Chapter 7

Linear Systems: Iterative Methods

In this chapter we return to the basic problem of Chapter 5, namely, solving a linear system of equations $A\mathbf{x} = \mathbf{b}$. The difference is that here we consider iterative methods for this purpose. The basic premise is that the nonsingular matrix A is large, and yet calculating the product $A\mathbf{v}$ for any given vector $\mathbf{v}$ of appropriate length is relatively inexpensive. Such a situation arises in many applications and typically happens when A is sparse, even if it is not tightly banded as in Section 5.6. In Section 7.1 we further justify the need for considering such problems, giving rise to iterative methods.

In Section 7.2 we describe **stationary** iterative methods, concentrating on some simple instances. The basic, so-called **relaxation** methods described in Section 7.2 are, well, basic. They are usually too slow to compete with more advanced methods in terms of efficiency but are often used instead as building blocks in more complex schemes. We analyze the performance of these methods in Section 7.3.

In Section 7.4 a solution technique for the special case where the matrix A is symmetric positive definite is described. This is the **conjugate gradient** (CG) method, and along the way we also introduce **gradient descent** methods such as steepest descent. These methods often require enhancement in the form of **preconditioning**, and we briefly consider this as well.

Extensions of the CG method to more general cases where the matrix A is not necessarily symmetric positive definite are considered in the more advanced Section 7.5.

Stationary methods do not have to be slow or simple, as is evidenced by the **multigrid method** considered in the more advanced Section 7.6. This family of methods is more context specific, and the exposition here is more demanding, but these methods can be very efficient in practice. The methods of Sections 7.4–7.6 all enjoy heavy, frequent use in practice today.

7.1 The need for iterative methods

Note: The Gaussian elimination algorithm and its variations such as the LU decomposition, the Cholesky method, adaptation to banded systems, etc., is the approach of choice for many problems. Please do not allow anything said in the present chapter to make you forget this.

There are situations that require a treatment that is different from the methods of Chapter 5. Here are a few drawbacks of direct methods:

- As observed in Chapter 5, the Gaussian elimination (or LU decomposition) process may introduce **fill-in**, i.e., L and U may have nonzero elements in locations where the original matrix A has zeros. If the amount of fill-in is significant, then applying the direct method may become costly. This in fact occurs often, in particular when a banded matrix is sparse within its band.

- Sometimes we do not really need to solve the system exactly. For example, in Chapter 9 we will discuss methods for nonlinear systems of equations in which each iteration involves a linear system solve. Frequently, it is sufficient to solve the linear system within the nonlinear iteration only to a low degree of accuracy. Direct methods cannot accomplish this because, by definition, to obtain a solution the process must be completed; there is no notion of an early termination or an inexact (yet acceptable) solution.

- Sometimes we have a pretty good idea of an approximate guess for the solution. For example, in time-dependent problems we may solve a linear system at a certain time level and then move on to the next time level. Often the solution for the previous time level is quite close to the solution in the current time level, and it is definitely worth using it as an initial guess. This is called a **warm start**. Direct methods cannot make good use of such information, especially when A also changes slightly from one time instance to the next.

- Sometimes only matrix-vector products are given. In other words, the matrix is not available explicitly or is very expensive to compute. For example, in digital signal processing applications it is often the case that only input and output signals are given, without the transformation itself being explicitly formulated and available.

Concrete instances leading to iterative methods

We next discuss some relevant instances of a matrix A. As mentioned above, a situation of fill-in arises often. Below we describe one major source of such problems. Many linear systems arising in practice involve values defined on a grid, or a mesh, in two or even three space dimensions. For instance, consider the brightness values at each pixel of your two-dimensional computer monitor screen. The collection of such brightness values forms a two-dimensional *grid function*, $\{u_{i,j},\ i = 1,\ldots,N,\ j = 1,\ldots,M\}$. As another instance, we endow a geophysical domain in three space dimensions with a discretization grid,[26] as depicted in Figure 7.1, before commencing with a computational analysis of its properties (such as conductivity σ). Imagine some function value associated with each of the grid nodes.

