Chapter 4
Iterative Methods for Sparse Linear Systems

This chapter contains an overview of several iterative methods for solving the large sparse linear systems that arise from discretizing elliptic equations. Large sparse linear systems arise from many other practical problems, too, of course, and the methods discussed here are useful in other contexts as well. Except when the matrix has very special structure and fast direct methods of the type discussed in Section 3.7 apply, iterative methods are usually the method of choice for large sparse linear systems.

The classical Jacobi, Gauss–Seidel, and successive overrelaxation (SOR) methods are introduced and briefly discussed. The bulk of the chapter, however, concerns more modern methods for solving linear systems that are typically much more effective for large-scale problems: preconditioned conjugate-gradient (CG) methods, Krylov space methods such as generalized minimum residual (GMRES), and multigrid methods.

4.1 Jacobi and Gauss–Seidel

In this section two classical iterative methods, Jacobi and Gauss–Seidel, are introduced to illustrate the main issues. It should be stressed at the beginning that these are poor methods in general which converge very slowly when used as standalone methods, but they have the virtue of being simple to explain. Moreover, these methods are sometimes used as building blocks in more sophisticated methods, e.g., Jacobi may be used as a smoother for the multigrid method, as discussed in Section 4.6.

We again consider the Poisson problem where we have the system of equations (3.10). We can rewrite this equation as

$$u_{ij} = \frac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}) - \frac{h^2}{4} f_{ij}. \tag{4.1}$$

In particular, note that for Laplace's equation (where $f_{ij} \equiv 0$), this simply states that the value of u at each grid point should be the average of its four neighbors. This is the discrete analogue of the well-known fact that a harmonic function has the following property: the value at any point (x, y) is equal to the average value around a closed curve containing the point, in the limit as the curve shrinks to the point. Physically this also makes sense if we

think of the heat equation. Unless the temperature at this point is equal to the average of the temperature at neighboring points, there will be a net flow of heat toward or away from this point.

The equation (4.1) suggests the following iterative method to produce a new estimate $u^{[k+1]}$ from a current guess $u^{[k]}$:

$$u_{ij}^{[k+1]} = \frac{1}{4}\left(u_{i-1,j}^{[k]} + u_{i+1,j}^{[k]} + u_{i,j-1}^{[k]} + u_{i,j+1}^{[k]}\right) - \frac{h^2}{4} f_{ij}. \tag{4.2}$$

This is the *Jacobi* iteration for the Poisson problem, and it can be shown that for this particular problem it converges from any initial guess $u^{[0]}$ (although very slowly).

Here is a short section of MATLAB code that implements the main part of this iteration:

```
for iter=0:maxiter
   for j=2:(m+1)
      for i=2:(m+1)
         unew(i,j) = 0.25*(u(i-1,j) + u(i+1,j) + ...
                     u(i,j-1) + u(i,j+1) - h^2 * f(i,j));
         end
      end
   u = unew;
   end
```

Here it is assumed that `u` initially contains the guess $u^{[0]}$ and that boundary data are stored in `u(1,:), u(m+2,:), u(:,1), and u(:,m+2)`. The indexing is off by 1 from what might be expected since MATLAB begins arrays with index 1, not 0.

Note that one might be tempted to dispense with the variable `unew` and replace the above code with

```
for iter=0:maxiter
   for j=2:(m+1)
      for i=2:(m+1)
         u(i,j) = 0.25*(u(i-1,j) + u(i+1,j) + ...
                    u(i,j-1) + u(i,j+1) - h^2 * f(i,j));
         end
      end
   end
```

This would not give the same results, however. In the correct code for Jacobi we compute new values of `u` based entirely on old data from the previous iteration, as required from (4.2). In the second code we have already updated `u(i-1,j)` and `u(i,j-1)` before updating `u(i,j)`, and these new values will be used instead of the old ones. The latter code thus corresponds to the method

$$u_{ij}^{[k+1]} = \frac{1}{4}\left(u_{i-1,j}^{[k+1]} + u_{i+1,j}^{[k]} + u_{i,j-1}^{[k+1]} + u_{i,j+1}^{[k]}\right) - \frac{h^2}{4} f_{ij}. \tag{4.3}$$

This is what is known as the *Gauss–Seidel* method, and it would be a lucky coding error since this method generally converges about twice as fast as Jacobi does. The Jacobi

method is sometimes called the *method of simultaneous displacements*, while Gauss–Seidel is known as the *method of successive displacements*. Later we'll see that Gauss–Seidel can be improved by using SOR.

Note that if one actually wants to implement Jacobi in MATLAB, looping over i and j is quite slow and it is much better to write the code in vectorized form, e.g.,

```
I = 2:(m+1);
J = 2:(m+1);
for iter=0:maxiter
   u(I,J) = 0.25*(u(I-1,J) + u(I+1,J) + u(I,J-1) ...
                          + u(I,J+1) - h^2 * f(I,J));
   end
```

It is somewhat harder to implement Gauss–Seidel in vectorized form.

Convergence of these methods will be discussed in Section 4.2. First we note some important features of these iterative methods:

- The matrix A is never stored. In fact, for this simple constant coefficient problem, we don't even store all the $5m^2$ nonzeros which all have the value $1/h^2$ or $-4/h^2$. The values 0.25 and h^2 in the code are the only values that are "stored." (For a variable coefficient problem where the coefficients are different at each point, we would in general have to store all the nonzeros.)

- Hence the storage is optimal—essentially only the m^2 solution values are stored in the Gauss–Seidel method. The above code for Jacobi uses $2m^2$ since `unew` is stored as well as `u`, but one could eliminate most of this with more careful coding.

- Each iteration requires $O(m^2)$ work. The total work required will depend on how many iterations are required to reach the desired level of accuracy. We will see that with these particular methods we require $O(m^2 \log m)$ iterations to reach a level of accuracy consistent with the expected global error in the solution (as $h \to 0$ we should require more accuracy in the solution to the linear system). Combining this with the work per iteration gives a total operation count of $O(m^4 \log m)$. This looks worse than Gaussian elimination with a banded solver, although since $\log m$ grows so slowly with m it is not clear which is really more expensive for a realistic-size matrix. (And the iterative method definitely saves on storage.)

Other iterative methods also typically require $O(m^2)$ work per iteration but may converge much faster and hence result in less overall work. The ideal would be to converge in a number of iterations that is independent of h so that the total work is simply $O(m^2)$. Multigrid methods (see Section 4.6) can achieve this, not only for Poisson's problem but also for many other elliptic equations.

4.2 Analysis of matrix splitting methods

In this section we study the convergence of the Jacobi and Gauss–Seidel methods. As a simple example we will consider the one-dimensional analogue of the Poisson problem, $u''(x) = f(x)$ as discussed in Chapter 2. Then we have a tridiagonal system of equations

(2.9) to solve. In practice we would never use an iterative method for this system, since it can be solved directly by Gaussian elimination in $O(m)$ operations, but it is easier to illustrate the iterative methods in the one-dimensional case, and all the analysis done here carries over almost unchanged to the two-dimensional and three-dimensional cases.

The Jacobi and Gauss–Seidel methods for this problem take the form

$$\text{Jacobi} \qquad u_i^{[k+1]} = \frac{1}{2}\left(u_{i-1}^{[k]} + u_{i+1}^{[k]} - h^2 f_i\right), \tag{4.4}$$

$$\text{Gauss–Seidel} \qquad u_i^{[k+1]} = \frac{1}{2}\left(u_{i-1}^{[k+1]} + u_{i+1}^{[k]} - h^2 f_i\right). \tag{4.5}$$

Both methods can be analyzed by viewing them as based on a splitting of the matrix A into

$$A = M - N, \tag{4.6}$$

where M and N are two $m \times m$ matrices. Then the system $Au = f$ can be written as

$$Mu - Nu = f \quad \Longrightarrow \quad Mu = Nu + f,$$

which suggests the iterative method

$$Mu^{[k+1]} = Nu^{[k]} + f. \tag{4.7}$$

In each iteration we assume $u^{[k]}$ is known and we obtain $u^{[k+1]}$ by solving a linear system with the matrix M. The basic idea is to define the splitting so that M contains as much of A as possible (in some sense) while keeping its structure sufficiently simple that the system (4.7) is much easier to solve than the original system with the full A. Since systems involving diagonal, lower, or upper triangular matrices are relatively simple to solve, there are some obvious choices for the matrix M. To discuss these in a unified framework, write

$$A = D - L - U \tag{4.8}$$

in general, where D is the diagonal of A, $-L$ is the strictly lower triangular part, and $-U$ is the strictly upper triangular part. For example, the tridiagonal matrix (2.10) would give

$$D = \frac{1}{h^2}\begin{bmatrix} -2 & 0 & & & & \\ 0 & -2 & 0 & & & \\ & 0 & -2 & 0 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 0 & -2 & 0 \\ & & & & 0 & -2 \end{bmatrix}, \quad L = -\frac{1}{h^2}\begin{bmatrix} 0 & 0 & & & & \\ 1 & 0 & 0 & & & \\ & 1 & 0 & 0 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & 0 & 0 \\ & & & & 1 & 0 \end{bmatrix}$$

with $-U = -L^T$ being the remainder of A.

In the Jacobi method, we simply take M to be the diagonal part of A, $M = D$, so that

$$M = -\frac{2}{h^2}I, \qquad N = L + U = D - A = -\frac{1}{h^2}\begin{bmatrix} 0 & 1 & & & & \\ 1 & 0 & 1 & & & \\ & 1 & 0 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & 0 & 1 \\ & & & & 1 & 0 \end{bmatrix}.$$

The system (4.7) is then diagonal and extremely easy to solve:

$$u^{[k+1]} = \frac{1}{2}\begin{bmatrix} 0 & 1 & & & & \\ 1 & 0 & 1 & & & \\ & 1 & 0 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & 0 & 1 \\ & & & & 1 & 0 \end{bmatrix} u^{[k]} - \frac{h^2}{2} f, \tag{4.9}$$

which agrees with (4.4).

In Gauss–Seidel, we take M to be the full lower triangular portion of A, so $M = D - L$ and $N = U$. The system (4.7) is then solved using forward substitution, which results in (4.5).

To analyze these methods, we derive from (4.7) the update formula

$$\begin{aligned} u^{[k+1]} &= M^{-1}Nu^{[k]} + M^{-1}f \\ &\equiv Gu^{[k]} + c, \end{aligned} \tag{4.10}$$

where $G = M^{-1}N$ is the *iteration matrix* and $c = M^{-1}f$.

Let u^* represent the true solution to the system $Au = f$. Then

$$u^* = Gu^* + c. \tag{4.11}$$

This shows that the true solution is a fixed point, or equilibrium, of the iteration (4.10), i.e., if $u^{[k]} = u^*$, then $u^{[k+1]} = u^*$ as well. However, it is not clear that this is a *stable equilibrium*, i.e., that we would converge toward u^* if we start from some incorrect initial guess.

If $e^{[k]} = u^{[k]} - u^*$ represents the error, then subtracting (4.11) from (4.10) gives

$$e^{[k+1]} = Ge^{[k]},$$

and so after k steps we have

$$e^{[k]} = G^k e^{[0]}. \tag{4.12}$$

From this we can see that the method will converge from any initial guess $u^{[0]}$, provided $G^k \to 0$ (an $m \times m$ matrix of zeros) as $k \to \infty$. When is this true?

For simplicity, assume that G is a diagonalizable matrix, so that we can write

$$G = R\Gamma R^{-1},$$

where R is the matrix of right eigenvectors of G and Γ is a diagonal matrix of eigenvalues $\gamma_1, \gamma_2, \ldots, \gamma_m$. Then

$$G^k = R\Gamma^k R^{-1}, \tag{4.13}$$

where

$$\Gamma^k = \begin{bmatrix} \gamma_1^k & & & \\ & \gamma_2^k & & \\ & & \ddots & \\ & & & \gamma_m^k \end{bmatrix}.$$

Clearly the method converges if $|\gamma_p| < 1$ for all $p = 1, 2, \ldots, m$, i.e., if $\rho(G) < 1$, where ρ is the spectral radius. See Appendix D for a more general discussion of the asymptotic properties of matrix powers.

4.2.1 Rate of convergence

From (4.12) we can also determine how rapidly the method can be expected to converge in cases where it is convergent. Using (4.13) in (4.12) and using the 2-norm, we obtain

$$\|e^{[k]}\|_2 \le \|\Gamma^k\|_2 \|R\|_2 \|R^{-1}\|_2 \|e^{[0]}\|_2 = \rho^k \kappa_2(R) \|e^{[0]}\|_2, \tag{4.14}$$

where $\rho \equiv \rho(G)$, and $\kappa_2(R) = \|R\|_2 \|R^{-1}\|_2$ is the condition number of the eigenvector matrix.

If the matrix G is a normal matrix (see Section C.4), then the eigenvectors are orthogonal and $\kappa_2(R) = 1$. In this case we have

$$\|e^{[k]}\|_2 \le \rho^k \|e^{[0]}\|_2. \tag{4.15}$$

If G is nonnormal, then the spectral radius of G gives information about the asymptotic rate of convergence as $k \to \infty$ but may not give a good indication of the behavior of the error for small k. See Section D.4 for more discussion of powers of nonnormal matrices and see Chapters 24–27 of [92] for some discussion of iterative methods on highly nonnormal problems.

Note: These methods are *linearly* convergent, in the sense that $\|e^{[k+1]}\| \le \rho \|e^{[k]}\|$ and it is the first power of $\|e^{[k]}\|$ that appears on the right. Recall that Newton's method is typically quadratically convergent, and it is the square of the previous error that appears on the right-hand side. But Newton's method is for a nonlinear problem and requires solving a linear system in each iteration. Here we are looking at solving such a linear system.

Example 4.1. For the Jacobi method we have

$$G = D^{-1}(D - A) = I - D^{-1}A.$$

If we apply this method to the boundary value problem $u'' = f$, then

$$G = I + \frac{h^2}{2}A.$$

The eigenvectors of this matrix are the same as the eigenvectors of A, and the eigenvalues are hence

$$\gamma_p = 1 + \frac{h^2}{2}\lambda_p,$$

where λ_p is given by (2.23). So

$$\gamma_p = \cos(p\pi h), \quad p = 1, 2, \ldots, m,$$

where $h = 1/(m+1)$. The spectral radius is

$$\rho(G) = |\gamma_1| = \cos(\pi h) \approx 1 - \frac{1}{2}\pi^2 h^2 + O(h^4). \tag{4.16}$$

The spectral radius is less than 1 for any $h > 0$ and the Jacobi method converges. Moreover, the G matrix for Jacobi is symmetric as seen in (4.9), and so (4.15) holds and the error is monotonically decreasing at a rate given precisely by the spectral radius. Unfortunately, though, for small h this value is very close to 1, resulting in very slow convergence.

