm (generalized minimum residual)* uses this decomposition to choose $x_k = Q_k y_k \in \mathcal{K}_k(A, b)$ to minimize the residual

$$
\begin{aligned}
\|r_k\|_2 &= \|b - Ax_k\|_2 \\
&= \|b - AQ_k y_k\|_2 \\
&= \|b - (QHQ^T)Q_k y_k\|_2 \quad \text{by equation (6.30)} \\
&= \|Q^T b - HQ^T Q_k y_k\|_2 \quad \text{since } Q \text{ is orthogonal} \\
&= \left\| e_1 \|b\|_2 - \begin{bmatrix} H_k & H_{uk} \\ H_{ku} & H_u \end{bmatrix} \cdot \begin{bmatrix} y_k \\ 0 \end{bmatrix} \right\|_2 \\
&\qquad \text{by equation (6.30) and since the first column of} \\
&\qquad Q = [Q_k, Q_u] \text{ is } b/\|b\|_2 \\
&= \left\| e_1 \|b\|_2 - \begin{bmatrix} H_k \\ H_{ku} \end{bmatrix} y_k \right\|_2 .
\end{aligned}
$$

Since only the first row of H_{ku} is nonzero, this is a $(k+1)$-by-k upper Hessenberg least squares problem for the entries of y_k. Since it is upper Hessenberg, the QR decomposition needed to solve it can be accomplished with k Givens rotations, at a cost of $O(k^2)$ instead of $O(k^3)$. Also, the storage required is $O(kn)$, since Q_k must be stored. One way to limit the growth in cost and storage is to *restart* GMRES, i.e., taking the answer x_k computed after k steps, restarting GMRES to solve the linear system $Ad = r_k = b - Ax_k$, and updating the solution to get $x_k + d$; this is called GMRES(k). Still, even GMRES(k) is more expensive than CG, where the cost of the inner loop does not depend on k at all.

Another approach to nonsymmetric linear systems is to abandon computing an orthonormal basis of $\mathcal{K}_k(A, b)$ and compute a nonorthonormal basis that again reduces A to (nonsymmetric) tridiagonal form. This is called the *nonsymmetric Lanczos method* and requires matrix-vector multiplication by both A and A^T. This is important because $A^T z$ is sometimes harder (or impossible)

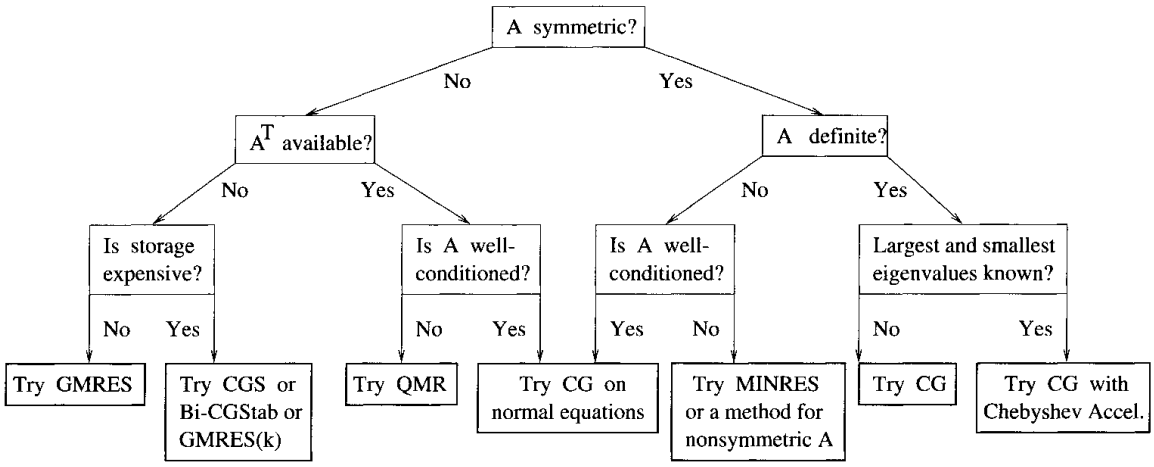


Fig. 6.8. *Decision tree for choosing an iterative algorithm for* $Ax = b$. *Bi-CGStab = bi-conjugate gradient stabilized. QMR = quasi-minimum residuals. CGS = CG squared.*

to compute (see Example 6.13). The advantage of tridiagonal form is that it is much easier to solve with a tridiagonal matrix than a Hessenberg one. The disadvantage is that the basis vectors may be very ill-conditioned and may in fact fail to exist at all, a phenomenon called *breakdown.* The potential efficiency has led to a great deal of research on avoiding or alleviating this instability (*look-ahead Lanczos*) and to competing methods, including *biconjugate gradients* and *quasi-minimum residuals.* There are also some versions that do not require multiplication by A^T, including *conjugate gradients squared* and *bi-conjugate gradient stabilized.* No one method is best in all cases.

Figure 6.8 shows a decision tree giving simple advice on which method to try first, assuming that we have no other deep knowledge of the matrix A (such as that it arises from the Poisson equation).

6.7. Fast Fourier Transform

In this section i will always denote $\sqrt{-1}$.

We begin by showing how to solve the two-dimensional Poisson's equation in a way requiring multiplication by the matrix of eigenvectors of T_N. A straightforward implementation of this matrix-matrix multiplication would cost $O(N^3) = O(n^{3/2})$ operations, which is expensive. Then we show how this multiplication can be implemented using the FFT in only $O(N^2 \log N) = O(n \log n)$ operations, which is within a factor of $\log n$ of optimal.

This solution is a discrete analogue of the Fourier series solution of the original differential equation (6.1) or (6.6). Later we will make this analogy more precise.

Let $T_N = Z \Lambda Z^T$ be the eigendecomposition of T_N, as defined in Lemma 6.1. We begin with the formulation of the two-dimensional Poisson's equation in

equation (6.11):

$$T_N V + V T_N = h^2 F.$$

Substitute $T_N = Z\Lambda Z^T$ and multiply by the Z^T on the left and Z on the right to get

$$Z^T(Z\Lambda Z^T)VZ + Z^T V(Z\Lambda Z^T)Z = Z^T(h^2 F)Z$$

or

$$\Lambda V' + V'\Lambda = h^2 F',$$

where $V' = Z^T V Z$ and $F' = Z^T F Z$. The (j,k)th entry of this last equation is

$$(\Lambda V' + V'\Lambda)_{jk} = \lambda_j v'_{jk} + v'_{jk}\lambda_k = h^2 f'_{jk},$$

which can be solved for v'_{jk} to get

$$v'_{jk} = \frac{h^2 f'_{jk}}{\lambda_j + \lambda_k}.$$

This yields the first version of our algorithm.

ALGORITHM 6.13. *Solving the two-dimensional Poisson's equation using the eigendecomposition* $T_N = Z\Lambda Z^T$*:*

1) $F' = Z^T F Z$
2) *For all* j *and* k, $v'_{jk} = \frac{h^2 f'_{jk}}{\lambda_j + \lambda_k}$
3) $V = Z V' Z^T$

The cost of step 2 is $3N^2 = 3n$ operations, and the cost of steps 1 and 3 is 4 matrix-matrix multiplications by Z and $Z^T = Z$, which is $8N^3 = 8n^{3/2}$ operations using a conventional algorithm. In the next section we show how multiplication by Z is essentially the same as computing a *discrete Fourier transform*, which can be done in $O(N^2 \log N) = O(n \log n)$ operations using the FFT.

(Using the language of Kronecker products introduced in section 6.3.3, and in particular the eigendecomposition of $T_{N\times N}$ from Proposition 6.1,

$$T_{N\times N} = I \otimes T_N + T_N \otimes I = (Z \otimes Z) \cdot (I \otimes \Lambda + \Lambda \otimes I) \cdot (Z \otimes Z)^T,$$

we can rewrite the formula justifying Algorithm 6.13 as follows:

$$\begin{aligned} \mathrm{vec}(V) &= (T_{N\times N})^{-1} \cdot \mathrm{vec}(h^2 F) \\ &= ((Z \otimes Z) \cdot (I \otimes \Lambda + \Lambda \otimes I) \cdot (Z \otimes Z)^T)^{-1} \cdot \mathrm{vec}(h^2 F) \\ &= (Z \otimes Z)^{-T} \cdot (I \otimes \Lambda + \Lambda \otimes I)^{-1} \cdot (Z \otimes Z)^{-1} \cdot \mathrm{vec}(h^2 F) \\ &= (Z \otimes Z) \cdot (I \otimes \Lambda + \Lambda \otimes I)^{-1} \cdot (Z^T \otimes Z^T) \cdot \mathrm{vec}(h^2 F). \quad (6.47) \end{aligned}$$

We claim that doing the indicated matrix-vector multiplications from right to left is mathematically the same as Algorithm 6.13; see Question 6.9. This also shows how to extend the algorithm to Poisson's equation in higher dimensions.)

6.7.1. The Discrete Fourier Transform

In this subsection, we will number the rows and columns of matrices from 0 to $N-1$ instead of from 1 to N.

DEFINITION 6.17. *The* discrete Fourier transform (DFT) *of an N-vector x is the vector $y = \Phi x$, where Φ is an N-by-N matrix defined as follows. Let $\omega = e^{\frac{-2\pi i}{N}} = \cos\frac{2\pi}{N} - i \cdot \sin\frac{2\pi}{N}$, a* principal Nth root of unity. *Then $\phi_{jk} = \omega^{jk}$. The* inverse discrete Fourier transform (IDFT) *of y is the vector $x = \Phi^{-1}y$.*

LEMMA 6.9. *$\frac{1}{\sqrt{N}}\Phi$ is a symmetric unitary matrix, so $\Phi^{-1} = \frac{1}{N}\Phi^* = \frac{1}{N}\bar{\Phi}$.*

Proof. Clearly $\Phi = \Phi^T$, so $\bar{\Phi} = \Phi^*$, and we need only show $\Phi \cdot \bar{\Phi} = N \cdot I$. Compute $(\Phi\bar{\Phi})_{lj} = \sum_{k=0}^{N-1} \phi_{lk}\bar{\phi}_{kj} = \sum_{k=0}^{N-1} \omega^{lk}\bar{\omega}^{kj} = \sum_{k=0}^{N-1} \omega^{k(l-j)}$, since $\bar{\omega} = \omega^{-1}$. If $l = j$, this sum is clearly N. If $l \neq j$, it is a geometric sum with value $\frac{1-\omega^{N(l-j)}}{1-\omega^{l-j}} = 0$, since $\omega^N = 1$. □

Thus, both the DFT and IDFT are just matrix-vector multiplications and can be straightforwardly implemented in $2N^2$ flops. This operation is called a DFT because of its close mathematical relationship to two other kinds of Fourier analyses:

the Fourier transform	$F(\zeta) = \int_{-\infty}^{\infty} e^{-2\pi i\zeta x} f(x)dx$
and its inverse	$f(x) = \int_{-\infty}^{\infty} e^{+2\pi i\zeta x} F(\zeta)d\zeta$
the Fourier series	$c_j = \int_0^1 e^{-2\pi ijx} f(x)dx$
where f is periodic on $[0,1]$	
and its inverse	$f(x) = \sum_{j=-\infty}^{\infty} e^{+2\pi ijx} c_j$
the DFT	$y_j = (\Phi x)_j = \sum_{k=0}^{N-1} e^{-2\pi ijk/N} x_k$
and its inverse	$x_k = (\Phi^{-1}y)_k = \frac{1}{N}\sum_{j=0}^{N-1} e^{+2\pi ijk/N} y_j$

We will make this close relationship more concrete in two ways. First, we will show how to solve the model problem using the DFT and then the original Poisson's equation (6.1) using Fourier series. This example will motivate us to find a fast way to multiply by Φ, because this will give us a fast way to solve the model problem. This fast way is called the *fast Fourier transform* or *FFT*. Instead of $2N^2$ flops, it will require only about $\frac{3}{2}N\log_2 N$ flops, which is much less. We will derive the FFT by stressing a second mathematical relationship shared among the different kinds of Fourier analyses: reducing convolution to multiplication.

In Algorithm 6.13 we showed that to solve the discrete Poisson equation $T_N V + V T_N = h^2 F$ for V required the ability to multiply by the N-by-N matrix Z, where

$$z_{jk} = \sqrt{\frac{2}{N+1}} \sin\frac{\pi(j+1)(k+1)}{N+1}.$$

(Recall that we number rows and columns from 0 to $N-1$ in this section.) Now consider the $(2N+2)$-by-$(2N+2)$ DFT matrix Φ, whose j,k entry is

$$\exp\left(\frac{-2\pi ijk}{2N+2}\right) = \exp\left(\frac{-\pi ijk}{N+1}\right) = \cos\frac{\pi jk}{N+1} - i\cdot\sin\frac{\pi jk}{N+1}.$$

Thus the N-by-N matrix Z consists of $-\sqrt{\frac{2}{N+1}}$ times the imaginary part of the second through $(N+1)$st rows and columns of Φ. So if we can multiply efficiently by Φ using the FFT, then we can multiply efficiently by Z. (To be most efficient, one modifies the FFT algorithm, which we describe below, to multiply by Z directly; this is called the *fast sine transform*. But one can also just use the FFT.) Thus, multiplying ZF quickly requires an FFT-like operation on each column of F, and multiplying FZ requires the same operation on each row. (In three dimensions, we would let V be an N-by-N-by-N array of unknowns and apply the same operation to each of the $3N^2$ sections parallel to the coordinate axes.)

6.7.2. Solving the Continuous Model Problem Using Fourier Series

We now return to numbering rows and columns of matrices from 1 to N.

In this section we show how the algorithm for solving the discrete model problem is a natural analogue of using Fourier series to solve the original differential equation (6.1). We will do this for the one-dimensional model problem.

Recall that Poisson's equation on $[0,1]$ is $-\frac{d^2v}{dx^2} = f(x)$ with boundary conditions $v(0) = v(1)$. To solve this, we will expand $v(x)$ in a Fourier series: $v(x) = \sum_{j=1}^{\infty}\alpha_j\sin(j\pi x)$. (The boundary condition $v(1)=0$ tells us that no cosine terms appear.) Plugging $v(x)$ into Poisson's equation yields

$$\sum_{j=1}^{\infty}\alpha_j(j^2\pi^2)\sin(j\pi x) = f(x).$$

Multiply both sides by $\sin(k\pi x)$, integrate from 0 to 1, and use the fact that $\int_0^1\sin(j\pi x)\sin(k\pi x)dx = 0$ if $j\neq k$ and $1/2$ if $j=k$ to get

$$\alpha_k = \frac{2}{k^2\pi^2}\int_0^1\sin(k\pi x)f(x)dx$$

and finally

$$v(x) = \sum_{j=1}^{\infty}\left(\frac{2}{j^2\pi^2}\int_0^1\sin(j\pi y)f(y)dy\right)\sin(j\pi x). \tag{6.48}$$

Now consider the discrete model problem $T_Nv = h^2f$. Since $T_N = Z\Lambda Z^T$, we can write $v = T_N^{-1}h^2f = Z\Lambda^{-1}Z^Th^2f$, so

$$v_k = \sum_{j=1}^{N} z_{kj}\frac{h^2}{\lambda_j}(Z^Tf)_j = \sum_{j=1}^{N}\sin\frac{\pi jk}{N+1}\left(\frac{h^2}{\lambda_j}\sqrt{\frac{2}{N+1}}(Z^Tf)_j\right), \tag{6.49}$$

where

$$\begin{aligned}\sqrt{\frac{2}{N+1}}(Z^T f)_j &= \sqrt{\frac{2}{N+1}}\sum_{l=1}^{N}\sqrt{\frac{2}{N+1}}\sin\left(\frac{\pi jl}{N+1}\right)f_l \\ &= 2\sum_{l=1}^{N}\frac{1}{N+1}\sin\left(\frac{\pi jl}{N+1}\right)f_l \\ &\approx 2\int_0^1 \sin(\pi jy)f(y)dy,\end{aligned}$$

since the last sum is just a Riemann sum approximation of the integral. Furthermore, for small j, recall that $\frac{h^2}{\lambda_j} \approx \frac{1}{j^2\pi^2}$. So we see how the solution of the discrete problem (6.49) approximates the solution of the continuous problem (6.48), with multiplication by Z^T corresponding to multiplication by $\sin(j\pi x)$ and integration, and multiplication by Z corresponding to summing the different Fourier components.

6.7.3. Convolutions

The *convolution* is an important operation in Fourier analysis, whose definition depends on whether we are doing Fourier transforms, Fourier series, or the DFT:

Fourier transform	$(f * g)(x) \equiv \int_{-\infty}^{\infty} f(x-y)g(y)dy$
Fourier series	$(f * g)(x) \equiv \int_0^1 f(x-y)g(y)dy$
DFT	If $a = [a_0, \ldots, a_{N-1}, 0, \ldots, 0]^T$ and $b = [b_0, \ldots, b_{N-1}, 0, \ldots, 0]^T$ are $2N$-vectors, then $a * b \equiv c = [c_0, \ldots, c_{2N-1}]^T$, where $c_k = \sum_{j=0}^{k} a_j b_{k-j}$

To illustrate the use of the discrete convolution, consider polynomial multiplication. Let $\mathrm{a}(x) = \sum_{k=0}^{N-1} a_k x^k$ and $\mathrm{b}(x) = \sum_{k=0}^{N-1} b_k x^k$ be degree-$(N-1)$ polynomials. Then their product $\mathrm{c}(x) \equiv \mathrm{a}(x) \cdot \mathrm{b}(x) = \sum_{k=0}^{2N-1} c_k x^k$, where the coefficients $c_0, \ldots, c_{2N-1}$ are given by the discrete convolution.

One purpose of the Fourier transform, Fourier series, or DFT is to convert convolution into multiplication. In the case of the Fourier transform, $\mathcal{F}(f*g) = \mathcal{F}(f) \cdot \mathcal{F}(g)$; i.e., the Fourier transform of the convolution is the product of the Fourier transforms. In the case of Fourier series, $c_j(f * g) = c_j(f) \cdot c_j(g)$; i.e., the Fourier coefficients of the convolution are the product of the Fourier coefficients. The same is true of the discrete convolution.

THEOREM 6.10. *Let* $a = [a_0, \ldots, a_{N-1}, 0, \ldots, 0]^T$ *and* $b = [b_0, \ldots, b_{N-1}, 0, \ldots, 0]^T$ *be vectors of dimension* $2N$, *and let* $c = a * b = [c_0, \ldots, c_{2N-1}]^T$. *Then* $(\Phi c)_k = (\Phi a)_k \cdot (\Phi b)_k$.

Proof. If $a' = \Phi a$, then $a'_k = \sum_{j=0}^{2N-1} a_j \omega^{kj}$, the value of the polynomial $\mathrm{a}(x) \equiv \sum_{j=0}^{N-1} a_j x^j$ at $x = \omega^k$. Similarly $b' = \Phi b$ means $b'_k = \sum_{j=0}^{N-1} b_j \omega^{kj} = \mathrm{b}(\omega^k)$ and $c' = \Phi c$ means $c'_k = \sum_{j=0}^{2N-1} c_j \omega^{kj} = \mathrm{c}(\omega^k)$. Therefore

$$a'_k \cdot b'_k = \mathrm{a}(\omega^k) \cdot \mathrm{b}(\omega^k) = \mathrm{c}(\omega^k) = c'_k$$

as desired. □

In other words, the DFT is polynomial evaluation at the points $\omega^0, \ldots, \omega^{N-1}$, and conversely the IDFT is polynomial interpolation, producing the coefficients of a polynomial given its values at $\omega^0, \ldots, \omega^{N-1}$.

6.7.4. Computing the Fast Fourier Transform

We will derive the FFT via its interpretation as polynomial evaluation just discussed. The goal is to evaluate $\mathrm{a}(x) = \sum_{k=0}^{N-1} a_k x^k$ at $x = \omega^j$ for $0 \le j \le N-1$. For simplicity we will assume $N = 2^m$. Now write

$$\begin{aligned} \mathrm{a}(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{N-1} x^{N-1} \\ &= (a_0 + a_2 x^2 + a_4 x^4 + \cdots) + x(a_1 + a_3 x^2 + a_5 x^4 + \cdots) \\ &\equiv \mathrm{a}_{even}(x^2) + x \cdot \mathrm{a}_{odd}(x^2). \end{aligned}$$

Thus, we need to evaluate two polynomials a_{even} and a_{odd} of degree $\frac{N}{2} - 1$ at $(\omega^j)^2$, $0 \le j \le N-1$. But this is really just $\frac{N}{2}$ points ω^{2j} for $0 \le j \le \frac{N}{2} - 1$, since $\omega^{2j} = \omega^{2(j+\frac{N}{2})}$.

Thus evaluating a polynomial of degree $N - 1 = 2^m - 1$ at all N Nth roots of unity is the same as evaluating two polynomials of degree $\frac{N}{2} - 1$ at all $\frac{N}{2}$ $\frac{N}{2}$th roots of unity and then combining the results with N multiplications and additions. This can be done recursively.

ALGORITHM 6.14. *FFT (recursive version):*

```
function FFT(a, N)
    if N = 1
        return a
    else
        a'_even = FFT(a_even, N/2)
        a'_odd = FFT(a_odd, N/2)
        ω = e^(-2πi/N)
        w = [ω^0, ..., ω^(N/2-1)]
        return a' = [a'_even + w. * a'_odd, a'_even - w. * a'_odd]
    endif
```

Here . means componentwise multiplication of arrays (as in Matlab), and we have used the fact that* $\omega^{j+N/2} = -\omega^j$.

Let the cost of this algorithm be denoted $C(N)$. Then we see that $C(N)$ satisfies the recurrence $C(N) = 2C(N/2) + 3N/2$ (assuming that the powers of ω are precomputed and stored in tables). To solve this recurrence write

$$\begin{aligned} C(N) &= 2C\left(\frac{N}{2}\right) + \frac{3N}{2} = 4C\left(\frac{N}{4}\right) + 2 \cdot \frac{3N}{2} = 8C\left(\frac{N}{8}\right) + 3 \cdot \frac{3N}{2} \\ &= \cdots \\ &= \log_2 N \cdot \frac{3N}{2}. \end{aligned}$$

To compute the FFT of each column (or each row) of an N-by-N matrix therefore costs $\log_2 N \cdot \frac{3N^2}{2}$. This complexity analysis justifies the entry for the FFT in Table 6.1.

In practice, implementations of the FFT use simple nested loops rather than recursion in order to be as efficient as possible; see NETLIB/fftpack. In addition, these implementations sometimes return the components in *bit-reversed* order: This means that instead of returning $y_0, y_1, \ldots, y_{N-1}$, where $y = \Phi x$, the subscripts j are reordered so that the bit patterns are reversed. For example, if $N = 8$, the subscripts run from $0 = 000_2$ to $7 = 111_2$. The following table shows the normal order and the bit-reversed order:

normal increasing order	bit-reversed order
$0 = 000_2$	$0 = 000_2$
$1 = 001_2$	$4 = 100_2$
$2 = 010_2$	$2 = 010_2$
$3 = 011_2$	$6 = 110_2$
$4 = 100_2$	$1 = 001_2$
$5 = 101_2$	$5 = 101_2$
$6 = 110_2$	$3 = 011_2$
$7 = 111_2$	$7 = 111_2$

The inverse FFT undoes this reordering and returns the results in their original order. Therefore, these algorithms can be used for solving the model problem, provided that we divide by the appropriate eigenvalues, whose subscripts correspond to bit-reversed order. (Note that Matlab always returns results in normal increasing order.)

6.8. Block Cyclic Reduction

Block cyclic reduction is another fast ($O(N^2 \log_2 N)$) method for the model problem but is slightly more generally applicable than the FFT-based solution. The fastest algorithms for the model problem on vector computers are often a hybrid of block cyclic reduction and FFT.

First we describe a simple but numerically unstable version version of the algorithm; then we say a little about how to stabilize it. Write the model problem as

$$\begin{bmatrix} A & -I & & \\ -I & \ddots & \ddots & \\ & \ddots & \ddots & -I \\ & & -I & A \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_N \end{bmatrix},$$

where we assume that N, the dimension of $A = T_N + 2I_N$, is odd. Note also that x_i and b_i are N-vectors.

We use block Gaussian elimination to combine three consecutive sets of equations,

$$\begin{array}{rl} + & [\ -x_{j-2} + Ax_{j-1} - x_j = b_{j-1}\ ], \\ +A* & [\ -x_{j-1} + Ax_j - x_{j+1} = b_j\ ], \\ + & [\ -x_j + Ax_{j+1} - x_{j+2} = b_{j+1}\ ], \end{array}$$

thus eliminating x_{j-1} and x_{j+1}:

$$-x_{j-2} + (A^2 - 2I)x_j - x_{j+2} = b_{j-1} + Ab_j + b_{j+1}.$$

Doing this for every set of three consecutive equations yields two sets of equations: one for the x_j with j even,

$$\begin{bmatrix} B & -I & & & \\ -I & B & -I & & \\ & -I & \ddots & \ddots & \\ & & \ddots & \ddots & -I \\ & & & -I & B \end{bmatrix} \begin{bmatrix} x_2 \\ x_4 \\ \vdots \\ x_{N-1} \end{bmatrix} = \begin{bmatrix} b_1 + Ab_2 + b_3 \\ b_3 + Ab_4 + b_5 \\ \vdots \\ b_{N-2} + Ab_{N-1} + b_N \end{bmatrix}, \quad (6.50)$$

where $B = A^2 - 2I$, and one set of equations for the x_j with j odd, which we can solve after solving equation (6.50) for the odd x_j:

$$\begin{bmatrix} A & & & \\ & A & & \\ & & \ddots & \\ & & & A \end{bmatrix} \begin{bmatrix} x_1 \\ x_3 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} b_1 + x_2 \\ b_3 + x_2 + x_4 \\ \vdots \\ b_N + x_{N-1} \end{bmatrix}.$$

Note that equation (6.50) has the same form as the original problem, so we may repeat this process recursively. For example, at the next step we get

$$\begin{bmatrix} C & -I & & & \\ -I & C & -I & & \\ & -I & \ddots & \ddots & \\ & & \ddots & \ddots & -I \\ & & & -I & C \end{bmatrix} \begin{bmatrix} x_4 \\ x_8 \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix}, \quad \text{where } C = B^2 - 2I,$$

and

$$\begin{bmatrix} B & & & \\ & B & & \\ & & \ddots & \\ & & & B \end{bmatrix} \begin{bmatrix} x_2 \\ x_6 \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix}.$$

We repeat this until only one equation is left, which we solve another way.

We formalize this algorithm as follows: Assume $N = N_0 = 2^{k+1} - 1$, and let $N_r = 2^{k+1-r} - 1$. Let $A^{(0)} = A$ and $b_j{}^{(0)} = b_j$ for $j = 1, \ldots, N$.

ALGORITHM 6.15. *Block cyclic reduction:*

1) *Reduce:*

for $r = 0$ *to* $k - 1$
  $A^{(r+1)} = (A^{(r)})^2 - 2I$
  for $j = 1$ *to* N_{r+1}
    $b_j{}^{(r+1)} = b_{2j-1}{}^{(r)} + A^{(r)} b_{2j}{}^{(r)} + b_{2j+1}{}^{(r)}$
  end for
end for

Comment: at the r*th step the problem is reduced to*

$$\begin{bmatrix} A^{(r)} & -I & & \\ -I & \ddots & \ddots & \\ & \ddots & \ddots & -I \\ & & -I & A^{(r)} \end{bmatrix} \begin{bmatrix} x_1{}^{(r)} \\ \vdots \\ x_{N_r}{}^{(r)} \end{bmatrix} = \begin{bmatrix} b_1{}^{(r)} \\ \vdots \\ b_{N_r}{}^{(r)} \end{bmatrix}$$

2) $A^{(k)} x^{(k)} = b^{(k)}$ *is solved another way.*

3) *Backsolve:*

for $r = k - 1, \ldots, 0$
  for $j = 1$ *to* N_{r+1}
    $x_{2j}{}^{(r)} = x_j{}^{(r+1)}$
  end for
  for $j = 1$ *to* N_r *step* 2
    solve $A^{(r)} x_j{}^{(r)} = b_j{}^{(r)} + x_{j-1}{}^{(r)} + x_{j+1}{}^{(r)}$ *for* $x_j^{(r)}$
      (*we take* $x_0^{(r)} = x_{N_r+1}^{(r)} \equiv 0$)
  end for
end for

Finally, $x = x^{(0)}$ *is the desired result.*

This simple approach has two drawbacks:

1) It is numerically unstable because $A^{(r)}$ grows quickly: $\|A^{(r)}\| \sim \|A^{(r-1)}\|^2 \approx 4^{2^r}$, so in computing $b_j^{(r+1)}$, the $b_{2j\pm1}^{(r)}$ are lost in roundoff.

2) $A^{(r)}$ has bandwidth $2^r + 1$ if A is tridiagonal, so it soon becomes dense and thus expensive to multiply or solve.

Here is a fix for the second drawback. Note that $A^{(r)}$ is a polynomial $p_r(A)$ of degree 2^r:

$$p_0(A) = A \text{ and } p_{r+1}(A) = (p_r(A))^2 - 2I.$$

LEMMA 6.10. *Let $t = 2\cos\theta$. Then $p_r(t) = p_r(2\cos\theta) = 2\cos(2^r\theta)$.*

Proof. This is a simple trigonometric identity. □

Note that $p_r(t) = 2\cos(2^r \arccos(\frac{t}{2})) = 2T_{2^r}(\frac{t}{2})$, where $T_{2^r}(\cdot)$ is a *Chebyshev polynomial* (see section 6.5.6).

LEMMA 6.11. *$p_r(t) = \prod_{j=1}^{2^r}(t - t_j)$, where $t_j = 2\cos(\pi\frac{2j-1}{2^r})$.*

Proof. The zeros of the Chebyshev polynomials are given in Lemma 6.7. □

Thus $A^{(r)} = \prod_{j=1}^{2^r}(A - 2\cos(\pi\frac{2j-1}{2^r}))$, so solving $A^{(r)}z = c$ is equivalent to solving 2^r tridiagonal systems with tridiagonal coefficient matrices $A + 2\cos(\pi\frac{2j-1}{2^r})$, each of which costs $O(N)$ via tridiagonal Gaussian elimination or Cholesky.

More changes are needed to have a numerically stable algorithm. The final algorithm is due to Buneman and described in [47, 46].