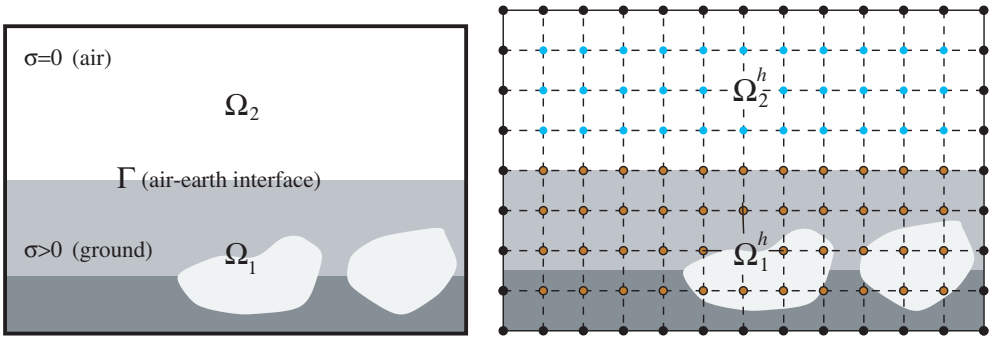
Often there are linear, algebraic relationships between values at neighboring grid nodes.[27] But neighboring locations in a two- or three-dimensional array do not necessarily correspond to consecutive locations when such a structure is reshaped into a one-dimensional array. Let us get more specific. The following case study is long but fundamental.

Example 7.1. An example of such a relationship as described above, arising in many applications, is the famous **Poisson equation**. This is a partial differential equation that in its simplest form is defined on the open *unit square*, $0 < x, y < 1$, and reads

$$-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = g(x,y).$$

[26] People also use the word *mesh* in place of grid. For us these words are synonymous.

[27] *Finite element methods* and *finite difference methods* for partial differential equations give rise to such local algebraic relationships.



(a) An earthly domain. (b) With a discretization grid.

Figure 7.1. *A two-dimensional cross section of a three-dimensional domain with a square grid added. (Reprinted from* [2] *with permission from World Scientific Publishing Co. Pte. Ltd.)*

Here $u(x,y)$ is the unknown function sought and $g(x,y)$ is a given *source*. Further to satisfying the differential equation, the sought function $u(x,y)$ is known to satisfy *homogeneous Dirichlet boundary conditions* along the entire boundary of the unit square, written as

$$u(x,0) = u(x,1) = u(0,y) = u(1,y) = 0.$$

In Example 4.17 we saw a one-dimensional version of this problem, but the present one involves many more issues, both mathematically and computationally. Nonetheless, we hasten to point out that you don't need to be an expert in partial differential equations to follow the exposition below.

Discretizing using centered differences for the partial derivatives we obtain the equations

$$4u_{i,j} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1} = b_{i,j}, \quad 1 \le i,j \le N, \tag{7.1}$$
$$u_{i,j} = 0 \quad \text{otherwise},$$

where $u_{i,j}$ is the value at the $(i,j)th$ node of a square (with $M = N$) planar grid, and $b_{i,j} = h^2 g(ih, jh)$ are given values at the same grid locations. Here $h = 1/(N+1)$ is the *grid width*; see Figure 7.2. The value of N can be fairly large, say, $N = 127$ or even $N = 1023$.

Obviously, these are linear relations, so we can express them as a system of linear equations

$$A\mathbf{u} = \mathbf{b},$$

where $\mathbf{u}$ consists of the $n = N^2$ unknowns $\{u_{i,j}\}$ somehow organized as a vector, and $\mathbf{b}$ is composed likewise from the values $\{b_{i,j}\}$. Notice that the resulting matrix A could be quite large. For example, for $N = 1023$ we have $n = N^2 > 1{,}000{,}000$(!). But what does the matrix A look like? In general this depends on the way we choose to order the grid values, as we see next.