How many iterations are required to obtain a good solution? Suppose we want to reduce the error to $\|e^{[k]}\| \approx \epsilon \|e^{[0]}\|$ (where typically $\|e^{[0]}\|$ is on the order of 1).[1] Then we want $\rho^k \approx \epsilon$ and so

$$k \approx \log(\epsilon)/\log(\rho). \tag{4.17}$$

How small should we choose ϵ? To get full machine precision we might choose ϵ to be close to the machine round-off level. However, this typically would be very wasteful. For one thing, we rarely need this many correct digits. More important, however, we should keep in mind that even the *exact* solution u^* of the linear system $Au = f$ is only an *approximate* solution of the differential equation we are actually solving. If we are using a second order accurate method, as in this example, then u_i^* differs from $u(x_i)$ by something on the order of h^2 and so we cannot achieve better accuracy than this no matter how well we solve the linear system. In practice we should thus take ϵ to be something related to the expected global error in the solution, e.g., $\epsilon = Ch^2$ for some fixed C.

To estimate the order of work required asymptotically as $h \to 0$, we see that the above choice gives

$$k = (\log(C) + 2\log(h))/\log(\rho). \tag{4.18}$$

For Jacobi on the boundary value problem we have $\rho \approx 1 - \frac{1}{2}\pi^2 h^2$ and hence $\log(\rho) \approx -\frac{1}{2}\pi^2 h^2$. Since $h = 1/(m+1)$, using this in (4.18) gives

$$k = O(m^2 \log m) \quad \text{as } m \to \infty. \tag{4.19}$$

Since each iteration requires $O(m)$ work in this one-dimensional problem, the total work required to solve the problem is

$$\text{total work} = O(m^3 \log m).$$

Of course this tridiagonal problem can be solved exactly in $O(m)$ work, so we would be foolish to use an iterative method at all here!

For a Poisson problem in two or three dimensions it can be verified that (4.19) still holds, although now the work required per iteration is $O(m^2)$ or $O(m^3)$, respectively, if there are m grid points in each direction. In two dimensions we would thus find that

$$\text{total work} = O(m^4 \log m). \tag{4.20}$$

Recall from Section 3.7 that Gaussian elimination on the banded matrix requires $O(m^4)$ operations, while other direct methods can do much better, so Jacobi is still not competitive. Luckily there are much better iterative methods.

[1] Assuming we are using some grid function norm, as discussed in Appendix A. Note that for the 2-norm in one dimension this requires introducing a factor of $\sqrt{h}$ in the definitions of both $\|e^{[k]}\|$ and $\|e^{[0]}\|$, but these factors cancel out in choosing an appropriate ϵ.

For the Gauss–Seidel method applied to the Poisson problem in any number of space dimensions, it can be shown that

$$\rho(G) = 1 - \pi^2 h^2 + O(h^4) \quad \text{as } h \to 0. \tag{4.21}$$

This still approaches 1 as $h \to 0$, but it is better than (4.16) by a factor of 2, and the number of iterations required to reach a given tolerance typically will be half the number required with Jacobi. The order of magnitude figure (4.20) still holds, however, and this method also is not widely used.

4.2.2 Successive overrelaxation

If we look at how iterates $u^{[k]}$ behave when Gauss–Seidel is applied to a typical problem, we will often see that $u_i^{[k+1]}$ is closer to u_i^* than $u_i^{[k]}$ was, but only by a little bit. The Gauss–Seidel update moves u_i in the right direction but is far too conservative in the amount it allows u_i to move. This suggests that we use the following two-stage update, illustrated again for the problem $u'' = f$:

$$\begin{aligned} u_i^{\mathrm{GS}} &= \frac{1}{2}\left(u_{i-1}^{[k+1]} + u_{i+1}^{[k]} - h^2 f_i\right), \\ u_i^{[k+1]} &= u_i^{[k]} + \omega\left(u_i^{\mathrm{GS}} - u_i^{[k]}\right), \end{aligned} \tag{4.22}$$

where ω is some scalar parameter. If $\omega = 1$, then $u_i^{[k+1]} = u_i^{\mathrm{GS}}$ is the Gauss–Seidel update. If $\omega > 1$, then we move farther than Gauss–Seidel suggests. In this case the method is known as *successive overrelaxation* (SOR).

If $\omega < 1$, then we would be underrelaxing, rather than overrelaxing. This would be even less effective than Gauss–Seidel as a standalone iterative method for most problems, although underrelaxation is sometimes used in connection with multigrid methods (see Section 4.6).

The formulas in (4.22) can be combined to yield

$$u_i^{[k+1]} = \frac{\omega}{2}\left(u_{i-1}^{[k+1]} + u_{i+1}^{[k]} - h^2 f_i\right) + (1-\omega)u_i^{[k]}. \tag{4.23}$$

For a general system $Au = f$ with $A = D - L - U$ it can be shown that SOR with forward sweeps corresponds to a matrix splitting method of the form (4.7) with

$$M = \frac{1}{\omega}(D - \omega L), \qquad N = \frac{1}{\omega}((1-\omega)D + \omega U). \tag{4.24}$$

Analyzing this method is considerably trickier than with the Jacobi or Gauss–Seidel methods because of the form of these matrices. A theorem of Ostrowski states that if A is symmetric positive definite (SPD) and $D - \omega L$ is nonsingular, then the SOR method converges for all $0 < \omega < 2$. Young [105] showed how to find the optimal ω to obtain the most rapid convergence for a wide class of problems (including the Poisson problem). This elegant theory can be found in many introductory texts. (For example, see [37], [42], [96], [106]. See also [67] for a different introductory treatment based on Fourier series

and modified equations in the sense of Section 10.9, and see [3] for applications of this approach to the 9-point Laplacian.)

For the Poisson problem in any number of space dimensions it can be shown that the SOR method converges most rapidly if ω is chosen as

$$\omega_{\text{opt}} = \frac{2}{1 + \sin(\pi h)} \approx 2 - 2\pi h.$$

This is nearly equal to 2 for small h. One might be tempted to simply set $\omega = 2$ in general, but this would be a poor choice since SOR does not then converge! In fact the convergence rate is quite sensitive to the value of ω chosen. With the optimal ω it can be shown that the spectral radius of the corresponding G matrix is

$$\rho_{\text{opt}} = \omega_{\text{opt}} - 1 \approx 1 - 2\pi h,$$

but if ω is changed slightly this can deteriorate substantially.

Even with the optimal ω we see that $\rho_{\text{opt}} \to 1$ as $h \to 0$, but only linearly in h rather than quadratically as with Jacobi or Gauss–Seidel. This makes a substantial difference in practice. The expected number of iterations to converge to the required $O(h^2)$ level, the analogue of (4.19), is now

$$k_{opt} = O(m \log m).$$

Figure 4.1 shows some computational results for the methods described above on the two-point boundary value problem $u'' = f$. The SOR method with optimal ω is

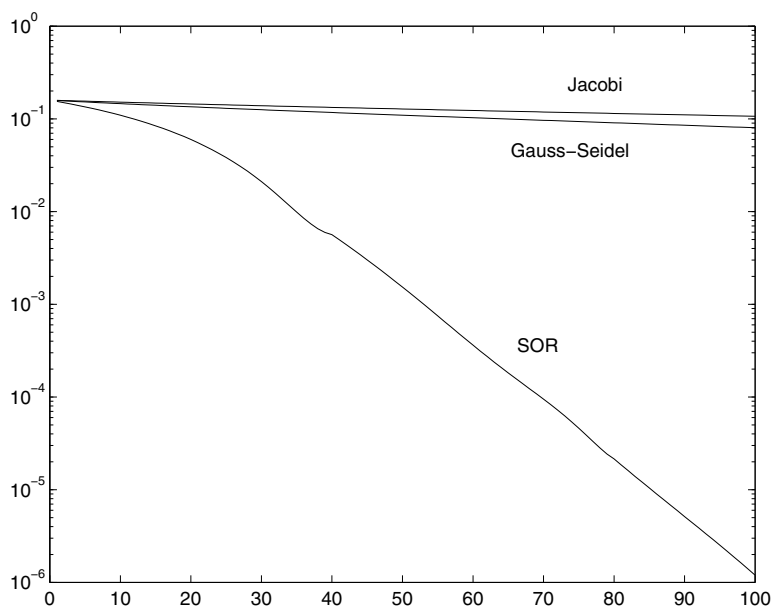

Figure 4.1. *Errors versus k for three methods.*

far superior to Gauss–Seidel or Jacobi, at least for this simple problem with a symmetric coefficient matrix. For more complicated problems it can be difficult to estimate the optimal ω, however, and other approaches are usually preferred.

4.3 Descent methods and conjugate gradients

The CG method is a powerful technique for solving linear systems $Au = f$ when the matrix A is SPD, or negative definite since negating the system then gives an SPD matrix. This may seem like a severe restriction, but SPD methods arise naturally in many applications, such as the discretization of elliptic equations. There are several ways to introduce the CG method and the reader may wish to consult texts such as [39], [79], [91] for other approaches and more analysis. Here the method is first motivated as a *descent method* for solving a *minimization problem*.

Consider the function $\phi : \mathbb{R}^m \to \mathbb{R}$ defined by

$$\phi(u) = \frac{1}{2}u^T Au - u^T f. \tag{4.25}$$

This is a quadratic function of the variables $u_1, \ldots, u_m$. For example, if $m = 2$, then

$$\phi(u) = \phi(u_1, u_2) = \frac{1}{2}(a_{11}u_1^2 + 2a_{12}u_1u_2 + a_{22}u_2^2) - u_1 f_1 - u_2 f_2.$$

Note that since A is symmetric, $a_{21} = a_{12}$. If A is positive definite, then plotting $\phi(u)$ as a function of u_1 and u_2 gives a parabolic bowl as shown in Figure 4.2(a). There is a unique value u^* that minimizes $\phi(u)$ over all choices of u. At the minimum, the partial derivative of ϕ with respect to each component of u is zero, which gives the equations

$$\begin{aligned} \frac{\partial \phi}{\partial u_1} &= a_{11}u_1 + a_{12}u_2 - f_1 = 0, \\ \frac{\partial \phi}{\partial u_2} &= a_{21}u_1 + a_{22}u_2 - f_2 = 0. \end{aligned} \tag{4.26}$$

This is exactly the linear system $Au = f$ that we wish to solve. So finding u^* that solves this system can equivalently be approached as finding u^* to minimize $\phi(u)$. This is true more generally when $u \in \mathbb{R}^m$ and $A \in \mathbb{R}^{m\times m}$ is SPD. The function $\phi(u)$ in (4.25) has a unique minimum at the point u^*, where $\nabla\phi(u^*) = 0$, and

$$\nabla\phi(u) = Au - f, \tag{4.27}$$

so the minimizer solves the linear system $Au = f$.

If A is negative definite, then $\phi(u)$ instead has a unique maximum at u^*, which again solves the linear system. If A is *indefinite* (neither positive nor negative definite), i.e., if the eigenvalues of A are not all of the same sign, then the function $\phi(u)$ still has a stationary point with $\nabla\phi(u^*) = 0$ at the solution to $Au = f$, but this is a saddle point rather than a minimum or maximum, as illustrated in Figure 4.2(b). It is much harder to find a saddle point than a minimum. An iterative method can find a minimum by always heading downhill, but if we are looking for a saddle point, it is hard to tell if we need to

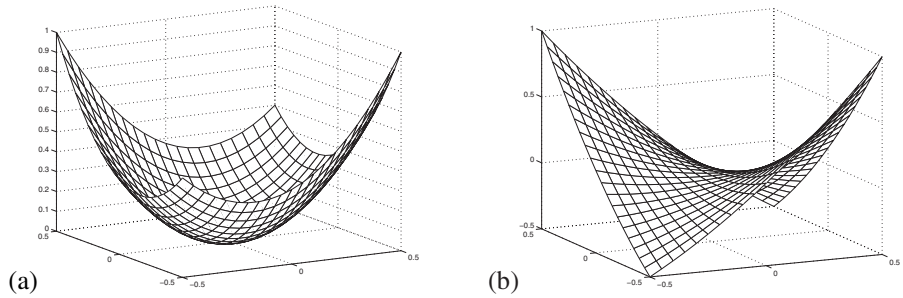

Figure 4.2. (a) *The function $\phi(u)$ for $m = 2$ in a case where A is symmetric and positive definite.* (b) *The function $\phi(u)$ for $m = 2$ in a case where A is symmetric but indefinite.*

head uphill or downhill from the current approximation. Since the CG method is based on minimization, it is necessary for the matrix to be SPD. By viewing CG in a different way it is possible to generalize it and obtain methods that also work on indefinite problems, such as the GMRES algorithm described in Section 4.4.

4.3.1 The method of steepest descent

As a prelude to studying CG, we first review the method of steepest descent for minimizing $\phi(u)$. As in all iterative methods we start with an initial guess u_0 and iterate to obtain $u_1, u_2, \ldots$. For notational convenience we now use subscripts to denote the iteration number: u_k instead of $u^{[k]}$. This is potentially confusing since normally we use subscripts to denote components of the vector, but the formulas below get too messy otherwise and we will not need to refer to the components of the vector in the rest of this chapter.

From one estimate u_{k-1} to u^* we wish to obtain a better estimate u_k by moving downhill, based on values of $\phi(u)$. It seems sensible to move in the direction in which ϕ is decreasing most rapidly, and go in this direction for as far as we can before $\phi(u)$ starts to increase again. This is easy to implement, since the gradient vector $\nabla\phi(u)$ always points in the direction of most rapid increase of ϕ. So we want to set

$$u_k = u_{k-1} - \alpha_{k-1}\nabla\phi(u_{k-1}) \tag{4.28}$$

for some scalar α_{k-1}, chosen to solve the minimization problem

$$\min_{\alpha\in\mathbb{R}} \phi\left(u_{k-1} - \alpha\nabla\phi(u_{k-1})\right). \tag{4.29}$$

We expect $\alpha_{k-1} \geq 0$ and $\alpha_{k-1} = 0$ only if we are already at the minimum of ϕ, i.e., only if $u_{k-1} = u^*$.

For the function $\phi(u)$ in (4.25), the gradient is given by (4.27) and so

$$\nabla\phi(u_{k-1}) = Au_{k-1} - f \equiv -r_{k-1}, \tag{4.30}$$

where $r_{k-1} = f - Au_{k-1}$ is the *residual vector* based on the current approximation u_{k-1}. To solve the minimization problem (4.29), we compute the derivative with respect to α and set this to zero. Note that

$$\phi(u+\alpha r) = \left(\frac{1}{2}u^T Au - u^T f\right) + \alpha(r^T Au - r^T f) + \frac{1}{2}\alpha^2 r^T Ar \tag{4.31}$$

and so

$$\frac{d\phi(u+\alpha r)}{d\alpha} = r^T Au - r^T f + \alpha r^T Ar.$$

Setting this to zero and solving for α gives

$$\alpha = \frac{r^T r}{r^T Ar}. \tag{4.32}$$

The steepest descent algorithm thus takes the form

```
choose a guess u_0
for k = 1, 2, ...
    r_{k-1} = f - A u_{k-1}
    if ||r_{k-1}|| is less than some tolerance then stop
    α_{k-1} = (r_{k-1}^T r_{k-1})/(r_{k-1}^T A r_{k-1})
    u_k = u_{k-1} + α_{k-1} r_{k-1}
    end
```

Note that implementing this algorithm requires only that we be able to multiply a vector by A, as with the other iterative methods discussed earlier. We do not need to store the matrix A, and if A is very sparse, then this multiplication can be done quickly.