We analyze the cost of the simple algorithm as follows; the stable algorithm is analogous. Multiplying by a tridiagonal matrix or solving a tridiagonal system of size N costs $O(N)$ flops. Therefore multiplying by $A^{(r)}$ or solving a system with $A^{(r)}$ costs $O(2^rN)$ flops, since $A^{(r)}$ is the product of 2^r tridiagonal matrices. The inner loop of step 1) of the algorithm therefore costs $\frac{N}{2^{r+1}} \cdot O(2^rN) = O(N^2)$ flops to update the $N_{r+1} \approx \frac{N}{2^{r+1}}$ vectors $b_j^{(r+1)}$. $A^{(r+1)}$ is not computed explicitly. Since the loop in step 1) is executed $k \approx \log_2 N$ times, the total cost of step 1) is $O(N^2 \log_2 N)$. For similar reasons, step 2) costs $O(2^kN) = O(N^2)$ flops, and step 3) costs $O(N^2 \log_2 N)$ flops, for a total cost of $O(N^2 \log_2 N)$ flops. This justifies the entry for block cyclic reduction in Table 6.1.

This algorithm generalizes to any block tridiagonal matrix with a symmetric matrix A repeated along the diagonal and a symmetric matrix F that commutes with A ($FA = AF$) repeated along the offdiagonals. See also Question 6.10. This is a common situation when solving linear systems arising from discretized differential equations such as Poisson's equation.

6.9. Multigrid

Multigrid methods were invented for partial differential equations such as Poisson's equation, but they work on a wider class of problems too. In contrast to other iterative schemes that we have discussed so far, multigrid's convergence rate is *independent* of the problem size N, instead of slowing down for larger problems. As a consequence, it can solve problems with n unknowns in $O(n)$ time or for a constant amount of work per unknown. This is optimal, modulo the (modest) constant hidden inside the $O(\cdot)$.

Here is why the other iterative algorithms that we have discussed *cannot* be optimal for the model problem. In fact, this is true of *any* iterative algorithm that computes approximation x_{m+1} by averaging values of x_m and the right-hand side b from neighboring grid points. This includes Jacobi's, Gauss–Seidel, SOR(ω), SSOR with Chebyshev acceleration (the last three with red-black ordering), and any Krylov subspace method based on matrix-vector multiplication with the matrix $T_{N\times N}$; this is because multiplying a vector by $T_{N\times N}$ is also equivalent to averaging neighboring grid point values. Suppose that we start with a right-hand side b on a 31-by-31 grid, with a single nonzero entry, as shown in the upper left of Figure 6.9. The true solution x is shown in the upper right of the same figure; note that it is everywhere nonzero and gets smaller as we get farther from the center. The bottom left plot in Figure 6.9 shows the solution $x_{J,5}$ after 5 steps of Jacobi's method, starting with an initial solution of all zeros. Note that the solution $x_{J,5}$ is zero more than 5 grid points away from the center, because averaging with neighboring grid points can "propagate information" only one grid point per iteration, and the only nonzero value is initially in the center of the grid. More generally, after k iterations only grid points within k of the center can be nonzero. The bottom right figure shows the best possible solution $x_{Best,5}$ obtainable by any "nearest neighbor" method after 5 steps: it agrees with x on grid points within 5 of the center and is necessarily 0 farther away. We see graphically that the error $x_{Best,5} - x$ is equal to the size of x at the sixth grid point away from the center. This is still a large error; by formalizing this argument, one can show that it would take at least $O(\log n)$ steps on an n-by-n grid to decrease the error by a constant factor less than 1, no matter what "nearest-neighbor" algorithm is used. If we want to do better than $O(\log n)$ steps (and $O(n \log n)$ cost), we need to "propagate information" farther than one grid point per iteration. Multigrid does this by communicating with nearest neighbors on coarser grids, where a nearest neighbor on a coarse grid can be much farther away than a nearest neighbor on a fine grid.

Multigrid uses coarse grids to do *divide-and-conquer* in two related senses. First, it obtains an initial solution for an N-by-N grid by using an $(N/2)$-by-$(N/2)$ grid as an approximation, taking every other grid point from the N-by-N grid. The coarser $(N/2)$-by-$(N/2)$ grid is in turn approximated by an $(N/4)$-by-$(N/4)$ grid, and so on recursively. The second way multigrid

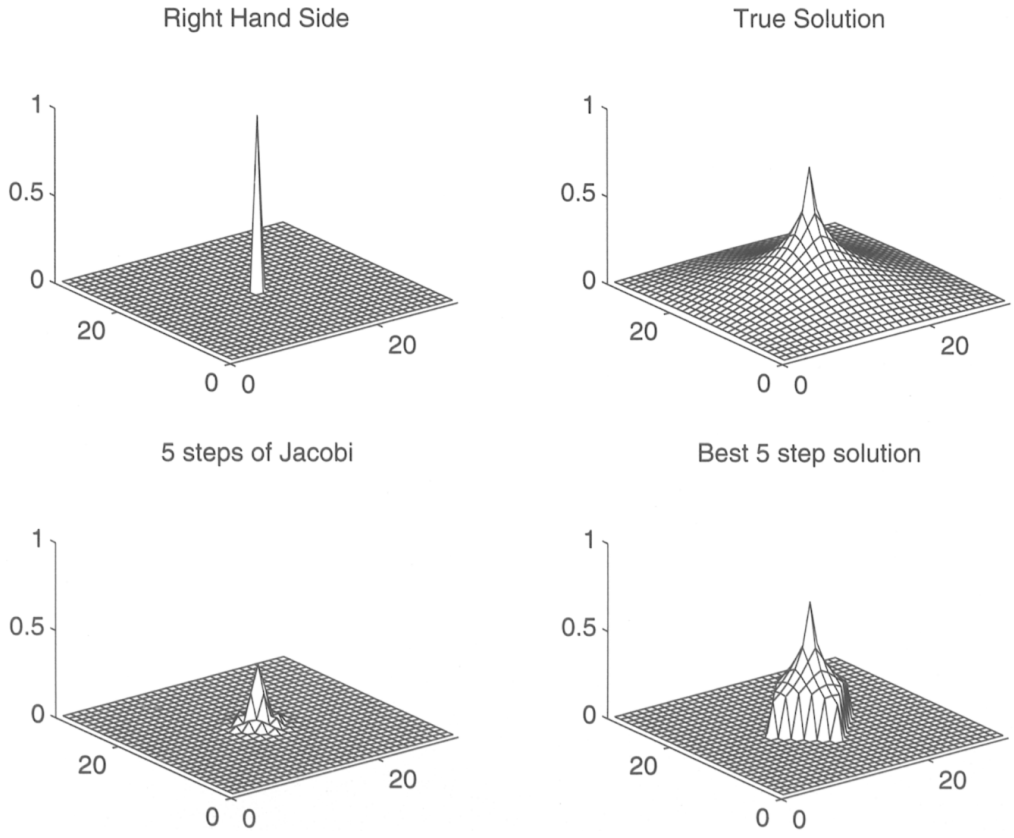


Fig. 6.9. *Limits of averaging neighboring grid points.*

uses divide-and-conquer is in the *frequency domain.* This requires us to think of the error as a sum of eigenvectors, or sine-curves of different frequencies. Then, intuitively, the work that we do on a particular grid will attenuate the error in half of the frequency components not attenuated on coarser grids. In particular, the work performed on a particular grid—averaging the solution at each grid point with its neighbors, a variation of Jacobi's method—makes the solution smoother, which is equivalent to getting rid of the high-frequency error. We will illustrate these notions further below.

6.9.1. Overview of Multigrid on the Two-Dimensional Poisson's Equation

We begin by stating the algorithm at a high level and then fill in details. As with block cyclic reduction (section 6.8), it turns out to be convenient to consider a $(2^k - 1)$-by-$(2^k - 1)$ grid of unknowns rather than the 2^k-by-2^k grid favored by the FFT (section 6.7). For understanding and implementation, it is convenient to add the nodes at the boundary, which have the known value 0, to get a $(2^k + 1)$-by-$(2^k + 1)$ grid, as shown in Figures 6.10 and 6.13. We also let $N_k = 2^k - 1$.

We will let $P^{(i)}$ denote the problem of solving a discrete Poisson equation on a $(2^i + 1)$-by-$(2^i + 1)$ grid with $(2^i - 1)^2$ unknowns, or equivalently a $(N_i + 2)$-

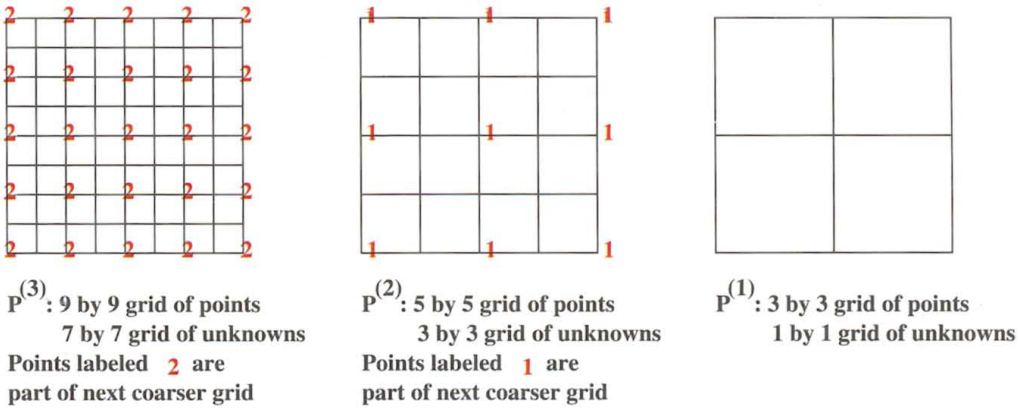


Fig. 6.10. *Sequence of grids used by two-dimensional multigrid.*

by-$(N_i + 2)$) grid with N_i^2 unknowns. The problem $P^{(i)}$ is specified by the right-hand side $b^{(i)}$ and implicitly the grid size $2^i - 1$ and the coefficient matrix $T^{(i)} \equiv T_{N_i \times N_i}$. An approximate solution of $P^{(i)}$ will be denoted $x^{(i)}$. Thus, $b^{(i)}$ and $x^{(i)}$ are $(2^i - 1)$-by-$(2^i - 1)$ arrays of values at each grid point. (The zero boundary values are implicit.) We will generate a sequence of related problems $P^{(i)}$, $P^{(i-1)}$, $P^{(i-2)}$, ..., $P^{(1)}$ on increasingly coarse grids, where the solution to $P^{(i-1)}$ is a good approximation to the error in the solution of $P^{(i)}$.

To explain how multigrid works, we need some operators that take a problem on one grid and either improve it or transform it to a related problem on another grid:

- The *solution operator* S takes a problem $P^{(i)}$ and its approximate solution $x^{(i)}$ and computes an improved $x^{(i)}$:

$$\text{improved } x^{(i)} = S(b^{(i)}, x^{(i)}). \tag{6.51}$$

  The improvement is to damp the "high-frequency components" of the error. We will explain what this means below. It is implemented by averaging each grid point value with its nearest neighbors and is a variation of Jacobi's method.

- The *restriction operator* R takes a right-hand side $b^{(i)}$ from problem $P^{(i)}$ and maps it to $b^{(i-1)}$, which is an approximation on the coarser grid:

$$b^{(i-1)} = R(b^{(i)}). \tag{6.52}$$

  Its implementation also requires just a weighted average with nearest neighbors on the grid.

- The *interpolation operator* In takes an approximate solution $x^{(i-1)}$ for $P^{(i-1)}$ and converts it to an approximate solution $x^{(i)}$ for the problem $P^{(i)}$ on the next finer grid:

$$x^{(i)} = In(x^{(i-1)}). \tag{6.53}$$

Its implementation also requires just a weighted average with nearest neighbors on the grid.

Since all three operators are implemented by replacing values at each grid point by some weighted averages of nearest neighbors, each operation costs just $O(1)$ per unknown, or $O(n)$ for n unknowns. This is the key to the low cost of the ultimate algorithm.

Multigrid V-Cycle

This is enough to state the basic algorithm, the *multigrid V-cycle (MGV).*

ALGORITHM 6.16. *MGV (the lines are numbered for later reference):*

function $MGV(b^{(i)}, x^{(i)})$ ... *replace an approximate solution* $x^{(i)}$
... *of* $P^{(i)}$ *with an improved one*
if $i = 1$ ... *only one unknown*
compute the exact solution $x^{(1)}$ *of* $P^{(1)}$
return $x^{(1)}$
else
1) $x^{(i)} = S(b^{(i)}, x^{(i)})$ ... *improve the solution*
2) $r^{(i)} = T^{(i)} \cdot x^{(i)} - b^{(i)}$ ... *compute the residual*
3) $d^{(i)} = In(MGV(4 \cdot R(r^{(i)}), 0))$ ... *solve recursively*
... *on coarser grids*
4) $x^{(i)} = x^{(i)} - d^{(i)}$ ... *correct fine grid solution*
5) $x^{(i)} = S(b^{(i)}, x^{(i)})$ ... *improve the solution again*
return $x^{(i)}$
endif

In words, the algorithm does the following:

1. Starts with a problem on a fine grid $(b^{(i)}, x^{(i)})$.
2. Improves it by damping the high-frequency error: $x^{(i)} = S(b^{(i)}, x^{(i)})$.
3. Computes the residual $r^{(i)}$ of the approximate solution $x^{(i)}$.
4. Approximates the fine grid residual $r^{(i)}$ on the next coarser grid: $R(r^{(i)})$.
5. Solves the coarser problem recursively, with a zero initial guess: $MGV(4 \cdot R(r^{(i)}), 0)$. The factor 4 appears because of the h^2 factor in the right-hand side of Poisson's equation, which changes by a factor of 4 from fine grid to coarse grid.
6. Maps the coarse solution back to the fine grid: $d_i = In(MGV(R(r^{(i)}), 0))$.
7. Subtracts the correction computed on the coarse grid from the fine grid solution: $x^{(i)} = x^{(i)} - d^{(i)}$.

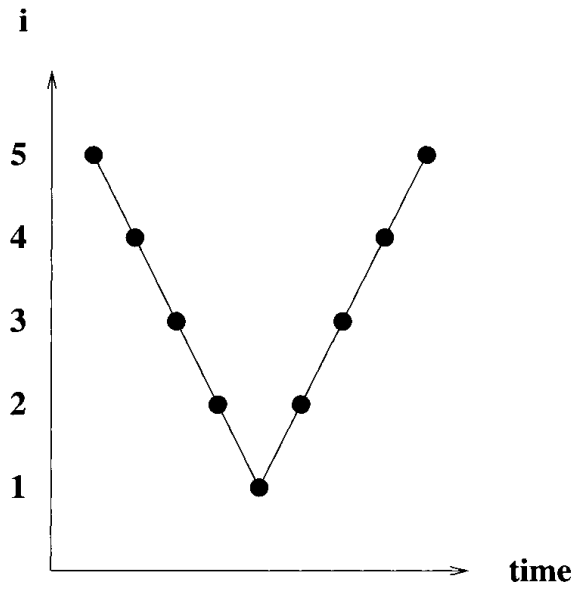


Fig. 6.11. *MGV.*

8. Improves the solution some more: $x^{(i)} = S(b^{(i)}, x^{(i)})$.

We justify the algorithm briefly as follows (we do the details later). Suppose (by induction) that $d^{(i)}$ is the *exact* solution to the equation

$$T^{(i)} \cdot d^{(i)} = r^{(i)} = T^{(i)} \cdot x^{(i)} - b^{(i)}.$$

Rearranging, we get

$$T^{(i)} \cdot (x^{(i)} - d^{(i)}) = b^{(i)}$$

so that $x^{(i)} - d^{(i)}$ is the desired solution.

The algorithm is called a V-cycle, because if we draw it schematically in (grid number i, time) space, with a point for each recursive call to MGV, it looks like Figure 6.11, starting with a call to MGV$(b^{(5)}, x^{(5)})$ in the upper left corner. This calls MGV on grid 4, then 3, and so on down to the coarsest grid 1 and then back up to grid 5 again.

Knowing only that the building blocks S, R, and In replace values at grid points by certain weighted averages of their neighbors, we know enough to do an $O(\cdot)$ complexity analysis of MGV. Since each building block does a constant amount of work per grid point, it does a total amount of work proportional to the number of grid points. Thus, each point at grid level i on the "V" in the V-cycle will cost $O((2^i - 1)^2) = O(4^i)$ operations. If the finest grid is at level k with $n = O(4^k)$ unknowns, then the total cost will be given by the geometric sum

$$\sum_{i=1}^{k} O(4^i) = O(4^k) = O(n).$$

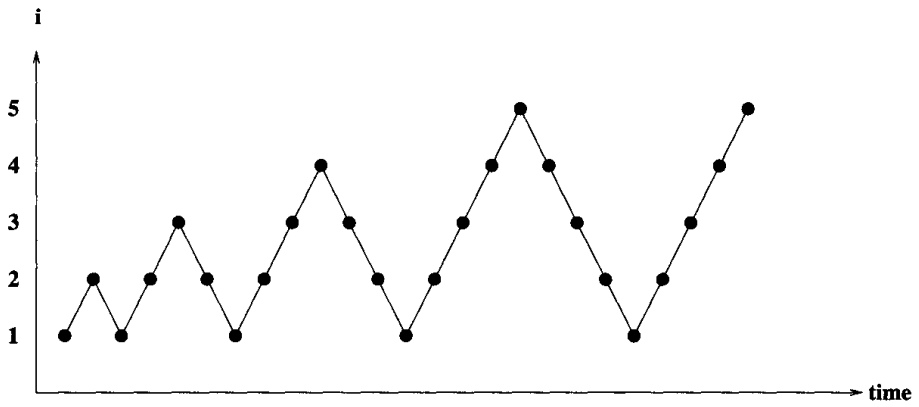


Fig. 6.12. *FMG cycle.*

Full Multigrid

The ultimate multigrid algorithm uses the MGV just described as a building block. It is called *full multigrid (FMG).*

ALGORITHM 6.17. *FMG:*

function $FMG(b^{(k)}, x^{(k)})$ ... *return an accurate solution* $x^{(k)}$ *of* $P^{(k)}$
  solve $P^{(1)}$ *exactly to get* $x^{(1)}$
  for $i = 2$ *to* k
    $x^{(i)} = MGV(b^{(i)}, In(x^{(i-1)}))$
  end for

In words, the algorithm does the following:

1. Solves the simplest problem $P^{(1)}$ exactly.
2. Given a solution $x^{(i-1)}$ of the coarse problem $P^{(i-1)}$, maps it to a starting guess $x^{(i)}$ for the next finer problem $P^{(i)}$: $In(x^{(i-1)})$.
3. Solves the finer problem using the MGV with this starting guess: $MGV(b^{(i)}, In(x^{(i-1)}))$.

Now we can do the overall $O(\cdot)$ complexity analysis of FMG. A picture of FMG in (grid number i, time) space is shown in Figure 6.12. There is one "V" in this picture for each call to MGV in the inner loop of FMG. The "V" starting at level i costs $O(4^i)$ as before. Thus the total cost is again given by the geometric sum

$$\sum_{i=1}^{k} O(4^i) = O(4^k) = O(n),$$

which is optimal, since it does a constant amount of work for each of the n unknowns. This explains the entry for multigrid in Table 6.1.

A Matlab implementation of multigrid (for both the one- and the two-dimensional model problems) is available at HOMEPAGE/Matlab/MG_README.html.

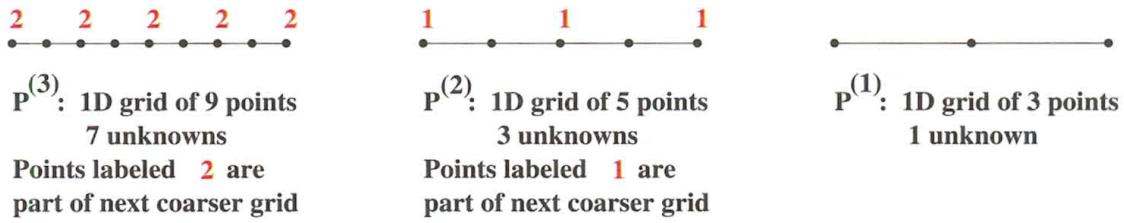


Fig. 6.13. *Sequence of grids used by one-dimensional multigrid.*

6.9.2. Detailed Description of Multigrid on the One-Dimensional Poisson's Equation

Now we will explain in detail the various operators S, R, and In composing the multigrid algorithm and sketch the convergence proof. We will do this for Poisson's equation in one dimension, since this will capture all the relevant behavior but is simpler to write. In particular, we can now consider a nested set of one-dimensional problems instead of two-dimensional problems, as shown in Figure 6.13.

As before we denote by $P^{(i)}$ the problem to be solved on grid i, namely, $T^{(i)} \cdot x^{(i)} = b^{(i)}$, where as before $N_i = 2^i - 1$ and $T^{(i)} \equiv T_{N_i}$. We begin by describing the solution operator S, which is a form of *weighted Jacobi's method.*

Solution Operator in One Dimension

In this subsection we drop the superscripts on $T^{(i)}$, $x^{(i)}$, and $b^{(i)}$ for simplicity of notation. Let $T = Z\Lambda Z^T$ be the eigendecomposition of T, as defined in Lemma 6.1. The standard Jacobi's method for solving $Tx = b$ is $x_{m+1} = Rx_m + c$, where $R = I - T/2$ and $c = b/2$. We consider *weighted Jacobi's method* $x_{m+1} = R_w x_m + c_w$, where $R_w = I - wT/2$ and $c_w = wb/2$; $w = 1$ corresponds to the standard Jacobi's method. Note that $R_w = Z(I - w\Lambda/2)Z^T$ is the eigendecomposition of R_w. The eigenvalues of R_w determine the convergence of weighted Jacobi in the usual way: Let $e_m = x_m - x$ be the error at the mth iteration of weighted Jacobi convergence so that

$$\begin{aligned} e_m &= R_w e_{m-1} \\ &= R_w^m e_0 \\ &= (Z(I - w\Lambda/2)Z^T)^m e_0 \\ &= Z(I - w\Lambda/2)^m Z^T e_0 \end{aligned}$$

so

$$Z^T e_m = (I - w\Lambda/2)^m Z^T e_0 \quad \text{or} \quad (Z^T e_m)_j = (I - w\Lambda/2)^m_{jj} (Z^T e_0)_j.$$

We call $(Z^T e_m)_j$ the *jth frequency component* of the error e_m, since $e_m = Z(Z^T e_m)$ is a sum of columns of Z weighted by the $(Z^T e_m)_j$, i.e., a sum of sinusoids of varying frequencies (see Figure 6.2). The eigenvalues $\lambda_j(R_w) = 1 -$

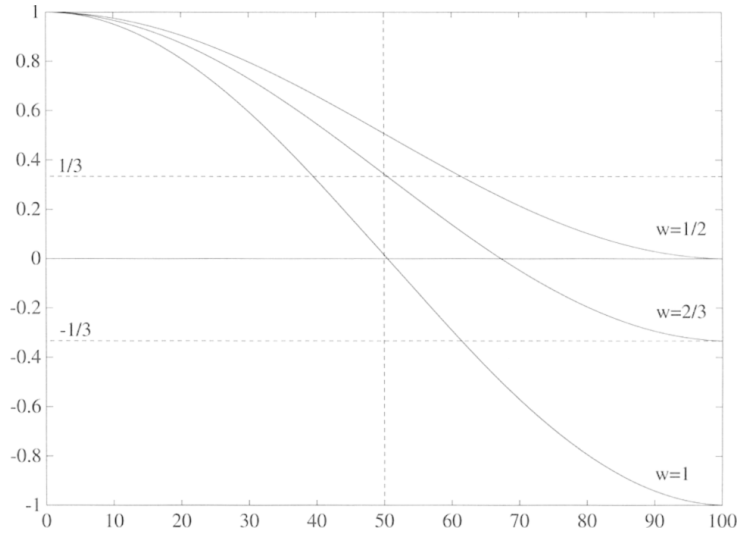


Fig. 6.14. *Graph of the spectrum of R_w for $N = 99$ and $w = 1$ (Jacobi's method), $w = 1/2$, and $w = 2/3$.*

$w\lambda_j/2$ determine how fast each frequency component goes to zero. Figure 6.14 plots $\lambda_j(R_w)$ for $N = 99$ and varying values of the weight w.

When $w = \frac{2}{3}$ and $j > \frac{N}{2}$, i.e., for the upper half of the frequencies λ_j, we have $|\lambda_j(R_w)| \leq \frac{1}{3}$. This means that the upper half of the error components $(Z^T e_m)_j$ are multiplied by $\frac{1}{3}$ or less at every iteration, independently of N. Low-frequency error components are not decreased as much, as we will see in Figure 6.15. So weighted Jacobi convergence with $w = \frac{2}{3}$ is good at decreasing the high-frequency error.

Thus, our solution operator S in equation (6.51) consists of taking one step of weighted Jacobi convergence with $w = \frac{2}{3}$:

$$S(b, x) = R_{2/3} \cdot x + b/3. \tag{6.54}$$

When we want to indicate the grid i on which $R_{2/3}$ operates, we will instead write $R^{(i)}_{2/3}$.

Figure 6.15 shows the effect of taking two steps of S for $i = 6$, where we have $2^i - 1 = 63$ unknowns. There are three rows of pictures, the first row showing the initial solution and error and the following two rows showing the solution x_m and error e_m after successive applications of S. The true solution is a sine curve, shown as a dotted line in the leftmost plot in each row. The approximate solution is shown as a solid line in the same plot. The middle plot shows the error alone, including its two-norm in the label at the bottom. The rightmost plot shows the frequency components of the error $Z^T e_m$. One can see in the rightmost plots that as S is applied, the right (upper) half of the frequency components are damped out. This can also be seen in the middle and left plots, because the approximate solution grows smoother. This is because

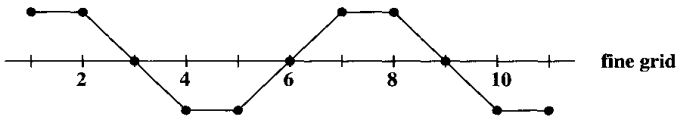
high-frequency error looks like "rough" error and low-frequency error looks like "smooth" error. Initially, the norm of the vector decreases rapidly, from 1.65 to 1.055, but then decays more gradually, because there is little more error in the high frequencies to damp. Thus, it only makes sense to do a few iterations of S at a time.

Recursive Structure of Multigrid

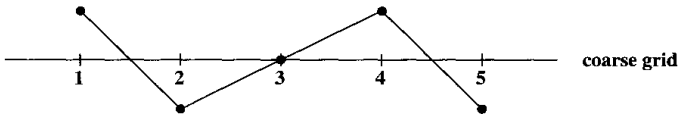
Using this terminology, we can describe the recursive structure of multigrid as follows. What multigrid does on the finest grid $P^{(k)}$, is to damp the upper half of the frequency components of the error in the solution. This is accomplished by the solution operator S, as just described. On the next coarser grid, with half as many points, multigrid damps the upper half of the remaining frequency components in the error. This is because taking a coarser grid, with half as many points, makes frequencies appear twice as high, as illustrated in the example below.

EXAMPLE 6.16.

$N = 12, \ k = 4$
low frequency, $k < \frac{N}{2}$
$\sin \frac{\pi \cdot 4 \cdot j}{12}$
for $1 \leq j \leq 11$



$N = 6, \ k = 4$
high frequency, $k > \frac{N}{2}$
$\sin \frac{\pi \cdot 4 \cdot j}{6}$
for $1 \leq j \leq 5$



◇

On the next coarser grid, the upper half of the remaining frequency components are damped, and so on, until we solve the exact (one-unknown) problem $P^{(1)}$. This is shown schematically in Figure 6.16. The purpose of the restriction and interpolation operators is to change an approximate solution on one grid to one on the next coarser or next finer grid.

Restriction Operator in One Dimension

Now we turn to the restriction operator R, which takes a right-hand side $r^{(i)}$ from problem $P^{(i)}$ and approximates it on the next coarse grid, yielding $r^{(i-1)}$.

The simplest way to compute $r^{(i-1)}$ would be to simply *sample* $r^{(i)}$ at the common grid points of the coarse and fine grids. But it is better to compute $r^{(i-1)}$ at a coarse grid point by averaging values of $r^{(i)}$ on neighboring fine grid points: the value at a coarse grid point is .5 times the value at the corresponding fine grid point, plus .25 times each of the fine grid point neighbors. We call this *averaging*. Both methods are illustrated in Figure 6.17.