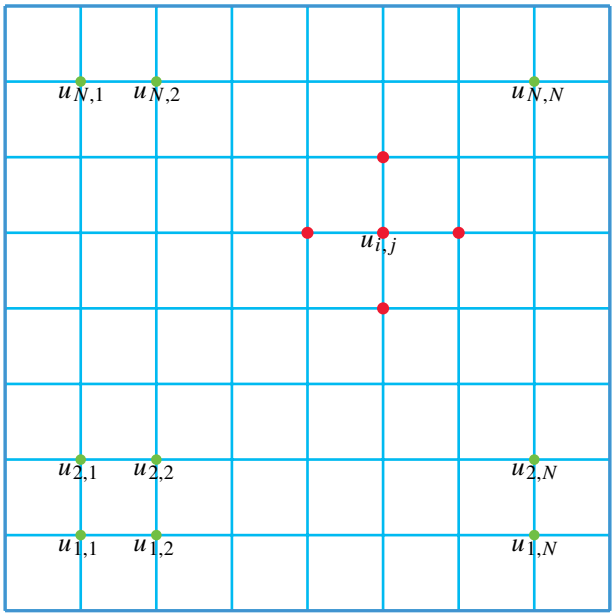


Figure 7.2. *A two-dimensional grid with grid function values at its nodes, discretizing the unit square. The length of each edge of the small squares is the grid width h, while their union is a square with edge length* 1. *The locations of* $u_{i,j}$ *and those of its neighbors that are participating in the difference formula* (7.1) *are marked in red.*

How should we order the grid unknowns $\{u_{i,j}\}_{i,j=1}^{N}$ into a vector $\mathbf{u}$? We can do this in many ways. A simple way is lexicographically, say, by columns, which yields

$$\mathbf{u} = \begin{pmatrix} u_{1,1} \\ u_{2,1} \\ \vdots \\ u_{N,1} \\ u_{1,2} \\ u_{2,2} \\ \vdots \\ u_{N,2} \\ u_{1,3} \\ \vdots \\ u_{N,N} \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} b_{1,1} \\ b_{2,1} \\ \vdots \\ b_{N,1} \\ b_{1,2} \\ b_{2,2} \\ \vdots \\ b_{N,2} \\ b_{1,3} \\ \vdots \\ b_{N,N} \end{pmatrix}.$$

In MATLAB this can be achieved by the instruction `reshape (b,n,1)`, where `b` is an $N \times N$

array and $n = N^2$. This ordering of an array of values in two dimensions into a vector (that is to say, a one-dimensional array), as well as any other ordering of this sort, separates neighbors to some extent. The resulting $n \times n$ matrix A (with $n = N^2$) has the form

$$A = \begin{pmatrix} J & -I & & & \\ -I & J & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & J & -I \\ & & & -I & J \end{pmatrix},$$

where J is the tridiagonal $N \times N$ matrix

$$J = \begin{pmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 4 & -1 \\ & & & -1 & 4 \end{pmatrix},$$

and I denotes the identity matrix of size N. For instance, if $N = 3$, then

$$A = \left(\begin{array}{ccc|ccc|ccc} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\ \hline -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ \hline 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{array}\right).$$

We can see that for any size N the matrix A is diagonally dominant and nonsingular. It can be verified directly that the $n = N^2$ eigenvalues of A are given by

$$\lambda_{l,m} = 4 - 2(\cos(l\pi h) + \cos(m\pi h)), \quad 1 \le l, m \le N$$

(recall $(N+1)h = 1$). Thus $\lambda_{l,m} > 0$ for all $1 \le l, m \le N$, and we see that the matrix A is also positive definite.

The MATLAB command `spy(A)` plots the nonzero structure of a matrix A. For the current example and $N = 10$, we obtain Figure 7.3.