It appears that in each iteration we must do two matrix-vector multiplies, Au_{k-1} to compute r_{k-1} and then Ar_{k-1} to compute α_{k-1}. However, note that

$$\begin{aligned} r_k &= f - Au_k \\ &= f - A(u_{k-1} + \alpha_{k-1} r_{k-1}) \\ &= r_{k-1} - \alpha_{k-1} Ar_{k-1}. \end{aligned} \tag{4.33}$$

So once we have computed Ar_{k-1} as needed for α_{k-1}, we can also use this result to compute r_k. A better way to organize the computation is thus:

```
choose a guess u_0
r_0 = f - A u_0
for k = 1, 2, ...
    w_{k-1} = A r_{k-1}
    α_{k-1} = (r_{k-1}^T r_{k-1})/(r_{k-1}^T w_{k-1})
    u_k = u_{k-1} + α_{k-1} r_{k-1}
    r_k = r_{k-1} - α_{k-1} w_{k-1}
    if ||r_k|| is less than some tolerance then stop
    end
```

Figure 4.3 shows how this iteration proceeds for a typical case with $m = 2$. This figure shows a contour plot of the function $\phi(u)$ in the u_1-u_2 plane (where u_1 and u_2 mean the components of u here), along with several iterates u_n of the steepest descent algorithm. Note that the gradient vector is always orthogonal to the contour lines. We move along the direction of the gradient (the "search direction" for this algorithm) to the

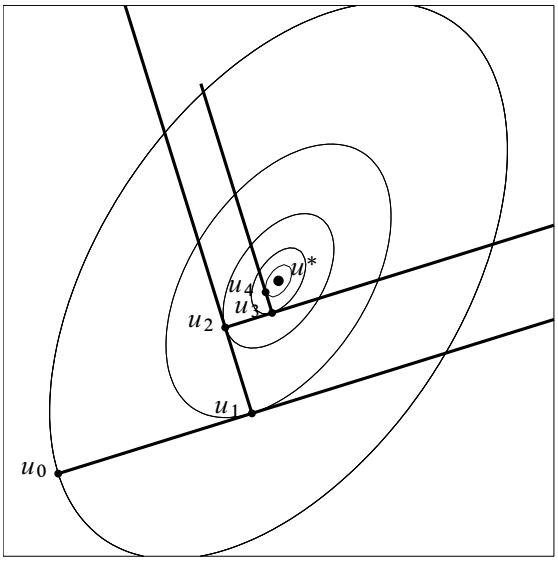

Figure 4.3. *Several iterates of the method of steepest descent in the case $m = 2$. The concentric ellipses are level sets of $\phi(u)$.*

point where $\phi(u)$ is minimized along this line. This will occur at the point where this line is tangent to a contour line. Consequently, the next search direction will be orthogonal to the current search direction, and in two dimensions we simply alternate between only two search directions. (Which particular directions depend on the location of u_0.)

If A is SPD, then the contour lines (level sets of ϕ) are always ellipses. How rapidly this algorithm converges depends on the geometry of these ellipses and on the particular starting vector u_0 chosen. Figure 4.4(a) shows the best possible case, where the ellipses are circles. In this case the iterates converge in one step from any starting guess, since the first search direction r_0 generates a line that always passes through the minimum u^* from any point.

Figure 4.4(b) shows a bad case, where the ellipses are long and skinny and the iteration slowly traverses back and forth in this shallow valley searching for the minimum. In general steepest descent is a slow algorithm, particularly when m is large, and should not be used in practice. Shortly we will see a way to improve this algorithm dramatically.

The geometry of the level sets of $\phi(u)$ is closely related to the eigenstructure of the matrix A. In the case $m = 2$ as shown in Figures 4.3 and 4.4, each ellipse can be characterized by a major and minor axis, as shown in Figure 4.5 for a typical level set. The points v_1 and v_2 have the property that the gradient $\nabla\phi(v_j)$ lies in the direction that connects v_j to the center u^*, i.e.,

$$Av_j - f = \lambda_j(v_j - u^*) \tag{4.34}$$

for some scalar λ_j. Since $f = Au^*$, this gives

$$A(v_j - u^*) = \lambda_j(v_j - u^*) \tag{4.35}$$

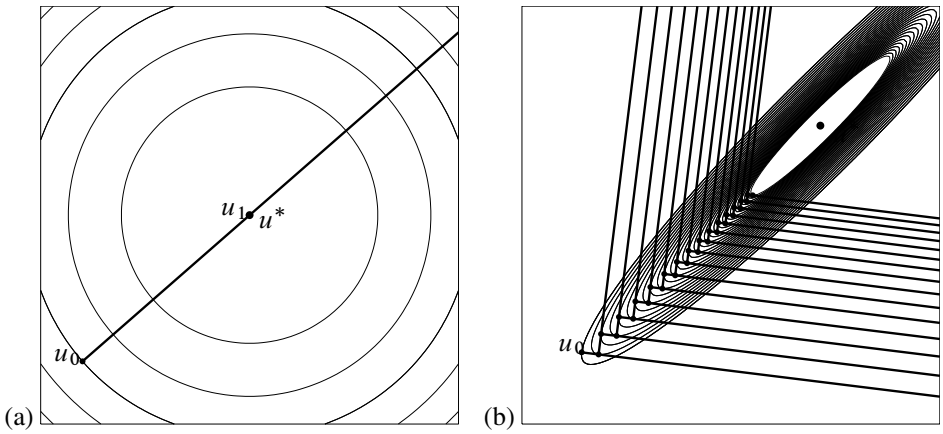

Figure 4.4. (a) *If A is a scalar multiple of the identity, then the level sets of $\phi(u)$ are circular and steepest descent converges in one iteration from any initial guess u_0.* (b) *If the level sets of $\phi(u)$ are far from circular, then steepest descent may converge slowly.*

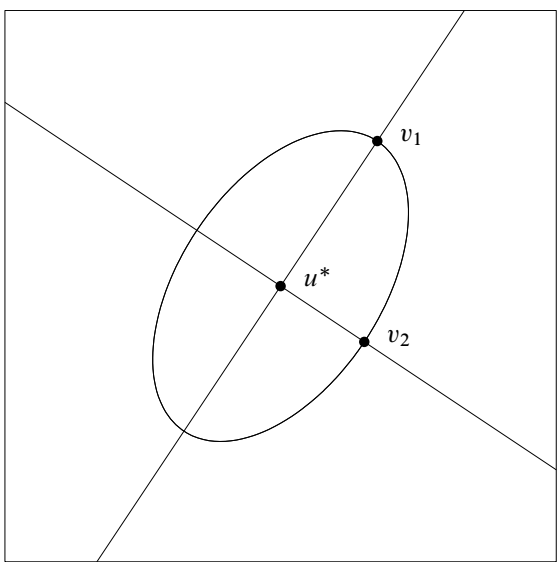

Figure 4.5. *The major and minor axes of the elliptical level set of $\phi(u)$ point in the directions of the eigenvectors of A.*

and hence each direction $v_j - u^*$ is an eigenvector of the matrix A, and the scalar λ_j is an eigenvalue.

If the eigenvalues of A are distinct, then the ellipse is noncircular and there are two unique directions for which the relation (4.34) holds, since there are two one-dimensional eigenspaces. Note that these two directions are always orthogonal since a symmetric matrix

A has orthogonal eigenvectors. If the eigenvalues of A are equal, $\lambda_1 = \lambda_2$, then every vector is an eigenvector and the level curves of $\phi(u)$ are circular. For $m = 2$ this happens only if A is a multiple of the identity matrix, as in Figure 4.4(a).

The length of the major and minor axes is related to the magnitude of λ_1 and λ_2. Suppose that v_1 and v_2 lie on the level set along which $\phi(u) = 1$, for example. (Note that $\phi(u^*) = -\frac{1}{2}u^{*T}Au^* \leq 0$, so this is reasonable.) Then

$$\frac{1}{2}v_j^T Av_j - v_j^T Au^* = 1. \tag{4.36}$$

Taking the inner product of (4.35) with $(v_j - u^*)$ and combining with (4.36) yields

$$\|v_j - u^*\|_2^2 = \frac{2 + u^{*T}Au^*}{\lambda_j}. \tag{4.37}$$

Hence the ratio of the length of the major axis to the length of the minor axis is

$$\frac{\|v_1 - u^*\|_2}{\|v_2 - u^*\|_2} = \sqrt{\frac{\lambda_2}{\lambda_1}} = \sqrt{\kappa_2(A)}, \tag{4.38}$$

where $\lambda_1 \leq \lambda_2$ and $\kappa_2(A)$ is the 2-norm condition number of A. (Recall that in general $\kappa_2(A) = \max_j |\lambda_j| / \min_j |\lambda_j|$ when A is symmetric.)

A multiple of the identity is perfectly conditioned, $\kappa_2 = 1$, and has circular level sets. Steepest descent converges in one iteration. An ill-conditioned matrix ($\kappa_2 \gg 1$) has long skinny level sets, and steepest descent may converge very slowly. The example shown in Figure 4.4(b) has $\kappa_2 = 50$, which is not particularly ill-conditioned compared to the matrices that often arise in solving differential equations.

When $m > 2$ the level sets of $\phi(u)$ are ellipsoids in m-dimensional space. Again the eigenvectors of A determine the directions of the principal axes and the spread in the size of the eigenvalues determines how stretched the ellipse is in each direction.

4.3.2 The A-conjugate search direction

The steepest descent direction can be generalized by choosing a search direction p_{k-1} in the kth iteration that might be different from the gradient direction r_{k-1}. Then we set

$$u_k = u_{k-1} + \alpha_{k-1} p_{k-1}, \tag{4.39}$$

where α_{k-1} is chosen to minimize $\phi(u_{k-1} + \alpha p_{k-1})$ over all scalars α. In other words, we perform a *line search* along the line through u_{k-1} in the direction p_{k-1} and find the minimum of ϕ on this line. The solution is at the point where the line is tangent to a contour line of ϕ, and

$$\alpha_{k-1} = \frac{p_{k-1}^T r_{k-1}}{p_{k-1}^T A p_{k-1}}. \tag{4.40}$$

A *bad* choice of search direction p_{k-1} would be a direction orthogonal to r_{k-1}, since then p_{k-1} would be tangent to the level set of ϕ at u_{k-1}, $\phi(u)$ could only increase along

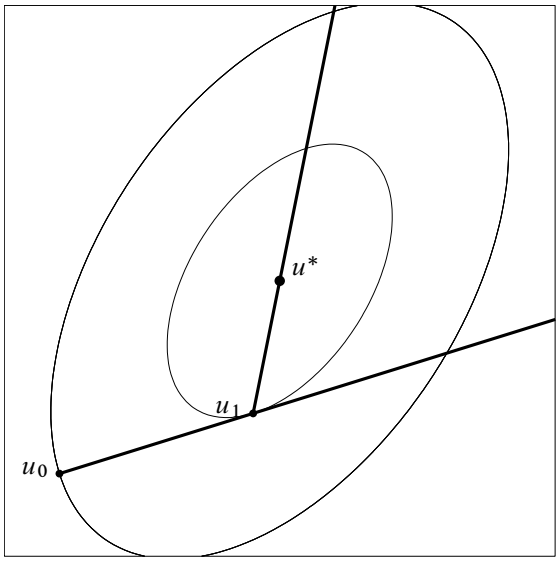

Figure 4.6. *The CG algorithm converges in two iterations from any initial guess u_0 in the case $m = 2$. The two search directions used are A-conjugate.*

this line, and so $u_k = u_{k-1}$. But as long as $p_{k-1}^T r_{k-1} \neq 0$, the new point u_k will be different from u_{k-1} and will satisfy $\phi(u_k) < \phi(u_{k-1})$.

Intuitively we might suppose that the best choice for p_{k-1} would be the direction of steepest descent r_{k-1}, but Figure 4.4(b) illustrates that this does not always give rapid convergence. A much better choice, if we could arrange it, would be to choose the direction p_{k-1} to point directly toward the solution u^*, as shown in Figure 4.6. Then minimizing ϕ along this line would give $u_k = u^*$, in which case we would have converged.

Since we don't know u^*, it seems there is little hope of determining this direction in general. But in two dimensions ($m = 2$) it turns out that we can take an arbitrary initial guess u_0 and initial search direction p_0 and then from the next iterate u_1 determine the direction p_1 that leads directly to the solution, as illustrated in Figure 4.6. Once we obtain u_1 by the formulas (4.39) and (4.40), we choose the next search direction p_1 to be a vector satisfying

$$p_1^T A p_0 = 0. \tag{4.41}$$

Below we will show that this is the optimal search direction, leading directly to $u_2 = u^*$. When $m > 2$ we generally cannot converge in two iterations, but we will see below that it is possible to define an algorithm that converges in at most m iterations to the exact solution (in exact arithmetic, at least).

Two vectors p_0 and p_1 that satisfy (4.41) are said to be *A-conjugate*. For any SPD matrix A, the vectors u and v are A-conjugate if the inner product of u with Av is zero, $u^T Av = 0$. If $A = I$, this just means the vectors are orthogonal, and A-conjugacy is a natural generalization of the notion of orthogonality. This concept is easily explained in terms of the ellipses that are level sets of the function $\phi(u)$ defined by (4.25). Consider

an arbitrary point on an ellipse. The direction tangent to the ellipse at this point and the direction that points toward the center of the ellipse are always A-conjugate. This is the fact that allows us to determine the direction toward the center once we know a tangent direction, which has been achieved by the line search in the first iteration. If $A = I$ then the ellipses are circles and the direction toward the center is simply the radial direction, which is orthogonal to the tangent direction.

To prove that the two directions shown in Figure 4.6 are A-conjugate, note that the direction p_0 is tangent to the level set of ϕ at u_1 and so p_0 is orthogonal to the residual $r_1 = f - Au_1 = A(u^* - u_1)$, which yields

$$p_0^T A(u^* - u_1) = 0. \tag{4.42}$$

On the other hand, $u^* - u_1 = \alpha p_1$ for some scalar $\alpha \neq 0$ and using this in (4.42) gives (4.41).