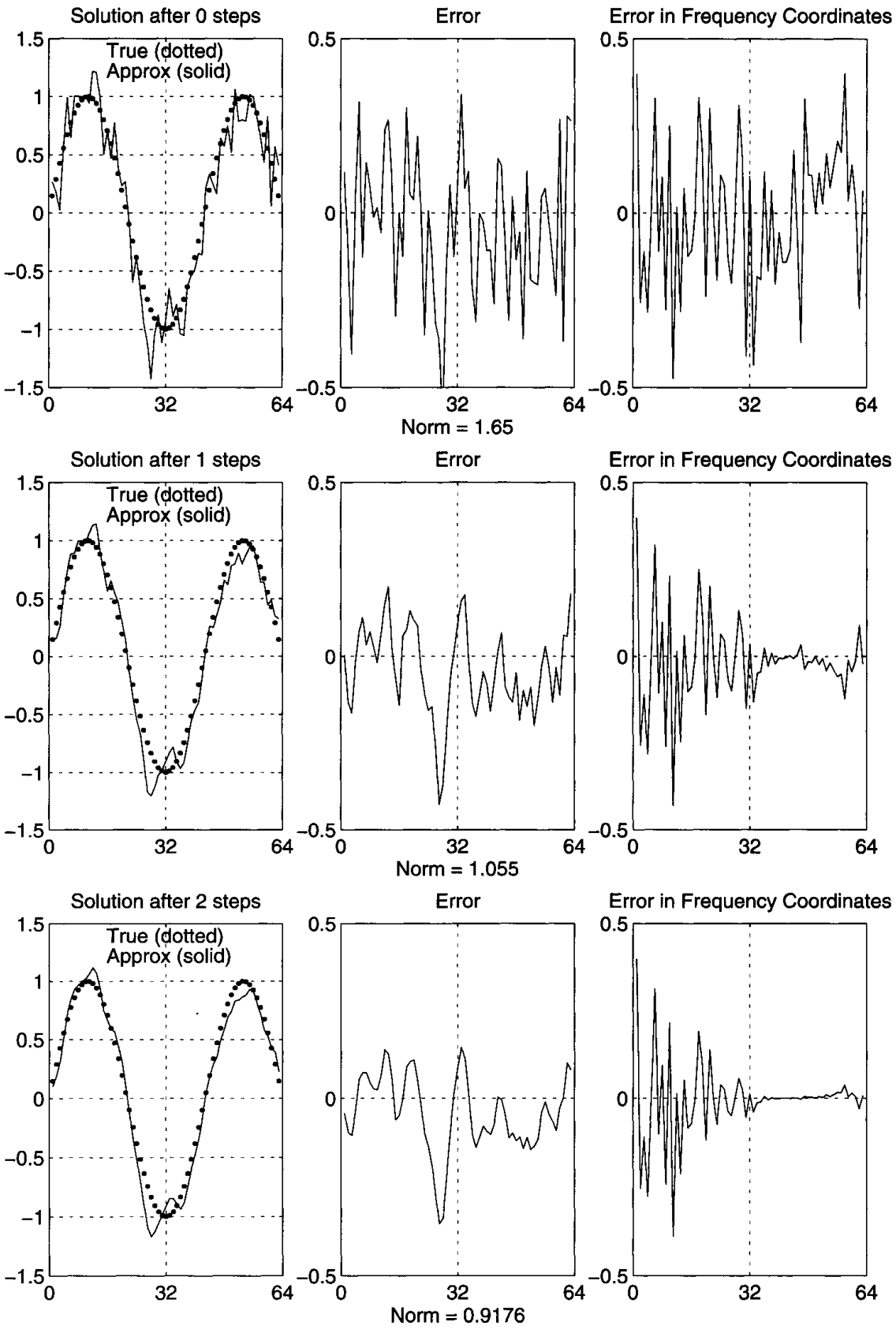


Fig. 6.15. *Illustration of weighted Jacobi convergence.*

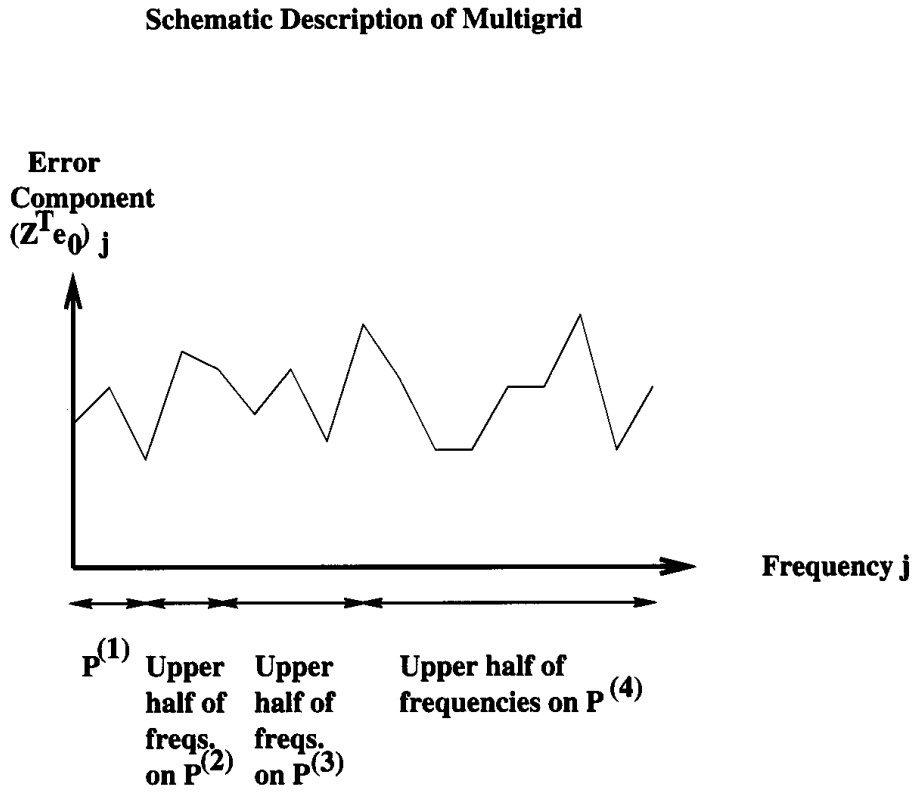


Fig. 6.16. *Schematic description of how multigrid damps error components.*

So altogether, we write the restriction operation as

$$\begin{aligned} r^{(i-1)} &= R(r^{(i)}) \\ &\equiv P_i^{i-1} \cdot r^{(i)} \\ &= \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} & & & & & & & \\ & & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} & & & & & \\ & & & & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} & & & \\ & & & & & \ddots & \ddots & \ddots & & \\ & & & & & & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix} \cdot r^{(i)}. \end{aligned} \tag{6.55}$$

The subscript i and superscript $i-1$ on the matrix P_i^{i-1} indicate that it maps from the grid with $2^i - 1$ points to the grid with $2^{i-1} - 1$ points.

In two dimensions, restriction involves averaging with the eight nearest neighbors of each grid points: $\frac{1}{4}$ times the grid cell value itself, plus $\frac{1}{8}$ times the four neighbors to the left, right, top, and bottom, plus $\frac{1}{16}$ times the four remaining neighbors at the upper left, lower left, upper right, and lower right.

Interpolation Operator in One Dimension

The interpolation operator In takes an approximate solution $d^{(i-1)}$ on a coarse grid and maps it to a function $d^{(i)}$ on the next finer grid. The solution $d^{(i-1)}$ is interpolated to the finer grid as shown in Figure 6.18: we do simple linear

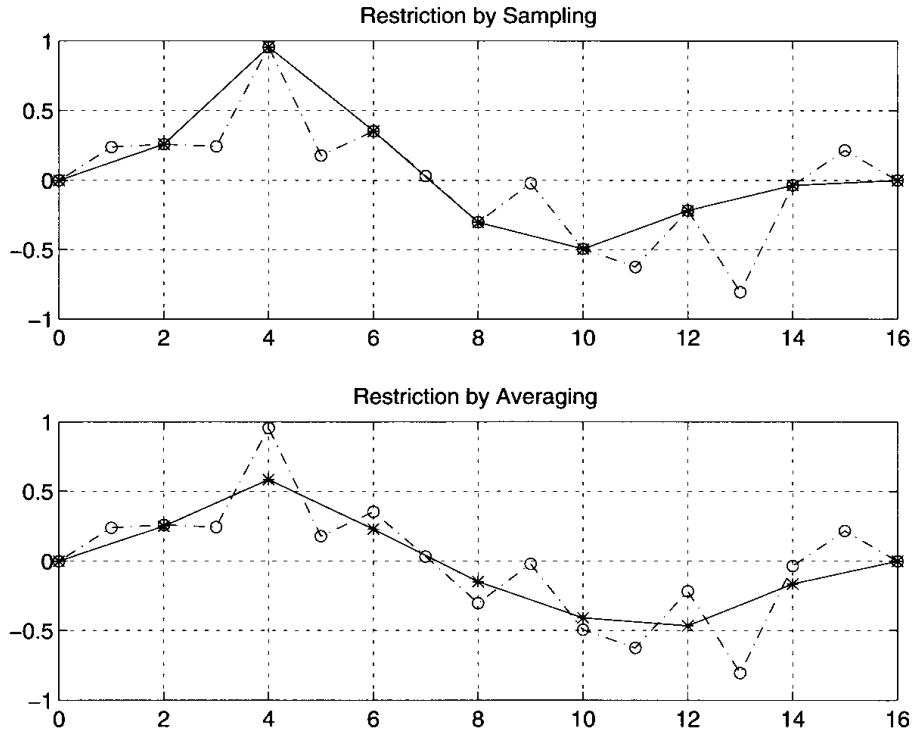


Fig. 6.17. *Restriction from a grid with* $2^4 - 1 = 15$ *points to a grid with* $2^3 - 1 = 7$ *points.* (0 *boundary values also shown.*)

interpolation to fill in the values on the fine grid (using the fact that the boundary values are known to be zero). Mathematically, we write this as

$$d^{(i)} = In(d^{(i-1)}) \equiv P_{i-1}^{i} \cdot d^{(i-1)} = \begin{bmatrix} \frac{1}{2} & & & \\ 1 & & & \\ \frac{1}{2} & \frac{1}{2} & & \\ & 1 & \ddots & \\ & \frac{1}{2} & \ddots & \frac{1}{2} \\ & & \ddots & 1 \\ & & & \frac{1}{2} \end{bmatrix} \cdot x^{(i-1)}. \tag{6.56}$$

The subscript $i-1$ and superscript i on the matrix P_{i-1}^{i} indicate that it maps from the grid with $2^{i-1} - 1$ points to the grid with $2^i - 1$ points.

Note that $P_{i-1}^{i} = 2 \cdot (P_i^{i-1})^T$. In other words, interpolation and smoothing are essentially transposes of one another. This fact will be important in the convergence analysis later.

In two dimensions, interpolation again involves averaging the values at coarse nearest neighbors of a fine grid point (one point if the fine grid point is also a coarse grid point; two neighbors if the fine grid point's nearest coarse neighbors are to the left and right or top and bottom; and four neighbors otherwise).

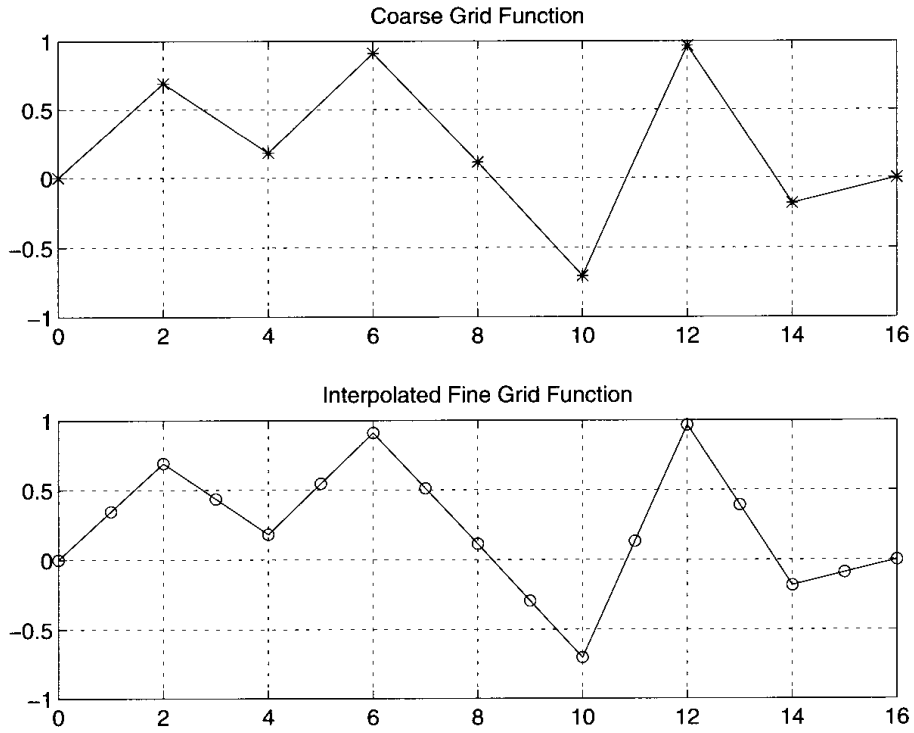


Fig. 6.18. *Interpolation from a grid with* $2^3 - 1 = 7$ *points to a grid with* $2^4 - 1 = 15$ *points.* (0 *boundary values also shown.*)

Putting It All Together

Now we run the algorithm just described for eight iterations on the problem pictured in the top two plots of Figure 6.19; both the true solution x (on the top left) and right-hand side b (on the top right) are shown. The number of unknowns is $2^7 - 1 = 127$. We show how multigrid converges in the bottom three plots. The middle left plot shows the ratio of consecutive residuals $\|r_{m+1}\|/\|r_m\|$, where the subscript m is the number of iterations of multigrid (i.e., calls to FMG, or Algorithm 6.17). These ratios are about .15, indicating that the residual decreases by more than a factor of 6 with each multigrid iteration. This quick convergence is indicated in the middle right plot, which shows a semilogarithmic plot of $\|r_m\|$ versus m; it is a straight line with slope $\log_{10}(.15)$ as expected. Finally, the bottom plot plots all eight error vectors $x_m - x$. We see how they smooth out and become parallel on a semilogarithmic plot, with a constant decrease between adjacent plots of $\log_{10}(.15)$.

Figure 6.20 shows a similar example for a two-dimensional model problem.

Convergence Proof

Finally, we sketch a convergence proof that shows that the overall error in an FMG "V"-cycle is decreased by a constant less than 1, independent of grid size $N_k = 2^k - 1$. This means that the number of FMG V-cycles needed to decrease the error by any factor less than 1 is independent of k, and so the total work

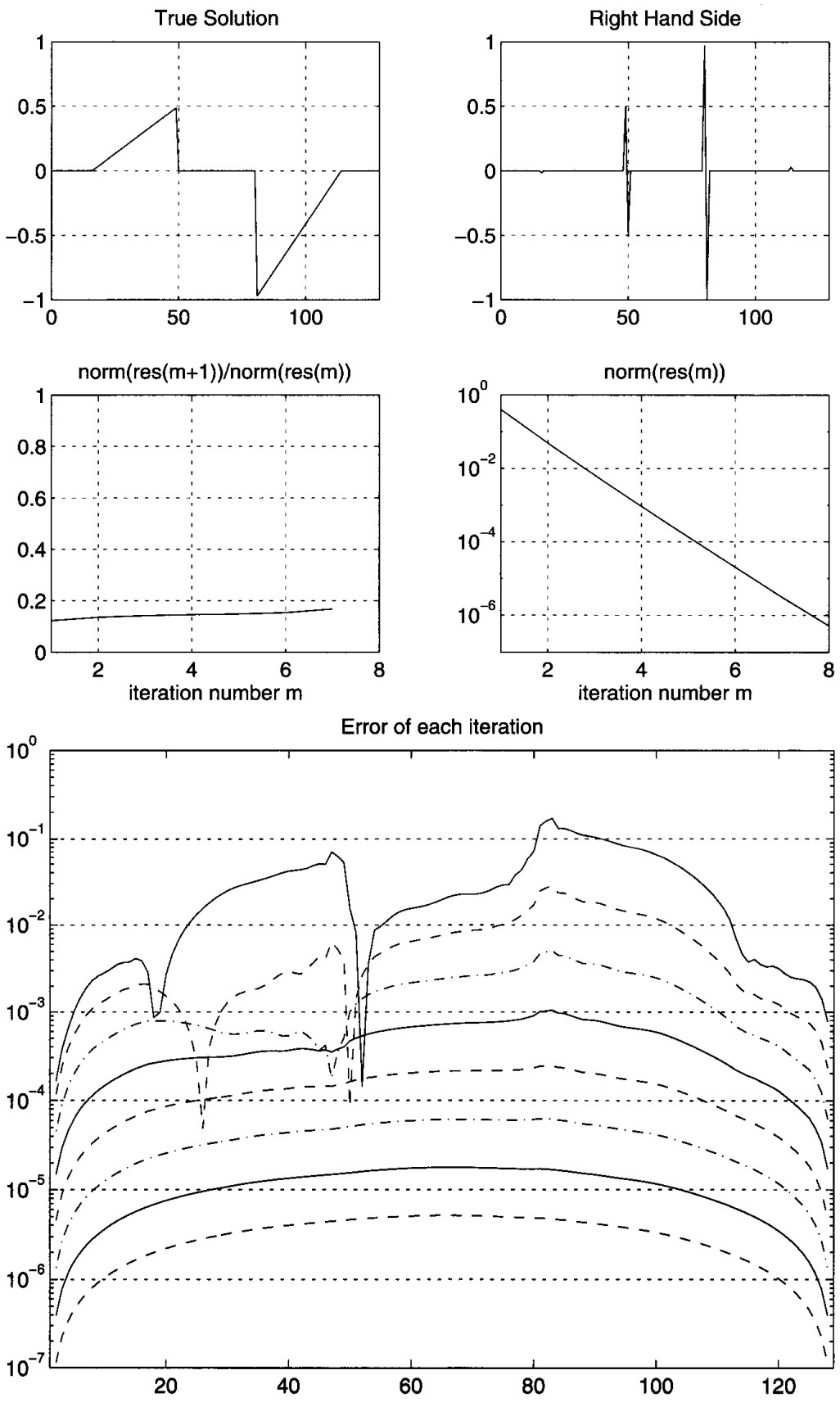


Fig. 6.19. *Multigrid solution of the one-dimensional model problem.*

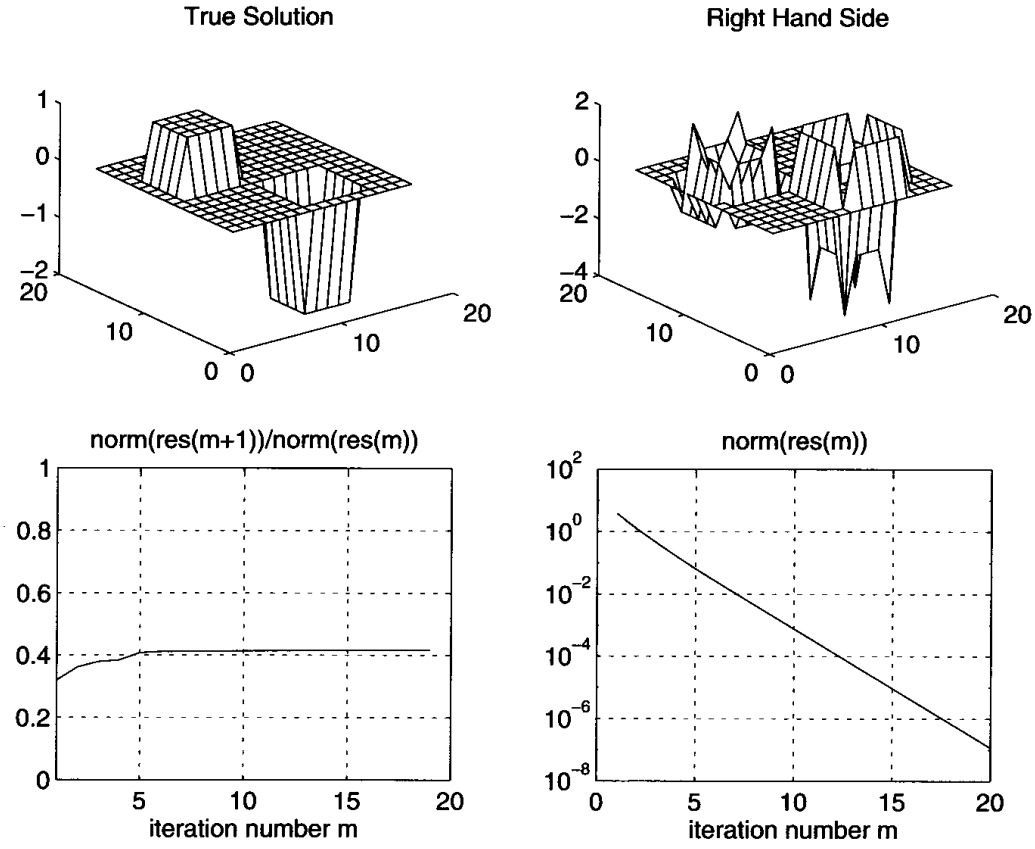


Fig. 6.20. *Multigrid solution of the two-dimensional model problem.*

is proportional to the cost of a single FMG V-cycle, i.e., proportional to the number of unknowns n.

We will simplify the proof by looking at one V-cycle and assuming by induction that the coarse grid problem is solved *exactly* [43]. In reality, the coarse grid problem is not solved quite exactly, but this rough analysis suffices to capture the spirit of the proof: that low-frequency error is attenuated on the coarser grid and high-frequency error is eliminated on the fine grid.

Now let us write all the formulas defining a V-cycle and combine them all to get a single formula of the form "new $e^{(i)} = M \cdot e^{(i)}$," where $e^{(i)} = x^{(i)} - x$ is the error and M is a matrix whose eigenvalues determine the rate of convergence; our goal is to show that they are bounded away from 1, independently of i. The line numbers in the following table refer to Algorithm 6.16.

(a) $x^{(i)} = S(b(i), x(i)) = R^{(i)}_{2/3} x^{(i)} + b^{(i)}/3$
by line 1) and equation (6.54),

(b) $r^{(i)} = T^{(i)} \cdot x^{(i)} - b^{(i)}$
by line 2),

$d^{(i)} = In(MGV(4 \cdot R(r^{(i)}), 0))$
by line 3)

$$
\begin{aligned}
&= In(\left[T^{(i-1)}\right]^{-1}(4 \cdot R(r^{(i)}))) \\
&\qquad \text{by our assumption that the} \\
&\qquad \text{coarse grid problem is solved exactly} \\
&= In(\left[T^{(i-1)}\right]^{-1}(4 \cdot P_i^{i-1} r^{(i)})) \\
&\qquad \text{by equation (6.55)} \\
\text{(c)} \qquad &= P_{i-1}^{i}(\left[T^{(i-1)}\right]^{-1}(4 \cdot P_i^{i-1} r^{(i)})) \\
&\qquad \text{by equation (6.56)} \\
\text{(d)} \quad x^{(i)} &= x^{(i)} - d^{(i)} \\
&\qquad \text{by line 4)} \\
\text{(e)} \quad x^{(i)} &= S(b(i), x(i)) = R_{2/3}^{(i)} x^{(i)} + b^{(i)}/3 \\
&\qquad \text{by line 5).}
\end{aligned}
$$

In order to get equations updating the error $e^{(i)}$, we subtract the identity $x = R_{2/3}^{(i)} x + b^{(i)}/3$ from lines (a) and (e) above, $0 = T^{(i)} \cdot x - b^{(i)}$ from line (b), and $x = x$ from line (d) to get

$$
\begin{aligned}
&\text{(a)} \quad e^{(i)} = R_{2/3}^{(i)} e^{(i)}, \\
&\text{(b)} \quad r^{(i)} = T^{(i)} \cdot e^{(i)}, \\
&\text{(c)} \quad d^{(i)} = P_{i-1}^{i}(\left[T^{(i-1)}\right]^{-1}(4 \cdot P_i^{i-1} r^{(i)})), \\
&\text{(d)} \quad e^{(i)} = e^{(i)} - d^{(i)}, \\
&\text{(e)} \quad e^{(i)} = R_{2/3}^{(i)} e^{(i)}.
\end{aligned}
$$

Substituting each of the above equations into the next yields the following formula, showing how the error is updated by a V-cycle:

$$
\begin{aligned}
\text{new } e^{(i)} &= R_{2/3}^{(i)} \left\{ I - P_{i-1}^{i} \cdot \left[T^{(i-1)}\right]^{-1} \cdot (4 \cdot P_i^{i-1} T^{(i)}) \right\} R_{2/3}^{(i)} \cdot e^{(i)} \\
&\equiv M \cdot e^{(i)}. \qquad (6.57)
\end{aligned}
$$

Now we need to compute the eigenvalues of M. We first simplify equation (6.57), using the facts that $P_{i-1}^{i} = 2 \cdot (P_i^{i-1})^T$ and

$$
T^{(i-1)} = 4 \cdot P_i^{i-1} T^{(i)} P_{i-1}^{i} = 8 \cdot P_i^{i-1} T^{(i)} (P_i^{i-1})^T \qquad (6.58)
$$

(see Question 6.15). Substituting these into the expression for M in equation (6.57) yields

$$
M = R_{2/3}^{(i)} \left\{ I - (P_i^{i-1})^T \cdot \left[P_i^{i-1} T^{(i)} (P_i^{i-1})^T \right]^{-1} \cdot (P_i^{i-1} T^{(i)}) \right\} R_{2/3}^{(i)}
$$

or, dropping indices to simplify notation,

$$
M = R_{2/3} \left\{ I - P^T \cdot \left[P T P^T \right]^{-1} \cdot P T \right\} R_{2/3}. \qquad (6.59)
$$

We continue, using the fact that all the matrices composing M (T, $R_{2/3}$, and P) can be (nearly) diagonalized by the eigenvector matrices $Z = Z^{(i)}$ and

$Z^{(i-1)}$ of $T = T^{(i)}$ and $T^{(i-1)}$, respectively: Recall that $Z = Z^T = Z^{-1}$, $T = Z\Lambda Z$, and $R_{2/3} = Z(I - \Lambda/3)Z \equiv Z\Lambda_R Z$. We leave it to the reader to confirm that $Z^{(i-1)}PZ^{(i)} = \Lambda_P$, where Λ_P is almost diagonal (see Question 6.15):

$$\lambda_{P,jk} = \begin{cases} (+1 + \cos\frac{\pi j}{2^i})/\sqrt{8} & \text{if } k = j, \\ (-1 + \cos\frac{\pi j}{2^i})/\sqrt{8} & \text{if } k = 2^i - j, \\ 0 & \text{otherwise.} \end{cases} \tag{6.60}$$

This lets us write

$$\begin{aligned} ZMZ &= (ZR_{2/3}Z) \\ &\quad \cdot \left\{ I - (ZP^TZ^{(i-1)}) \cdot \left[(Z^{(i-1)}PZ)(ZTZ)(ZP^TZ^{(i-1)}) \right]^{-1} \right. \\ &\quad \left. \cdot (Z^{(i-1)}PZ)(ZTZ) \right\} \cdot (ZR_{2/3}Z) \\ &= \Lambda_R \cdot \left\{ I - \Lambda_P^T \left[\Lambda_P \Lambda \Lambda_P^T \right]^{-1} \Lambda_P \Lambda \right\} \cdot \Lambda_R. \end{aligned}$$

The matrix ZMZ is similar to M since $Z = Z^{-1}$ and so has the same eigenvalues as M. Also, ZMZ is nearly diagonal: it has nonzeros only on its main diagonal and "perdiagonal" (the diagonal from the lower left corner to the upper right corner of the matrix). This lets us compute the eigenvalues of M explicitly.

THEOREM 6.11. *The matrix M has eigenvalues $1/9$ and 0, independent of i. Therefore multigrid converges at a fixed rate independent of the number of unknowns.*

For a proof, see Question 6.15. For a more general analysis, see [268].

For an implementation of this algorithm, see Question 6.16. The Web site [91] contains pointers to an extensive literature, software, and so on.

6.10. Domain Decomposition

Domain decomposition for solving sparse systems of linear equations is a topic of current research. See [49, 116, 205] and especially [232] for recent surveys. We will give only simple examples.

The need for methods beyond those we have discussed arises from of the irregularity and size of real problems and also from the need for algorithms for parallel computers. The fastest methods that we have discussed so far, those based on block cyclic reduction, the FFT, and multigrid, work best (or only) on particularly regular problems such as the model problem, i.e., Poisson's equation discretized with a uniform grid on a rectangle. But the region of solution of a real problem may not be a rectangle but more irregular, representing a physical object like a wing (see Figure 2.12). Figure 2.12 also

illustrates that there may be more grid points in regions where the solution is expected to be less smooth than in regions with a smooth solution. Also, we may have more complicated equations than Poisson's equation or even different equations in different regions. Independent of whether the problem is regular, it may be too large to fit in the computer memory and may have to be solved "in pieces." Or we may want to break the problem into pieces that can be solved in parallel on a parallel computer.

Domain decomposition addresses all these issues by showing how to systematically create "hybrid" methods from the simpler methods discussed in previous sections. These simpler methods are applied to smaller and more regular subproblems of the overall problem, after which these partial solutions are "pieced together" to get the overall solution. These subproblems can be solved one at a time if the whole problem does not fit into memory, or in parallel on a parallel computer. We give examples below. There are generally many ways to break a large problem into pieces, many ways to solve the individual pieces, and many ways to piece the solutions together. Domain decomposition theory does not provide a magic way to choose the best way to do this in all cases but rather a set of reasonable possibilities to try. There are some cases (such as problems sufficiently like Poisson's equation) where the theory does yield "optimal methods" (costing $O(1)$ work per unknown).

We divide our discussion into two parts, *nonoverlapping methods* and *overlapping* methods.

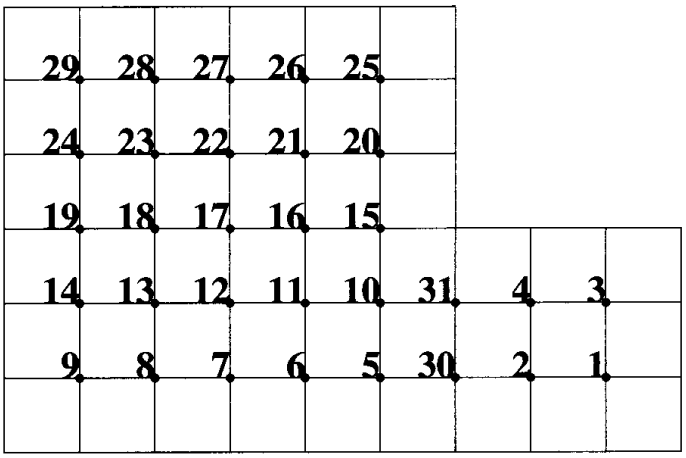
6.10.1. Nonoverlapping Methods

This method is also called *substructuring* or a *Schur complement method* in the literature. It has been used for decades, especially in the structural analysis community, to break large problems into smaller ones that fit into computer memory.

For simplicity we will illustrate this method using the usual Poisson's equation with Dirichlet boundary conditions discretized with a 5-point stencil but on an *L-shaped region* rather than a square. This region may be decomposed into two domains: a small square and a large square of twice the side length, where the small square is connected to the bottom of the right side of a larger square. We will design a solver that can exploit our ability to solve problems quickly on squares.

In the figure below, the number of each grid point is shown for a coarse discretization (the number is above and to the left of the corresponding grid

point; only grid points interior to the "L" are numbered).



Note that we have numbered first the grid points inside the two subdomains (1 to 4 and 5 to 29) and then the grid points on the boundary (30 and 31). The resulting matrix is

$$\equiv A \equiv \left[\begin{array}{c||c||c} A_{11} & 0 & A_{13} \\ \hline\hline 0 & A_{22} & A_{23} \\ \hline\hline A_{13}^T & A_{23}^T & A_{33} \end{array}\right].$$

Here, $A_{11} = T_{2\times 2}$, $A_{22} = T_{5\times 5}$, and $A_{33} = T_{2\times 1} \equiv T_2 + 2I_2$, where T_N is defined in equation (6.3) and $T_{N\times N}$ is defined in equation (6.14). One of the most important properties of this matrix is that $A_{12} = 0$, since there is no direct coupling between the interior grid points of the two subdomains. The only coupling is through the boundary, which is numbered last (grid points 30 and 31). Thus A_{13} contains the coupling between the small square and the boundary, and A_{23} contains the coupling between the large square and the boundary.