We can see that the matrix remains banded, as we have seen before in Example 4.17, but now there are also $N-2$ zero diagonals *between* the nonzero diagonals. Gaussian elimination without pivoting can be stably applied to this diagonally dominant system, retaining the sparsity

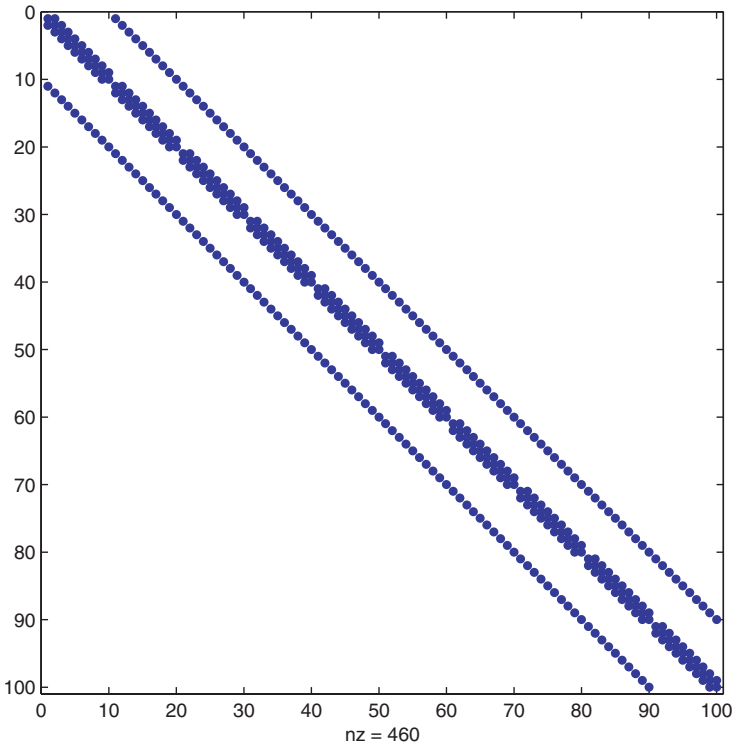


Figure 7.3. *The sparsity pattern of A for Example* 7.1 *with* $N = 10$. *Note that there are* `nz` $= 460$ *nonzeros out of* 10,000 *matrix locations.*

structure of the outer band. However, the inner zero diagonals get replaced, in general, by nonzeros. The leftmost plots in Figures 5.6 and 5.7 (page 126) depict this. It can be seen that the Cholesky decomposition costs $\mathcal{O}(N^4)$ flops (because the matrix is banded, with semibandwidth N) and, perhaps worse, requires $\mathcal{O}(N^3)$ storage locations instead of the original $5N^2$. For $N = 1{,}000$, say, the storage requirement $N^3 = 10^9$ is often prohibitive for a fast memory, whereas $N^2 = 10^6$ is acceptable. ∎

The situation for three-dimensional grids (arising, for instance, from discretizing the Poisson equation in three space variables) is even worse for the direct methods of Chapter 5, because for those problems the semibandwidth is larger (compared to the dimensions of the matrix) and within the band there is a much higher degree of sparsity. Hence we consider here alternative, *iterative methods*. These methods in their basic form do not require the storage of A. They are suitable mainly for special matrices, but such special matrices often arise in practice.

Fixed point iteration

Several methods considered in this chapter can be written in the form of a *fixed point iteration*, which is a vector version of the approach considered in Section 3.3. Thus, for a given linear system of equations

$$A\mathbf{x} = \mathbf{b},$$

rewritten as a *vector equation* $\mathbf{f}(\mathbf{x}) = \mathbf{b} - A\mathbf{x} = \mathbf{0}$, we seek an equivalent form $\mathbf{g}(\mathbf{x}) = \mathbf{x}$, and define the iteration

$$\mathbf{x}_{k+1} = \mathbf{g}(\mathbf{x}_k), \quad k = 0, 1, \ldots,$$

starting from an initial guess $\mathbf{x}_0$. Moreover, the convergence of all these methods is *linear*, and the difference between them in terms of speed of convergence, although it can be significant, is only in the *rate of convergence*.