Now consider the case $m = 3$, from which the essential features of the general algorithm will be more apparent. In this case the level sets of the function $\phi(u)$ are concentric ellipsoids, two-dimensional surfaces in $\mathbb{R}^3$ for which the cross section in any two-dimensional plane is an ellipse. We start at an arbitrary point u_0 and choose a search direction p_0 (typically $p_0 = r_0$, the residual at u_0). We minimize $\phi(u)$ along the one-dimensional line $u_0 + \alpha p_0$, which results in the choice (4.40) for α_0, and we set $u_1 = u_0 + \alpha_0 p_0$. We now choose the search direction p_1 to be A-conjugate to p_0. In the previous example with $m = 2$ this determined a unique direction, which pointed straight to u^*. With $m = 3$ there is a two-dimensional space of vectors p_1 that are A-conjugate to p_0 (the plane orthogonal to the vector Ap_0). In the next section we will discuss the full CG algorithm, where a specific choice is made that is computationally convenient, but for the moment suppose p_1 is any vector that is both A-conjugate to p_0 and also linearly independent from p_0. We again use (4.40) to determine α_1 so that $u_2 = u_1 + \alpha_1 p_1$ minimizes $\phi(u)$ along the line $u_1 + \alpha p_1$.

We now make an observation that is crucial to understanding the CG algorithm for general m. The two vectors p_0 and p_1 are linearly independent and so they span a plane that cuts through the ellipsoidal level sets of $\phi(u)$, giving a set of concentric ellipses that are the contour lines of $\phi(u)$ within this plane. The fact that p_0 and p_1 are A-conjugate means that the point u_2 lies at the *center* of these ellipses. In other words, when restricted to this plane the algorithm so far looks exactly like the $m = 2$ case illustrated in Figure 4.6.

This means that u_2 not only minimizes $\phi(u)$ over the one-dimensional line $u_1 + \alpha p_1$ but in fact minimizes $\phi(u)$ over the entire two-dimensional plane $u_0 + \alpha p_0 + \beta p_1$ for all choices of α and β (with the minimum occurring at $\alpha = \alpha_0$ and $\beta = \alpha_1$).

The next step of the algorithm is to choose a new search direction p_2 that is A-conjugate to *both* p_0 and p_1. It is important that it be A-conjugate to both the previous directions, not just the most recent direction. This defines a unique direction (the line orthogonal to the plane spanned by Ap_0 and Ap_1). We now minimize $\phi(u)$ over the line $u_2 + \alpha p_2$ to obtain $u_3 = u_2 + \alpha_2 p_2$ (with α_2 given by (4.40)). It turns out that this always gives $u_3 = u^*$, the center of the ellipsoids and the solution to our original problem $Au = f$.

In other words, the direction p_2 always points from u_2 directly through the center of the concentric ellipsoids. This follows from the three-dimensional version of the result we showed above in two dimensions, that the direction tangent to an ellipse and the direction

toward the center are always A-conjugate. In the three-dimensional case we have a plane spanned by p_0 and p_1 and the point u_2 that minimized $\phi(u)$ over this plane. This plane must be the tangent plane to the level set of $\phi(u)$ through u_2. This tangent plane is always A-conjugate to the line connecting u_2 to u^*.

Another way to interpret this process is the following. After one step, u_1 minimizes $\phi(u)$ over the one-dimensional line $u_0 + \alpha p_0$. After two steps, u_2 minimizes $\phi(u)$ over the two-dimensional plane $u_0 + \alpha p_0 + \beta p_1$. After three steps, u_3 minimizes $\phi(u)$ over the three-dimensional space $u_0 + \alpha p_0 + \beta p_1 + \gamma p_2$. But this is all of $\mathbb{R}^3$ (provided p_0, p_1, and p_2 are linearly independent) and so $u_3 = u_0 + \alpha_0 p_0 + \alpha_1 p_1 + \alpha_2 p_2$ must be the global minimizer u^*.

For $m = 3$ this procedure always converges in *at most* three iterations (in exact arithmetic, at least). It may converge to u^* in fewer iterations. For example, if we happen to choose an initial guess u_0 that lies along one of the axes of the ellipsoids, then r_0 will already point directly toward u^*, and so $u_1 = u^*$ (although this is rather unlikely).

However, there are certain matrices A for which it will always take fewer iterations no matter what initial guess we choose. For example, if A is a multiple of the identity matrix, then the level sets of $\phi(u)$ are concentric *circles*. In this case r_0 points toward u^* from any initial guess u_0 and we always obtain convergence in one iteration. Note that in this case all three eigenvalues of A are equal, $\lambda_1 = \lambda_2 = \lambda_3$.

In the "generic" case (i.e., a random SPD matrix A), all the eigenvalues of A are distinct and three iterations are typically required. An intermediate case is if there are only two distinct eigenvalues, *e.g.*, $\lambda_1 = \lambda_2 \neq \lambda_3$. In this case the level sets of ϕ appear circular when cut by certain planes but appear elliptical when cut at other angles. As we might suspect, it can be shown that the CG algorithm always converges in at most *two* iterations in this case, from any initial u_0.

This generalizes to the following result for the analogous algorithm in m dimensions: in exact arithmetic, an algorithm based on A-conjugate search directions as discussed above converges in at most n iterations, where n is the number of distinct eigenvalues of the matrix $A \in \mathbb{R}^{m\times m}$ $(n \leq m)$.

4.3.3 The conjugate-gradient algorithm

In the above description of algorithms based on A-conjugate search directions we required that each search direction p_k be A-conjugate to all previous search directions, but we did not make a specific choice for this vector. In this section the full "conjugate gradient algorithm" is presented, in which a specific recipe for each p_k is given that has very nice properties both mathematically and computationally. The CG method was first proposed in 1952 by Hestenes and Stiefel [46], but it took some time for this and related methods to be fully understood and widely used. See Golub and O'Leary [36] for some history of the early developments.

This method has the feature mentioned at the end of the previous section: it always converges to the exact solution of $Au = f$ in a finite number of iterations $n \leq m$ (in exact arithmetic). In this sense it is not really an iterative method mathematically. We can view it as a "direct method" like Gaussian elimination, in which a finite set of operations produces the exact solution. If we programmed it to always take m iterations, then in principle we would always obtain the solution, and with the same asymptotic work estimate

as for Gaussian elimination (since each iteration takes at most $O(m^2)$ operations for matrix-vector multiplies, giving $O(m^3)$ total work). However, there are two good reasons why CG is better viewed as an iterative method than a direct method:

- In theory it produces the exact solution in n iterations (where n is the number of distinct eigenvalues) but in finite precision arithmetic u_n will not be the exact solution, and may not be substantially better than u_{n-1}. Hence it is not clear that the algorithm converges at all in finite precision arithmetic, and the full analysis of this turns out to be quite subtle [39].
- On the other hand, in practice CG frequently "converges" to a sufficiently accurate approximation to u^* in *far less* than n iterations. For example, consider solving a Poisson problem using the 5-point Laplacian on a 100×100 grid, which gives a linear system of dimension $m = 10{,}000$ and a matrix A that has $n \approx 5000$ distinct eigenvalues. An approximation to u^* consistent with the truncation error of the difference formula is obtained after approximately 150 iterations, however (after preconditioning the matrix appropriately).

That effective convergence often is obtained in far fewer iterations is crucial to the success and popularity of CG, since the operation count of Gaussian elimination is far too large for most sparse problems and we wish to use an iterative method that is much quicker. To obtain this rapid convergence it is often necessary to *precondition* the matrix, which effectively moves the eigenvalues around so that they are distributed more conducively for rapid convergence. This is discussed in Section 4.3.5, but first we present the basic CG algorithm and explore its convergence properties more fully.

The CG algorithm takes the following form:

Choose initial guess u_0 (possibly the zero vector)
$r_0 = f - Au_0$
$p_0 = r_0$
for $k = 1, 2, \ldots$
  $w_{k-1} = Ap_{k-1}$
  $\alpha_{k-1} = (r_{k-1}^T r_{k-1})/(p_{k-1}^T w_{k-1})$
  $u_k = u_{k-1} + \alpha_{k-1} p_{k-1}$
  $r_k = r_{k-1} - \alpha_{k-1} w_{k-1}$
  if $\|r_k\|$ is less than some tolerance then stop
  $\beta_{k-1} = (r_k^T r_k)/(r_{k-1}^T r_{k-1})$
  $p_k = r_k + \beta_{k-1} p_{k-1}$
  end

As with steepest descent, only one matrix-vector multiply is required at each iteration in computing w_{k-1}. In addition, two inner products must be computed each iteration. (By more careful coding than above, the inner product of each residual with itself can be computed once and reused twice.) To arrange this, we have used the fact that

$$p_{k-1}^T r_{k-1} = r_{k-1}^T r_{k-1}$$

to rewrite the expression (4.40).

Compare this algorithm to the steepest descent algorithm presented on page 80. Up through the convergence check it is essentially the same except that the A-conjugate search direction p_{k-1} is used in place of the steepest descent search direction r_{k-1} in several places.

The final two lines in the loop determine the next search direction p_k. This simple choice gives a direction p_k with the required property that p_k is A-conjugate to all the previous search directions p_j for $j = 0, 1, , \ldots, k-1$. This is part of the following theorem, which is similar to Theorem 38.1 of Trefethen and Bau [91], although there it is assumed that $u_0 = 0$. See also Theorem 2.3.2 in Greenbaum [39].

Theorem 4.1. *The vectors generated in the CG algorithm have the following properties, provided $r_k \neq 0$ (if $r_k = 0$, then we have converged):*

1. *p_k is A-conjugate to all the previous search directions,* i.e., $p_k^T A p_j = 0$ *for* $j = 0, 1, , \ldots, k-1$.
2. *The residual r_k is orthogonal to all previous residuals,* $r_k^T r_j = 0$ *for* $j = 0, 1, , \ldots, k-1$.
3. *The following three subspaces of $\mathbb{R}^m$ are identical:*

$$\begin{aligned} &\operatorname{span}(p_0, p_1, p_2, \ldots, p_{k-1}), \\ &\operatorname{span}(r_0, Ar_0, A^2 r_0, \ldots, A^{k-1} r_0), \\ &\operatorname{span}(Ae_0, A^2 e_0, A^3 e_0, \ldots, A^k e_0). \end{aligned} \tag{4.43}$$

The subspace $\mathcal{K}_k = \operatorname{span}(r_0, Ar_0, A^2 r_0, \ldots, A^{k-1} r_0)$ spanned by the vector r_0 and the first $k-1$ powers of A applied to this vector is called a *Krylov space* of dimension k associated with this vector.

The iterate u_k is formed by adding multiples of the search directions p_j to the initial guess u_0 and hence must lie in the affine spaces $u_0 + \mathcal{K}_k$ (i.e., the vector $u_k - u_0$ is in the linear space $\mathcal{K}_k$).

We have seen that the CG algorithm can be interpreted as minimizing the function $\phi(u)$ over the space $u_0 + \operatorname{span}(p_0, p_1, \ldots, p_{k-1})$ in the kth iteration, and by the theorem above this is equivalent to minimizing $\phi(u)$ over the $u_0 + \mathcal{K}_k$. Many other iterative methods are also based on the idea of solving problems on an expanding sequence of Krylov spaces; see Section 4.4.

4.3.4 Convergence of conjugate gradient

The convergence theory for CG is related to the fact that u_k minimizes $\phi(u)$ over the affine space $u_0 + \mathcal{K}_k$ defined in the previous section. We now show that a certain norm of the error is also minimized over this space, which is useful in deriving estimates about the size of the error and rate of convergence.

Since A is assumed to be SPD, the A-norm defined by

$$\|e\|_A = \sqrt{e^T A e} \tag{4.44}$$

satisfies the requirements of a vector norm in Section A.3, as discussed further in Section C.10. This is a natural norm to use because

$$\begin{aligned} \|e\|_A^2 &= (u-u^*)^T A(u-u^*) \\ &= u^T Au - 2u^T Au^* + u^{*T} Au^* \\ &= 2\phi(u) + u^{*T} Au^*. \end{aligned} \tag{4.45}$$

Since $u^{*T} Au^*$ is a fixed number, we see that minimizing $\|e\|_A$ is equivalent to minimizing $\phi(u)$.

Since

$$u_k = u_0 + \alpha_0 p_0 + \alpha_1 p_1 + \cdots + \alpha_{k-1} p_{k-1},$$

we find by subtracting u^* that

$$e_k = e_0 + \alpha_0 p_0 + \alpha_1 p_1 + \cdots + \alpha_{k-1} p_{k-1}.$$

Hence $e_k - e_0$ is in $\mathcal{K}_k$ and by Theorem 4.1 lies in $\text{span}(Ae_0, A^2e_0, \ldots, A^k e_0)$. So $e_k = e_0 + c_1 Ae_0 + c_2 A^2 e_0 + \cdots + c_k A^k e_0$ for some coefficients $c_1, \ \ldots, \ c_k$. In other words,

$$e_k = P_k(A)e_0, \tag{4.46}$$

where

$$P_k(A) = I + c_1 A + c_2 A^2 + \cdots + c_k A^k \tag{4.47}$$

is a polynomial in A. For a scalar value x we have

$$P_k(x) = 1 + c_1 x + c_2 x^2 + \cdots + c_k x^k \tag{4.48}$$

and $P_k \in \mathcal{P}_k$, where

$$\mathcal{P}_k = \{\text{polynomials } P(x) \text{ of degree at most } k \text{ satisfying } P(0) = 1\}. \tag{4.49}$$

The polynomial P_k constructed implicitly by the CG algorithm solves the minimization problem

$$\min_{P \in \mathcal{P}_k} \|P(A)e_0\|_A. \tag{4.50}$$

To understand how a polynomial function of a diagonalizable matrix behaves, recall that

$$A = V\Lambda V^{-1} \quad \Longrightarrow \quad A^j = V\Lambda^j V^{-1},$$

where V is the matrix of right eigenvectors, and so

$$P_k(A) = VP_k(\Lambda)V^{-1},$$

where

$$P_k(\Lambda) = \begin{bmatrix} P_k(\lambda_1) & & & \\ & P_k(\lambda_2) & & \\ & & \ddots & \\ & & & P_k(\lambda_m) \end{bmatrix}.$$

Note, in particular, that if $P_k(x)$ has a root at each eigenvalue $\lambda_1, \ldots, \lambda_m$, then $P_k(\Lambda)$ is the zero matrix and so $e_k = P_k(A)e_0 = 0$. If A has only $n \le m$ distinct eigenvalues $\lambda_1, \ldots, \lambda_n$, then there is a polynomial $P_n \in \mathcal{P}_n$ that has these roots, and hence the CG algorithm converges in at most n iterations, as was previously claimed.

To get an idea of how small $\|e_0\|_A$ will be at some earlier point in the iteration, we will show that for any polynomial $P(x)$ we have

$$\frac{\|P(A)e_0\|_A}{\|e_0\|_A} \le \max_{1\le j\le m} |P(\lambda_j)| \tag{4.51}$$

and then exhibit one polynomial $\tilde{P}_k \in \mathcal{P}_k$ for which we can use this to obtain a useful upper bound on $\|e_k\|_A/\|e_0\|_k$.