To see how to take advantage of the special structure of A to solve $Ax = b$, write the block LDU decomposition of A as follows:

$$A = \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \\ A_{13}^T A_{11}^{-1} & A_{23}^T A_{22}^{-1} & I \end{bmatrix} \cdot \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & S \end{bmatrix} \cdot \begin{bmatrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ 0 & 0 & I \end{bmatrix},$$

where

$$S = A_{33} - A_{13}^T A_{11}^{-1} A_{13} - A_{23}^T A_{22}^{-1} A_{23} \tag{6.61}$$

is called the *Schur complement* of the leading principal submatrix containing A_{11} and A_{22}. Therefore, we may write

$A^{-1} =$

$$\begin{bmatrix} A_{11}^{-1} & 0 & -A_{11}^{-1} A_{13} \\ 0 & A_{22}^{-1} & -A_{22}^{-1} A_{23} \\ 0 & 0 & I \end{bmatrix} \cdot \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & S^{-1} \end{bmatrix} \cdot \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \\ -A_{13}^T A_{11}^{-1} & -A_{23}^T A_{22}^{-1} & I \end{bmatrix}.$$

Therefore, to multiply a vector by A^{-1} we need to multiply by the blocks in the entries of this factored form of A^{-1}, namely, A_{13} and A_{23} (and their transposes), A_{11}^{-1} and A_{22}^{-1}, and S^{-1}. Multiplying by A_{13} and A_{23} is cheap because they are very sparse. Multiplying by A_{11}^{-1} and A_{22}^{-1} is also cheap because we chose these subdomains to be solvable by FFT, block cyclic reduction, multigrid, or some other fast method discussed so far. It remains to explain how to multiply by S^{-1}.

Since there are many fewer grid points on the boundary than in the subdomains, A_{33} and S have a much smaller dimension than A_{11} and A_{22}; this effect grows for finer grid spacings. S is symmetric positive definite, as is A, and (in this case) dense. To compute it explicitly one would need to solve with each subdomain once per boundary grid point (from the $A_{11}^{-1} A_{13}$ and $A_{22}^{-1} A_{23}$ terms in (6.61)). This can certainly be done, after which one could factor S using dense Cholesky and proceed to solve the system. But this is expensive, much more so than just multiplying a vector by S, which requires just one solve per subdomain using equation (6.61). This makes a Krylov subspace–based iterative method such as CG look attractive (section 6.6), since these methods require only multiplying a vector by S. The number of matrix-vector multiplications CG requires depends on the condition number of S. What makes

domain decomposition so attractive is that S turns out to be much better conditioned that the original matrix A (a condition number that grows like $O(N)$ instead of $O(N^2)$), and so convergence is fast [116, 205].

More generally, one has $k > 2$ subdomains, separated by boundaries (see Figure 6.21, where the heavy lines separate subdomains). If we number the nodes in each subdomain consecutively, followed by the boundary nodes, we get the matrix

$$A = \left[\begin{array}{ccc|c} A_{1,1} & & 0 & A_{1,k+1} \\ & \ddots & & \vdots \\ 0 & & A_{k,k} & A_{k,k+1} \\ \hline A_{1,k+1}^T & \cdots & A_{k,k+1}^T & A_{k+1,k+1} \end{array}\right], \tag{6.62}$$

where again we can factor it by factoring each $A_{i,i}$ independently and forming the Schur complement $S = A_{k+1,k+1} - \sum_{i=1}^{k} A_{i,k+1}^T A_{i,i}^{-1} A_{i,k+1}$.

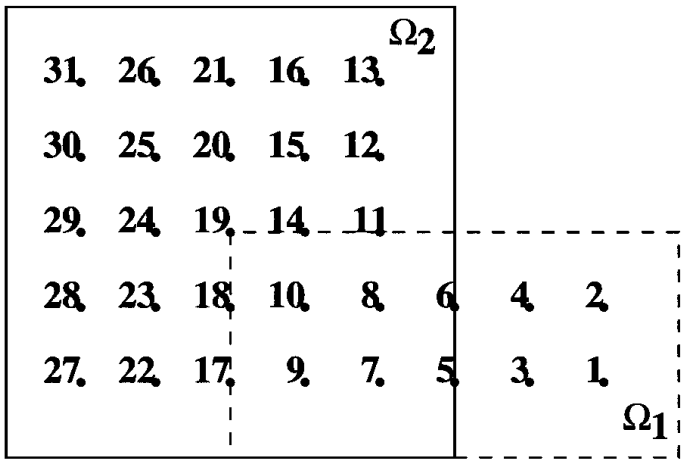
In this case, when there is more than one boundary segment, S has further structure that can be exploited to precondition it. For example, by numbering the grid points in the interior of each boundary segment before the grid points at the intersection of boundary segments, one gets a block structure as in A. The diagonal blocks of S are complicated but may be approximated by $T_N^{1/2}$, which may be inverted efficiently using the FFT [36, 37, 38, 39, 40]. To summarize the state of the art, by choosing the preconditioner for S appropriately, one can make the number of steps of CG independent of the number of boundary grid points N [231].

6.10.2. Overlapping Methods

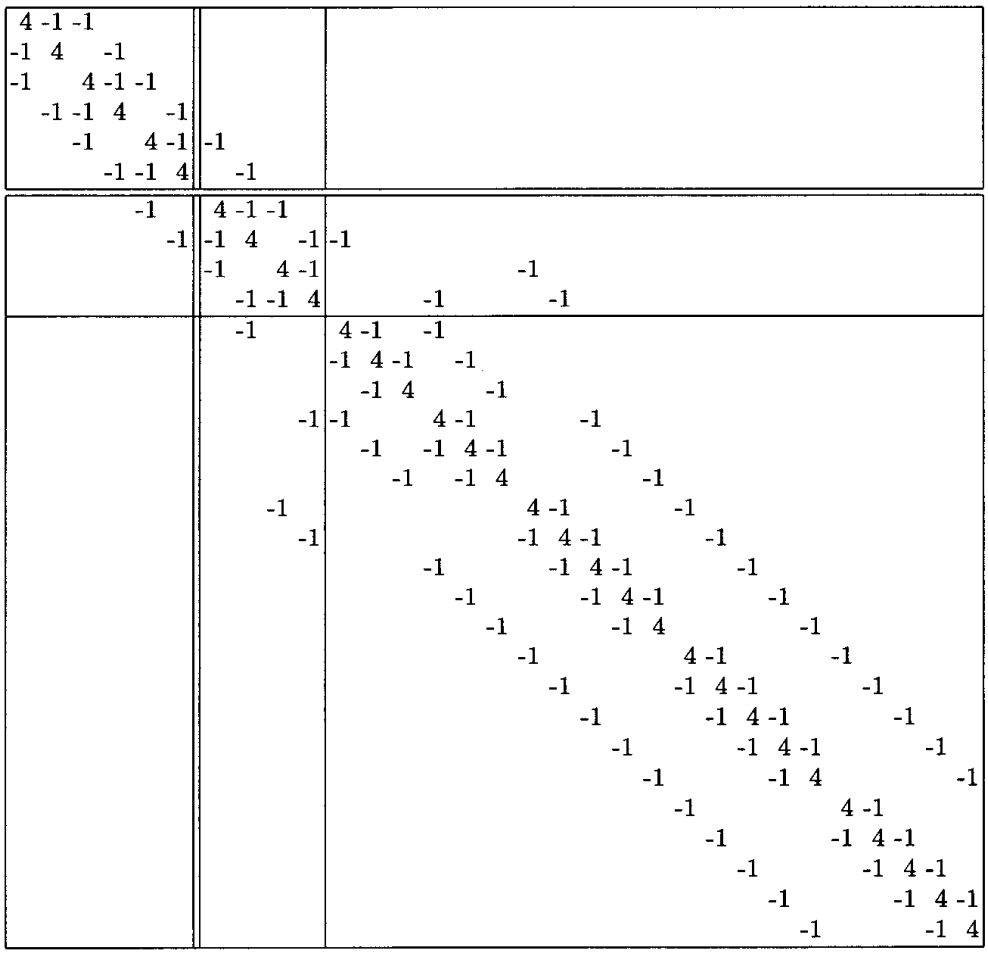
The methods in the last section were called *nonoverlapping* because the domains corresponding to the nodes in $A_{i,i}$ were disjoint, leading to the block diagonal structure in equation (6.62). In this section we permit overlapping domains, as shown in the figure below. As we will see, this overlap permits us to design an algorithm comparable in speed with multigrid but applicable to a wider set of problems.

The rectangle with a dashed boundary in the figure is domain Ω_1, and the square with a solid boundary is domain Ω_2. We have renumbered the nodes so that the nodes in Ω_1 are numbered first and the nodes in Ω_2 are numbered

last, with the nodes in the overlap $\Omega_1 \cap \Omega_2$ in the middle.



These domains are shown in the matrix A below, which is the same matrix as in section 6.10.1 but with its rows and columns ordered as shown above:



We have indicated the boundaries between domains in the way that we have partitioned the matrix: The single lines divide the matrix into the nodes associated with Ω_1 (1 through 10) and the rest $\Omega \setminus \Omega_1$ (11 through 31). The double lines divide the matrix into the nodes associated with Ω_2 (7 through 31) and the rest $\Omega \setminus \Omega_2$ (1 through 6). The submatrices below are subscripted

accordingly:

$$A = \left[\begin{array}{c|c} A_{\Omega_1,\Omega_1} & A_{\Omega_1,\Omega\setminus\Omega_1} \\ \hline A_{\Omega\setminus\Omega_1,\Omega_1} & A_{\Omega\setminus\Omega_1,\Omega\setminus\Omega_1} \end{array}\right] = \left[\begin{array}{c||c} A_{\Omega\setminus\Omega_2,\Omega\setminus\Omega_2} & A_{\Omega\setminus\Omega_2,\Omega_2} \\ \hline\hline A_{\Omega_2,\Omega\setminus\Omega_2} & A_{\Omega_2,\Omega_2} \end{array}\right].$$

We conformally partition vectors such as

$$\begin{aligned} x &= \left[\begin{array}{c} x_{\Omega_1} \\ \hline x_{\Omega\setminus\Omega_1} \end{array}\right] = \left[\begin{array}{c} x(1:10) \\ \hline x(11:31) \end{array}\right] \\ &= \left[\begin{array}{c} x_{\Omega\setminus\Omega_2} \\ \hline\hline x_{\Omega_2} \end{array}\right] = \left[\begin{array}{c} x(1:6) \\ \hline\hline x(7:31) \end{array}\right]. \end{aligned}$$

Now we have enough notation to state two basic overlapping domain decomposition algorithms. The simplest one is called the *additive Schwarz method* for historical reasons but could as well be called *overlapping block Jacobi iteration* because of its similarity to (block) Jacobi iteration from sections 6.5 and 6.6.5.

ALGORITHM 6.18. *Additive Schwarz method for updating an approximate solution x_i of $Ax = b$ to get a better solution x_{i+1}:*

$r = b - Ax_i$ /* *compute the residual* */
$x_{i+1} = 0$
$x_{i+1,\Omega_1} = x_{i,\Omega_1} + A^{-1}_{\Omega_1,\Omega_1} \cdot r_{\Omega_1}$ /* *update the solution on* Ω_1 */
$x_{i+1,\Omega_2} = x_{i+1,\Omega_2} + A^{-1}_{\Omega_2,\Omega_2} \cdot r_{\Omega_2}$ /* *update the solution on* Ω_2 */

This algorithm also be written in one line as

$$x_{i+1} = x_i + \left[\begin{array}{c} A^{-1}_{\Omega_1,\Omega_1} \cdot r_{\Omega_1} \\ \hline 0 \end{array}\right] + \left[\begin{array}{c} 0 \\ \hline\hline A^{-1}_{\Omega_2,\Omega_2} \cdot r_{\Omega_2} \end{array}\right].$$

In words, the algorithm works as follows: The update $A^{-1}_{\Omega_1,\Omega_1} r_{\Omega_1}$ corresponds to solving Poisson's equation just on Ω_1, using boundary conditions at nodes 11, 14, 17, 18, and 19, which depend on the previous approximate solution x_i. The update $A^{-1}_{\Omega_2,\Omega_2} r_{\Omega_2}$ is analogous, using boundary conditions at nodes 5 and 6 depending on x_i.

In our case the Ω_i are rectangles, so any one of our earlier fast methods, such as multigrid, could be used to solve $A^{-1}_{\Omega_i,\Omega_i} r_{\Omega_i}$. Since the additive Schwarz method is iterative, it is not necessary to solve the problems on Ω_i exactly.

Indeed, the additive Schwarz method is typically used as a preconditioner for a Krylov subspace method like conjugate gradients (see section 6.6.5). In the notation of section 6.6.5, the preconditioner M is given by

$$M^{-1} = \left[\begin{array}{c|c} A^{-1}_{\Omega_1,\Omega_1} & 0 \\ \hline 0 & 0 \end{array}\right] + \left[\begin{array}{c||c} 0 & 0 \\ \hline\hline 0 & A^{-1}_{\Omega_2,\Omega_2} \end{array}\right].$$

If Ω_1 and Ω_2 did not overlap, then M^{-1} would simplify to

$$\begin{bmatrix} A^{-1}_{\Omega_1,\Omega_1} & 0 \\ 0 & A^{-1}_{\Omega_2,\Omega_2} \end{bmatrix}$$

and we would be doing block Jacobi iteration. But we know that Jacobi's method does not converge particularly quickly, because "information" about the solution from one domain can only move slowly to the other domain across the boundary between them (see the discussion at the beginning of section 6.9). But as long as the overlap is a large enough fraction of the two domains, information will travel quickly enough to guarantee fast convergence. Of course we do not want too large an overlap, because this increases the work significantly. The goal in designing a good domain decomposition method is to choose the domains and the overlaps so as to have fast convergence while doing as little work as possible; we say more on how convergence depends on overlap below.

From the discussion in section 6.5, we know that the Gauss–Seidel method is likely to be more effective than Jacobi's method. This is the case here as well, with the *overlapping block Gauss–Seidel method* (more commonly called the *multiplicative Schwarz method*) often being twice as fast as additive block Jacobi iteration (the additive Schwarz method).

ALGORITHM 6.19. *Multiplicative Schwarz method for updating an approximate solution x_i of $Ax = b$:*

(1) $r_{\Omega_1} = (b - Ax_i)_{\Omega_1}$ /* *compute residual of* x_i *on* Ω_1 */

(2) $x_{i+\frac{1}{2},\Omega_1} = x_{i,\Omega_1} + A^{-1}_{\Omega_1,\Omega_1} \cdot r_{\Omega_1}$ /* *update solution on* Ω_1 */

(2′) $x_{i+\frac{1}{2},\Omega\setminus\Omega_1} = x_{i,\Omega\setminus\Omega_1}$

(3) $r_{\Omega_2} = (b - Ax_{i+\frac{1}{2}})_{\Omega_2}$ /* *compute residual of* $x_{i+\frac{1}{2}}$ *on* Ω_2 */

(4) $x_{i+1,\Omega_2} = x_{i+\frac{1}{2},\Omega_2} + A^{-1}_{\Omega_2,\Omega_2} \cdot r_{\Omega_2}$ /* *update solution on* Ω_2 */

(4′) $x_{i+1,\Omega\setminus\Omega_2} = x_{i+\frac{1}{2},\Omega\setminus\Omega_2}$

Note that lines (2′) *and* (4′) *do not require any data movement, provided that* $x_{i+\frac{1}{2}}$ *and* x_{i+1} *overwrite* x_i.

This algorithm first solves Poisson's equation on Ω_1 using boundary data from x_i, just like Algorithm 6.18. It then solves Poisson's equation on Ω_2, but using boundary data that has just been updated. It may also be used as a preconditioner for a Krylov subspace method.

In practice more domains than just two (Ω_1 and Ω_2) are used. This is done if the domain of solution is more complicated or if there are many independent parallel processors available to solve independent problems $A^{-1}_{\Omega_i,\Omega_i} r_{\Omega_i}$ or just to keep the subproblems $A^{-1}_{\Omega_i,\Omega_i} r_{\Omega_i}$ small and inexpensive to solve.

Here is a summary of the theoretical convergence analysis of these methods for the model problem and similar elliptic partial differential equations. Let h

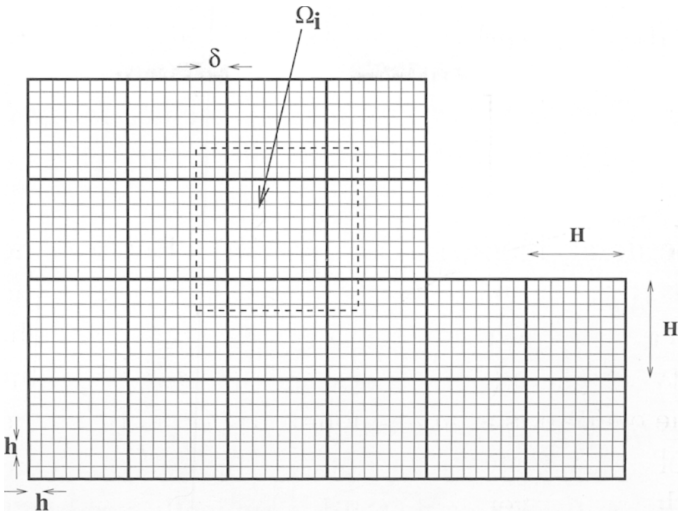


Fig. 6.21. *Coarse and fine discretizations of an L-shaped region.*

be the mesh spacing. The theory predicts how many iterations are necessary to converge as a function of h as h decreases to 0. With two domains, as long as the overlap region $\Omega_1 \cap \Omega_2$ is a nonzero fraction of the total domain $\Omega_1 \cup \Omega_2$, the number of iterations required for convergence is independent of h as h goes to zero. This is an attractive property and is reminiscent of multigrid, which also converged at a rate independent of mesh size h. But the cost of an iteration includes solving subproblems on Ω_1 and Ω_2 *exactly*, which may be comparable in expense to the original problem. So unless the solutions on Ω_1 and Ω_2 are very cheap (as with the L-shaped region above), the cost is still high.

Now suppose we have many domains Ω_i, each of size $H \gg h$. In other words, think of the Ω_i as the regions bounded by a coarse mesh with spacing H, plus some cells beyond the boundary, as shown by the dashed line in Figure 6.21.

Let $\delta < H$ be the amount by which adjacent domains overlap. Now let H, δ, and h all go to zero such that the overlap fraction δ/H remains constant, and $H \gg h$. Then the number of iterations required for convergence grows like $1/H$, i.e., *independently* of the fine mesh spacing h. This is close to, but still not as good as, multigrid, which does a constant number of iterations and $O(1)$ work per unknown.

Attaining the performance of multigrid requires one more idea, which, perhaps not surprisingly, is similar to multigrid. We use an approximation A_H of the problem on the coarse grid with spacing H to get a *coarse grid preconditioner* in addition to the fine grid preconditioners $A^{-1}_{\Omega_i,\Omega_i}$. We need three matrices to describe the algorithm. First, let A_H be the matrix for the model problem discretized with coarse mesh spacing H. Second, we need a *restriction operator* R to take a residual on the fine mesh and restrict it to values on the coarse mesh; this is essentially the same as in multigrid (see section 6.9.2). Finally, we need an *interpolation operator* to take values on the coarse mesh and interpolate them to the fine mesh; as in multigrid this also turns out to

be R^T.

ALGORITHM 6.20. *Two-level additive Schwarz method for updating an approximate solution x_i of $Ax = b$ to get a better solution x_{i+1}:*

$x_{i+1} = x_i$
for $i = 1$ *to the number of domains* Ω_i
  $r_{\Omega_i} = (b - Ax_i)_{\Omega_i}$
  $x_{i+1,\Omega_i} = x_{i+1,\Omega_i} + A^{-1}_{\Omega_i,\Omega_i} \cdot r_{\Omega_i}$
endfor
$x_{i+1} = x_{i+1} + R^T A_C^{-1} Rr$

As with Algorithm 6.18, this method is typically used as a preconditioner for a Krylov subspace method.

Convergence theory for this algorithm, which is applicable to more general problems than Poisson's equation, says that as H, δ, and h shrink to 0 with δ/H staying fixed, the number of iterations required to converge is independent of H, h, or δ. This means that as long as the work to solve the subproblems $A^{-1}_{\Omega_i,\Omega_i}$ and A_H^{-1} is proportional to the number of unknowns, the complexity is as good as multigrid.

It is probably evident to the reader that implementing these methods in a real world problem can be complicated. There is software available on-line that implements many of the building blocks described here and also runs on parallel machines. It is called PETSc, for Portable Extensible Toolkit for Scientific computing. PETSc is available at http://www.mcs.anl.gov/petsc/petsc.html and is described briefly in [232].

6.11. References and Other Topics for Chapter 6

Up-to-date surveys of modern iterative methods are given in [15, 107, 136, 214], and their parallel implementations are also surveyed in [76]. Classical methods such as Jacobi's, Gauss–Seidel, and SOR methods are discussed in detail in [249, 137]. Multigrid methods are discussed in [43, 185, 186, 260, 268] and the references therein; [91] is a Web site with pointers to an extensive bibliography, software, and so on. Domain decomposition are discussed in [49, 116, 205, 232]. Chebyshev and other polynomials are discussed in [240]. The FFT is discussed in any good textbook on computer science algorithms, such as [3] and [248]. A stabilized version of block cyclic reduction is found in [47, 46].

6.12. Questions for Chapter 6

QUESTION 6.1. *(Easy)* Prove Lemma 6.1.

QUESTION 6.2. *(Easy)* Prove the following formulas for triangular factorizations of T_N.

1. The Cholesky factorization $T_N = B_N^T B_N$ has a upper bidiagonal Cholesky factor B_N with

$$B_N(i,i) = \sqrt{\frac{i+1}{i}} \quad \text{and} \quad B_N(i,i+1) = \sqrt{\frac{i}{i+1}}.$$

2. The result of Gaussian elimination with partial pivoting on T_N is $T_N = L_N U_N$, where the triangular factors are bidiagonal:

$$L_N(i,i) = 1 \quad \text{and} \quad L_N(i+1,i) = -\frac{i}{i+1},$$

$$U_N(i,i) = \frac{i+1}{i} \quad \text{and} \quad U_N(i,i+1) = -1.$$

3. $T_N = D_N D_N^T$, where D_N is the N-by-$(N+1)$ upper bidiagonal matrix with 1 on the main diagonal and -1 on the superdiagonal.

QUESTION 6.3. *(Easy)* Confirm equation (6.13).

QUESTION 6.4. *(Easy)*

1. Prove Lemma 6.2.
2. Prove Lemma 6.3.
3. Prove that the Sylvester equation $AX - XB = C$ is equivalent to $(I_n \otimes A - B^T \otimes I_m)\text{vec}(X) = \text{vec}(C)$.
4. Prove that $\text{vec}(AXB) = (B^T \otimes A) \cdot \text{vec}(X)$.

QUESTION 6.5. *(Medium)* Suppose that $A^{n\times n}$ is diagonalizable, so A has n independent eigenvectors: $Ax_i = \alpha_i x_i$, or $AX = X\Lambda_A$, where $X = [x_1, \ldots, x_n]$ and $\Lambda_A = \text{diag}(\alpha_i)$. Similarly, suppose that $B^{m\times m}$ is diagonalizable, so b has m independent eigenvectors: $By_i = \beta_i y_i$, or $BY = Y\Lambda_B$, where $Y = [y_1, \ldots, y_m]$ and $\Lambda_B = \text{diag}(\beta_j)$. Prove the following results.

1. The mn eigenvalues of $I_m \otimes A + B \otimes I_n$ are $\lambda_{ij} = \alpha_i + \beta_j$, i.e., all possible sums of pairs of eigenvalues of A and B. The corresponding eigenvectors are z_{ij}, where $z_{ij} = x_i \otimes y_j$, whose $(km+l)$th entry is $x_i(k)y_j(l)$. Written another way,

$$(I_m \otimes A + B \otimes I_n)(Y \otimes X) = (Y \otimes X) \cdot (I_m \otimes \Lambda_A + \Lambda_B \otimes I_n). \quad (6.63)$$

2. The Sylvester equation $AX + XB^T = C$ is nonsingular (solvable for X, given any C) if and only if the sum $\alpha_i + \beta_j \neq 0$ for all eigenvalues α_i of A and β_j of B. The same is true for the slightly different Sylvester equation $AX + XB = C$ (see also Question 4.6).

3. The mn eigenvalues of $A \otimes B$ are $\lambda_{ij} = \alpha_i \beta_j$, i.e., all possible products of pairs of eigenvalues of A and B. The corresponding eigenvectors are z_{ij}, where $z_{ij} = x_i \otimes y_j$, whose $(km + l)$th entry is $x_i(k)y_j(l)$. Written another way,

$$(B \otimes A)(Y \otimes X) = (Y \otimes X) \cdot (\Lambda_B \otimes \Lambda_A). \tag{6.64}$$

QUESTION 6.6. *(Easy; Programming)* Write a one-line Matlab program to implement Algorithm 6.2: one step of Jacobi's algorithm for Poisson's equation. Test it by confirming that it converges as fast as predicted in section 6.5.4.

QUESTION 6.7. *(Hard)* Prove Lemma 6.7.

QUESTION 6.8. *(Medium; Programming)* Write a Matlab program to solve the discrete model problem on a square using FFTs. The inputs should be the dimension N and a square N-by-N matrix of values of f_{ij}. The outputs should be an N-by-N matrix of solution v_{ij} and the residual $\|T_{N\times N}v - h^2 f\|_2/(\|T_{N\times N}\|_2 \cdot \|v\|)$. You should also produce three-dimensional plots of f and v. Use the FFT built into Matlab. Your program should not have to be more than a few lines long if you use all the features of Matlab that you can. Solve it for several problems whose solutions you know and several you do not:

1. $f_{jk} = \sin(j\pi/(N+1)) \cdot \sin(k\pi/(N+1))$.

2. $f_{jk} = \sin(j\pi/(N+1)) \cdot \sin(k\pi/(N+1)) + \sin(3j\pi/(N+1)) \cdot \sin(5k\pi/(N+1))$.

3. f has a few sharp spikes (both positive and negative) and is 0 elsewhere. This approximates the electrostatic potential of charged particles located at the spikes and with charges proportional to the heights (positive or negative) of the spikes. If the spikes are all positive, this is also the gravitational potential.

QUESTION 6.9. *(Medium)* Confirm that evaluating the formula in (6.47) by performing the matrix-vector multiplications from right to left is mathematically the same as Algorithm 6.13.

QUESTION 6.10. *(Medium; Hard)*

1. *(Hard)* Let A and H be real symmetric n-by-n matrices that *commute*, i.e., $AH = HA$. Show that there is an orthogonal matrix Q such that $QAQ^T = \mathrm{diag}(\alpha_1, \ldots, \alpha_n)$ and $QHQ^T = \mathrm{diag}(\theta_1, \ldots, \theta_n)$ are both diagonal. In other words, A and H have the same eigenvectors. Hint: First assume A has distinct eigenvalues, and then remove this assumption.

2. *(Medium)* Let

$$\hat{T} = \begin{bmatrix} \alpha & \theta & & \\ \theta & \ddots & \ddots & \\ & \ddots & \ddots & \theta \\ & & \theta & \alpha \end{bmatrix}$$

be a symmetric tridiagonal Toeplitz matrix, i.e., a symmetric tridiagonal matrix with constant α along the diagonal and θ along the offdiagonals. Write down *simple* formulas for the eigenvalues and eigenvectors of $\hat{T}$. Hint: Use Lemma 6.1.

3. *(Hard)* Let

$$T = \begin{bmatrix} A & H & & \\ H & \ddots & \ddots & \\ & \ddots & \ddots & H \\ & & H & A \end{bmatrix}$$

be an n^2-by-n^2 block tridiagonal matrix, with n copies of A along the diagonal. Let $QAQ^T = \text{diag}(\alpha_1, \ldots, \alpha_n)$ be the eigendecomposition of A, and let $QHQ^T = \text{diag}(\theta_1, \ldots, \theta_n)$ be the eigendecomposition of H as above. Write down *simple* formulas for the n^2 eigenvalues and eigenvectors of T in terms of the α_i, θ_i, and Q. Hint: Use Kronecker products.

4. *(Medium)* Show how to solve $Tx = b$ in $O(n^3)$ time. In contrast, how much bigger are the running times of dense LU factorization and band LU factorization?

5. *(Medium)* Suppose that A and H are (possibly different) symmetric tridiagonal Toeplitz matrices, as defined above. Show how to use the FFT to solve $Tx = b$ in just $O(n^2 \log n)$ time.

QUESTION 6.11. *(Easy)* Suppose that R is upper triangular and nonsingular and that C is upper Hessenberg. Confirm that RCR^{-1} is upper Hessenberg.

QUESTION 6.12. *(Medium)* Confirm that the Krylov subspace $\mathcal{K}_k(A, y_1)$ has dimension k if and only if the Arnoldi algorithm (Algorithm 6.9) or the Lanczos algorithm (Algorithm 6.10) can compute q_k without quitting first.

QUESTION 6.13. *(Medium)* Confirm that when $A^{n \times n}$ is symmetric positive definite and $Q^{n \times k}$ has full column rank, then $T = Q^T AQ$ is also symmetric positive definite. (For this question, Q need not be orthogonal.)

QUESTION 6.14. *(Medium)* Prove Theorem 6.9.

QUESTION 6.15. *(Medium; Hard)*

1. *(Medium)* Confirm equation (6.58).
2. *(Medium)* Confirm equation (6.60).
3. *(Hard)* Prove Theorem 6.11.

QUESTION 6.16. *(Medium; Programming)* A Matlab program implementing multigrid to solve the discrete model problem on a square is available on the class homepage at HOMEPAGE/Matlab/MG_README.html. Start by running the demonstration (type "makemgdemo" and then "testfmgv"). Then, try running testfmg for different right-hand sides (input array b), different numbers of weighted Jacobi iterations before and after each recursive call to the multigrid solver (inputs jac1 and jac2), and different numbers of iterations (input iter). The software will plot the convergence rate (ratio of consecutive residuals); does this depend on the size of b? the frequencies in b? the values of jac1 and jac2? For which values of jac1 and jac2 is the solution most efficient?

QUESTION 6.17. *(Medium; Programming)* Using a fast model problem solver from either Question 6.8 or Question 6.16, use domain decomposition to build a fast solver for Poisson's equation on an L-shaped region, as described in section 6.10. The large square should be 1-by-1 and the small square should be .5-by-.5, attached at the bottom right of the large square. Compute the residual in order to show that your answer is correct.

QUESTION 6.18. *(Hard)* Fill in the entries of a table like Table 6.1, but for solving Poisson's equation in three dimensions instead of two. Assume that the grid of unknowns is $N \times N \times N$, with $n = N^3$. Try to fill in as many entries of columns 2 and 3 as you can.

7

Iterative Methods for Eigenvalue Problems

7.1. Introduction

In this chapter we discuss iterative methods for finding eigenvalues of matrices that are too large to use the direct methods of Chapters 4 and 5. In other words, we seek algorithms that take far less than $O(n^2)$ storage and $O(n^3)$ flops. Since the eigenvectors of most n-by-n matrices would take n^2 storage to represent, this means that we seek algorithms that compute just a few user-selected eigenvalues and eigenvectors of a matrix.