Specific exercises for this section: Exercises 1–2.

7.2 Stationary iteration and relaxation methods

In this section we consider the solution of our prototype system of linear equations by *stationary* methods, defined below.

Stationary methods using splitting

For a given *splitting* of our matrix A, $A = M - N$, where obviously $N = M - A$, we can write $M\mathbf{x} = N\mathbf{x} + \mathbf{b}$. This leads to a fixed point iteration of the form

$$\mathbf{x}_{k+1} = M^{-1}N\mathbf{x}_k + M^{-1}\mathbf{b} = \mathbf{x}_k + M^{-1}(\mathbf{b} - A\mathbf{x}_k).$$

We can write the iteration as $\mathbf{x}_{k+1} = \mathbf{g}(\mathbf{x}_k)$, where $\mathbf{g}(\mathbf{x}) = \mathbf{x} + M^{-1}(\mathbf{b} - A\mathbf{x})$. It is called stationary because the matrix multiplying $\mathbf{x}$ and the fixed vector appearing in the definition of the function $\mathbf{g}(\mathbf{x})$ are constant and do not depend on the iteration.

In Section 3.3 we use the notation $\mathbf{x}^*$ for the exact solution where $\mathbf{x}^* = \mathbf{g}(\mathbf{x}^*)$, hence $A\mathbf{x}^* = \mathbf{b}$ here. In this chapter, as in the previous three, we drop the superscript * for notational convenience and agree that the notation $\mathbf{x}$, which in the definition of $\mathbf{g}(\mathbf{x})$ above is the independent variable, doubles up as the exact (unknown) solution of the given linear system.

Clearly, the question of how to choose M is central. But before diving into it, let us spend a moment on figuring out how the error behaves in this general setting and how it is related to the residual. We start our iteration with an initial guess $\mathbf{x}_0$. The error is $\mathbf{e}_0 = \mathbf{x} - \mathbf{x}_0$, and the residual equation is $A\mathbf{e}_0 = \mathbf{b} - A\mathbf{x}_0 = \mathbf{r}_0$. Thus, $\mathbf{x} = \mathbf{x}_0 + A^{-1}\mathbf{r}_0$. Obviously, getting $\mathbf{x}$ this way is equivalent to solving $A\mathbf{x} = \mathbf{b}$, so we have nothing to write home about yet. But now we *approximate this error equation*. Instead of solving $A\mathbf{e}_0 = \mathbf{r}_0$, we solve $M\mathbf{p}_0 = \mathbf{r}_0$, where hopefully $\mathbf{p}_0 \approx \mathbf{e}_0$. We can thus write our iteration as calculating for each k the residual $\mathbf{r}_k = \mathbf{b} - A\mathbf{x}_k$ and then computing

$$\mathbf{x}_{k+1} = \mathbf{x}_k + M^{-1}\mathbf{r}_k.$$

This gives us precisely the fixed point iteration written above. It is the form in which we cast all our stationary iterations when implementing them.

Choosing the splitting matrix *M*

The matrix M should be chosen so that on one hand $\mathbf{x}_{k+1}$ can be easily found and on the other hand M^{-1} is "close to A^{-1}." These two seemingly contradictory requirements allow lots of room to play. Different choices of M lead to a variety of methods, from the simple iterative methods described below to very complicated multiresolution ones. Furthermore, by following this strategy we are giving up computing the exact solution $\mathbf{x}$. The matrix M^{-1} is sometimes referred to as an *approximate inverse* of A. Note that we do not have to form M^{-1} in order to apply the iteration—only the solutions of problems of the form $M\mathbf{p}_k = \mathbf{b} - A\mathbf{x}_k$ for $\mathbf{p}_k$ are required (followed by setting $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{p}_k$). For instance, we may design M such that solving for $\mathbf{p}_k$ can be done efficiently

using a direct solution method. This will become obvious below, when we discuss specific choices of M.