Since A is SPD, the eigenvectors are orthogonal and we can choose the matrix V so that $V^{-1} = V^T$ and $A = V\Lambda V^{-1}$. In this case we obtain

$$\begin{aligned}
\|P(A)e_0\|_A^2 &= e_0^T P(A)^T A P(A) e_0 \\
&= e_0^T V P(\Lambda) V^T A V P(\Lambda) V^T e_0 \\
&= e_0^T V \operatorname{diag}(\lambda_j P(\lambda_j)^2) V^T e_0 \\
&\le \max_{1\le j\le m} P(\lambda_j)^2 \left(e_0^T V \Lambda V^T e_0\right).
\end{aligned} \tag{4.52}$$

Taking square roots and rearranging results in (4.51).

We will now show that for a particular choice of polynomials $\tilde{P}_k \in \mathcal{P}_k$ we can evaluate the right-hand side of (4.51) and obtain a bound that decreases with increasing k. Since the polynomial P_k constructed by CG solves the problem (4.50), we know that

$$\|P_k(A)e_0\|_A \le \|\tilde{P}_k(A)e_0\|_A,$$

and so this will give a bound for the convergence rate of the CG algorithm.

Consider the case $k = 1$, after one step of CG. We choose the linear function

$$\tilde{P}_1(x) = 1 - \frac{2x}{\lambda_m + \lambda_1}, \tag{4.53}$$

where we assume the eigenvalues are ordered $0 < \lambda_1 \le \lambda_2 \le \cdots \le \lambda_m$. A typical case is shown in Figure 4.7(a). The linear function $\tilde{P}_1(x) = 1 + c_1 x$ must pass through $P_1(0) = 1$ and the slope c_1 has been chosen so that

$$\tilde{P}_1(\lambda_1) = -\tilde{P}_1(\lambda_m),$$

which gives

$$1 + c_1\lambda_1 = -1 - c_1\lambda_m \quad \Longrightarrow \quad c_1 = -\frac{2}{\lambda_m + \lambda_1}.$$

If the slope were made any larger or smaller, then the value of $|\tilde{P}_1(\lambda)|$ would increase at either λ_m or λ_1, respectively; see Figure 4.7(a). For this polynomial we have

$$\begin{aligned}
\max_{1\le j\le m} |\tilde{P}_1(\lambda_j)| = \tilde{P}_1(\lambda_1) &= 1 - \frac{2\lambda_1}{\lambda_m + \lambda_1} = \frac{\lambda_m/\lambda_1 - 1}{\lambda_m/\lambda_1 + 1} \\
&= \frac{\kappa - 1}{\kappa + 1},
\end{aligned} \tag{4.54}$$

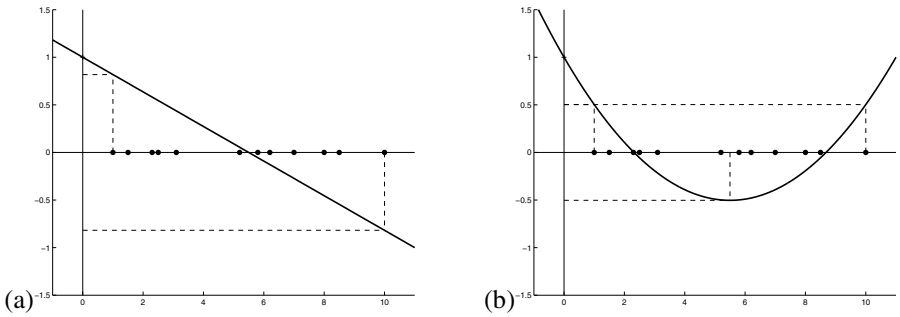

Figure 4.7. (a) *The polynomial* $\tilde{P}_1(x)$ *based on a sample set of eigenvalues marked by dots on the* x*-axis.* (b) *The polynomial* $\tilde{P}_2(x)$ *for the same set of eigenvalues.*

where $\kappa = \kappa_2(A)$ is the condition number of A. This gives an upper bound on the reduction of the error in the first step of the CG algorithm and is the best estimate we can obtain by knowing only the distribution of eigenvalues of A. The CG algorithm constructs the actual $P_1(x)$ based on e_0 as well as A and may do better than this for certain initial data. For example, if $e_0 = a_j v_j$ has only a single eigencomponent, then $P_1(x) = 1 - x/\lambda_j$ reduces the error to zero in one step. This is the case where the initial guess lies on an axis of the ellipsoid and the residual points directly to the solution $u^* = A^{-1} f$. But the above bound is the best we can obtain that holds for any e_0.

Now consider the case $k = 2$, after two iterations of CG. Figure 4.7(b) shows the quadratic function $\tilde{P}_2(x)$ that has been chosen so that

$$\tilde{P}_2(\lambda_1) = -\tilde{P}_1\left((\lambda_m + \lambda_1)/2\right) = \tilde{P}_2(\lambda_m).$$

This function equioscillates at three points in the interval $[\lambda_1, \lambda_m]$, where the maximum amplitude is taken. This is the polynomial from $\mathcal{P}_2$ that has the smallest maximum value on this interval, i.e., it minimizes

$$\max_{\lambda_1 \le x \le \lambda_m} |P(x)|.$$

This polynomial does not necessarily solve the problem of minimizing

$$\max_{1 \le j \le m} |P(\lambda_j)|$$

unless $(\lambda_1 + \lambda_m)/2$ happens to be an eigenvalue, since we could possibly reduce this quantity by choosing a quadratic with a slightly larger magnitude near the midpoint of the interval but a smaller magnitude at each eigenvalue. However, it has the great virtue of being easy to compute based only on λ_1 and λ_m. Moreover, we can compute the analogous polynomial $\tilde{P}_k(x)$ for arbitrary degree k, the polynomial from $\tilde{P}_k$ with the property of minimizing the maximum amplitude over the entire interval $[\lambda_1, \lambda_m]$. The resulting maximum amplitude also can be computed in terms of λ_1 and λ_m and in fact depends only on the ratio of these and hence depends only on the condition number of A. This gives an upper bound for the convergence rate of CG in terms of the condition number of A that often is quite realistic.

The polynomials we want are simply shifted and scaled versions of the Chebyshev polynomials discussed in Section B.3.2. Recall that $T_k(x)$ equioscillates on the interval $[-1, 1]$ with the extreme values ± 1 being taken at $k+1$ points, including the endpoints. We shift this to the interval $[\lambda_1, \lambda_m]$, scale it so that the value at $x = 0$ is 1, and obtain

$$\tilde{P}_k(x) = \frac{T_k\left(\frac{\lambda_m+\lambda_1-2x}{\lambda_m-\lambda_1}\right)}{T_k\left(\frac{\lambda_m+\lambda_1}{\lambda_m-\lambda_1}\right)}. \tag{4.55}$$

For $k = 1$ this gives (4.53) since $T_1(x) = x$. We now need only compute

$$\max_{1\le j\le m} |\tilde{P}_k(\lambda_j)| = \tilde{P}_k(\lambda_1)$$

to obtain the desired bound on $\|e_k\|_A$. We have

$$\tilde{P}_k(\lambda_1) = \frac{T_k(1)}{T_k\left(\frac{\lambda_m+\lambda_1}{\lambda_m-\lambda_1}\right)} = \frac{1}{T_k\left(\frac{\lambda_m+\lambda_1}{\lambda_m-\lambda_1}\right)}. \tag{4.56}$$

Note that

$$\frac{\lambda_m+\lambda_1}{\lambda_m-\lambda_1} = \frac{\lambda_m/\lambda_1+1}{\lambda_m/\lambda_1-1} = \frac{\kappa+1}{\kappa-1} > 1$$

so we need to evaluate the Chebyshev polynomial at a point outside the interval $[-1, 1]$, which according to (B.27) is

$$T_k(x) = \cosh(k \cosh^{-1} x).$$

We have

$$\cosh(z) = \frac{e^z + e^{-z}}{2} = \frac{1}{2}(y + y^{-1})$$

where $y = e^z$, so if we make the change of variables $x = \frac{1}{2}(y + y^{-1})$, then $\cosh^{-1} x = z$ and

$$T_k(x) = \cosh(kz) = \frac{e^{kz} + e^{-kz}}{2} = \frac{1}{2}(y^k + y^{-k}).$$

We can find y from any given x by solving the quadratic equation $y^2 - 2xy + 1 = 0$, yielding

$$y = x \pm \sqrt{x^2 - 1}.$$

To evaluate (4.56) we need to evaluate T_k at $x = (\kappa+1)/(\kappa-1)$, where we obtain

$$\begin{aligned} y &= \frac{\kappa+1}{\kappa-1} \pm \sqrt{\left(\frac{\kappa+1}{\kappa-1}\right)^2 - 1} \\ &= \frac{\kappa+1 \pm \sqrt{4\kappa}}{\kappa-1} \\ &= \frac{(\sqrt{\kappa} \pm 1)^2}{(\sqrt{\kappa}+1)(\sqrt{\kappa}-1)} \\ &= \frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1} \quad \text{or} \quad \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}. \end{aligned} \tag{4.57}$$

Either choice of y gives the same value for

$$T_k\left(\frac{\kappa+1}{\kappa-1}\right) = \frac{1}{2}\left[\left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^k + \left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k\right]. \tag{4.58}$$

Using this in (4.56) and combining with (4.51) gives

$$\frac{\|P(A)e_0\|_A}{\|e_0\|_A} \le 2\left[\left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^k + \left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k\right]^{-1} \le 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k. \tag{4.59}$$

This gives an upper bound on the error when the CG algorithm is used. In practice the error may be smaller, either because the initial error e_0 happens to be deficient in some eigencoefficients or, more likely, because the optimal polynomial $P_k(x)$ is much smaller at all the eigenvalues λ_j than our choice $\tilde{P}_k(x)$ used to obtain the above bound. This typically happens if the eigenvalues of A are clustered near fewer than m points. Then the $P_k(x)$ constructed by CG will be smaller near these points and larger on other parts of the interval $[\lambda_1, \lambda_m]$ where no eigenvalues lie. As an iterative method it is really the number of clusters, not the number of mathematically distinct eigenvalues, that then determines how rapidly CG converges in practical terms.

The bound (4.59) is realistic for many matrices, however, and shows that in general the convergence rate depends on the size of the condition number κ. If κ is large, then

$$2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k \approx 2\left(1-\frac{2}{\sqrt{\kappa}}\right)^k \approx 2e^{-2k/\sqrt{\kappa}}, \tag{4.60}$$

and we expect that the number of iterations required to reach a desired tolerance will be $k = O(\sqrt{\kappa})$.

For example, the standard second order discretization of the Poisson problem on a grid with m points in each direction gives a matrix with $\kappa = O(1/h^2)$, where $h = 1/(m+1)$. The bound (4.60) suggests that CG will require $O(m)$ iterations to converge, which is observed in practice. This is true in any number of space dimensions. In one dimension where there are only m unknowns this does not look very good (and of course it's best just to solve the tridiagonal system by elimination). In two dimensions there are m^2 unknowns and m^2 work per iteration is required to compute Ap_{k-1}, so CG requires $O(m^3)$ work to converge to a fixed tolerance, which is significantly better than Gauss elimination and comparable to SOR with the optimal ω. Of course for this problem a fast Poisson solver could be used, requiring only $O(m^2 \log m)$ work. But for other problems, such as variable coefficient elliptic equations with symmetric coefficient matrices, CG may still work very well while SOR works well only if the optimal ω is found, which may be impossible, and fast Fourier transform (FFT) methods are inapplicable. Similar comments apply in three dimensions.

4.3.5 Preconditioners

We saw in Section 4.3.4 that the convergence rate of CG generally depends on the condition number of the matrix A. Often *preconditioning* the system can reduce the condition

number of the matrix involved and speed up convergence. In fact preconditioning is absolutely essential for most practical problems, and there are many papers in the literature on the development of effective preconditioners for specific applications or general classes of problems.

If M is any nonsingular matrix, then

$$Au = f \quad \Longleftrightarrow \quad M^{-1}Au = M^{-1}f. \tag{4.61}$$

So we could solve the system on the right instead of the system on the left. If M is some approximation to A, then $M^{-1}A$ may have a much smaller condition number than A. If $M = A$, then $M^{-1}A$ is perfectly conditioned but we'd still be faced with the problem of computing $M^{-1}f = A^{-1}f$.

Of course in practice we don't actually form the matrix $M^{-1}A$. As we will see below, the preconditioned conjugate gradient (PCG) algorithm has the same basic form as CG, but a step is added in which a system of the form $Mz = r$ is solved, and it is here that the preconditioner is "applied." The idea is to choose an M for which $M^{-1}A$ is better conditioned than A but for which systems involving M are much easier to solve than systems involving A. Often this can be done by solving some approximation to the original physical problem (e.g., by solving on a coarser grid and then interpolating, by solving a nearby constant-coefficient problem).

A very simple preconditioner that is effective for some problems is simply to use $M = \text{diag}(A)$, a diagonal matrix for which solving linear systems is trivial. This doesn't help for the Poisson problem on a rectangle, where this is just a multiple of the identity matrix, and hence doesn't change the condition number at all, but for other problems such as variable coefficient elliptic equations with large variation in the coefficients, this can make a significant difference.

Another popular approach is to use an incomplete Cholesky factorization of the matrix A, as discussed briefly in Section 4.3.6. Other iterative methods are sometimes used as a preconditioner, for example, the multigrid algorithm of Section 4.6. Other preconditioners are discussed in many places; for example, there is a list of possible approaches in Trefethen and Bau [91].

A problem with the approach to preconditioning outlined above is that $M^{-1}A$ may not be symmetric, even if M^{-1} and A are, in which case CG could not be applied to the system on the right in (4.61). Instead we can consider solving a different system, again equivalent to the original:

$$(C^{-T}AC^{-1})(Cu) = C^{-T}f, \tag{4.62}$$

where C is a nonsingular matrix. Write this system as

$$\tilde{A}\tilde{u} = \tilde{f}. \tag{4.63}$$

Note that since $A^T = A$, the matrix $\tilde{A}$ is also symmetric even if C is not. Moreover $\tilde{A}$ is positive definite (provided A is) since

$$u^T\tilde{A}u = u^TC^{-T}AC^{-1}u = (C^{-1}u)^TA(C^{-1}u) > 0$$

for any vector $u \neq 0$.

Now the problem is that it may not be clear how to choose a reasonable matrix C in this formulation. The goal is to make the condition number of $\tilde{A}$ small, but C appears twice in the definition of $\tilde{A}$ so C should be chosen as some sort of "square root" of A. But note that the condition number of $\tilde{A}$ depends only on the eigenvalues of this matrix, and we can apply a similarity transformation to $\tilde{A}$ without changing its eigenvalues, e.g.,

$$C^{-1}\tilde{A}C = C^{-1}C^{-T}A = (C^T C)^{-1}A. \tag{4.64}$$

The matrix $\tilde{A}$ thus has the same condition number as $(C^T C)^{-1}A$. So if we have a sensible way to choose a preconditioner M in (4.61) that is SPD, we could in principle determine C by a Cholesky factorization of the matrix M.

In practice this is not necessary, however. There is a way to write the PCG algorithm in such a form that it only requires solving systems involving M (without ever computing C) but that still corresponds to applying CG to the SPD system (4.63).