We will depend on the material on Krylov subspace methods developed in section 6.6, the material on symmetric eigenvalue problems in section 5.2, and the material on the power method and inverse iteration in section 5.3. The reader is advised to review these sections.

The simplest eigenvalue problem is to compute just the largest eigenvalue in absolute value, along with its eigenvector. The power method (Algorithm 4.1) is the simplest algorithm suitable for this task: Recall that its inner loop is

$$\begin{aligned} y_{i+1} &= Ax_i, \\ x_{i+1} &= y_{i+1}/\|y_{i+1}\|_2, \end{aligned}$$

where x_i converges to the eigenvector corresponding to the desired eigenvector (provided that there is only one eigenvalue of largest absolute value, and x_1 does not lie in an invariant subspace not containing its eigenvector). Note that the algorithm uses A only to perform matrix-vector multiplication, so all that we need to run the algorithm is a "black-box" that takes x_i as input and returns Ax_i as output (see Example 6.13).

A closely related problem is to find the eigenvalue closest to a user-supplied value σ, along with its eigenvector. This is precisely the situation inverse iteration (Algorithm 4.2) was designed to handle. Recall that its inner loop is

$$\begin{aligned} y_{i+1} &= (A - \sigma I)^{-1} x_i, \\ x_{i+1} &= y_{i+1}/\|y_{i+1}\|_2, \end{aligned}$$

i.e., solving a linear system of equations with coefficient matrix $A-\sigma I$. Again x_i converges to the desired eigenvector, provided that there is just one eigenvalue closest to σ (and x_1 satisfies the same property as before). Any of the sparse matrix techniques in Chapter 6 or section 2.7.4 could be used to solve for y_{i+1}, although this is usually much more expensive than simply multiplying by A. When A is symmetric Rayleigh quotient iteration (Algorithm 5.1) can also be used to accelerate convergence (although it is not always guaranteed to converge to the eigenvalue of A closest to σ).

Starting with a given x_1, $k-1$ iterations of either the power method or inverse iteration produce a sequence of vectors $x_1, x_2, \ldots, x_k$. These vectors span a *Krylov subspace*, as defined in section 6.6.1. In the case of the power method, this Krylov subspace is $\mathcal{K}_k(A, x_1) = \text{span}[x_1, Ax_1, A^2x_1, \ldots, A^{k-1}x_1]$, and in the case of inverse iteration this Krylov subspace is $\mathcal{K}_k((A-\sigma I)^{-1}, x_1)$. Rather than taking x_k as our approximate eigenvector, it is natural to ask for the "best" approximate eigenvector in $\mathcal{K}_k$, i.e., the best linear combination $\sum_{i=1}^{k} \alpha_i x_i$. We took the same approach for solving $Ax = b$ in section 6.6.2, where we asked for the best approximate solution to $Ax = b$ from $\mathcal{K}_k$. We will see that the best eigenvector (and eigenvalue) approximations from $\mathcal{K}_k$ are much better than x_k alone. Since $\mathcal{K}_k$ has dimension k (in general), we can actually use it to compute k best approximate eigenvalues and eigenvectors. These best approximations are called the *Ritz values* and *Ritz vectors*.

We will concentrate on the symmetric case $A = A^T$. In the last section we will briefly describe the nonsymmetric case.

The rest of this chapter is organized as follows. Section 7.2 discusses the Rayleigh–Ritz method, our basic technique for extracting information about eigenvalues and eigenvectors from a Krylov subspace. Section 7.3 discusses our main algorithm, the Lanczos algorithm, in exact arithmetic. Section 7.4 analyzes the rather different behavior of the Lanczos algorithm in floating point arithmetic, and sections 7.5 and 7.6 describe practical implementations of Lanczos that compute reliable answers despite roundoff. Finally, section 7.7 briefly discusses algorithms for the nonsymmetric eigenproblem.

7.2. The Rayleigh–Ritz Method

Let $Q = [Q_k, Q_u]$ be any n-by-n orthogonal matrix, where Q_k is n-by-k and Q_u is n-by-$(n-k)$. In practice the columns of Q_k will be computed by the Lanczos algorithm (Algorithm 6.10 or Algorithm 7.1 below) and span a Krylov subspace $\mathcal{K}_k$, and the subscript u indicates that Q_u is (mostly) unknown. But for now we do not care where we get Q.

We will use the following notation (which was also used in equation (6.31)):

$$T = Q^T A Q = [Q_k, Q_u]^T A [Q_k, Q_u] = \begin{bmatrix} Q_k^T A Q_k & Q_k^T A Q_u \\ Q_u^T A Q_k & Q_u^T A Q_u \end{bmatrix}$$

$$\begin{aligned} & & k & n-k \\ \equiv & \begin{matrix} k \\ n-k \end{matrix} & \begin{pmatrix} T_k & T_{uk} \\ T_{ku} & T_u \end{pmatrix} \\ = & & \begin{bmatrix} T_k & T_{ku}^T \\ T_{ku} & T_u \end{bmatrix}. \end{aligned} \tag{7.1}$$

When $k = 1$, T_k is just the Rayleigh quotient $T_1 = \rho(Q_1, A)$ (see Definition 5.1). So for $k > 1$, T_k is a natural generalization of the Rayleigh quotient.

DEFINITION 7.1. *The* Rayleigh–Ritz procedure *is to approximate the eigenvalues of A by the eigenvalues of $T_k = Q_k^T A Q_k$. These approximations are called* Ritz values. *Let $T_k = V \Lambda V^T$ be the eigendecomposition of T_k. The corresponding eigenvector approximations are the columns of $Q_k V$ and are called* Ritz vectors.

The Ritz values and Ritz vectors are considered *optimal* approximations to the eigenvalues and eigenvectors of A for several reasons. First, when Q_k and so T_k are known but Q_u and so T_{ku} and T_u are unknown, the Ritz values and vectors are the natural approximations from the known part of the matrix. Second, they satisfy the following generalization of Theorem 5.5. (Theorem 5.5 showed that the Rayleigh quotient was a "best approximation" to a single eigenvalue.) Recall that the columns of Q_k span an invariant subspace of A if and only if $AQ_k = Q_k R$ for some matrix R.

THEOREM 7.1. *The minimum of $\|AQ_k - Q_k R\|_2$ over all k-by-k symmetric matrices R is attained by $R = T_k$, in which case $\|AQ_k - Q_k R\|_2 = \|T_{ku}\|_2$. Let $T_k = V \Lambda V^T$ be the eigendecomposition of T_k. The minimum of $\|AP_k - P_k D\|_2$ over all n-by-k orthogonal matrices P_k where* $\text{span}(P_k) = \text{span}(Q_k)$ *and over diagonal D is also $\|T_{ku}\|_2$ and is attained by $P_k = Q_k V$ and $D = \Lambda$.*

In other words, the columns of $Q_k V$ (the Ritz vectors) are the "best" approximate eigenvectors and the diagonal entries of Λ (the Ritz values) are the "best" approximate eigenvalues in the sense of minimizing the residual $\|AP_k - P_k D\|_2$.

Proof. We temporarily drop the subscripts k on T_k and Q_k to simplify notation, so we can write the k-by-k matrix $T = Q^T A Q$. Let $R = T + Z$. We want to show $\|AQ - QR\|_2^2$ is minimized when $Z = 0$. We do this by using a disguised form of the Pythagorean theorem:

$$\begin{aligned} \|AQ - QR\|_2^2 &= \lambda_{\max}\left[(AQ - QR)^T (AQ - QR)\right] \\ &\qquad \text{by Part 7 of Lemma 1.7} \\ &= \lambda_{\max}\left[(AQ - Q(T+Z))^T (AQ - Q(T+Z))\right] \\ &= \lambda_{\max}\left[(AQ - QT)^T (AQ - QT) - (AQ - QT)^T (QZ)\right. \\ &\quad \left. -(QZ)^T (AQ - QT) + (QZ)^T (QZ)\right] \end{aligned}$$

$$
\begin{aligned}
&= \lambda_{\max}\left[(AQ-QT)^T(AQ-QT) - (Q^TAQ-T)Z\right. \\
&\quad \left. -Z^T(Q^TAQ-T) + Z^TZ\right] \\
&= \lambda_{\max}\left[(AQ-QT)^T(AQ-QT) + Z^TZ\right] \\
&\qquad \text{because } Q^TAQ = T \\
&\geq \lambda_{\max}\left[(AQ-QT)^T(AQ-QT)\right] \\
&\qquad \text{by Question 5.5, since } Z^TZ \text{ is} \\
&\qquad \text{symmetric positive semidefinite} \\
&= \|AQ-QT\|_2^2 \quad \text{by Part 7 of Lemma 1.7.}
\end{aligned}
$$

Restoring subscripts, it is easy to compute the minimum value

$$\|AQ_k - Q_kT_k\|_2 = \|(Q_kT_k + Q_uT_{ku}) - (Q_kT_k)\|_2 = \|Q_uT_{ku}\|_2 = \|T_{ku}\|_2.$$

If we replace Q_k with any product Q_kU, where U is another orthogonal matrix, then the columns of Q_k and Q_kU span the same space, and

$$\|AQ_k - Q_kR\|_2 = \|AQ_kU - Q_kRU\|_2 = \|A(Q_kU) - (Q_kU)(U^TRU)\|_2.$$

These quantities are still minimized when $R = T_k$, and by choosing $U = V$ so that U^TT_kU is diagonal, we solve the second minimization problem in the statement of the theorem. □

This theorem justifies using Ritz values as eigenvalue approximations. When Q_k is computed by the Lanczos algorithm, in which case (see equation (6.31))

$$T = \left[\begin{array}{c|c} T_k & T_{ku}^T \\ \hline T_{ku} & T_u \end{array}\right] = \left[\begin{array}{ccccc|ccccc} \alpha_1 & \beta_1 & & & & & & & & \\ \beta_1 & \ddots & \ddots & & & & & & & \\ & \ddots & \ddots & \beta_{k-1} & & & & & & \\ & & \beta_{k-1} & \alpha_k & \beta_k & & & & & \\ \hline & & & \beta_k & \alpha_{k+1} & \beta_{k+1} & & & & \\ & & & & \beta_{k+1} & \ddots & \ddots & & & \\ & & & & & \ddots & \ddots & \beta_{n-1} & & \\ & & & & & & \beta_{n-1} & \alpha_n & & \end{array}\right],$$

then it is easy to compute all the quantities in Theorem 7.1. This is because there are good algorithms for finding eigenvalues and eigenvectors of the symmetric tridiagonal matrix T_k (see section 5.3) and because the residual norm is simply $\|T_{ku}\|_2 = \beta_k$. (From the Lanczos algorithm we know that β_k is nonnegative.) This simplifies the error bounds on the approximate eigenvalues and eigenvectors in the following theorem.

THEOREM 7.2. *Let T_k, T_{ku}, and Q_k be as in equation* (7.1). *Let $T_k = V\Lambda V^T$ be the eigendecomposition of T_k, where $V = [v_1, \ldots, v_k]$ is orthogonal and $\Lambda = \operatorname{diag}(\theta_1, \ldots, \theta_k)$. Then*

1. *There are k eigenvalues $\alpha_1, \ldots, \alpha_k$ of A (not necessarily the largest k) such that $|\theta_i - \alpha_i| \leq \|T_{ku}\|_2$ for $i = 1, \ldots, k$. If Q_k is computed by the Lanczos algorithm, then $|\theta_i - \alpha_i| \leq \|T_{ku}\|_2 = \beta_k$, where β_k is the single (possibly) nonzero entry in the upper right corner of T_{ku}.*

2. *$\|A(Q_k v_i) - (Q_k v_i)\theta_i\|_2 = \|T_{ku} v_i\|_2$. Thus, the difference between the Ritz value θ_i and some eigenvalue α of A is at most $\|T_{ku} v_i\|_2$, which may be much smaller than $\|T_{ku}\|_2$. If Q_k is computed by the Lanczos algorithm, then $\|T_{ku} v_i\|_2 = \beta_k |v_i(k)|$, where $v_i(k)$ is the kth (bottom) entry of v_i. This formula lets us compute the residual $\|A(Q_k v_i) - (Q_k v_i)\theta_i\|_2$ cheaply, i.e., without multiplying any vector by Q_k or by A.*

3. *Without any further information about the spectrum of T_u, we cannot deduce any useful error bound on the Ritz vector $Q_k v_i$. If we know that the gap between θ_i and any other eigenvalue of T_k or T_u is at least g, then we can bound the angle θ between $Q_k v_i$ and a true eigenvector of A by*

$$\frac{1}{2} \sin 2\theta \leq \frac{\|T_{ku}\|_2}{g}. \tag{7.2}$$

If Q_k is computed by the Lanczos algorithm, then the bound simplifies to

$$\frac{1}{2} \sin 2\theta \leq \frac{\beta_k}{g}.$$

Proof.

1. The eigenvalues of $\hat{T} = \begin{bmatrix} T_k & 0 \\ 0 & T_u \end{bmatrix}$ include θ_1 through θ_k. Since

$$\|\hat{T} - T\|_2 = \left\| \begin{bmatrix} 0 & T_{ku}^T \\ T_{ku} & 0 \end{bmatrix} \right\|_2 = \|T_{ku}\|_2,$$

Weyl's theorem, Theorem 5.1, tells us that the eigenvalues of $\hat{T}$ and T differ by at most $\|T_{ku}\|_2$. But the eigenvalues of T and A are identical, proving the result.

2. We compute

$$\begin{aligned} \|A(Q_k v_i) - (Q_k v_i)\theta_i\|_2 &= \|Q^T A(Q_k v_i) - Q^T (Q_k v_i)\theta_i\|_2 \\ &= \left\| \begin{bmatrix} T_k v_i \\ T_{ku} v_i \end{bmatrix} - \begin{bmatrix} v_i \theta_i \\ 0 \end{bmatrix} \right\|_2 = \left\| \begin{bmatrix} 0 \\ T_{ku} v_i \end{bmatrix} \right\|_2 \\ &\quad \text{since } T_k v_i = \theta_i v_i \\ &= \|T_{ku} v_i\|_2. \end{aligned}$$

Then by Theorem 5.5, A has some eigenvalue α satisfying $|\alpha - \theta_i| \leq \|T_{ku} v_i\|_2$. If Q_k is computed by the Lanczos algorithm, then $\|T_{ku} v_i\|_2 = \beta_k |v_i(k)|$, because only the top right entry of T_{ku}, namely, β_k, is nonzero.

3. We reuse Example 5.4 to show that we cannot deduce a useful error bound on the Ritz vector without further information about the spectrum of T_u:

$$T = \begin{bmatrix} 1+g & \epsilon \\ \epsilon & 1 \end{bmatrix},$$

where $0 < \epsilon < g$. We let $k = 1$ and $Q_1 = [e_1]$, so $T_1 = 1 + g$ and the approximate eigenvector is simply e_1. But as shown in Example 5.4, the eigenvectors of T are close to $[1, \epsilon/g]^T$ and $[-\epsilon/g, 1]^T$. So without a lower bound on g, i.e., the gap between the eigenvalue of T_k and all the other eigenvalues, including those of T_u, we cannot bound the error in the computed eigenvector. If we do have such a lower bound, we can apply the second bound of Theorem 5.4 to T and $T + E = \text{diag}(T_k, T_u)$ to derive equation (7.2). ◇

7.3. The Lanczos Algorithm in Exact Arithmetic

The Lanczos algorithm for finding eigenvalues of a symmetric matrix A combines the Lanczos algorithm for building a Krylov subspace (Algorithm 6.10) with the Rayleigh–Ritz procedure of the last section. In other words, it builds an orthogonal matrix $Q_k = [q_1, \ldots, q_k]$ of orthogonal Lanczos vectors and approximates the eigenvalues of A by the Ritz values (the eigenvalues of the symmetric tridiagonal matrix $T_k = Q_k^T A Q_k$), as in equation (7.1).

ALGORITHM 7.1. *Lanczos algorithm in exact arithmetic for finding eigenvalues and eigenvectors of* $A = A^T$*:*

$q_1 = b/\|b\|_2$, $\beta_0 = 0$, $q_0 = 0$
for $j = 1$ *to* k
  $z = Aq_j$
  $\alpha_j = q_j^T z$
  $z = z - \alpha_j q_j - \beta_{j-1} q_{j-1}$
  $\beta_j = \|z\|_2$
  if $\beta_j = 0$, *quit*
  $q_{j+1} = z/\beta_j$
  Compute eigenvalues, eigenvectors, and error bounds of T_j
end for

In this section we explore the convergence of the Lanczos algorithm by describing a numerical example in some detail. This example has been chosen to illustrate both typical convergence behavior, as well as some more problematic behavior, which we call *misconvergence*. Misconvergence can occur because the starting vector q_1 is nearly orthogonal to the eigenvector of the desired eigenvalue or when there are multiple (or very close) eigenvalues.

The title of this section indicates that we have (nearly) eliminated the effects of roundoff error on our example. Of course, the Matlab code (HOMEPAGE/Matlab/LanczosFullReorthog.m) used to produce the example below ran in floating point arithmetic, but we implemented the Lanczos algorithm (in particular the inner loop of Algorithm 7.1) in a particularly careful and expensive way in order to make it mimic the exact result as closely as possible. This careful implementation is called *Lanczos with full reorthogonalization*, as indicated in the titles of the figures below.

In the next section we will explore the same numerical example using the original, inexpensive implementation of Algorithm 7.1, which we call *Lanczos with no reorthogonalization* in order to contrast it with *Lanczos with full reorthogonalization.* (We will also explain the difference in the two implementations.) We will see that the original Lanczos algorithm can behave significantly differently from the more expensive "exact" algorithm. Nevertheless, we will show how to use the less expensive algorithm to compute eigenvalues reliably.

EXAMPLE 7.1. We illustrate the Lanczos algorithm and its error bounds by running a large example, a 1000-by-1000 diagonal matrix A, most of whose eigenvalues were chosen randomly from a normal Gaussian distribution. Figure 7.1 is a plot of the eigenvalues. To make later plots easy to understand, we have also sorted the diagonal entries of A from largest to smallest, so $\lambda_i(A) = a_{ii}$, with corresponding eigenvector e_i, the ith column of the identity matrix. There are a few extreme eigenvalues, and the rest cluster near the center of the spectrum. The starting Lanczos vector q_1 has all equal entries, except for one, as described below.

There is no loss in generality in experimenting with a diagonal matrix, since running the Lanczos algorithm on A with starting vector q_1 is equivalent to running the Lanczos algorithm on $Q^T A Q$ with starting vector $Q^T q_1$ (see Question 7.1).

To illustrate convergence, we will use several plots of the sort shown in Figure 7.2. In this figure the eigenvalues of each T_k are shown plotted in column k, for $k = 1$ to 9 on the top, and for $k = 1$ to 29 on the bottom, with the eigenvalues of A plotted in an extra column at the right. Thus, column k has k pluses, one marking each eigenvalue of T_k. We have also color-coded the eigenvalues as follows: The largest and smallest eigenvalues of each T_k are shown in black, the second largest and second smallest eigenvalues are red, the third largest and third smallest eigenvalues are green, and the fourth largest and fourth smallest eigenvalues are blue. Then these colors recycle into the interior of the spectrum.

To understand convergence, consider the largest eigenvalue of each T_k; these black pluses are on the top of each column. Note that they increase monotonically as k increases; this is a consequence of the Cauchy interlace theorem, since T_k is a submatrix of T_{k+1} (see Question 5.4). In fact, the Cauchy interlace theorem tells us more, that the eigenvalues of T_k *interlace* those of T_{k+1},

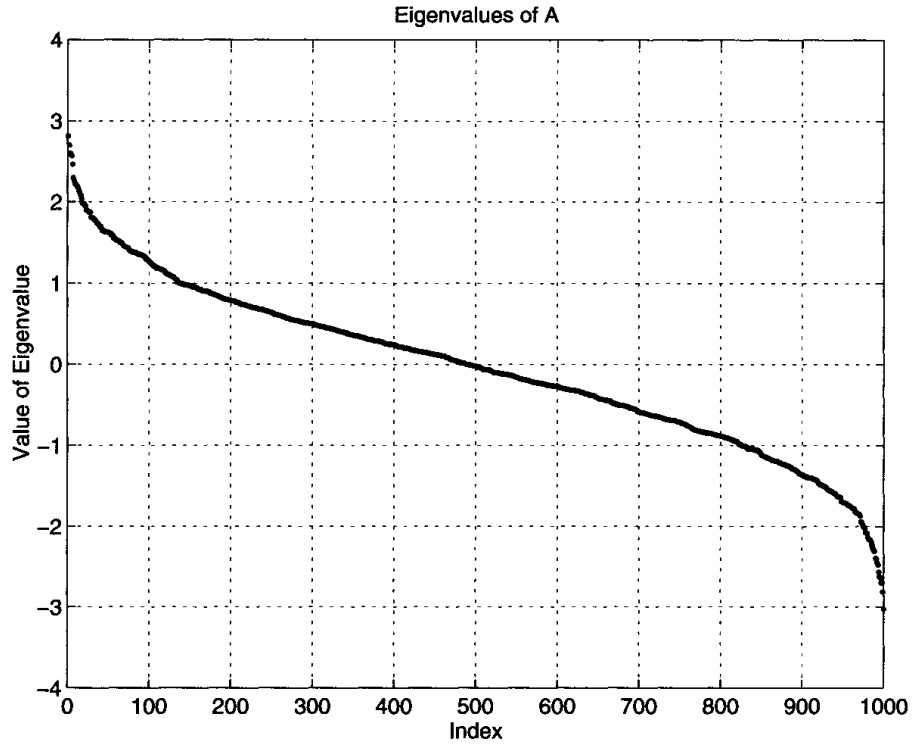


Fig. 7.1. *Eigenvalues of the diagonal matrix A.*

or that $\lambda_i(T_{k+1}) \geq \lambda_i(T_k) \geq \lambda_{i+1}(T_{k+1}) \geq \lambda_{i+1}(T_k)$. In other words, $\lambda_i(T_k)$ increases monotonically with k for any fixed i, not just $i = 1$ (the largest eigenvalue). This is illustrated by the colored sequences of pluses moving right and up in the figure.

A completely analogous phenomenon occurs with the smallest eigenvalues: The bottom black plus sign in each column of Figure 7.2 shows the smallest eigenvalue of each T_k, and these are monotonically decreasing as k increases. Similarly, the ith smallest eigenvalue is also monotonically decreasing. This is also a simple consequence of the Cauchy interlace theorem.

Now we can ask to which eigenvalue of A the eigenvalue $\lambda_i(T_k)$ can converge as k increases. Clearly the largest eigenvalue of T_k, $\lambda_1(T_k)$, ought to converge to the largest eigenvalue of A, $\lambda_1(A)$. Indeed, if the Lanczos algorithm proceeds to step $k = n$ (without quitting early because some $\beta_k = 0$), then T_n and A are similar, and so $\lambda_1(T_n) = \lambda_1(A)$. Similarly, the ith largest eigenvalue $\lambda_i(T_k)$ of T_k must increase monotonically and converge to the ith largest eigenvalue $\lambda_i(A)$ of A (provided that the Lanczos algorithm does not quit early). And the ith smallest eigenvalue $\lambda_{k+1-i}(T_k)$ of T_k must similarly decrease monotonically and converge to the ith smallest eigenvalue $\lambda_{n+1-i}(A)$ of A.

All these converging sequences are represented by sequences of pluses of a common color in Figure 7.2 and other figures in this section. Consider the bottom graph in Figure 7.2: For k larger than about 15, the topmost and bottom-most black pluses form horizontal rows next to the extreme eigenvalues of A, which are plotted in the rightmost column; this demonstrates conver-

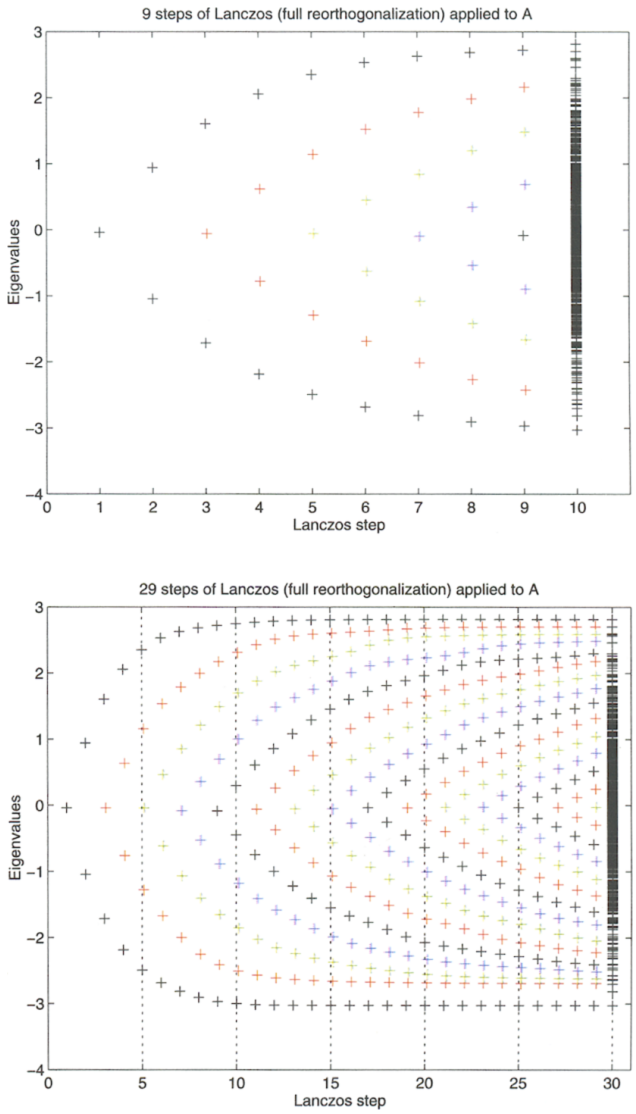


Fig. 7.2. *The Lanczos algorithm applied to A. The first* 9 *steps are shown on the top, and the first* 29 *steps are shown on the bottom. Column k shows the eigenvalues of T_k, except that the rightmost columns (column* 10 *on the top and column* 30 *on the bottom) show all the eigenvalues of A.*

gence. Similarly, the top sequence of red pluses forms a horizontal row next to the second largest eigenvalue of A in the rightmost column; they converge later than the outermost eigenvalues. A blow-up of this behavior for more Lanczos algorithm steps is shown in the top two graphs of Figure 7.3.

To summarize the above discussion, *extreme eigenvalues, i.e., the largest and smallest ones, converge first, and the interior eigenvalues converge last. Furthermore, convergence is monotonic, with the ith largest (smallest) eigenvalue of T_k increasing (decreasing) to the ith largest (smallest) eigenvalue of A, provided that the Lanczos algorithm does not stop prematurely with some $\beta_k = 0$.*

Now we examine the convergence behavior in more detail, compute the actual errors in the Ritz values, and compare these errors with the error bounds

in part 2 of Theorem 7.2. We run the Lanczos algorithm for 99 steps on the same matrix pictured in Figure 7.2 and display the results in Figure 7.3. The top left graph in Figure 7.3 shows only the largest eigenvalues, and the top right graph shows only the smallest eigenvalues.

The middle two graphs in Figure 7.3 show the errors in the four largest computed eigenvalues (on the left) and the four smallest computed eigenvalues (on the right). The colors in the middle graphs match the colors in the top graphs. We measure and plot the errors in three ways:

- The *global errors* (the solid lines) are given by $|\lambda_i(T_k) - \lambda_i(A)|/|\lambda_i(A)|$. We divide by $|\lambda_i(A)|$ in order to normalize all the errors to lie between 1 (no accuracy) and about 10^{-16} (machine epsilon, or full accuracy). As k increases, the global error decreases monotonically, and we expect it to decrease to machine epsilon, unless the Lanczos algorithm quits prematurely.

- The *local errors* (the dotted lines) are given by $\min_j |\lambda_i(T_k) - \lambda_j(A)|/|\lambda_i(A)|$. The local error measures the smallest distance between $\lambda_i(T_k)$ and the *nearest* eigenvalue $\lambda_j(A)$ of A, not just the ultimate value $\lambda_i(A)$. We plot this because sometimes the local error is much smaller than the global error.

- The *error bounds* (the dashed lines) are the quantities $|\beta_k v_i(k)|/|\lambda_i(A)|$ computed by the algorithm (except for the normalization by $|\lambda_i(A)|$, which of course the algorithm does not know!).

The bottom two graphs in Figure 7.3 show the eigenvector components of the Lanczos vectors q_k for the four eigenvectors corresponding to the four largest eigenvalues (on the left) and for the four eigenvectors corresponding to the four smallest eigenvalues (on the right). In other words, they plot $q_k^T e_j = q_k(j)$, where e_j is the jth eigenvector of the diagonal matrix A, for $k = 1$ to 99 and for $j = 1$ to 4 (on the left) and $j = 997$ to 1000 (on the right). The components are plotted on a logarithmic scale, with "+" and "o" to indicate whether the component is positive or negative, respectively. We use these plots to help explain convergence below.