Let us denote by D the diagonal matrix consisting of the diagonal elements of A and by E the $n \times n$ lower triangular matrix consisting of corresponding elements of A and zeros elsewhere. In MATLAB you can define these by `D = diag(diag(A)); E = tril(A)`. (Note that in cases where A is not available explicitly, it may not always be possible to actually evaluate these matrices.) Next, we describe methods based on specific choices of M that are related to D and E.

Two basic relaxation methods

The following iterative methods are often referred to as *relaxation methods*. To refresh our memory, we rewrite the system $A\mathbf{x} = \mathbf{b}$ in component form as

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2, \\ \vdots \qquad\qquad\qquad\quad &= \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n. \end{aligned}$$

Let us denote the kth iterate as above by $\mathbf{x}_k$ but use superscripts if the ith component of the vector $\mathbf{x}_k$ is referred to: $x_i^{(k)}$. Thus, $\mathbf{x}_0 = (x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)})^T$ is a given *initial iterate* (or *guess*).

- The **Jacobi method** (*simultaneous relaxation*). We choose $M = D$, yielding the kth iteration

$$\mathbf{x}_{k+1} = \mathbf{x}_k + D^{-1}\mathbf{r}_k.$$

  Thus, assuming we have a routine that returns $A\mathbf{y}$ for any vector $\mathbf{y}$ of suitable size, at each iteration we form the residual $\mathbf{r}_k$ using one call to this routine, scale the residual by D^{-1}, and add to the current iterate.

  In component form, which may be used for understanding but should not be used for computing, we have

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k)} \right], \quad i = 1, \ldots, n.$$

  The basic idea is that for each solution component i in turn, $x_i^{(k)}$ is adjusted to zero out the ith residual (or defect) with all other unknowns kept at their value in iteration k. Note that this can be done simultaneously and therefore also *in parallel* for all i.

  Example 7.2. In practice we will never solve a 3×3 dense linear system using the Jacobi method (or any other iterative method, for that matter), but let us do it here anyway, just for the purpose of illustrating the mechanism of the method. Consider the linear system

$$\begin{aligned} 7x_1 + 3x_2 + x_3 &= 3, \\ -3x_1 + 10x_2 + 2x_3 &= 4, \\ x_1 + 7x_2 - 15x_3 &= 2\,. \end{aligned}$$

Here

$$A = \begin{pmatrix} 7 & 3 & 1 \\ -3 & 10 & 2 \\ 1 & 7 & -15 \end{pmatrix}, \text{ hence } D = \begin{pmatrix} 7 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & -15 \end{pmatrix}. \text{ Also } \mathbf{b} = \begin{pmatrix} 3 \\ 4 \\ 2 \end{pmatrix}.$$

Note that the matrix A is strictly diagonally dominant.

The kth Jacobi iteration in component form reads

$$\begin{aligned} x_1^{(k+1)} &= \frac{3 - 3x_2^{(k)} - x_3^{(k)}}{7}, \\ x_2^{(k+1)} &= \frac{4 + 3x_1^{(k)} - 2x_3^{(k)}}{10}, \\ x_3^{(k+1)} &= \frac{2 - x_1^{(k)} - 7x_2^{(k)}}{-15} \end{aligned}$$

for $k = 0, 1, \ldots$. ∎

Example 7.3. For the model problem of Example 7.1 the Jacobi iteration reads

$$u_{l,m}^{(k+1)} = \frac{1}{4}\left[u_{l+1,m}^{(k)} + u_{l-1,m}^{(k)} + u_{l,m+1}^{(k)} + u_{l,m-1}^{(k)} + b_{l,m}\right],$$

where we *sweep* through the grid, $l, m = 1, \ldots, N$, in some order or in parallel. See Table 7.1 and Figure 7.5. ∎