To see this, suppose we apply CG to (4.63) and generate vectors $\tilde{u}_k$, $\tilde{p}_k$, $\tilde{w}_k$, and $\tilde{r}_k$. Now define

$$u_k = C^{-1}\tilde{u}_k, \quad p_k = C^{-1}\tilde{p}_k, \quad w_k = C^T\tilde{w}_k, \text{ and } r_k = C^T\tilde{r}_k.$$

Note that $\tilde{r}_k$ is multiplied by C^T, not C^{-1}. Here $\tilde{r}_k$ is the residual when $\tilde{u}_k$ is used in the system (4.63). Note that if $\tilde{u}_k$ approximates the solution to (4.62), then u_k will approximate the solution to the original system $Au = f$. Moreover, we find that

$$r_k = C(\tilde{f} - \tilde{A}\tilde{u}_k) = f - Au_k$$

and so r_k is the residual for the original system. Rewriting this CG algorithm in terms of the variables u_k, p_k, w_k, and r_k, we find that it can be rewritten as the following PCG algorithm:

$r_0 = f - Au_0$
Solve $Mz_0 = r_0$ for z_0
$p_0 = z_0$
for $k = 1, 2, \ldots$
    $w_{k-1} = Ap_{k-1}$
    $\alpha_{k-1} = (z_{k-1}^T r_{k-1})/(p_{k-1}^T w_{k-1})$
    $u_k = u_{k-1} + \alpha_{k-1}p_{k-1}$
    $r_k = r_{k-1} - \alpha_{k-1}w_{k-1}$
    if $\|r_k\|$ is less than some tolerance then stop
    Solve $Mz_k = r_k$ for z_k
    $\beta_{k-1} = (z_k^T r_k)/(z_{k-1}^T r_{k-1})$
    $p_k = z_k + \beta_{k-1}p_{k-1}$
    end

Note that this is essentially the same as the CG algorithm on page 87, but we solve the system $Mz_k = r_k$ for $z_k = M^{-1}r_k$ in each iteration and then use the z vector in place of r in several places in the algorithm.

4.3.6 Incomplete Cholesky and ILU preconditioners

There is one particular preconditioning strategy where the matrix C is in fact computed and used. Since A is SPD it has a Cholesky factorization of the form $A = R^T R$, where R is an upper triangular matrix (this is just a special case of the LU factorization). The problem with computing and using this factorization to solve the original system $Au = f$ is that the elimination process used to compute R generates a lot of nonzeros in the R matrix, so that it is typically much less sparse than A.

A popular preconditioner that is often very effective is to do an *incomplete Cholesky factorization* of the matrix A, in which nonzeros in the factors are allowed to appear only in positions where the corresponding element of A is nonzero, simply throwing away the other elements as we go along. This gives an approximate factorization of the form $A \approx C^T C$. This defines a preconditioner $M = C^T C$. To solve systems of the form $Mz = r$ required in the PCG algorithm we use the known Cholesky factorization of M and only need to do forward and back substitutions for these lower and upper triangular systems. This approach can be generalized by specifying a *drop tolerance* and dropping only those elements of R that are smaller than this tolerance. A smaller drop tolerance will give a better approximation to A but a denser matrix C.

Methods for nonsymmetric linear systems (e.g., the GMRES algorithm in the next section) also generally benefit greatly from preconditioners and this idea can be extended to *incomplete LU (ILU) factorizations* as a preconditioner for nonsymmetric systems.

4.4 The Arnoldi process and GMRES algorithm

For linear systems that are not SPD, many other iterative algorithms have been developed. We concentrate here on just one of these, the popular GMRES (generalized minimum residual) algorithm. In the course of describing this method we will also see the Arnoldi process, which is useful in other applications.

In the kth step of GMRES a least squares problem is solved to find the best approximation to the solution of $Au = f$ from the affine space $u_0 + \mathcal{K}_k$, where again $\mathcal{K}_k$ is the k-dimensional Krylov space $\mathcal{K}_k = \text{span}(r_0, Ar_0, A^2 r_0, \ldots, A^{k-1} r_0)$ based on the initial residual $r_0 = f - Au_0$. To do this we build up a matrix of the form

$$Q_k = [q_1\ q_2\ \cdots\ q_k] \in \mathbb{R}^{m \times k},$$

whose columns form an orthonormal basis for the space $\mathcal{K}_k$. In the kth iteration we determine the vector q_{k+1} by starting with some vector v_j that is not in $\mathcal{K}_k$ and orthogonalizing it to $q_1,\ q_2,\ \ldots,\ q_k$ using a Gram–Schmidt-type procedure. How should we choose v_k? One obvious choice might be $v_k = A^k r_0$. This is a bad choice, however. The vectors $r_0,\ Ar_0,\ A^2 r_0,\ \ldots$, although linearly independent and a natural choice from our definition of the Krylov space, tend to become more and more closely aligned (nearly linearly dependent) as k grows. (In fact they converge to the eigenvector direction of the dominant eigenvalue of A since this is just the power method.) In other words the Krylov matrix

$$K_{k+1} = [r_0\ Ar_0\ A^2 r_0\ \cdots\ A^k r_0]$$

has rank $k + 1$ but has some very small singular values. Applying the orthogonalization procedure using $v_k = A^k r_0$ would amount to doing a QR factorization of the matrix K_{k+1},

which is numerically unstable in this case. Moreover, it is not clear how we would use the resulting basis to find the least square approximation to $Au = f$ in the affine space $u_0 + \mathcal{K}_k$.

Instead we choose $v_k = Aq_k$ as the starting point in the kth step. Since q_k has already been orthogonalized to all the previous basis vectors, this does not tend to be aligned with an eigendirection. In addition, the resulting procedure can be viewed as building up a factorization of the matrix A itself that can be directly used to solve the desired least squares problem.

This procedure is called the *Arnoldi process*. This algorithm is important in other applications as well as in the solution of linear systems, as we will see below. Here is the basic algorithm, with an indication of where a least squares problem should be solved in each iteration to compute the GMRES approximations u_k to the solution of the linear system:

```
q_1 = r_0/||r_0||_2
for k = 1, 2, ...
    v = Aq_k
    for i = 1 : k
        h_ik = q_i^T v
        v = v - h_ik q_i        % orthogonalize to previous vectors
        end
    h_{k+1,k} = ||v||_2
    q_{k+1} = v/h_{k+1,k}       % normalize
    % For GMRES: Check residual of least squares problem (4.75).
    % If it's sufficiently small, halt and compute u_k
    end
```

Before discussing the least squares problem, we must investigate the form of the matrix factorization we are building up with this algorithm. After k iterations we have

$$Q_k = [q_1\ q_2\ \cdots\ q_k] \in \mathbb{R}^{m\times k}, \qquad Q_{k+1} = [Q_k\ q_{k+1}] \in \mathbb{R}^{m\times(k+1)},$$

which form orthonormal bases for $\mathcal{K}_k$ and $\mathcal{K}_{k+1}$, respectively. Let

$$H_k = \begin{bmatrix} h_{11} & h_{12} & h_{13} & \cdots & h_{1,k-1} & h_{1k} \\ h_{21} & h_{22} & h_{23} & \cdots & h_{2,k-1} & h_{2k} \\ & h_{32} & h_{33} & \cdots & h_{3,k-1} & h_{3k} \\ & & \ddots & \ddots & & \vdots \\ & & & & h_{k,k-1} & h_{kk} \end{bmatrix} \in \mathbb{R}^{k\times k} \tag{4.65}$$

be the upper Hessenberg matrix consisting of the h values computed so far. We will also need the matrix $\tilde{H}_k \in \mathbb{R}^{(k+1)\times k}$ consisting of H_k with an additional row that is all zeros except for the $h_{k+1,k}$ entry, also computed in the kth step of Arnoldi.

Now consider the matrix product

$$AQ_k = [Aq_1\ Aq_2\ \cdots\ Aq_k].$$

The jth column of this matrix has the form of the starting vector v used in the jth iteration of Arnoldi, and unraveling the computations done in the jth step shows that

$$h_{j+1,j}q_{j+1} = Aq_j - h_{1j}q_1 - h_{2j}q_2 - \cdots - h_{jj}q_j.$$

This can be rearranged to give

$$Aq_j = h_{1j}q_1 + h_{2j}q_2 + \cdots + h_{jj}q_j + h_{j+1,j}q_{j+1}. \tag{4.66}$$

The left-hand side is the jth column of AQ_k and the right-hand side, at least for $j < k$, is the jth column of the matrix $Q_k H_k$. We find that

$$AQ_k = Q_k H_k + h_{k+1,k}q_{k+1}e_k^T. \tag{4.67}$$

In the final term the vector $e_k^T = [0\ 0\ \cdots\ 0\ 1]$ is the vector of length k with a 1 in the last component and $h_{k+1,k}q_{k+1}e_k^T$ is the $m \times k$ matrix that is all zeros except the last column, which is $h_{k+1,k}q_{k+1}$. This term corresponds to the last term in the expression (4.66) for $j = k$. The expression (4.67) can also be rewritten as

$$AQ_k = Q_{k+1}\tilde{H}_k. \tag{4.68}$$

If we run the Arnoldi process to completion (i.e., up to $k = m$, the dimension of A), then we will find in the final step that $v = Aq_m$ lies in the Krylov space $\mathcal{K}_m$ (which is already all of $\mathbb{R}^m$), so orthogonalizing it to each of the q_i for $i = 1 : m$ will leave us with $v = 0$. So in this final step there is no $h_{m+1,m}$ value or q_{m+1} vector and setting $Q = Q_m$ and $H = H_m$ gives the result

$$AQ = QH,$$

which yields

$$Q^T AQ = H \qquad \text{or} \qquad A = QHQ^T. \tag{4.69}$$

We have reduced A to Hessenberg form by a similarity transformation.

Our aim at the moment is not to reduce A all the way by running the algorithm to $k = m$ but rather to approximate the solution to $Au = f$ well in a few iterations. After k iterations we have (4.67) holding. We wish to compute u_k, an approximation to $u = A^{-1}f$ from the affine space $u_0 + \mathcal{K}_k$, by minimizing the 2-norm of the residual $r_k = f - Au_k$ over this space. Since the columns of Q_k form a basis for $\mathcal{K}_k$, we must have

$$u_k = u_0 + Q_k y_k \tag{4.70}$$

for some vector $y_k \in \mathbb{R}^k$, and so the residual is

$$\begin{aligned} r_k &= f - A(u_0 + Q_k y_k) \\ &= r_0 - AQ_k y_k \\ &= r_0 - Q_{k+1}\tilde{H}_k y_k, \end{aligned} \tag{4.71}$$

where we have used (4.68). But recall that the first column of Q_{k+1} is just $q_1 = r_0/\|r_0\|_2$ so we have $r_0 = Q_{k+1}\eta$, where η is the vector

$$\eta = \begin{bmatrix} \|r_0\|_2 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \in \mathbb{R}^{k+1}. \tag{4.72}$$

Hence

$$r_k = Q_{k+1}(\eta - \tilde{H}_k y_k). \tag{4.73}$$

Since $Q_{k+1}^T Q_{k+1} = I$, computing $r_k^T r_k$ shows that

$$\|r_k\|_2 = \|\eta - \tilde{H}_k y_k\|_2. \tag{4.74}$$

In the kth iteration of GMRES we choose y_k to solve the least squares problem

$$\min_{y \in \mathbb{R}^k} \|\eta - \tilde{H}_k y\|_2, \tag{4.75}$$

and the approximation u_k is then given by (4.70).

Note the following (see, e.g., Greenbaum [39] for details):

- $\tilde{H}_k \in \mathbb{R}^{(k+1)\times k}$ and $\eta \in \mathbb{R}^{k+1}$, so this is a small least squares problem when $k \ll m$.
- $\tilde{H}_k$ is already nearly upper triangular, so solving the least squares problem by computing the QR factorization of this matrix is relatively cheap.
- Moreover, in each iteration $\tilde{H}_k$ consists of $\tilde{H}_{k-1}$ with one additional row and column added. Since the QR factorization of $\tilde{H}_{k-1}$ has already been computed in the previous iteration, the QR factorization of $\tilde{H}_k$ is easily computed with little additional work.
- Once the QR factorization is known, it is possible to compute the residual in the least squares problem (4.75) without actually solving for y_k (which requires solving an upper triangular system of size k using the R matrix from QR). So in practice only the residual is checked each iteration and the final y_k and u_k are actually computed only after the convergence criterion is satisfied.

Notice, however, one drawback of GMRES, and the Arnoldi process more generally, for nonsymmetric matrices: in the kth iteration we must orthogonalize v to all k previous basis vectors, so we must keep all these vectors in storage. For practical problems arising from discretizing a multidimensional partial differential equation (PDE), each of these "vectors" is an approximation to the solution over the full grid, which may consist of millions of grid points. Taking more than a few iterations may consume a great deal of storage.

Often in GMRES the iteration is restarted periodically to save storage: the approximation u_k at some point is used as the initial guess for a new GMRES iteration. There's a large literature on this and other variations of GMRES.

4.4.1 Krylov methods based on three term recurrences

Note that if A is symmetric, then so is the Hessenberg matrix H, since

$$H^T = (Q^T A Q)^T = Q^T A^T Q = Q^T A Q = H,$$

and hence H must be tridiagonal. In this case the Arnoldi iteration simplifies in a very important way: $h_{ik} = 0$ for $i = 1, 2, \ldots, (k-2)$ and in the kth iteration of Arnoldi v

only needs to be orthogonalized to the previous two basis vectors. There is a three-term recurrence relation for each new basis vector in terms of the previous two. This means only the two previous vectors need to be stored at any time, rather than all the previous q_i vectors, which is a dramatic improvement for systems of large dimension.

The special case of Arnoldi on a symmetric matrix (or more generally a complex Hermitian matrix) is called the *Lanczos iteration* and plays an important role in many numerical algorithms, not just for linear systems but also for eigenvalue problems and other applications.

There are also several iterative methods for nonsymmetric systems of equations that are based on three-term recurrence relations using the idea of *biorthogonalization*—in addition to building up a Krylov space based on powers of the matrix A, a second Krylov space based on powers of the matrix A^H is simultaneously determined. Basis vectors v_i and w_i for the two spaces are found that are not orthogonal sets separately, but are instead "biorthogonal" in the sense that

$$v_i^H w_j = 0 \quad \text{if } i \neq j.$$

It turns out that there are three-term recurrence relations for these sets of basis vectors, eliminating the need for storing long sequences of vectors. The disadvantage is that two matrix-vector multiplies must be performed each iteration, one involving A and another involving A^H. One popular method of this form is Bi-CGSTAB (bi-conjugate gradient stabilized), introduced by Van der Vorst [95]. See, e.g., [39], [91] for more discussion of this method and other variants.