Now we use Figure 7.3 to examine convergence in more detail. The largest eigenvalue of T_k (topmost black pluses in the top left graph of Figure 7.3) begins converging to its final value (about 2.81) right away, is correct to six decimal places after 25 Lanczos steps, and is correct to machine precision by step 50. The global error is shown by the solid black line in the middle left graph. The local error (the dotted black line) is the same as the global error after not too many steps, although it can be "accidentally" much smaller if an eigenvalue $\lambda_i(T_k)$ happens to fall close to some other $\lambda_j(A)$ on its way to $\lambda_i(A)$. The dashed black line in the same graph is the relative error bound computed by the algorithm, which overestimates the true error up to about

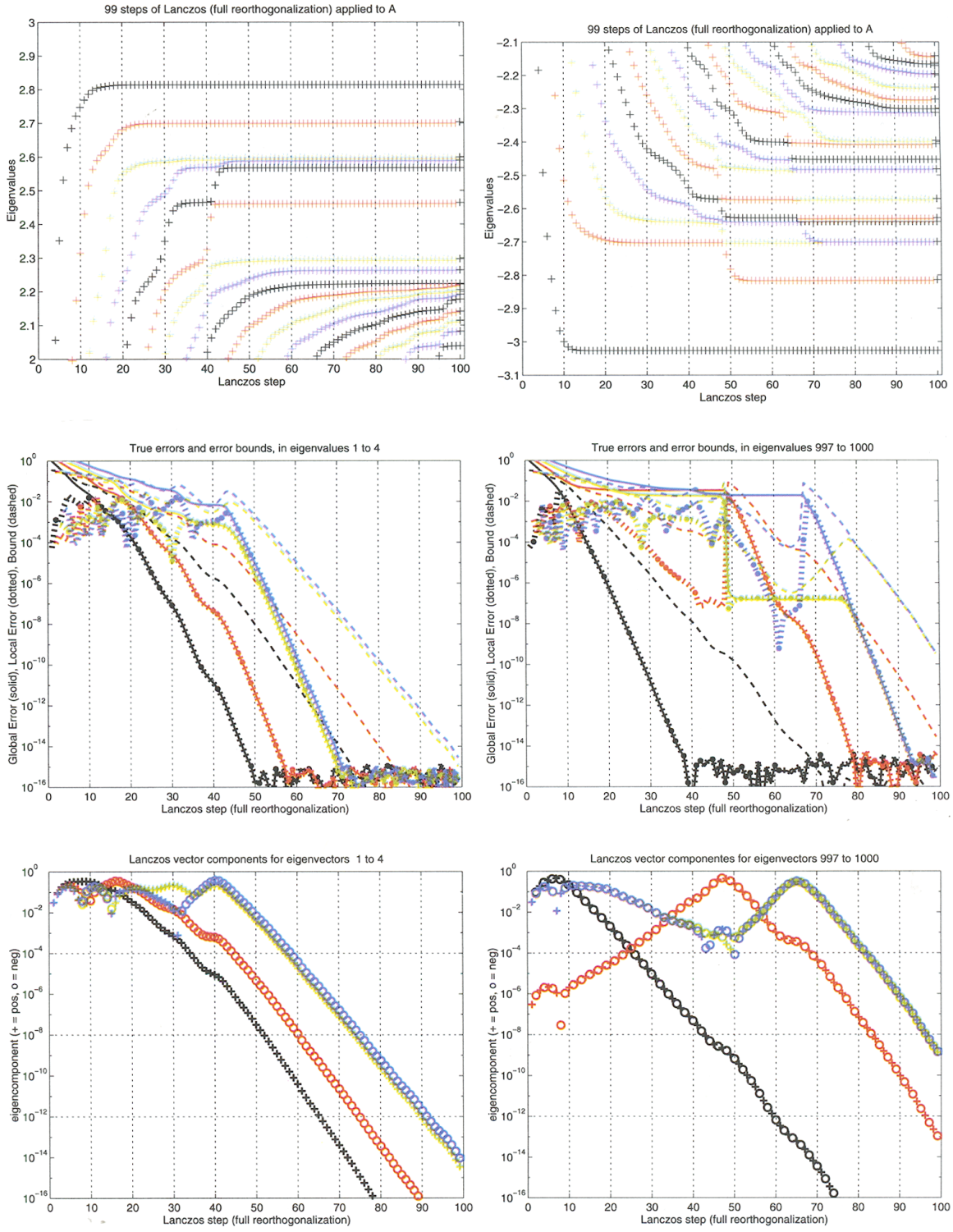


Fig. 7.3. *99 steps of the Lanczos algorithm applied to A. The largest eigenvalues are shown on the left, and the smallest on the right. The top two graphs show the eigenvalues themselves, the middle two graphs the errors (global = solid, local = dotted, bounds = dashed), and the bottom two graphs show eigencomponents of Lanczos vectors. The colors in a column of three graphs match.*

step 75. Still, the relative error bound correctly indicates that the largest eigenvalue is correct to several decimal digits.

The second through fourth largest eigenvalues (the topmost red, green and blue pluses in the top left graph of Figure 7.3) converge in a similar fashion, with eigenvalue i converging slightly faster than eigenvalue $i+1$. This is typical behavior of the Lanczos algorithm.

The bottom left graph of Figure 7.3 measures convergence in terms of the eigenvector components $q_k^T e_j$. To explain this graph, consider what happens to the Lanczos vectors q_k as the first eigenvalue converges. Convergence means that the corresponding eigenvector e_1 nearly lies in the Krylov subspace spanned by the Lanczos vectors. In particular, since the first eigenvalue has converged after $k = 50$ Lanczos steps, this means that e_1 must very nearly be a linear combination of q_1 through q_{50}. Since the q_k are mutually orthogonal, this means q_k must also be orthogonal to e_1 for $k > 50$. This is borne out by the black curve in the bottom left graph, which has decreased to less than 10^{-7} by step 50. The red curve is the component of e_2 in q_k, and this reaches 10^{-8} by step 60. The green curve (third eigencomponent) and blue curve (fourth eigencomponent) get comparably small a few steps later.

Now we discuss the smallest four eigenvalues, whose behavior is described by the three graphs on the right of Figure 7.3. We have chosen the matrix A and starting vector q_1 to illustrate certain difficulties that can arise in the convergence of the Lanczos algorithm to show that convergence is not always as straightforward as in the case of the four eigenvalues just examined.

In particular, we have chosen $q_1(999)$, the eigencomponent of q_1 in the direction of the second smallest eigenvalue (-2.81), to be about 10^{-7}, which is 10^5 times smaller than all the other components of q_1, which are equal. Also, we have chosen the third and fourth smallest eigenvalues (numbers 998 and 997) to be nearly the same: -2.700001 and -2.7.

The convergence of the smallest eigenvalue of T_k to $\lambda_{1000}(A) \approx -3.03$ is uneventful, similar to the largest eigenvalues. It is correct to 16 digits by step 40.

The *second* smallest eigenvalue of T_k, shown in red, begins by *misconverging* to the *third* smallest eigenvalue of A, near -2.7. Indeed, the dotted red line in the middle right graph of Figure 7.3 shows that $\lambda_{999}(T_k)$ agrees with $\lambda_{998}(A)$ to six decimal places for Lanczos steps $40 < k < 50$. The corresponding error bound (the red dashed line) tells us that $\lambda_{999}(T_k)$ equals *some* eigenvalue of A to three or four decimal places for the same values of k. The reason $\lambda_{999}(T_k)$ misconverges is that the Krylov subspace starts with a very small component of the corresponding Krylov subspace e_{999}, namely, 10^{-7}. This can be seen by the red curve in bottom right graph, which starts at 10^{-7} and takes until step 45 before a large component of e_{999} appears. Only at this point, when the Krylov subspace contains a sufficiently large component of the eigenvector e_{999}, can $\lambda_{999}(T_k)$ start converging again to its final value $\lambda_{999}(A) \approx -2.81$, as shown in the top and middle right graphs. Once this convergence has set in again,

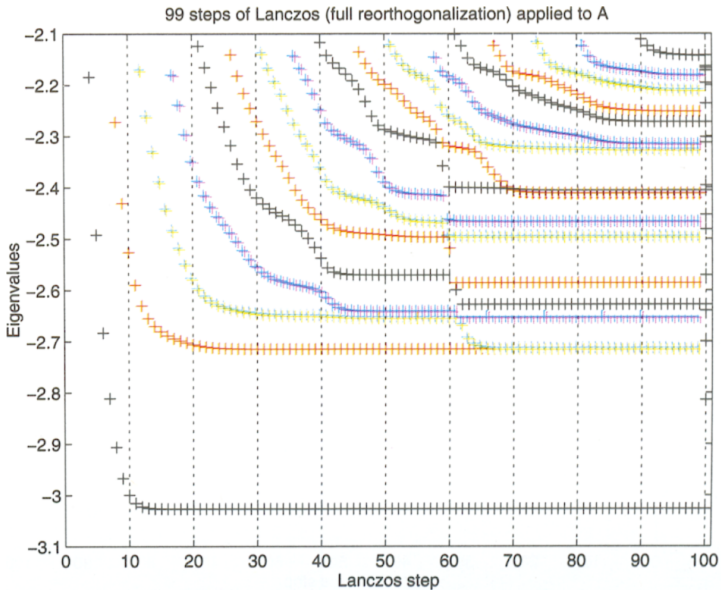


Fig. 7.4. *The Lanczos algorithm applied to A, where the starting vector q_1 is orthogonal to the eigenvector corresponding to the second smallest eigenvalue -2.81. No approximation to this eigenvalue is computed.*

the component of e_{999} starts decreasing again and becomes very small once $\lambda_{999}(T_k)$ has converged to $\lambda_{999}(A)$ sufficiently accurately. (For a quantitative relationship between the convergence rate and the eigencomponent $q_1^T e_{999}$, see the theorem of Kaniel and Saad discussed below.)

Indeed, if q_1 were *exactly* orthogonal to e_{999}, so $q_1^T e_{999} = 0$ rather than just $q_1^T e_{999} = 10^{-7}$, then all later Lanczos vectors would also be orthogonal to e_{999}. This means $\lambda_{999}(T_k)$ would never converge to $\lambda_{999}(A)$. (For a proof, see Question 7.3.) We illustrate this in Figure 7.4, where we have modified q_1 just slightly so that $q_1^T e_{999} = 0$. Note that no approximation to $\lambda_{999}(A) \approx -2.81$ ever appears.

Fortunately, if we choose q_1 at random, it is extremely unlikely to be orthogonal to an eigenvector. We can always rerun the Lanczos algorithm with a different random q_1 to provide more "statistical" evidence that we have not missed any eigenvalues.

Another source of "misconvergence" are (nearly) multiple eigenvalues, such as the the third smallest eigenvalue $\lambda_{998}(A) = -2.700001$ and the fourth smallest eigenvalue $\lambda_{997}(A) = -2.7$. By examining $\lambda_{998}(T_k)$, the bottommost green curve in the top right and middle right graphs of Figure 7.3, we see that during Lanczos steps $50 < k < 75$, $\lambda_{998}(T_k)$ *misconverges* to about -2.7000005, *halfway* between the two closest eigenvalues of A. This is not visible at the resolution provided by the top right graph but is evident from the horizontal segment of the solid green line in the middle right graph during Lanczos steps $50 < k < 75$. At step 76 rapid convergence to the final value $\lambda_{998}(A) = -2.700001$ sets in again.

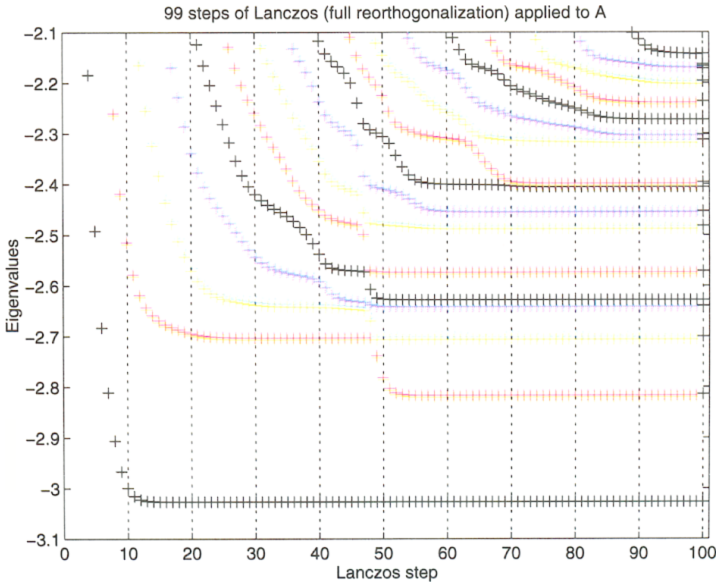


Fig. 7.5. *The Lanczos algorithm applied to A, where the third and fourth smallest eigenvalues are equal. Only one approximation to this double eigenvalue is computed.*

Meanwhile, the fourth smallest eigenvalue $\lambda_{997}(T_k)$, shown in blue, has misconverged to a value near $\lambda_{996}(A) \approx -2.64$; the blue dotted line in the middle right graph indicates that $\lambda_{997}(T_k)$ and $\lambda_{996}(A)$ agree to up to nine decimal places near step $k = 61$. At step $k = 65$ rapid convergence sets in again to the final value $\lambda_{997}(A) = -2.7$. This can also be seen in the bottom right graph, where the eigenvector components of e_{997} and e_{998} grow again during step $50 < k < 65$, after which rapid convergence sets in and they again decrease.

Indeed, if $\lambda_{997}(A)$ were *exactly* a double eigenvalue, we claim that T_k would never have two eigenvalues near that value but only one (in exact arithmetic). (For a proof, see Question 7.3.) We illustrate this in Figure 7.5, where we have modified A just slightly so that it has two eigenvalues exactly equal to -2.7. Note that only one approximation to $\lambda_{998}(A) = \lambda_{997}(A) = -2.7$ ever appears.

Fortunately, there are many applications where it is sufficient to find one copy of each eigenvalue rather than all multiple copies. Also, it is possible to use "block Lanczos" to recover multiple eigenvalues (see the algorithms cited in section 7.6).

Examining other eigenvalues in the top right graph of Figure 7.3, we see that misconvergence is quite common, as indicated by the frequent short horizontal segments of like-colored pluses, which then drop off to the right to the next smaller eigenvalue. For example, the seventh smallest eigenvalue is well-approximated by the fifth (black), sixth (red), and seventh (green) smallest eigenvalues of T_k at various Lanczos steps.

These misconvergence phenomena explain why the computable error bound provided by part 2 of Theorem 7.2 is essential to monitor convergence [198]. If the error bound is small, the computed eigenvalue is indeed a good approx-

imation to some eigenvalue, even if one is "missing." ◇

There is another error bound, due to Kaniel and Saad, that sheds light on why misconvergence occurs. This error bound depends on the angle between the starting vector q_1 and the desired eigenvectors, the Ritz values, and the desired eigenvalues. In other words, it depends on quantities unknown during the computation, so it is not of practical use. But it shows that if q_1 is nearly orthogonal to the desired eigenvector, or if the desired eigenvalue is nearly multiple, then we can expect slow convergence. See [197, sect. 12-4] for details.

7.4. The Lanczos Algorithm in Floating Point Arithmetic

The example in the last section described the behavior of the "ideal" Lanczos algorithm, essentially without roundoff. We call the corresponding careful but expensive implementation of Algorithm 6.10 *Lanczos with full reorthogonalization* to contrast it with the original inexpensive implementation, which we call *Lanczos with no reorthogonalization* (HOMEPAGE/Matlab/LanczosNoReorthog.m). Both algorithms are shown below.

ALGORITHM 7.2. *Lanczos algorithm with full or no reorthogonalization for finding eigenvalues and eigenvectors of* $A = A^T$*:*

$q_1 = b/\|b\|_2$, $\beta_0 = 0$, $q_0 = 0$
for $j = 1$ *to* k
  $z = Aq_j$
  $\alpha_j = q_j^T z$
  $\begin{cases} z = z - \sum_{i=1}^{j-1}(z^T q_i)q_i, \quad z = z - \sum_{i=1}^{j-1}(z^T q_i)q_i & \textit{full reorthogonalization} \\ z = z - \alpha_j q_j - \beta_{j-1}q_{j-1} & \textit{no reorthogonalization} \end{cases}$
  $\beta_j = \|z\|_2$
  if $\beta_j = 0$, *quit*
  $q_{j+1} = z/\beta_j$
  Compute eigenvalues, eigenvectors, and error bounds of T_k
end for

Full reorthogonalization corresponds to applying the Gram–Schmidt orthogonalization process "$z = z - \sum_{i=1}^{j-1}(z^T q_i)q_i$" *twice* in order to almost surely make z orthogonal to q_1 through q_{j-1}. (See Algorithm 3.1 as well as [197, sect. 6-9] and [171, chap. 7] for discussions of when "twice is enough.") In exact arithmetic, we showed in section 6.6.1 that z is orthogonal to q_1 through q_{j-1} without reorthogonalization. Unfortunately, we will see that roundoff destroys this orthogonality property, upon which all of our analysis has depended so far.

This loss of orthogonality does not cause the algorithm to behave completely unpredictably. Indeed, we will see that the price we pay is to get

multiple copies of converged Ritz values. In other words, instead of T_k having one eigenvalue nearly equal to $\lambda_i(A)$ for k large, it may have many eigenvalues nearly equal to $\lambda_i(A)$. This is not a disaster if one is not concerned about computing multiplicities of eigenvalues and does not mind the resulting delayed convergence of interior eigenvalues. See [57] for a detailed description of a Lanczos implementation that operates in this fashion, and NETLIB/lanczos for the software itself. (This software has heuristics for estimating multiplicities of eigenvalues.)

But if accurate multiplicities are important, then one needs to keep the Lanczos vectors (nearly) orthogonal. So one could use the Lanczos algorithm with full reorthogonalization, as we did in the last section. But one can easily confirm that this costs $O(k^2n)$ flops instead of $O(kn)$ flops for k steps, and $O(kn)$ space instead of $O(n)$ space, which may be too high a price to pay.

Fortunately, there is a middle ground between no reorthogonalization and full reorthogonalization, which nearly gets the best of both worlds. It turns out that the q_k lose their orthogonality in a very systematic way by developing large components in the directions of already converged Ritz vectors. (This is what leads to multiple copies of converged Ritz values.) This systematic loss of orthogonality is illustrated by the next example and explained by Paige's theorem below. We will see that by monitoring the computed error bounds, we can conservatively predict which q_k will have large components of which Ritz vectors. Then we can *selectively orthogonalize* q_k against just those few prior Ritz vectors, rather than against all the earlier q_is at each step, as with full reorthogonalization. This keeps the Lanczos vectors (nearly) orthogonal for very little extra work. The next section discusses selective orthogonalization in detail.

EXAMPLE 7.2. Figure 7.7 shows the convergence behavior of 149 steps of Lanczos on the matrix in Example 7.1. The graphs on the right are with full reorthogonalization, and the graphs on the left are with no reorthogonalization. These graphs are similar to those in Figure 7.3, except that the global error is omitted, since this clutters the middle graphs.

Figure 7.6 plots the smallest singular value $\sigma_{\min}(Q_k)$ versus Lanczos step k. In exact arithmetic, Q_k is orthogonal and so $\sigma_{\min}(Q_k) = 1$. With roundoff, Q_k loses orthogonality starting at around step $k = 70$, and $\sigma_{\min}(Q_k)$ drops to .01 by step $k = 80$, which is where the top two graphs in Figure 7.7 begin to diverge visually.

In particular, starting at step $k = 80$ in the top left graph of Figure 7.7, the second smallest (red) eigenvalue $\lambda_2(T_k)$, which had converged to $\lambda_2(A) \approx 2.7$ to almost 16 digits, leaps up to $\lambda_1(A) \approx 2.81$ in just a few steps, yielding a "second copy" of $\lambda_1(A)$ along with $\lambda_1(T_k)$ (in black). (This may be hard to see, since the red pluses overwrite and so obscure the black pluses.) This transition can be seen in the leap in the dashed red error bound in the middle left graph. Also, this transition was "foreshadowed" by the increasing component of e_1

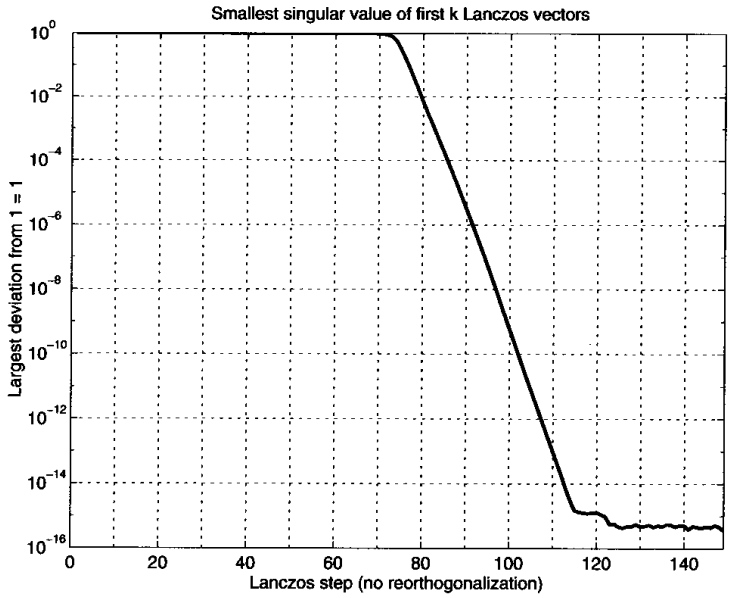


Fig. 7.6. *Lanczos algorithm without reorthogonalization applied to A. The smallest singular value* $\sigma_{\min}(Q_k)$ *of the Lanczos vector matrix* Q_k *is shown for* $k = 1$ *to* 149. *In the absence of roundoff,* Q_k *is orthogonal, and so all singular values should be one. With roundoff,* Q_k *becomes rank deficient.*

in the bottom left graph, where the black curve starts rising again at step $k = 50$ rather than continuing to decrease to machine epsilon, as it does with full reorthogonalization in the bottom right graph. Both of these indicate that the algorithm is diverging from its exact path (and that some selective orthogonalization is called for). After the second copy of $\lambda_1(A)$ has converged, the component of e_1 in the Lanczos vectors starts dropping again, starting a little after step $k = 80$.

Similarly, starting at about step $k = 95$, a second copy of $\lambda_2(A)$ appears when the blue curve $(\lambda_4(T_k))$ in the upper left graph moves from about $\lambda_3(A) \approx 2.6$ to $\lambda_2(A) \approx 2.7$. At this point we have two copies of $\lambda_1(A) \approx 2.81$ and two copies of $\lambda_2(A)$. This is a bit hard to see on the graphs, since the pluses of one color obscure the pluses of the other color (red overwrites black, and blue overwrites green). This transition is indicated by the dashed blue error bound for $\lambda_4(T_k)$ in the middle left graph rising sharply near $k = 95$ and is foreshadowed by the rising red curve in the bottom left graph, which indicates that the component of e_2 in the Lanczos vectors is rising. This component peaks near $k = 95$ and starts dropping again.

Finally, around step $k = 145$, a *third* copy of $\lambda_1(A)$ appears, again indicated and foreshadowed by changes in the two bottom left graphs. If we were to continue the Lanczos process, we would periodically get additional copies of many other converged Ritz values. $\diamond$

The next theorem provides an explanation for the behavior seen in the above example, and hints at a practical criterion for selectively orthogonalizing Lanczos vectors. In order not to be overwhelmed by taking all possible roundoff

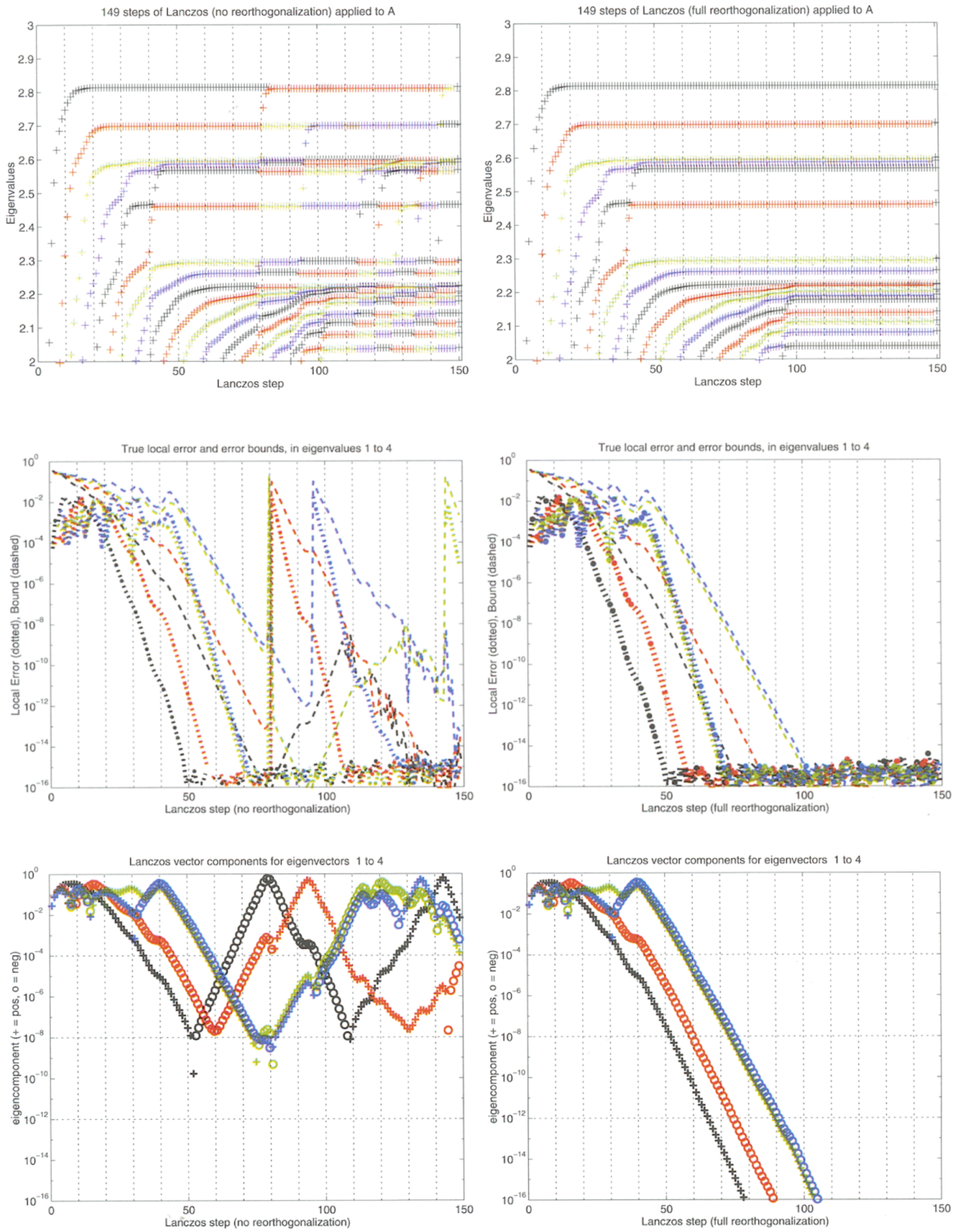


Fig. 7.7. 149 *steps of Lanczos algorithm applied to A. Column* 150 *(at the right of the top graphs) shows the eigenvalues of A. In the left graphs, no reorthogonalization is done. In the right graphs, full reorthogonalization is done.*

errors into account, we will draw on others' experience to identify those few rounding errors that are important, and simply ignore the rest [197, sect. 13-4]. This lets us summarize the Lanczos algorithm with no reorthogonalization in one line:

$$\beta_j q_{j+1} + f_j = Aq_j - \alpha_j q_j - \beta_{j-1} q_{j-1}. \tag{7.3}$$

In this equation the variables represent the values actually stored in the machine, except for f_j, which represents the roundoff error incurred by evaluating the right-hand side and then computing β_j and q_{j+1}. The norm $\|f_j\|_2$ is bounded by $O(\varepsilon\|A\|)$, where ε is machine epsilon, which is all we need to know about f_j. In addition, we will write $T_k = V\Lambda V^T$ *exactly*, since we know that the roundoff errors occurring in this eigendecomposition are not important. Thus, Q_k is not necessarily an orthogonal matrix, but V is.

THEOREM 7.3. Paige. *We use the notation and assumptions of the last paragraph. We also let* $Q_k = [q_1, \ldots, q_k]$, $V = [v_1, \ldots, v_k]$, *and* $\Lambda = \mathrm{diag}(\theta_1, \ldots, \theta_k)$. *We continue to call the columns* $y_{k,i} = Q_k v_i$ *of* $Q_k V$ *the Ritz vectors and the* θ_i *the Ritz values. Then*

$$y_{k,i}^T q_{k+1} = \frac{O(\varepsilon\|A\|)}{\beta_k |v_i(k)|}.$$

In other words the component $y_{k,i}^T q_{k+1}$ of the computed Lanczos vector q_{k+1} in the direction of the Ritz vector $y_{k,i} = Q_k v_i$ is proportional to the reciprocal of $\beta_k |v_i(k)|$, which is the error bound on the corresponding Ritz value θ_i (see Part 2 of Theorem 7.2). Thus, when the Ritz value θ_i converges and its error bound $\beta_k |v_i(k)|$ goes to zero, the Lanczos vector q_{k+1} acquires a large component in the direction of Ritz vector $y_{k,i}$. Thus, the Ritz vectors become linearly dependent, as seen in Example 7.2. Indeed, Figure 7.8 plots both the error bound $|\beta_k v_i(k)|/|\lambda_i(A)| \approx |\beta_k v_i(k)|/\|A\|$ and the Ritz vector component $y_{k,i}^T q_{k+1}$ for the largest Ritz value ($i = 1$, the top graph) and for the second largest Ritz value ($i = 2$, the bottom graph) of our 1000-by-1000 diagonal example. According to Paige's theorem, the product of these two quantities should be $O(\varepsilon)$. Indeed it is, as can be seen by the symmetry of the curves about the middle line $\sqrt{\varepsilon}$ of these semilogarithmic graphs.

Proof of Paige's theorem. We start with equation (7.3) for $j = 1$ to $j = k$, and write these k equations as the single equation

$$\begin{aligned} AQ_k &= Q_k T_k + [0, \ldots, 0, \beta_k q_{k+1}] + F_k \\ &= Q_k T_k + \beta_k q_{k+1} e_k^T + F_k, \end{aligned}$$

where e_k^T is the k-dimensional row vector $[0, \ldots, 0, 1]$ and $F_k = [f_1, \ldots, f_k]$ is the matrix of roundoff errors. We simplify notation by dropping the subscript k to get $AQ = QT + \beta q e^T + F$. Multiply on the left by Q^T to get $Q^T AQ =$

$Q^TQT + \beta Q^T q e^T + Q^T F$. Since Q^TAQ is symmetric, we get that $Q^TQT + \beta Q^T q e^T + Q^T F$ equals its transpose or, rearranging this equality,

$$0 = (Q^TQT - TQ^TQ) + \beta(Q^T q e^T - e q^T Q) + (Q^T F - F^T Q). \tag{7.4}$$

If θ and v are a Ritz value and Ritz vector, respectively, so that $Tv = \theta v$, then note that

$$v^T \beta(e q^T Q) v = [\beta v(k)] \cdot [q^T(Qv)] \tag{7.5}$$

is the product of error bound $\beta v(k)$ and the Ritz vector component $q^T(Qv) = q^T y$, which Paige's theorem says should be $O(\varepsilon \|A\|)$. Our goal is now to manipulate equation (7.4) to get an expression for eq^TQ alone, and then use equation (7.5).