In MATLAB we can have a function for performing matrix-vector products, say, `y = mvp(A,x)`, which returns for any vector `x` the vector `y = A*x` calculated without necessarily having the matrix `A` stored in full. Assume also that a one-dimensional array `d` has been created which contains the diagonal elements of `A`. Then the Jacobi iteration can be written as

```
r = b - mvp(A,x);
x = x + r./d;
```

- The **Gauss–Seidel method**. We choose $M = E$, yielding the kth iteration

$$\mathbf{x}_{k+1} = \mathbf{x}_k + E^{-1}\mathbf{r}_k.$$

Since E is a lower triangular matrix, once the residual $\mathbf{r}_k$ has been formed a forward substitution produces the vector that gets added to the current iterate $\mathbf{x}_k$ to form the next one $\mathbf{x}_{k+1}$. In component form the iteration reads

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left[b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}\right], \quad i = 1, \ldots, n.$$

Here, newly obtained values are put to use during the same iteration, or sweep, in the hope that $|x_j^{(k+1)} - x_j| < |x_j^{(k)} - x_j|$. Note that, unlike for the Jacobi case, here the order in which the

sweep is made, which corresponds to permuting rows and columns of A before the solution process begins, does matter.

It turns out that under certain conditions, Gauss–Seidel converges whenever the Jacobi iteration converges and (when they do converge) typically twice as fast. But the sweep is harder to execute in parallel.

Example 7.4. Back to Example 7.2, we have

$$E = \begin{pmatrix} 7 & 0 & 0 \\ -3 & 10 & 0 \\ 1 & 7 & -15 \end{pmatrix}.$$

In component form the Gauss–Seidel iteration reads

$$\begin{aligned} x_1^{(k+1)} &= \frac{3 - 3x_2^{(k)} - x_3^{(k)}}{7}, \\ x_2^{(k+1)} &= \frac{4 + 3x_1^{(k+1)} - 2x_3^{(k)}}{10}, \\ x_3^{(k+1)} &= \frac{2 - x_1^{(k+1)} - 7x_2^{(k+1)}}{-15} \end{aligned}$$

for $k = 0, 1, \ldots$. ∎

Example 7.5. Referring again to Example 7.1 for a grid-related case study, the resulting formula depends on the way in which the grid function **u** is ordered into a vector. If this is done by row, as in Example 7.1 (or by column; either way this is called a *lexicographic* ordering), then the resulting formula is

$$u_{l,m}^{(k+1)} = \frac{1}{4}\left[u_{l+1,m}^{(k)} + u_{l-1,m}^{(k+1)} + u_{l,m+1}^{(k)} + u_{l,m-1}^{(k+1)} + b_{l,m}\right],$$

where we *sweep* through the grid in some order, $l, m = 1, \ldots, N$.

A better ordering than lexicographic for this example can be *red-black* or *checkerboard* ordering; see Figure 7.4. Here the neighbors of a "red" grid point are all "black," and vice versa. So, a Gauss–Seidel half-sweep over all red points can be followed by a half-sweep over all black points, and each of these can be done in parallel. The general algorithm, *for this particular sparsity pattern*, can then be written as two Jacobi half-sweeps:

for $i = 1 : 2 : n$

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left[b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k)}\right]$$

end

for $i = 2 : 2 : n$

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left[b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k+1)}\right]$$