4.4.2 Other applications of Arnoldi

The Arnoldi process has other applications besides the approximate solution of linear systems. Note from (4.67) that

$$Q_k^T A Q_k = H_k \tag{4.76}$$

since $Q_k^T Q_k = I$ and $Q_k^T q_{k+1} = 0$. This looks much like (4.69), but here Q_k is a rectangular matrix (for $k < m$) and so this is not a similarity transformation and H_k does not have the same eigenvalues as A (or even the same number of eigenvalues, since it has only k). However, a very useful fact is that the eigenvalues of H_k are typically good approximations to the dominant eigenvalues of A (those with largest magnitude). In many eigenvalue applications where A is a large sparse matrix, the primary interest is in determining the dominant eigenvalues (e.g., in determining stability or asymptotic growth properties of matrix iterations or exponentials). In this case we can run the Arnoldi process (which requires only matrix-vector multiplies with A) and then calculate the eigenvalues of the small matrix H_k in the kth iteration as an approximation to the dominant eigenvalues of A. This approach is implemented in the ARPACK software [62], which is used, for example, by the `eigs` command in MATLAB.

Also note that from (4.76), by multiplying on the left by Q_k and on the right by Q_k^T we obtain

$$Q_k Q_k^T A Q_k Q_k^T = Q_k H_k Q_k^T. \tag{4.77}$$

If $k = m$, then $Q_k Q_k^T = I$ and this is simply (4.69). For $k < m$, $Q_k Q_k^T$ is the projection matrix that projects any vector z in $\mathbb{R}^m$ onto the k-dimensional Krylov space $\mathcal{K}_k$.

So the operator on the left of (4.77), when applied to any vector in $z \in \mathbb{R}^m$, has the following effect: the vector is first projected to $\mathcal{K}_k$, then A is applied, and then the result is again projected to $\mathcal{K}_k$. The operator on the right does the same thing in a different form: $Q_k^T z \in \mathbb{R}^k$ consists of the coefficients of the basis vectors of Q_k for the projected vector. Multiplying by H_k transforms these coefficients according to the effect of A, and $H_k Q_k^T z$ are then the modified coefficients used to form a linear combination of the basis vectors when this is multiplied by Q_k. Hence we can view H_k as the restriction of A to the k-dimensional Krylov space $\mathcal{K}_k$. Thus it is not so surprising, for example, that the eigenvalues of H_k approximate the dominant eigenvalues of A. As commented above, the basis vectors $f,\ Af,\ A^2 f,\ \ldots$ for $\mathcal{K}_k$ tend to align with the dominant eigenvectors of A, and if an eigenvector of A lies in $\mathcal{K}_k$, then it is also an eigenvector of the restriction of A to this space.

We will see another use of Krylov space methods in Section 11.6, where we consider exponential time differencing methods for time-dependent ordinary differential equations (ODEs). The matrix exponential applied to a vector, $e^{At}v$, arises in solving linear systems of ODEs. This often can be effectively approximated by $Q_k e^{H_k t} Q_k^T v$ for $k \ll m$. More generally, other functions $\phi(z)$ can be extended to matrix arguments (using the Cauchy integral formula (D.4), for example) and their action often approximated by $\phi(A)v \approx Q_k \phi(H_k t) Q_k^T v$.

4.5 Newton–Krylov methods for nonlinear problems

So far in this chapter we have considered only linear problems and a variety of iterative methods that can be used to solve sparse linear systems of the form $Au = f$. However, many differential equations are nonlinear and these naturally give rise to nonlinear systems of equations after discretization. In Section 2.16 we considered a nonlinear boundary value problem and discussed the use of Newton's method for its solution. Recall that Newton's method is an iterative method based on linearizing the problem about the current approximation to the solution and then solving a linear system of equations involving the Jacobian matrix to determine the next update to the approximation. If the nonlinear system is written as $G(u) = 0$, then the Newton update is

$$u^{[j+1]} = u^{[j]} - \delta^{[j]}, \tag{4.78}$$

where $\delta^{[j]}$ is the solution to the linear system

$$J^{[j]}\delta^{[j]} = G(u^{[j]}). \tag{4.79}$$

Here $J^{[j]} = G'(u^{[j]})$ is the Jacobian matrix evaluated at the current iterate. For the one-dimensional problem of Section 2.16 the Jacobian matrix is tridiagonal and the linear system is easily solved in each iteration by a direct method.

For a nonlinear problem in more space dimensions the Jacobian matrix typically will have the same nonzero structure as the matrices discussed in the context of linear elliptic equations in Chapter 3. (Of course for a linear problem $Au = f$ we have $G(u) = Au - f$ and the matrix A is the Jacobian matrix.) Hence when solving a nonlinear elliptic equation by a Newton method we must solve, in each Newton iteration, a sparse linear system of the type we are tackling in this chapter. For practical problems the Jacobian matrix is often

nonsymmetric and Krylov space methods such as GMRES are a popular choice. This gives an obvious way to combine Newton's method with Krylov space methods: in each iteration of Newton's method determine all the elements of the Jacobian matrix $J^{[j]}$ and then apply a Krylov space method to solve the system (4.79).

However, the term *Newton–Krylov method* often refers to something slightly different, in which the calculation of the full Jacobian matrix is avoided in performing the Krylov space iteration. These methods are also called *Jacobian–free Newton–Krylov methods* (JFNK), and a good survey of these methods and their history and applicability is given in the review paper of Knoll and Keyes [57].

To explain the basic idea, consider a single iteration of the Newton method and drop the superscript j for notational convenience. So we need to solve a linear system of the form

$$J(u)\delta = G(u), \tag{4.80}$$

where u a fixed vector (the current Newton iterate $u^{[j]}$).

When the GMRES algorithm (or any other iterative method requiring only matrix vector products) is applied to the linear system (4.80), we require only the product $J(u)q_k$ for certain vectors q_k (where k is the iteration index of the linear solver). The key to JFNK is to recognize that since $J(u)$ is a Jacobian matrix, the vector $J(u)q_k$ is simply the directional derivative of the nonlinear function G at this particular u in the direction q_k. The Jacobian matrix contains all the information needed to compute the directional derivative in any arbitrary direction, but there is no need to compute the full matrix if, in the course of the Krylov iteration, we are only going to need the directional derivative in relatively few directions. This is the case if we hope that the Krylov iteration will converge in very few iterations relative to the dimension of the system.

How do we compute the directional derivative $J(u)q_k$ without knowing $J(u)$? The standard approach is to use a simple finite difference approximation,

$$J(u)q_k \approx (G(u + \epsilon q_k) - G(u))/\epsilon, \tag{4.81}$$

where ϵ is some small real number. This approximation is first order accurate in ϵ but is sufficiently accurate for the needs of the Krylov space method if we take ϵ quite small. If ϵ is too small, however, then numerical cancellation can destroy the accuracy of the approximation in finite precision arithmetic. For scalar problems the optimal trade-off typically occurs at $\epsilon = \sqrt{\epsilon_{\text{mach}}}$, the square root of the machine precision (i.e., $\epsilon \approx 10^{-8}$ for 64-bit double precision calculations). See [57] for some comments on good choices.

JFNK is particularly advantageous for problems where the derivatives required in the Jacobian matrix cannot be easily computed analytically, for example, if the computation of $G(u)$ involves table look-ups or requires solving some other nonlinear problem. A subroutine evaluating $G(u)$ is already needed for a Krylov space method in order to evaluate the right-hand side of (4.80), and the JFNK method simply calls this in each iteration of the Krylov method to compute $G(u + \epsilon q_k)$.

Good preconditioners generally are required to obtain good convergence properties and limit the number of Krylov iterations (and hence nonlinear G evaluations) required. As with Newton's method in other contexts, a good initial guess is often required to achieve convergence of the Newton iteration, regardless of how the system (4.79) is solved in each iteration. See [57] for more comments on these and other issues.

4.6 Multigrid methods

We return to the solution of linear systems $Au = f$ and discuss a totally different approach to the solution of such systems. Multigrid methods also can be applied directly to nonlinear problems and there is a vast literature on variations of these methods and applications to a variety of problems. Here we concentrate on understanding the main idea of multigrid methods in the context of the one-dimensional model problem $u''(x) = f(x)$. For more discussion, see, for example, [11], [52], [41], [101].

4.6.1 Slow convergence of Jacobi

Let

$$f(x) = -20 + a\phi''(x)\cos(\phi(x)) - a(\phi'(x))^2 \sin(\phi(x)), \tag{4.82}$$

where $a = 0.5$, $\phi(x) = 20\pi x^3$, and consider the boundary value problem $u''(x) = f(x)$ with Dirichlet boundary conditions $u(0) = 1$ and $u(1) = 3$. The true solution is

$$u(x) = 1 + 12x - 10x^2 + a\sin(\phi(x)), \tag{4.83}$$

which is plotted in Figure 4.8(a). This function has been chosen because it clearly contains variations on many different spatial scales, i.e., large components of many different frequencies.

We discretize this problem with the standard tridiagonal systems (2.10) and apply the Jacobi iterative method of Section 4.1 to the linear initial guess u_0 with components $1 + 2x_i$, which is also shown in Figure 4.8(a). Figure 4.8(b) shows the error e_0 in this initial guess on a grid with $m = 255$ grid points.

The left column of Figure 4.9 shows the approximations obtained after $k = 20$, 100, and 1000 iterations of Jacobi. This method converges very slowly and it would take about 10^5 iterations to obtain a useful approximation to the solution. However, notice something very interesting in Figure 4.9. The more detailed features of the solution develop relatively quickly and it is the larger-scale features that are slow to appear. At first this may seem counterintuitive since we might expect the small-scale features to be harder to capture.

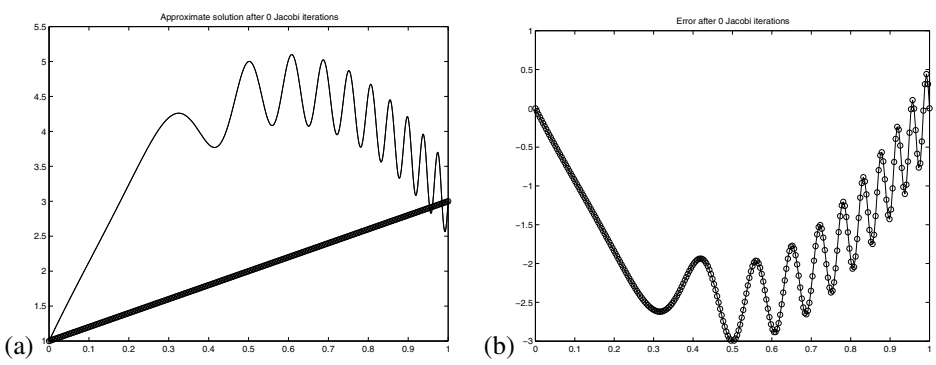

Figure 4.8. (a) *The solution $u(x)$ (solid line) and initial guess u_0 (circles).* (b) *The error e_0 in the initial guess.*

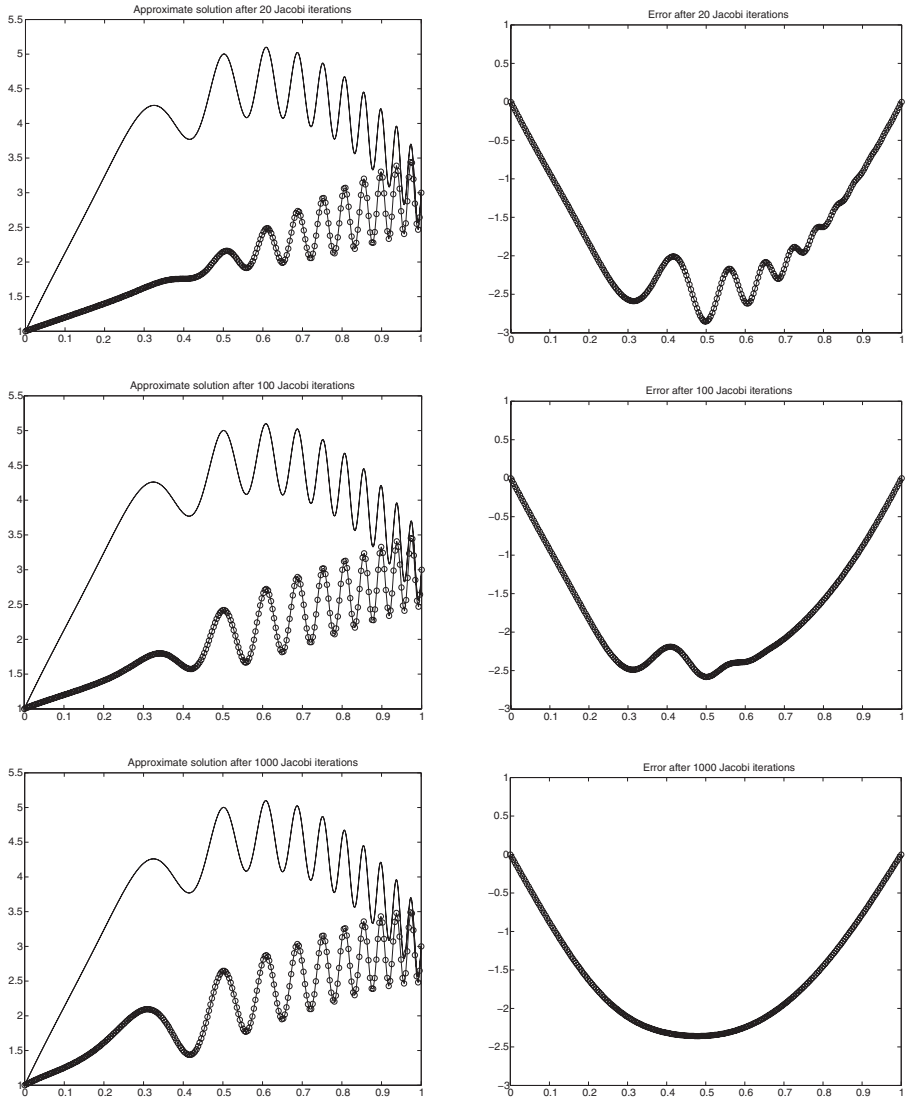

Figure 4.9. *On the left: The solution $u(x)$ (solid line) and Jacobi iterate u_k after k iterations. On the right: The error e_k, shown for $k = 20$ (top), $k = 100$ (middle), and $k = 1000$ (bottom).*

This is easier to understand if we look at the errors shown on the right. The initial error is highly oscillatory but these oscillations are rapidly damped by the Jacobi iteration, and after only 20 iterations the error is much smoother than the initial error. After 100 iterations it is considerably smoother and after 1000 iterations only the smoothest components of the error remain. This component takes nearly forever to be damped out, and it is this component that dominates the error and renders the approximate solution worthless.