To this end, we now invoke more simplifying assumptions about roundoff: Since each column of Q is gotten by dividing a vector z by its norm, the diagonal of Q^TQ is equal to 1 to full machine precision; we will suppose that it is exactly 1. Furthermore, the vector $z' = z - \alpha_j q_j = z - (q_j^T z) q_j$ computed by the Lanczos algorithm is constructed to be orthogonal to q_j, so it is also true that q_{j+1} and q_j are orthogonal to nearly full machine precision. Thus $q_{j+1}^T q_j = (Q^TQ)_{j+1,j} = O(\varepsilon)$; we will simply assume $(Q^TQ)_{j+1,j} = 0$. Now write $Q^TQ = I + C + C^T$, where C is lower triangular. Because of our assumptions about roundoff, C is in fact nonzero only on the second subdiagonal and below. This means

$$Q^TQT - TQ^TQ = (CT - TC) + (C^TT - TC^T),$$

where we can use the zero structures of C and T to easily show that $CT - TC$ is strictly lower triangular and $C^TT - TC^T$ is strictly upper triangular. Also, since e is nonzero only in its last entry, eq^TQ is nonzero only in the last row. Furthermore, the structure of Q^TQ just described implies that the last entry of the last row of eq^TQ is zero. So in particular, eq^TQ is also strictly lower triangular and Q^Tqe^T is strictly upper triangular. Applying the fact that eq^TQ and $CT - TC$ are both strictly lower triangular to equation (7.4) yields

$$0 = (CT - TC) - \beta e q^T Q + L, \tag{7.6}$$

where L is the strict lower triangle of $Q^TF - F^TQ$. Multiplying equation (7.6) on the left by v^T and on the right by v, using equation (7.5) and the fact that $v^T(CT - TC)v = v^TCv\theta - \theta v^TCv = 0$, yields

$$v^T \beta(e q^T Q) v = [\beta v(k)] \cdot [q^T(Qv)] = v^T L v.$$

Since $|v^TLv| \leq \|L\| = O(\|Q^TF - F^TQ\|) = O(\|F\|) = O(\varepsilon\|A\|)$, we get

$$[\beta v(k)] \cdot [q^T(Qv)] = O(\varepsilon \|A\|),$$

which is equivalent to Paige's theorem. □

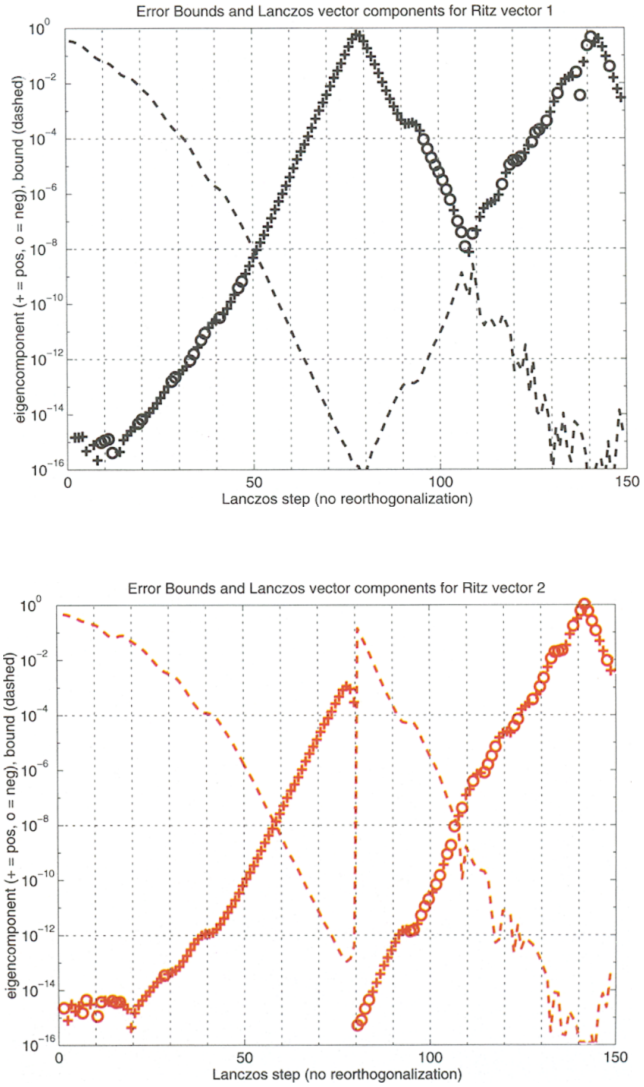


Fig. 7.8. *Lanczos with no reorthogonalization applied to A. The first* 149 *steps are shown for the largest eigenvalue (in black, at top) and for the second largest eigenvalue (in red, at bottom). The dashed lines are error bounds as before. The lines marked by pluses and o's show* $y_{k,i}^T q_{k+1}$, *the component of Lancos vector* $k+1$ *in the direction of the Ritz vector for the largest Ritz value* ($i = 1$, *at top) or for the second largest Ritz value* ($i = 2$, *at bottom).*

7.5. The Lanczos Algorithm with Selective Orthogonalization

We discuss a variation of the Lanczos algorithm which has (nearly) the high accuracy of the Lanczos algorithm with full reorthogonalization but (nearly) the low cost of the Lanczos algorithm with no reorthogonalization. This algorithm is called the Lanczos algorithm with *selective orthogonalization.* As discussed in the last section, our goal is to keep the computed Lanczos vectors q_k as nearly orthogonal as possible (for high accuracy) by orthogonalizing them against as few other vectors as possible at each step (for low cost). Paige's theorem (Theorem 7.3 in the last section) tells us that the q_k lose orthogonality because they acquire large components in the direction of Ritz vectors $y_{i,k} = Q_k v_i$ whose Ritz values θ_i have converged, as measured by the error bound $\beta_k |v_i(k)|$ becoming small. This phenomenon was illustrated in Example 7.2.

Thus, the simplest version of selective orthogonalization simply monitors the error bound $\beta_k |v_i(k)|$ at each step, and when it becomes small enough, the vector z in the inner loop of the Lanczos algorithm is orthogonalized against $y_{i,k}$: $z = z - (y_{i,k}^T z) y_{i,k}$. We consider $\beta_k |v_i(k)|$ to be small when it is less than $\sqrt{\varepsilon}\|A\|$, since Paige's theorem tells us that the vector component $|y_{i,k}^T q_{k+1}| = |y_{i,k}^T z / \|z\|_2|$ is then likely to exceed $\sqrt{\varepsilon}$. (In practice we may replace $\|A\|$ by $\|T_k\|$, since $\|T_k\|$ is known and $\|A\|$ may not be.) This leads to the following algorithm.

ALGORITHM 7.3. *The Lanczos algorithm with selective orthogonalization for finding eigenvalues and eigenvectors of* $A = A^T$:

$q_1 = b/\|b\|_2$, $\beta_0 = 0$, $q_0 = 0$
for $j = 1$ *to* k
  $z = Aq_j$
  $\alpha_j = q_j^T z$
  $z = z - \alpha_j q_j - \beta_{j-1} q_{j-1}$
  /* *Selectively orthogonalize against converged Ritz vectors* */
  for all $i \le k$ *such that* $\beta_k |v_i(k)| \le \sqrt{\varepsilon}\|T_k\|$
    $z = z - (y_{i,k}^T z) y_{i,k}$
  end for
  $\beta_j = \|z\|_2$
  if $\beta_j = 0$, *quit*
  $q_{j+1} = z/\beta_j$
  Compute eigenvalues, eigenvectors, and error bounds of T_k
end for

The following example shows what will happen to our earlier 1000-by-1000 diagonal matrix when this algorithm is used (HOMEPAGE/Matlab/LanczosSelectOrthog.m).

EXAMPLE 7.3. The behavior of the Lanczos algorithm with selective orthogonalization is visually indistinguishable from the behavior of the Lanczos algorithm with full orthogonalization shown in the three graphs on the right of Figure 7.7. In other words, selective orthogonalization provided as much accuracy as full orthogonalization.

The smallest singular values of all the Q_k were greater than $1-10^{-8}$, which means that selective orthogonalization did keep the Lanczos vectors orthogonal to about half precision, as desired.

Figure 7.9 shows the Ritz values corresponding to the Ritz vectors selected for reorthogonalization. Since the selected Ritz vectors correspond to converged Ritz values and the largest and smallest Ritz values converge first, there are two graphs: the large converged Ritz values are at the top, and the small converged Ritz values are at the bottom. The top graph matches the Ritz values shown in the upper right graph in Figure 7.7 that have converged to at least half precision. All together, 1485 Ritz vectors were selected for orthogonalization of a total possible 149*150/2 = 11175. Thus, selective orthogonalization did only 1485/11175 $\approx$ 13% as much work reorthogonalizing to keep the Lanczos vectors (nearly) orthogonal as full reorthogonalization.

Figure 7.10 shows how the Lanczos algorithm with selective reorthogonalization keeps the Lanczos vectors orthogonal just to the Ritz vectors for the largest two Ritz values. The graph at the top is a superposition of the two graphs in Figure 7.8, which show the error bounds and Ritz vectors components for the Lanczos algorithm with no reorthogonalization. The graph at the bottom is the corresponding graph for the Lanczos algorithm with selective orthogonalization. Note that at step $k = 50$, the error bound for the largest eigenvalue (the dashed black line) has reached the threshold of $\sqrt{\varepsilon}$. The Ritz vector is selected for orthogonalization (as shown by the top black pluses in the top of Figure 7.9), and the component in this Ritz vector direction disappears from the bottom graph of Figure 7.10. A few steps later, at $k = 58$, the error bound for the second largest Ritz value reaches $\sqrt{\varepsilon}$, and it too is selected for orthogonalization. The error bounds in the bottom graph continue to decrease to machine epsilon ε and stay there, whereas the error bounds in the top graph eventually grow again. $\diamond$

7.6. Beyond Selective Orthogonalization

Selective orthogonalization is not the end of the story, because the symmetric Lanczos algorithm can be made even less expensive. It turns out that once a Lanczos vector has been orthogonalized against a particular Ritz vector y, it takes many steps before the Lanczos vector again requires orthogonalization against y. So much of the orthogonalization work in Algorithm 7.3 can be eliminated. Indeed, there is a simple and inexpensive recurrence for deciding when to reorthogonalize [224, 192]. Another enhancement is to use the error bounds to efficiently distinguish between converged and "misconverged" eigen-

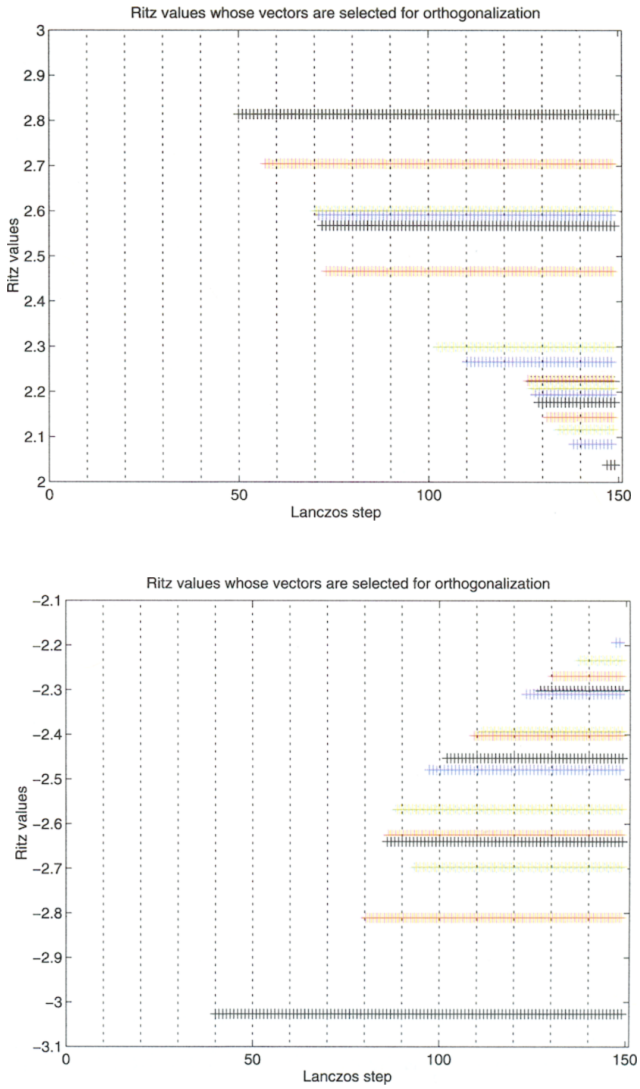


Fig. 7.9. *The Lanczos algorithm with selective orthogonalization applied to A. The Ritz values whose Ritz vectors are selected for orthogonalization are shown.*

values [198]. A state-of-the-art implementation of the Lanczos algorithm is described in [125]. A different software implementation is available in ARPACK (NETLIB/scalapack/readme.arpack [171, 233]).

If we apply the Lanczos algorithm to the shifted and inverted matrix $(A - \sigma I)^{-1}$, then we expect the eigenvalues closest to σ to converge first. There are other methods to "precondition" a matrix A to converge to certain eigenvalues more quickly. For example, Davidson's method [60] is used in quantum chemistry problems, where A is strongly diagonally dominant. It is also possible to combine Davidson's method with Jacobi's method [229].

7.7. Iterative Algorithms for the Nonsymmetric Eigenproblem

When A is nonsymmetric, the Lanczos algorithm described above is no longer applicable. There are two alternatives.

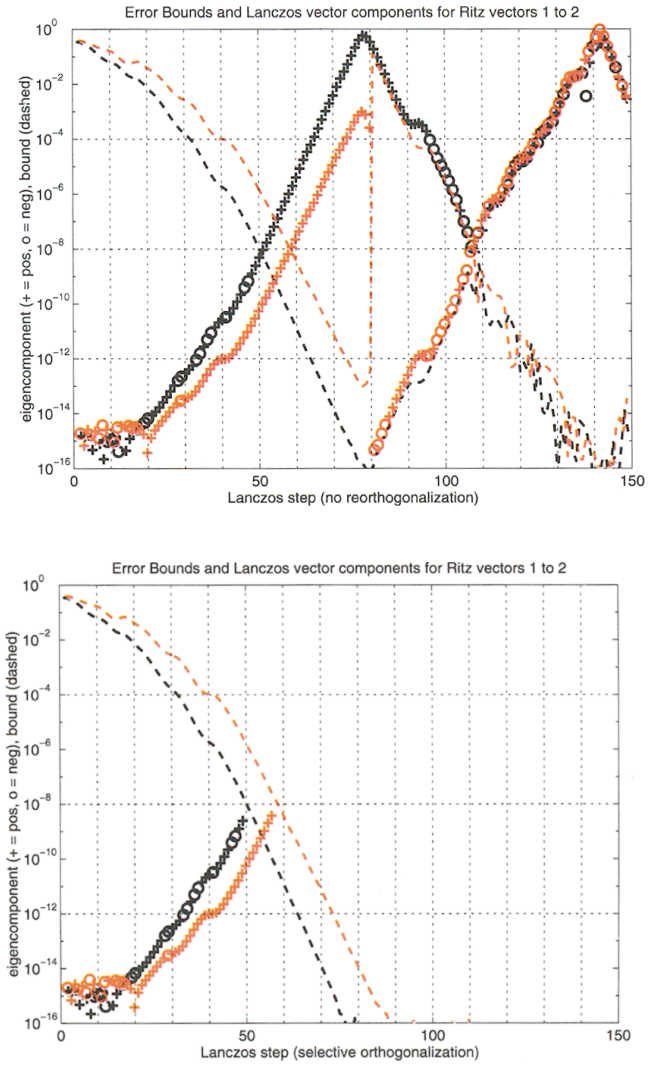


Fig. 7.10. *The Lanczos algorithm with selective orthogonalization applied to A. The top graph shows the first* 149 *steps of the Lanczos algorithm with no reorthogonalization, and the bottom graph shows the Lanczos algorithm with selective orthogonalization. The largest eigenvalue is shown in black, and the second largest eigenvalue is shown in red. The dashed lines are error bounds as before. The lines marked by pluses and o's show* $y_{k,i}^T q_{k+1}$, *the component of Lancos vector* $k+1$ *in the direction of the Ritz vector for the largest Ritz value* ($i = 1$, *in black) or for the second largest Ritz value* ($i = 2$, *in red). Note that selective orthogonalization eliminates these components after the first selective orthogonalizations at steps* 50 ($i = 1$) *and* 58 ($i = 2$).

The first alternative is to use the *Arnoldi algorithm* (Algorithm 6.9). Recall that the Arnoldi algorithm computes an orthogonal basis Q_k of a Krylov subspace $\mathcal{K}_k(A, q_1)$ such that $Q_k^T A Q_k = H_k$ is upper Hessenberg rather than symmetric tridiagonal. The analogue of the Rayleigh–Ritz procedure is again to approximate the eigenvalues of A by the eigenvalues of H_k. Since A is nonsymmetric, its eigenvalues may be complex and/or badly conditioned, so many of the attractive error bounds and monotonic convergence properties enjoyed by the Lanczos algorithm and described in section 7.3 no longer hold. Nonetheless, effective algorithms and implementations exist. Good references include [154, 171, 212, 216, 217, 233] and the book [213]. The latest software is described in [171, 233] and may be found in NETLIB/scalapack/readme.arpack. The Matlab command `eigs` (for "sparse eigenvalues") uses this software.

A second alternative is to use the *nonsymmetric Lanczos algorithm.* This algorithm attempts to reduce A to nonsymmetric tridiagonal form by a nonorthogonal similarity. The hope is that it will be easier to find the eigenvalues of a (sparse!) nonsymmetric tridiagonal matrix than the Hessenberg matrix produced by the Arnoldi algorithm. Unfortunately, the similarity transformations can be quite ill-conditioned, which means that the eigenvalues of the tridiagonal and of the original matrix may greatly differ. In fact, it is not always possible to find an appropriate similarity because of a phenomenon known as "breakdown" [42, 134, 135, 199]. Attempts to repair breakdown by a process called "look-ahead" have been proposed, implemented, and analyzed in [16, 18, 55, 56, 64, 108, 202, 265, 266].

Finally, it is possible to apply subspace iteration (Algorithm 4.3) [19], Davidson's algorithm [216], or the Jacobi–Davidson algorithm [230] to the sparse nonsymmetric eigenproblem.

7.8. References and Other Topics for Chapter 7

In addition to the references in sections 7.6 and 7.7, there are a number of good surveys available on algorithms for sparse eigenvalues problems: see [17, 51, 125, 163, 197, 213, 262]. Parallel implementations are also discussed in [76].

In section 6.2 we discussed the existence of on-line help to choose from among the variety of iterative methods available for solving $Ax = b$. A similar project is underway for eigenproblems and will be incorporated in a future edition of this book.

7.9. Questions for Chapter 7

QUESTION 7.1. *(Easy)* Confirm that running the Arnoldi algorithm (Algorithm 6.9) or the Lanczos algorithm (Algorithm 6.10) on A with starting vector q yields the identical tridiagonal matrices T_k (or Hessenberg matrices H_k) as running on $Q^T A Q$ with starting vector $Q^T q$.

QUESTION 7.2. *(Medium)* Let λ_i be a simple eigenvalue of A. Confirm that if q_1 is orthogonal to the corresponding eigenvector of A, then the eigenvalues of the tridiagonal matrices T_k computed by the Lanczos algorithm in exact arithmetic cannot converge to λ_i in the sense that the largest T_k computed cannot have λ_i as an eigenvalue. Show, by means of a 3-by-3 example, that an eigenvalue of some other T_k can equal λ_i "accidentally."

QUESTION 7.3. *(Medium)* Confirm that no symmetric tridiagonal matrix T_k computed by the Lanczos algorithm can have an exactly multiple eigenvalue. Show that if A has a multiple eigenvalue, then the Lanczos algorithm applied to A must break down before the last step.

Bibliography

[1] R. Agarwal, F. Gustavson, and M. Zubair. Exploiting functional parallelism of POWER2 to design high performance numerical algorithms. *IBM J. Res. Development*, 38:563–576, 1994.

[2] L. Ahlfors. *Complex Analysis.* McGraw-Hill, New York, 1966.

[3] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[4] G. Alefeld and J. Herzberger. *Introduction to Interval Computations.* Academic Press, New York, 1983.

[5] P. R. Amestoy and I. S. Duff. Vectorization of a multiprocessor multifrontal code. *International Journal of Supercomputer Applications*, 3:41–59, 1989.

[6] P. R. Amestoy. Factorization of large unsymmetric sparse matrices based on a multifrontal approach in a multiprocessor environment. Technical Report TH/PA/91/2, CERFACS, Toulouse, France, February 1991. Ph.D. thesis.

[7] A. Anda and H. Park. Fast plane rotations with dynamic scaling. *SIAM J. Matrix Anal. Appl.*, 15:162–174, 1994.

[8] A. Anda and H. Park. Self scaling fast rotations for stiff least squares problems. *Linear Algebra Appl.*, 234:137–162, 1996.

[9] A. Anderson, D. Culler, D. Patterson, and the NOW Team. A case for networks of workstations: NOW. *IEEE Micro*, 15(1):54–64, February 1995.

[10] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide* (*2nd edition*). SIAM, Philadelphia, PA, 1995.

[11] ANSI/IEEE, New York. *IEEE Standard for Binary Floating Point Arithmetic*, Std 754-1985 edition, 1985.

[12] ANSI/IEEE, New York. *IEEE Standard for Radix Independent Floating Point Arithmetic*, Std 854-1987 edition, 1987.

[13] P. Arbenz and G. Golub. On the spectral decomposition of Hermitian matrices modified by row rank perturbations with applications. *SIAM J. Matrix Anal. Appl.*, 9:40–58, 1988.

[14] M. Arioli, J. Demmel, and I. S. Duff. Solving sparse linear systems with sparse backward error. *SIAM J. Matrix Anal. Appl.*, 10:165–190, 1989.

[15] O. Axelsson. *Iterative Solution Methods.* Cambridge University Press, Cambridge, UK, 1994.

[16] Z. Bai. Error analysis of the Lanczos algorithm for the nonsymmetric eigenvalue problem. *Math. Comp.*, 62:209–226, 1994.

[17] Z. Bai. Progress in the numerical solution of the nonsymmetric eigenvalue problem. *J. Numer. Linear Algebra Appl.*, 2:219–234, 1995.

[18] Z. Bai, D. Day, and Q. Ye. ABLE: An adaptive block Lanczos method for non-Hermitian eigenvalue problems. Mathematics Dept. Report 95-04, University of Kentucky, May 1995. Submitted to *SIAM J. Matrix Anal. Appl.*

[19] Z. Bai and G. W. Stewart. SRRIT: A Fortran subroutine to calculate the dominant invariant subspace of a nonsymmetric matrix. Computer Science Dept. Report TR 2908, University of Maryland, April 1992. Available as pub/reports for reports and pub/srrit for programs via anonymous ftp from thales.cs.umd.edu.

[20] D. H. Bailey. Multiprecision translation and execution of Fortran programs. *ACM Trans. Math. Software*, 19:288–319, 1993.

[21] D. H. Bailey. A Fortran-90 based multiprecision system. *ACM Trans. Math. Software*, 21:379–387, 1995.

[22] D. H. Bailey, K. Lee, and H. D. Simon. Using Strassen's algorithm to accelerate the solution of linear systems. *J. Supercomputing*, 4:97–371, 1991.

[23] J. Barnes and P. Hut. A hierarchical $O(n \log n)$ force calculation algorithm. *Nature*, 324:446–449, 1986.

[24] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, V. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia, PA, 1994. Also available electronically at http://www.netlib.org/templates.

[25] S. Batterson. Convergence of the shifted QR algorithm on 3 by 3 normal matrices. *Numer. Math.*, 58:341–352, 1990.

[26] F. L. Bauer. Genauigkeitsfragen bei der Lösung linearer Gleichungssysteme. *Z. Angew. Math. Mech.*, 46:409–421, 1966.

[27] T. Beelen and P. Van Dooren. An improved algorithm for the computation of Kronecker's canonical form of a singular pencil. *Linear Algebra Appl.*, 105:9–65, 1988.

[28] C. Bischof. Incremental condition estimation. *SIAM J. Matrix Anal. Appl.*, 11:312–322, 1990.

[29] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1:11–29, 1992. Software available at http://www.mcs.anl.gov/adifor/.

[30] C. Bischof and G. Quintana-Orti. Computing rank-revealing QR factorizations of dense matrices. Argonne Preprint ANL-MCS-P559-0196, Argonne National Laboratory, Argonne, IL, 1996.

[31] Å. Björck. *Solution of Equations in* $\mathbb{R}^n$, volume 1 of *Handbook of Numerical Analysis*, chapter Least Squares Methods. Elsevier/North Holland, Amsterdam, 1987.

[32] Å. Björck. Least squares methods. Mathematics Department Report, Linköping University, 1991.

[33] Å. Björck. *Numerical Methods for Least Squares Problems.* SIAM, Philadelphia, PA, 1996.

[34] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide.* Software, Environments, and Tools 4. SIAM, Philadelphia, PA, 1997.

[35] J. Blue. A portable FORTRAN program to find the Euclidean norm of a vector. *ACM Trans. Math. Software*, 4:15–23, 1978.

[36] J. H. Bramble, J. E. Pasciak, and A. H. Schatz. The construction of preconditioners for elliptic problems by substructuring, I. *Math. Comp.*, 47:103–134, 1986.

[37] J. H. Bramble, J. E. Pasciak, and A. H. Schatz. An iterative method for elliptic problems on regions partitioned into substructures. *Math. Comp.*, 46:361–369, 1986.

[38] J. H. Bramble, J. E. Pasciak, and A. H. Schatz. The construction of preconditioners for elliptic problems by substructuring, II. *Math. Comp.*, 49:1–16, 1987.

[39] J. H. Bramble, J. E. Pasciak, and A. H. Schatz. The construction of preconditioners for elliptic problems by substructuring, III. *Math. Comp.*, 51:415–430, 1988.

[40] J. H. Bramble, J. E. Pasciak, and A. H. Schatz. The construction of preconditioners for elliptic problems by substructuring, IV. *Math. Comp.*, 53:1–24, 1989.

[41] K. Brenan, S. Campbell, and L. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations.* North Holland, New York, 1989.

[42] C. Brezinski, M. Redivo Zaglia, and H. Sadok. Avoiding breakdown and near-breakdown in Lanczos type algorithms. *Numer. Algorithms*, 1:261–284, 1991.

[43] W. Briggs. *A Multigrid Tutorial.* SIAM, Philadelphia, PA, 1987.

[44] J. Bunch and L. Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Math. Comp.*, 31:163–179, 1977.

[45] J. Bunch, P. Nielsen, and D. Sorensen. Rank-one modification of the symmetric eigenproblem. *Numer. Math.*, 31:31–48, 1978.

[46] B. Buzbee, F. Dorr, J. George, and G. Golub. The direct solution of the discrete Poisson equation on irregular regions. *SIAM J. Numer. Anal.*, 8:722–736, 1971.

[47] B. Buzbee, G. Golub, and C. Nielsen. On direct methods for solving Poisson's equation. *SIAM J. Numer. Anal.*, 7:627–656, 1970.

[48] T. Chan. Rank revealing QR factorizations. *Linear Algebra Appl.*, 88/89:67–82, 1987.

[49] T. Chan and T. Mathew. Domain decomposition algorithms. In A. Iserles, editor, *Acta Numerica, Volume* 3. Cambridge University Press, Cambride, UK, 1994.

[50] S. Chandrasekaran and I. Ipsen. On rank-revealing factorisations. *SIAM J. Matrix Anal. Appl.*, 15:592–622, 1994.

[51] F. Chatelin. *Eigenvalues of Matrices.* Wiley, Chichester, England, 1993. English translation of the original 1988 French edition.

[52] F. Chaitin-Chatelin and V. Frayssé. *Lectures on Finite Precision Computations.* SIAM, Philadelphia, PA, 1996.

[53] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers—Design issues and performance. Computer Science Dept. Technical Report CS-95-283, University of Tennessee, Knoxville, TN, March 1995. (LAPACK Working Note 95.)

[54] J. Coonen. Underflow and the denormalized numbers. *Computer*, 14:75–87, 1981.

[55] J. Cullum, W. Kerner, and R. Willoughby. A generalized nonsymmetric Lanczos procedure. *Comput. Phys. Comm.*, 53:19–48, 1989.

[56] J. Cullum and R. Willoughby. A practical procedure for computing eigenvalues of large sparse nonsymmetric matrices. In J. Cullum and R. Willoughby, editors, *Large Scale Eigenvalue Problems.* North Holland, Amsterdam, 1986. Mathematics Studies Series Vol. 127, Proceedings of the IBM Institute Workshop on Large Scale Eigenvalue Problems, July 8–12, 1985, Oberlech, Austria.

[57] J. Cullum and R. A. Willoughby. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations.* Birkhaüser, Basel, 1985. Vol. 1, Theory, Vol. 2, Program.

[58] J. J. M. Cuppen. The singular value decomposition in product form. *SIAM J. Sci. Statist. Comput.*, 4:216–221, 1983.

[59] J. J. M. Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numer. Math.*, 36:177–195, 1981.

[60] E. Davidson. The iteration calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real symmetric matrices. *J. Comp. Phys.*, 17:87–94, 1975.

[61] P. Davis. *Interpolation and Approximation.* Dover, New York, 1975.

[62] T. A. Davis and I. S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. Technical Report RAL 93-036, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, UK, 1994.

[63] T. A. Davis and I. S. Duff. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. Technical Report TR-95-020, Computer and Information Sciences Department, University of Florida, 1995.

[64] D. Day. *Semi-duality in the two-sided Lanczos algorithm.* Ph.D. thesis, University of California, Berkeley, CA, 1993.

[65] D. Day. How the QR algorithm fails to converge and how to fix it. Technical Report 96-0913J, Sandia National Laboratory, Albuquerque, NM, April 1996.

[66] A. Deichmoller. *Über die Berechnung verallgemeinerter singulärer Werte mittles Jacobi-ähnlicher Verfahren.* Ph.D. thesis, Fernuniversität-Hagen, Hagen, Germany, 1991.

[67] P. Deift, J. Demmel, L.-C. Li, and C. Tomei. The bidiagonal singular values decomposition and Hamiltonian mechanics. *SIAM J. Numer. Anal.*, 28:1463–1516, 1991. (LAPACK Working Note 11.)

[68] P. Deift, T. Nanda, and C. Tomei. ODEs and the symmetric eigenvalue problem. *SIAM J. Numer. Anal.*, 20:1–22, 1983.

[69] J. Demmel. The condition number of equivalence transformations that block diagonalize matrix pencils. *SIAM J. Numer. Anal.*, 20:599–610, 1983.

[70] J. Demmel. Underflow and the reliability of numerical software. *SIAM J. Sci. Statist. Comput.*, 5:887–919, 1984.

[71] J. Demmel. On condition numbers and the distance to the nearest ill-posed problem. *Numer. Math.*, 51:251–289, 1987.

[72] J. Demmel. The componentwise distance to the nearest singular matrix. *SIAM J. Matrix Anal. Appl.*, 13:10–19, 1992.

[73] J. Demmel, I. Dhillon, and H. Ren. On the correctness of some bisection-like parallel eigenvalue algorithms in floating point arithmetic. *Electronic Trans. Numer. Anal.*, 3:116–140, December 1995. (LAPACK Working Note 70.)

[74] J. Demmel and W. Gragg. On computing accurate singular values and eigenvalues of acyclic matrices. *Linear Algebra Appl.*, 185:203–218, 1993.

[75] J. Demmel, M. Gu, S. Eisenstat, I. Slapničar, K. Veselić, and Z. Drmač. Computing the singular value decomposition with high relative accuracy. Technical Report CSD-97-934, Computer Science Division, University of California, Berkeley, CA, February 1997. LAPACK Working Note 119. Submitted to *Linear Algebra Appl.*

[76] J. Demmel, M. Heath, and H. van der Vorst. Parallel numerical linear algebra. In A. Iserles, editor, *Acta Numerica, Volume* 2. Cambridge University Press, Cambridge, UK, 1993.

[77] J. Demmel and N. J. Higham. Stability of block algorithms with fast Level 3 BLAS. *ACM Trans. Math. Software*, 18:274–291, 1992.

[78] J. Demmel and B. Kågström. Accurate solutions of ill-posed problems in control theory. *SIAM J. Matrix Anal. Appl.*, 9:126–145, 1988.

[79] J. Demmel and B. Kågström. The generalized Schur decomposition of an arbitrary pencil $A - \lambda B$: Robust software with error bounds and applications. Parts I and II. *ACM Trans. Math. Software*, 19:160–201, June 1993.

[80] J. Demmel and W. Kahan. Accurate singular values of bidiagonal matrices. *SIAM J. Sci. Statist. Comput.*, 11:873–912, 1990.

[81] J. Demmel and X. Li. Faster numerical algorithms via exception handling. *IEEE Trans. Comput.*, 43:983–992, 1994. (LAPACK Working Note 59.)

[82] J. Demmel and K. Veselić. Jacobi's method is more accurate than QR. *SIAM J. Matrix Anal. Appl.*, 13:1204–1246, 1992. (LAPACK Working Note 15.)

[83] I. S. Dhillon. *A New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem.* Ph.D. thesis, Computer Science Division, University of California, Berkeley, May 1997.

[84] P. Dierckx. *Curve and Surface Fitting with Splines.* Oxford University Press, Oxford, UK, 1993.

[85] J. Dongarra. Performance of various computers using standard linear equations software. Computer Science Dept. Technical Report, University of Tennessee, Knoxville, April 1996. Up-to-date version available at NETLIB/benchmark.

[86] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 16:18–28, 1990.

[87] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 16:1–17, 1990.

[88] J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. Algorithm 656: An extended set of FORTRAN Basic Linear Algebra Subroutines. *ACM Trans. Math. Software*, 14:18–32, 1988.

[89] J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subroutines. *ACM Trans. Math. Software*, 14:1–17, 1988.

[90] J. Dongarra and D. Sorensen. A fully parallel algorithm for the symmetric eigenproblem. *SIAM J. Sci. Statist. Comput.*, 8:139–154, 1987.

[91] C. Douglas. MGNET: Multi-Grid net. http://NA.CS.Yale.EDU/mgnet/www/mgnet.html.

[92] Z. Drmač. *Computing the Singular and the Generalized Singular Values.* Ph.D. thesis, Fernuniversität-Hagen, Hagen, Germany, 1994.

[93] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, London, 1986.

[94] I. S. Duff. Sparse numerical linear algebra: Direct methods and preconditioning. Technical Report RAL-TR-96-047, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, UK, 1996.

[95] I. S. Duff and J. K. Reid. MA47, a Fortran code for direct solution of indefinite sparse symmetric linear systems. Technical Report RAL-95-001, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, UK, 1995.

[96] I. S. Duff and J. K. Reid. The design of MA48, a code for the direct solution of sparse unsymmetric linear systems of equations. *ACM Trans. Math. Software*, 22:187–226, 1996.

[97] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Math. Software*, 9:302–325, 1983.

[98] I. S. Duff and J. A. Scott. The design of a new frontal code for solving sparse unsymmetric systems. *ACM Trans. Math. Software*, 22:30–45, 1996.

[99] A. Edelman. The complete pivoting conjecture for Gaussian elimination is false. *The Mathematica Journal*, 2:58–61, 1992.

[100] A. Edelman and H. Murakami. Polynomial roots from companion matrices. *Math. Comp.*, 64:763–776, 1995.

[101] S. Eisenstat and I. Ipsen. Relative perturbation techniques for singular value problems. *SIAM J. Numer. Anal.*, 32:1972–1988, 1995.

[102] V. Faber and T. Manteuffel. Necessary and sufficient conditions for the existence of a conjugate gradient method. *SIAM J. Numer. Anal.*, 21:315–339, 1984.

[103] D. M. Fenwick, D. J. Foley, W. B. Gist, S. R. VanDoren, and D. Wissel. The AlphaServer 8000 series: High-end server platform development. *Digital Technical Journal*, 7:43–65, 1995.

[104] K. Fernando and B. Parlett. Accurate singular values and differential qd algorithms. *Numer. Math.*, 67:191–229, 1994.

[105] V. Fernando, B. Parlett, and I. Dhillon. A way to find the most redundant equation in a tridiagonal system. Berkeley Mathematics Dept. Preprint, 1995.

[106] H. Flaschka. *Dynamical Systems, Theory and Applications*, volume 38 of Lecture Notes in Physics, chapter Discrete and periodic solutions of some aspects of the inverse method. Springer-Verlag, New York, 1975.

[107] R. Freund, G. Golub, and N. Nachtigal. Iterative solution of linear systems. In A. Iserles, editor, *Acta Numerica* 1992, pages 57–100. Cambridge University Press, Cambridge, UK, 1992.

[108] R. Freund, M. Gutknecht, and N. Nachtigal. An implementation of the look-ahead Lanczos algorithm for non-Hermitian matrices. *SIAM J. Sci. Comput.*, 14:137–158, 1993.

[109] X. Sun, G. Quintana-Orti, and C. Bischof. A blas-3 version of the QR factorization with column pivoting. Argonne Preprint MCS-P551-1295, Argonne National Laboratory, Argonne, IL, 1995.

[110] F. Gantmacher. *The Theory of Matrices, vol.* II *(translation).* Chelsea, New York, 1959.

[111] M. Garey and D. Johnson. *Computers and Intractability.* W. H. Freeman, San Francisco, 1979.

[112] A. George. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.*, 10:345–363, 1973.

[113] A. George, M. Heath, J. Liu, and E. Ng. Solution of sparse positive definite systems on a shared memory multiprocessor. *Internat. J. Parallel Programming*, 15:309–325, 1986.

[114] A. George and J. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[115] A. George and E. Ng. Parallel sparse Gaussian elimination with partial pivoting. *Ann. Oper. Res.*, 22:219–240, 1990.

[116] R. Glowinski, G. Golub, G. Meurant, and J. Periaux, editors. *Domain Decomposition Methods for Partial Differential Equations*, SIAM, Philadelphia, PA, 1988. Proceedings of the First International Symposium on Domain Decomposition Methods for Partial Differential Equations, Paris, France, January 1987.

[117] S. Goedecker. Remark on algorithms to find roots of polynomials. *SIAM J. Sci. Statist. Comp.*, 15:1059–1063, 1994.

[118] I. Gohberg, P. Lancaster, and L. Rodman. *Matrix Polynomials.* Academic Press, New York, 1982.

[119] D. Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23:5–48, 1991.

[120] G. Golub and W. Kahan. Calculating the singular values and pseudo-inverse of a matrix. *SIAM J. Numer. Anal. (Series B)*, 2:205–224, 1965.

[121] G. Golub and C. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, MD, 3rd edition, 1996.

[122] N. Gould. On growth in Gaussian elimination with complete pivoting. *SIAM J. Matrix Anal. Appl.*, 12:354–361, 1991. See also editor's note in *SIAM J. Matrix Anal. Appl.*, 12(3), 1991.

[123] A. Greenbaum and Z. Strakos. Predicting the behavior of finite precision Lanczos and conjugate gradient computations. *SIAM J. Matrix Anal. Appl.*, 13:121–137, 1992.

[124] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73:325–348, 1987.

[125] R. Grimes, J. Lewis, and H. Simon. A shifted block Lanczos algorithm for solving sparse symmetric generalized eigenproblems. *SIAM J. Matrix Anal. Appl.*, 15:228–272, 1994.

[126] M. Gu. *Numerical Linear Algebra Computations.* Ph.D. thesis, Dept. of Computer Science, Yale University, November 1993.

[127] M. Gu and S. Eisenstat. A stable algorithm for the rank-1 modification of the symmetric eigenproblem. Computer Science Dept. Report YALEU/DCS/RR-916, Yale University, September 1992.

[128] M. Gu and S. Eisenstat. An efficient algorithm for computing a rank-revealing QR decomposition. Computer Science Dept. Report YALEU/DCS/RR-967, Yale University, June 1993.

[129] M. Gu and S. C. Eisenstat. A stable and efficient algorithm for the rank-1 modification of the symmetric eigenproblem. *SIAM J. Matrix Anal. Appl.*, 15:1266–1276, 1994. Yale Technical Report YALEU/DCS/RR-916, September 1992.

[130] M. Gu and S. C. Eisenstat. A divide-and-conquer algorithm for the bidiagonal SVD. *SIAM J. Matrix Anal. Appl.*, 16:79–92, 1995.

[131] M. Gu and S. C. Eisenstat. A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem. *SIAM J. Matrix Anal. Appl.*, 16:172–191, 1995.

[132] A. Gupta and V. Kumar. Optimally scalable parallel sparse Cholesky factorization. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 442–447. SIAM, Philadelphia, PA, 1995.

[133] A. Gupta, E. Rothberg, E. Ng, and B. W. Peyton. Parallel sparse Cholesky factorization algorithms for shared-memory multiprocessor systems. In R. Vichnevetsky, D. Knight, and G. Richter, editors, *Advances in Computer Methods for Partial Differential Equations*—VII. IMACS, 1992.

[134] M. Gutknecht. A completed theory of the unsymmetric Lanczos process and related algorithms, Part I. *SIAM J. Matrix Anal. Appl.*, 13:594–639, 1992.

[135] M. Gutknecht. A completed theory of the unsymmetric Lanczos process and related algorithms, Part II. *SIAM J. Matrix Anal. Appl.*, 15:15–58, 1994.

[136] W. Hackbusch. *Iterative Solution of Large Sparse Linear Systems of Equations.* Springer-Verlag, Berlin, 1994.

[137] L. A. Hageman and D. M. Young. *Applied Iterative Methods.* Academic Press, New York, 1981.

[138] W. W. Hager. Condition estimators. *SIAM J. Sci. Statist. Comput.*, 5:311–316, 1984.

[139] P. Halmos. *Finite Dimensional Vector Spaces.* Van Nostrand, New York, 1958.

[140] E. R. Hansen. *Global Optimization Using Interval Analysis.* Marcel Dekker, New York, 1992.

[141] P. C. Hansen. The truncated SVD as a method for regularization. *BIT*, 27:534–553, 1987.

[142] P. C. Hansen. Truncated singular value decomposition solutions to discrete ill-posed problems ill-determined numerical rank. *SIAM J. Sci. Statist. Comput.*, 11:503–518, 1990.

[143] M. T. Heath and P. Raghavan. Performance of a fully parallel sparse solver. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 334–341, IEEE, Los Alamitos, CA, 1994.

[144] M. Hénon. Integrals of the Toda lattice. *Phys. Rev. B*, 9:1421–1423, 1974.

[145] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Natl. Bur. Stand.*, 49:409–436, 1954.

[146] N. J. Higham. A survey of condition number estimation for triangular matrices. *SIAM Rev.*, 29:575–596, 1987.

[147] N. J. Higham. FORTRAN codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation. *ACM Trans. Math. Software*, 14:381–396, 1988.

[148] N. J. Higham. Experience with a matrix norm estimator. *SIAM J. Sci. Statist. Comput.*, 11:804–809, 1990.

[149] N. J. Higham. *Accuracy and Stability of Numerical Algorithms.* SIAM, Philadelphia, PA, 1996.

[150] P. Hong and C. T. Pan. The rank revealing QR and SVD. *Math. Comp.*, 58:575–232, 1992.

[151] X. Hong and H. T. Kung. I/O complexity: The red blue pebble game. In *Proceedings of the 13th Symposium on the Theory of Computing*, pages 326–334. ACM, New York, 1981.

[152] A. K. Jain. *Fundamentals of Digital Image Processing.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[153] E. Jessup and D. Sorensen. A divide and conquer algorithm for computing the singular value decomposition of a matrix. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 61–66, SIAM, Philadelphia, PA, 1989.

[154] Z. Jia. *Some Numerical Methods for Large Unsymmetric Eigenproblems.* Ph.D. thesis, Universität Bielefeld, Bielefeld, Germany, 1994.

[155] W.-D. Webber, J. P. Singh, and A. Gupta. Splash: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20:5–44, 1992.

[156] W. Kahan. Accurate eigenvalues of a symmetric tridiagonal matrix. Computer Science Dept. Technical Report CS41, Stanford University, Stanford, CA, July 1966 (revised June 1968).

[157] W. Kahan. A survey of error analysis. In *Information Processing* 71, pages 1214–1239, North Holland, Amsterdam, 1972.

[158] W. Kahan. The baleful effect of computer benchmarks upon applied mathematics, physics and chemistry. http://HTTP.CS.Berkeley.EDU/~wkahan/ieee754status/baleful.ps, 1995.

[159] W. Kahan. Lecture notes on the status of IEEE standard 754 for binary floating point arithmetic. http://HTTP.CS.Berkeley.EDU/~wkahan/ieee754status/ieee754.ps, 1995.

[160] T. Kailath and A. H. Sayed. Displacement structure: Theory and applications. *SIAM Rev.*, 37:297–386, 1995.

[161] T. Kato. *Perturbation Theory for Linear Operators.* Springer-Verlag, Berlin, 2nd edition, 1980.

[162] R. B. Kearfott. *Rigorous Global Search: Continuous Problems.* Kluwer, Dordrecht, the Netherlands, 1996. See also http://interval.usl.edu/euromath.html.

[163] W. Kerner. Large-scale complex eigenvalue problems. *J. Comput. Phys.*, 85:1–85, 1989.

[164] G. Kolata. Geodesy: Dealing with an enormous computer task. *Science*, 200:421–422, 1978.

[165] S. Krishnan, A. Narkhede, and D. Manocha. BOOLE: A system to compute Boolean combinations of sculptured solids. Computer Science Dept. Technical Report TR95-008, University of North Carolina, Chapel Hill, 1995. http://www.cs.unc.edu/~geom/geom.html.

[166] M. Kruskal. *Dynamical Systems, Theory and Applications*, volume 38 of Lecture Notes in Physics, chapter Nonlinear Wave Equations. Springer-Verlag, New York, 1975.

[167] K. Kundert. Sparse matrix techniques. In A. Ruehli, editor, *Circuit Analysis, Simulation and Design.* North Holland, Amsterdam, 1986.

[168] C. Lawson and R. Hanson. *Solving Least Squares Problems.* Prentice-Hall, Englewood Cliffs, NJ, 1974.

[169] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Software*, 5:308–323, 1979.

[170] P. Lax. Integrals of nonlinear equations of evolution and solitary waves. *Comm. Pure Appl. Math.*, 21:467–490, 1968.

[171] R. Lehoucq. *Analysis and Implementation of an Implicitly Restarted Arnoldi Iteration.* Ph.D. thesis, Rice University, Houston, TX, 1995.

[172] R.-C. Li. Solving secular equations stably and efficiently. Computer Science Dept. Technical Report CS-94-260, University of Tennessee, Knoxville, TN, November 1994. (LAPACK Working Note 89.)

[173] T.-Y. Li and Z. Zeng. Homotopy-determinant algorithm for solving nonsymmetric eigenvalue problems. *Math. Comp.*, 59:483–502, 1992.

[174] T.-Y. Li and Z. Zeng. Laguerre's iteration in solving the symmetric tridiagonal eigenproblem—a revisit. Michigan State University Preprint, 1992.

[175] T.-Y. Li, Z. Zeng, and L. Cong. Solving eigenvalue problems of nonsymmetric matrices with real homotopies. *SIAM J. Numer. Anal.*, 29:229–248, 1992.

[176] T.-Y. Li, H. Zhang, and X.-H. Sun. Parallel homotopy algorithm for symmetric tridiagonal eigenvalue problem. *SIAM J. Sci. Statist. Comput.*, 12:469–487, 1991.

[177] X. Li. *Sparse Gaussian Elimination on High Performance Computers.* Ph.D. thesis, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, September 1996.

[178] S.-S. Lo, B. Phillipe, and A. Sameh. A multiprocessor algorithm for the symmetric eigenproblem. *SIAM J. Sci. Statist. Comput.*, 8:155–165, 1987.

[179] K. Löwner. Über monotone matrixfunctionen. *Math. Z.*, 38:177–216, 1934.

[180] R. Lucas, W. Blank, and J. Tieman. A parallel solution method for large sparse systems of equations. *IEEE Trans. Computer Aided Design*, CAD-6:981–991, 1987.

[181] D. Manocha and J. Demmel. Algorithms for intersecting parametric and algebraic curves i: simple intersections. *ACM Transactions on Graphics*, 13:73–100, 1994.

[182] D. Manocha and J. Demmel. Algorithms for intersecting parametric and algebraic curves ii: Higher order intersections. *Computer Vision, Graphics and Image Processing: Graphical Models and Image Processing*, 57:80–100, 1995.

[183] R. Mathias. Accurate eigensystem computations by Jacobi methods. *SIAM J. Matrix Anal. Appl.*, 16:977–1003, 1996.

[184] The MathWorks, Inc., Natick, MA. *MATLAB Reference Guide*, 1992.

[185] S. McCormick, editor. *Multigrid Methods*, volume 3 of SIAM Frontiers in Applied Mathematics. SIAM, Philadelphia, PA, 1987.

[186] S. McCormick. *Multilevel Adaptive Methods for Partial Differential Equations*, volume 6 of SIAM Frontiers in Applied Mathematics. SIAM, Philadelphia, PA, 1989.

[187] J. Moser. *Dynamical Systems, Theory and Applications*, volume 38 of Lecture Notes in Physics, chapter Finitely many mass points on the line under the influence of an exponential potential—an integrable system. Springer-Verlag, New York, 1975.

[188] J. Moser, editor. *Dynamical Systems, Theory and Applications*, volume 38 of Lecture Notes in Physics. Springer-Verlag, New York, 1975.

[189] A. Netravali and B. Haskell. *Digital Pictures.* Plenum Press, New York, 1988.

[190] A. Neumaier. *Interval Methods for Systems of Equations.* Cambridge University Press, Cambridge, UK, 1990.

[191] E. G. Ng and B. W. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM J. Sci. Statist. Comp.*, 14:1034–1056, 1993.

[192] B. Nour-Omid, B. Parlett, and A. Liu. How to maintain semi-orthogonality among Lanczos vectors. CPAM Technical Report 420, University of California, Berkeley, CA, 1988.

[193] W. Oettli and W. Prager. Compatibility of approximate solution of linear equations with given error bounds for coefficients and right hand sides. *Numer. Math.*, 6:405–409, 1964.

[194] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM J. Numer. Anal.*, 12:617–629, 1975.

[195] V. Pan. How can we speed up matrix multiplication. *SIAM Rev.*, 26:393–416, 1984.

[196] V. Pan and P. Tang. Bounds on singular values revealed by QR factorization. Technical Report MCS-P332-1092, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1992.

[197] B. Parlett. *The Symmetric Eigenvalue Problem.* Prentice Hall, Englewood Cliffs, NJ, 1980.

[198] B. Parlett. Misconvergence in the Lanczos algorithm. In M. G. Cox and S. Hammarling, editors, *Reliable Numerical Computation*, chapter 1. Clarendon Press, Oxford, UK, 1990.

[199] B. Parlett. Reduction to tridiagonal form and minimal realizations. *SIAM J. Matrix Anal. Appl.*, 13:567–593, 1992.

[200] B. Parlett. *Acta Numerica*, The new qd algorithms, pages 459–491. Cambridge University Press, Cambridge, UK, 1995.

[201] B. Parlett. The construction of orthogonal eigenvectors for tight clusters by use of submatrices. Center for Pure and Applied Mathematics PAM-664, University of California, Berkeley, CA, January 1996. Submitted to *SIAM J. Matrix Anal. Appl.*

[202] B. N. Parlett, D. R. Taylor, and Z. A. Liu. A look-ahead Lanczos algorithm for unsymmetric matrices. *Math. Comp.*, 44:105–124, 1985.

[203] B. N. Parlett and I. S. Dhillon. Fernando's solution to Wilkinson's problem: An application of double factorization. *Linear Algebra Appl.*, 1997. To appear.

[204] D. Priest. Algorithms for arbitrary precision floating point arithmetic. In P. Kornerup and D. Matula, editors, *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 132–145, Grenoble, France, June 26–28, 1991. IEEE Computer Society Press, Los Alamitos, CA.

[205] A. Quarteroni, editor. *Domain Decomposition Methods*, AMS, Providence, RI, 1993. Proceedings of the Sixth International Symposium on Domain Decomposition Methods, Como, Italy, 1992.

[206] H. Ren. *On Error Analysis and Implementation of Some Eigenvalue and Singular Value Algorithms.* Ph.D. thesis, University of California at Berkeley, 1996.

[207] E. Rothberg and R. Schreiber. Improved load distribution in parallel sparse Cholesky factorization. In *Supercomputing*, pages 783–792, November 1994.

[208] S. Rump. Bounds for the componentwise distance to the nearest singular matrix. *SIAM J. Matrix Anal. Appl.*, 18:83–103, 1997.

[209] H. Rutishauser. *Lectures on Numerical Mathematics.* Birkhäuser, Basel, 1990.

[210] J. Rutter. A serial implementation of Cuppen's divide and conquer algorithm for the symmetric eigenvalue problem. Mathematics Dept. Master's Thesis, University of California, 1994. Available by anonymous ftp from tr-ftp.cs.berkeley.edu, directory pub/tech-reports/csd/csd-94-799, file all.ps.

[211] Y. Saad. Krylov subspace methods for solving large unsymmetric linear system. *Math. Comp.*, 37:105–126, 1981.

[212] Y. Saad. Numerical solution of large nonsymmetric eigenvalue problems. *Comput. Phys. Comm.*, 53:71–90, 1989.

[213] Y. Saad. *Numerical Methods for Large Eigenvalue Problems.* Manchester University Press, Manchester, UK, 1992.

[214] Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Co., Boston, 1996.

[215] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7:856–869, 1986.

[216] M. Sadkane. Block-Arnoldi and Davidson methods for unsymmetric large eigenvalue problems. *Numer. Math.*, 64:195–211, 1993.

[217] M. Sadkane. A block Arnoldi-Chebyshev method for computing the leading eigenpairs of large sparse unsymmetric matrices. *Numer. Math.*, 64:181–193, 1993.

[218] J. R. Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. Technical Report CMU-CS-96-140, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1996.

[219] M. Shub and S. Smale. Complexity of Bezout's theorem I: Geometric aspects. *J. Amer. Math. Soc.*, 6:459–501, 1993.

[220] M. Shub and S. Smale. Complexity of Bezout's theorem II: Volumes and probabilities. In F. Eyssette and A. Galligo, editors, *Progress in Mathematics, Vol.* 109—*Computational Algebraic Geometry.* Birkhäuser, Basel, 1993.

[221] M. Shub and S. Smale. Complexity of Bezout's theorem III: Condition number and packing. *J. Complexity*, 9:4–14, 1993.

[222] M. Shub and S. Smale. Complexity of Bezout's theorem IV: Probability of success; extensions. Mathematics Department Preprint, University of California, 1993.

[223] SGI Power Challenge. Technical Report, Silicon Graphics, 1995.

[224] H. Simon. The Lanczos algorithm with partial reorthogonalization. *Math. Comp.*, 42:115–142, 1984.

[225] R. D. Skeel. Scaling for numerical stability in Gaussian elimination. *Journal of the ACM*, 26:494–526, 1979.

[226] R. D. Skeel. Iterative refinement implies numerical stability for Gaussian elimination. *Math. Comp.*, 35:817–832, 1980.

[227] R. D. Skeel. Effect of equilibration on residual size for partial pivoting. *SIAM J. Numer. Anal.*, 18:449–454, 1981.

[228] I. Slapničar. *Accurate Symmetric Eigenreduction by a Jacobi Method.* Ph.D. thesis, Fernuniversität-Hagen, Hagen, Germany, 1992.

[229] G. Sleijpen and H. van der Vorst. A Jacobi-Davidson iteration method for linear eigenvalue problems. Dept. of Mathematics Report 856, University of Utrecht, 1994.

[230] G. Sleijpen, A. Booten, D. Fokkema, and H. van der Vorst. Jacobi-Davidson type methods for generalized eigenproblems and polynomial eigenproblems, Part I. Dept. of Mathematics Report 923, University of Utrecht, 1995.

[231] B. Smith. Domain decomposition algorithms for partial differential equations of linear elasticity. Technical Report 517, Department of Computer Science, Courant Institute, September 1990. Ph.D. thesis.

[232] B. Smith, P. Bjorstad, and W. Gropp. *Domain decomposition: Parallel multilevel methods for elliptic partial differential equations.* Cambridge University Press, Cambridge, UK, 1996. Corresponding PETSc software available at http://www.mcs.anl.gov/petsc/petsc.html.

[233] D. Sorensen. Implicit application of polynomial filters in a k-step Arnoldi method. *SIAM J. Matrix Anal. Appl.*, 13:357–385, 1992.

[234] D. Sorensen and P. Tang. On the orthogonality of eigenvectors computed by divide-and-conquer techniques. *SIAM J. Numer. Anal.*, 28:1752–1775, 1991.

[235] G. W. Stewart. *Introduction to Matrix Computations.* Academic Press, New York, 1973.

[236] G. W. Stewart. Rank degeneracy. *SIAM J. Sci. Statist. Comput.*, 5:403–413, 1984.

[237] G. W. Stewart and J.-G. Sun. *Matrix Perturbation Theory.* Academic Press, New York, 1990.

[238] SPARCcenter 2000 architecture and implementation. Sun Microsystems, Inc., November 1993. Technical White Paper.

[239] W. Symes. The QR algorithm for the finite nonperiodic Toda lattice. *Phys. D*, 4:275–280, 1982.

[240] G. Szegö. *Orthogonal Polynomials.* AMS, Providence, RI, 1967.

[241] K.-C. Toh and L. N. Trefethen. Pseudozeros of polynomials and pseudospectra of companion matrices. *Numer. Math.*, 68:403–425, 1994.

[242] L. Trefethen and R. Schreiber. Average case analysis of Gaussian elimination. *SIAM J. Matrix Anal. Appl.*, 11:335–360, 1990.

[243] L. N. Trefethen and D. Bau. *Numerical Linear Algebra.* SIAM, Philadelphia, PA, 1997.

[244] A. Van Der Sluis. Condition numbers and equilibration of matrices. *Numer. Math.*, 14:14–23, 1969.

[245] A. F. van der Stappen, R. H. Bisseling, and J. G. G. van der Vorst. Parallel sparse LU decomposition on a mesh network of transputers. *SIAM J. Matrix Anal. Appl.*, 14:853–879, 1993.

[246] P. Van Dooren. The computation of Kronecker's canonical form of a singular pencil. *Linear Algebra Appl.*, 27:103–141, 1979.

[247] P. Van Dooren. The generalized eigenstructure problem in linear system theory. *IEEE Trans. Automat. Control*, AC-26:111–128, 1981.

[248] C. V. Van Loan. *Computational Frameworks for the Fast Fourier Transform.* SIAM, Philadelphia, 1992.

[249] R. S. Varga. *Matrix Iterative Analysis.* Prentice-Hall, Englewood Cliffs, NJ, 1962.

[250] K. Veselić and I. Slapničar. Floating point perturbations of Hermitian matrices. *Linear Algebra Appl.*, 195:81–116, 1993.

[251] V. Voevodin. The problem of non-self-adjoint generalization of the conjugate gradient method is closed. *Comput. Math. Math. Phys.*, 23:143–144, 1983.

[252] D. Watkins. *Fundamentals of Matrix Computations.* Wiley, Chichester, UK, 1991.

[253] The Cray C90 series. http://www.cray.com/PUBLIC/product-info/C90/. Cray Research, Inc.

[254] The Cray J90 series. http://www.cray.com/PUBLIC/product-info/J90/. Cray Research, Inc.

[255] The Cray T3E series. http://www.cray.com/PUBLIC/product-info/T3E/. Cray Research, Inc.

[256] The IBM SP-2. http://www.rs6000.ibm.com/software/sp_products/sp2.html. IBM.

[257] The Intel Paragon. http://www.ssd.intel.com/homepage.html. Intel.

[258] P.-Å. Wedin. Perturbation theory for pseudoinverses. *BIT*, 13:217–232, 1973.

[259] S. Weisberg. *Applied Linear Regression.* Wiley, Chichester, UK, 2nd edition, 1985.

[260] P. Wesseling. *An Introduction to Multigrid Methods.* Wiley, Chichester, UK, 1992.

[261] J. H. Wilkinson. *Rounding Errors in Algebraic Processes.* Prentice Hall, Englewood Cliffs, NJ, 1963.

[262] J. H. Wilkinson. *The Algebraic Eigenvalue Problem.* Oxford University Press, Oxford, UK, 1965.

[263] S. Winograd and D. Coppersmith. Matrix multiplication via arithmetic progressions. In *Proceedings of the Nineteenth Annual ACM Symposium on the Theory of Computing*, pages 1–6. ACM, New York, 1987.

[264] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, Reading, MA, 1996.

[265] Q. Ye. A convergence analysis for nonsymmetric Lanczos algorithms. *Math. Comp.*, 56:677–691, 1991.

[266] Q. Ye. A breakdown-free variation of the nonsymmetric Lanczos algorithm. *Math. Comp.*, 62:179–207, 1994.

[267] D. Young. *Iterative Solution of Large Linear Systems.* Academic Press, New York, 1971.

[268] H. Yserentant. Old and new convergence proofs for multigrid methods. In A. Iserles, editor, *Acta Numerica* 1993, pages 285–326. Cambridge University Press, Cambrigde, UK, 1993.

[269] Z. Zeng. *Homotopy-Determinant Algorithm for Solving Matrix Eigenvalue Problems and Its Parallelizations.* Ph.D. thesis, Michigan State University, East Lansing, MI, 1991.

[270] Z. Zlatev, J. Waśniewski, P. C. Hansen, and Tz. Ostromsky. PARASPAR: a package for the solution of large linear algebraic equations on parallel computers with shared memory. Technical Report 95-10, Technical University of Denmark, Lyngby, September 1995.

[271] Z. Zlatev. *Computational Methods for General Sparse Matrices.* Kluwer Academic, Dordrecht, Boston, 1991.

Index