To understand why higher frequency components of the error are damped most rapidly, recall from Section 4.2 that the error $e_k = u_k = u^*$ satisfies

$$e_k = Ge_{k-1},$$

where, for the tridiagonal matrix A,

$$G = I + \frac{h^2}{2}A = \begin{bmatrix} 0 & 1/2 & & & & \\ 1/2 & 0 & 1/2 & & & \\ & 1/2 & 0 & 1/2 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1/2 & 0 & 1/2 \\ & & & & 1/2 & 0 \end{bmatrix}.$$

The ith element of e_k is simply obtained by averaging the $(i-1)$ and $(i+1)$ elements of e_{k-1} and this averaging damps out higher frequencies more rapidly than low frequencies. This can be quantified by recalling from Section 4.1 that the eigenvectors of G are the same as the eigenvectors of A. The eigenvector u^p has components

$$u_j^p = \sin(\pi p x_j) \qquad (x_j = jh, \;\; j = 1, 2, \ldots, m), \tag{4.84}$$

while the corresponding eigenvalue is

$$\gamma_p = \cos(p\pi h) \tag{4.85}$$

for $p = 1, 2, \ldots, m$. If we decompose the initial error e_0 into eigencomponents,

$$e_0 = c_1 u^1 + c_2 u^2 + \cdots + c_m u^m, \tag{4.86}$$

then we have

$$e_k = c_1 \gamma_1^k u^1 + c_2 \gamma_2^k u^2 + \cdots + c_m \gamma_m^k u^m. \tag{4.87}$$

Hence the pth eigencomponent decays at the rate γ_p^k as k increases. For large k the error is dominated by the components $c_1\gamma_1^k u^1$ and $c_m \gamma_m^k u^m$, since these eigenvalues are closest to 1:

$$\gamma_1 = -\gamma_m \approx 1 - \frac{1}{2}\pi^2 h^2.$$

This determines the overall convergence rate, as discussed in Section 4.1.

Other components of the error, however, decay much more rapidly. In fact, for half the eigenvectors, those with $m/4 \le p \le 3m/4$, the eigenvalue γ_p satisfies

$$|\gamma_p| \le \frac{1}{\sqrt{2}} \approx 0.7$$

and $|\gamma_p|^{20} < 10^{-3}$, so that 20 iterations are sufficient to reduce these components of the error by a factor of 1000. Decomposing the error e_0 as in (4.86) gives a Fourier sine series representation of the error, since u^p in (4.84) is simply a discretized version of the sine

function with frequency p. Hence eigencomponents $c_p u^p$ for larger p represent higher-frequency components of the initial error e_0, and so we see that higher-frequency components decay more rapidly.

Actually it is the middle range of frequencies, those nearest $p \approx m/2$, that decay most rapidly. The highest frequencies $p \approx m$ decay just as slowly as the lowest frequencies $p \approx 1$. The error e_0 shown in Figure 4.9 has a negligible component of these highest frequencies, however, and we are observing the rapid decay of the intermediate frequencies in this figure.

For this reason Jacobi is not the best method to use in the context of multigrid. A better choice is *underrelaxed Jacobi*, where

$$u_{k+1} = (1-\omega)u_k + \omega G u_k \tag{4.88}$$

with $\omega = 2/3$. The iteration matrix for this method is

$$G_\omega = (1-\omega)I + \omega G \tag{4.89}$$

with eigenvalues

$$\gamma_p = (1-\omega) + \omega\cos(p\pi h). \tag{4.90}$$

The choice $\omega = 2/3$ minimizes $\max_{m/2<p\le m} |\gamma_p|$, giving optimal smoothing of high frequencies. With this choice of ω, all frequencies above the midpoint $p = m/2$ have $|\gamma_p| \le 1/3$.

As a standalone iterative method this would be even worse than Jacobi, since low-frequency components of the error decay even more slowly (γ_1 is now $\frac{1}{3} + \frac{2}{3}\cos(\pi h) \approx 1 - \frac{1}{3}\pi^2 h^2$), but in the context of multigrid this does not concern us. What is important is that the upper half of the range frequencies are all damped by a factor of at least $1/3$ per iteration, giving a reduction by a factor of $(1/3)^3 \approx 0.037$ after only three iterations, for example.

4.6.2 The multigrid approach

We are finally ready to introduce the multigrid algorithm. If we use underrelaxed Jacobi, then after only three iterations the high-frequency components of the error have already decayed significantly, but convergence starts to slow down because of the lower-frequency components. But because the error is now much smoother, we can represent the remaining part of the problem on a coarser grid. The key idea in multigrid is to switch now to a coarser grid to estimate the remaining error. This has two advantages. Iterating on a coarser grid takes less work than iterating further on the original grid. This is nice but is a relatively minor advantage. Much more important, the convergence rate for some components of the error is greatly improved by transferring the error to a coarser grid.

For example, consider the eigencomponent $p = m/4$ that is not damped so much by underrelaxed Jacobi, $\gamma_{m/4} \approx 0.8$, and after three iterations on this grid this component of the error is damped only by a factor $(0.8)^3 = 0.512$. The value $p = m/4$ is not in the upper half of frequencies that can be represented on a grid with m points—it is right in the middle of the lower half.

However, if we transfer this function to a grid with only half as many points, it is suddenly at the halfway point of the frequencies we can represent on the coarser grid

($p \approx m_c/2$ now, where $m_c = (m-1)/2$ is the number of grid points on the coarser grid). Hence this same component of the error is damped by a factor of $(1/3)^3 \approx 0.037$ after only three iterations on this coarser grid. This is the essential feature of multigrid.

But how do we transfer the remaining part of the problem to a coarser grid? We don't try to solve the original problem on a coarser grid. Instead we solve an equation for the error. Suppose we have taken ν iterations on the original grid and now want to estimate the error $e_\nu = u_\nu - u^*$. This is related to the residual vector $r_\nu = f - Au_\nu$ by the linear system

$$Ae_\nu = -r_\nu. \tag{4.91}$$

If we can solve this equation for e_ν, then we can subtract e_ν from u_ν to obtain the desired solution u^*. The system (4.91) is the one we approximate on a coarsened grid. After taking a few iterations of Jacobi on the original problem, we know that e_ν is smoother than the solution u to the original problem, and so it makes sense that we can approximate this problem well on a coarser grid and then interpolate back to the original grid to obtain the desired approximation to e_ν. As noted above, iterating on the coarsened version of this problem leads to much more rapid decay of some components of the error.

The basic multigrid algorithm can be informally described as follows:

1. Take a fixed number of iterations (e.g., $\nu = 3$) of a simple iterative method (e.g., underrelaxed Jacobi or another choice of "smoother") on the original $m \times m$ system $Au = f$. This gives an approximation $u_\nu \in \mathbb{R}^m$.

2. Compute the residual $r_\nu = f - Au_\nu \in \mathbb{R}^m$.

3. Coarsen the residual: approximate the grid function r_ν on a grid with $m_c = (m-1)/2$ points to obtain $\tilde{r} \in \mathbb{R}^{m_c}$.

4. Approximately solve the system $\tilde{A}\tilde{e} = -\tilde{r}$, where $\tilde{A}$ is the $m_c \times m_c$ version of A (the tridiagonal approximation to d^2/dx^2 on a grid with m_c points).

5. The vector $\tilde{e}$ approximates the error in u_ν but only at m_c points on the coarse grid. Interpolate this grid function back to the original grid with m points to obtain an approximation to e_ν. Subtract this from u_ν to get a better approximation to u^*.

6. Using this as a starting guess, take a few more iterations (e.g., $\nu = 3$) of a simple iterative method (e.g., underrelaxed Jacobi) on the original $m \times m$ system $Au = f$ to smooth out errors introduced by this interpolation procedure.

The real power of multigrid comes from recursively applying this idea. In step 4 of the algorithm above we must approximately solve the linear system $\tilde{A}\tilde{e} = -\tilde{r}$ of size m_c. As noted, some components of the error that decayed slowly when iterating on the original system will now decay quickly. However, if m_c is still quite large, then there will be other lower-frequency components of the error that still decay abysmally slowly on this coarsened grid. The key is to recurse. We only iterate a few times on this problem before resorting to a coarser grid with $(m_c - 1)/2$ grid points to speed up the solution to this problem. In other words, the entire algorithm given above is applied within step 4 to solve the linear system $\tilde{A}\tilde{e} = -\tilde{r}$. In a recursive programming language (such as MATLAB) this is not hard to implement and allows one to recurse back as far as possible. If $m + 1$ is a

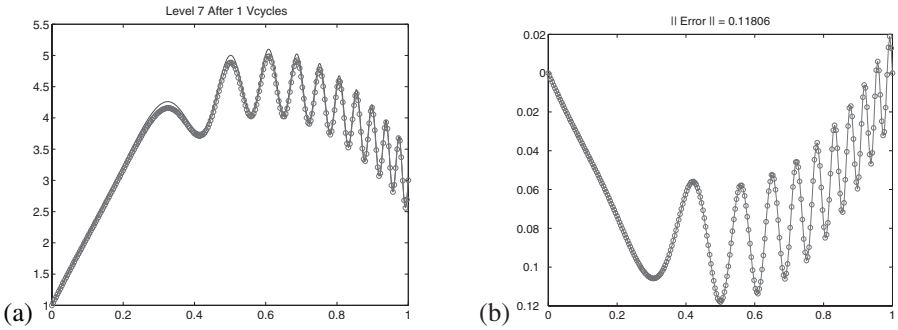

Figure 4.10. (a) *The solution $u(x)$ (solid line) and approximate solution (circles) obtained after one V-cycle of the multigrid algorithm with $\nu = 3$.* (b) *The error in this approximation. Note the change in scale from* Figure 4.9(b).

power of 2, then in principle one could recurse all the way back to a coarse grid with only a single grid point, but in practice the recursion is generally stopped once the problem is small enough that an iterative method converges very quickly or a direct method such as Gaussian elimination is easily applied.

Figure 4.10 shows the results obtained when the above algorithm is used starting with $m = 2^8 - 1 = 255$, using $\nu = 3$, and recursing down to a grid with three grid points, i.e., seven levels of grids. On each level we apply three iterations of underrelaxed Jacobi, do a coarse grid correction, and then apply three more iterations of under-relaxed Jacobi. Hence a total of six Jacobi iterations are used on each grid, and this is done on grids with $2^j - 1$ points for $j = 8, 7, 6, 5, 4, 3, 2$, since the coarse grid correction at each level requires doing this recursively at coarser levels. A total of 42 underrelaxed Jacobi iterations are performed, but most of these are on relatively coarse grids. The total number of grid values that must be updated in the course of these iterations is

$$6 \sum_{j=2}^{8} 2^j \approx 6 \cdot 2^9 = 3072,$$

roughly the same amount of work as 12 iterations on the original grid would require. But the improvement in accuracy is dramatic—compare Figure 4.10 to the results in Figure 4.9 obtained by simply iterating on the original grid with Jacobi.

More generally, suppose we start on a grid with $m + 1 = 2^J$ points and recurse all the way down, taking ν iterations of Jacobi both before and after the coarse grid correction on each level. Then the work is proportional to the total number of grid values updated, which is

$$2\nu \sum_{j=2}^{J} 2^j \approx 4\nu 2^J \approx 4\nu m = O(m). \tag{4.92}$$

Note that this is *linear* in the number of grid points m, although as m increases we are using an increasing number of coarser grids. The number of grids grows at the rate of $\log_2(m)$ but the work on each grid is half as much as the previous finer grid and, so the total work

is $O(m)$. This is the work required for one "V-cycle" of the multigrid algorithm, starting on the finest grid, recursing down to the coarsest grid and then back up as illustrated in Figure 4.11(a) and (b). Taking a single V-cycle often results in a significant reduction in the error, as illustrated in Figure 4.10, but more than one V-cycle might be required to obtain a sufficiently accurate solution. In fact, it can be shown that for this model problem $O(\log(m))$ V-cycles would be needed to reach a given level of error, so that the total work would grow like $O(m \log m)$.

We might also consider taking more that one iteration of the cycle on each of the coarser grids to solve the coarse grid problems within each cycle on the finest grid. Suppose, for example, that we take two cycles at each stage on each of the finer grids. This gives the W-cycle illustrated in Figure 4.11(c).

Even better results are typically obtained by using the "full multigrid" (FMG) algorithm, which consists of starting the process on the coarsest grid level instead of the finest grid. The original problem $u''(x) = f(x)$ is discretized and solved on the coarsest level first, using a direct solver or a few iterations of some iterative method. This approximation to $u(x)$ is then interpolated to the next finer grid to obtain a good initial guess for solving the problem on this grid. The two-level multigrid algorithm is used on this level to solve the problem. The result is then interpolated to the next-level grid to give good initial data there, and so on. By the time we get to the finest grid (our original grid, where we want the solution), we have a very good initial guess to start the multigrid process described above. This process is illustrated using the V-cycle in Figure 4.12.

This start-up phase of the computation adds relatively little work since it is mostly iterating on coarser grids. The total work for FMG with one V-cycle is only about 50% more than for the V-cycle alone. With this initialization process it often turns out that one V-cycle then suffices to obtain good accuracy, regardless of the number of grid points. In

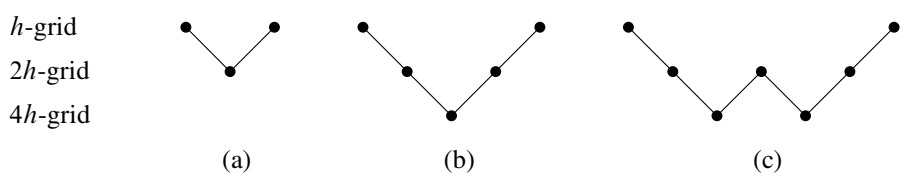

Figure 4.11. (a) *One V-cycle with two levels.* (b) *One V-cycle with three levels.* (c) *One W-cycle with three levels.*

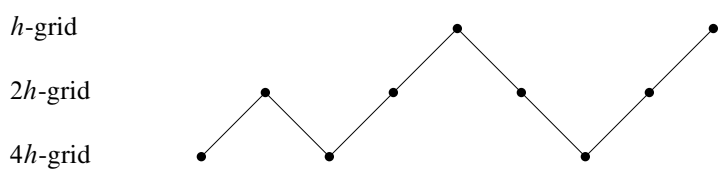

Figure 4.12. *FMG with one V-cycle on three levels.*

this case the total work is $O(m)$, which is optimal. For the example shown in Figure 4.10, switching to FMG gives an error of magnitude 6×10^{-3} after a single V-cycle.

Of course in one dimension simply solving the tridiagonal system requires only $O(m)$ work and is easier to implement, so this is not so impressive. But the same result carries over to more space dimensions. The FMG algorithm for the Poisson problem on an $m \times m$ grid in two dimensions requires $O(m^2)$ work, which is again optimal since there are this many unknowns to determine. Recall that fast Poisson solvers based on the FFT require $O(m^2 \log m)$ work, while the best possible direct method would require (m^3). Applying multigrid to more complicated problems can be more difficult, but optimal results of this sort have been achieved for a wide variety of problems.

The multigrid method described here is intimately linked to the finite difference grid being used and the natural manner in which a vector and matrix can be "coarsened" by discretizing the same differential operator on a coarser grid. However, the ideas of multigrid can also be applied to other sparse matrix problems arising from diverse applications where it may not be at all clear how to coarsen the problem. This more general approach is called *algebraic multigrid* (AMG); see, for example, [76], [86].