end ∎

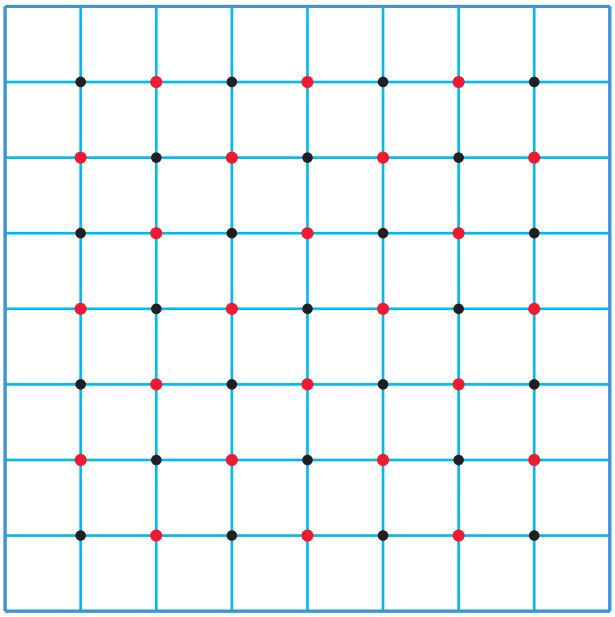


Figure 7.4. *Red-black ordering. First sweep simultaneously over the black points, then over the red.*

Successive over-relaxation

For each of the two basic methods introduced above, we can apply a simple modification leading to an ω**-Jacobi** and an ω**-Gauss–Seidel** scheme. Such a modification is defined, depending on a parameter $\omega > 0$, by replacing at the end of each sweep

$$\mathbf{x}_{k+1} \leftarrow \omega \mathbf{x}_{k+1} + (1-\omega)\mathbf{x}_k.$$

This can of course be worked directly into the definition of $\mathbf{x}_{k+1}$, i.e., the iteration does not have to be executed as a two-stage process.

It turns out that a much faster iteration method can be obtained using ω-Gauss–Seidel if ω is chosen carefully in the range $1 < \omega < 2$. This is the *successive over-relaxation (SOR)* method given by

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left[b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right], \quad i = 1,\ldots,n.$$

In matrix form, we have for SOR the lower triangular matrix $M = \frac{1-\omega}{\omega} D + E$, and the iteration reads

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \omega((1-\omega)D + \omega E)^{-1}\mathbf{r}_k.$$

No corresponding improvement in speed is available when using an ω-Jacobi method.

Example 7.6. Back, yet again, to the linear system of Example 7.2, the SOR iteration reads

$$x_1^{(k+1)} = (1-\omega)x_1^{(k)} + \omega\,\frac{3 - 3x_2^{(k)} - x_3^{(k)}}{7},$$
$$x_2^{(k+1)} = (1-\omega)x_2^{(k)} + \omega\,\frac{4 + 3x_1^{(k+1)} - 2x_3^{(k)}}{10},$$
$$x_3^{(k+1)} = (1-\omega)x_3^{(k)} + \omega\,\frac{2 - x_1^{(k+1)} - 7x_2^{(k+1)}}{-15}$$

for $k = 0, 1, \ldots$. ∎

Example 7.7. For the model problem of Example 7.1, choosing

$$\omega = \frac{2}{1 + \sin(\pi h)}$$

turns out to yield particularly rapid convergence. (See more on the convergence of SOR in Section 7.3, and in particular in Example 7.8.) Let us demonstrate this: we set $N = 15$, $h = 1/16$, and pick the right-hand-side $b_{l,m} = h^2$, $l, m = 1, \ldots, N$. Note that $n = N^2 = 225$, i.e., our matrix A is 225×225. Then we start iterating from the grid function $\mathbf{u}_0$ given by

$$u_{l,m}^{(0)} = 0, \quad l = 1, \ldots, N,\ m = 1, \ldots, N.$$

Let us denote as before by $\mathbf{u}$ and $\mathbf{b}$ the corresponding grid functions reshaped as vectors. The maximum errors $\|\mathbf{x}_k - \mathbf{x}\|_\infty = \|\mathbf{u}_k - \mathbf{u}\|_\infty$ are recorded in Table 7.1. Clearly the errors for the SOR method with $\omega \approx 1.67$ are significantly smaller than for the other two, with Gauss–Seidel being somewhat better per iteration than Jacobi (ignoring other implementation considerations). ∎

Table 7.1. *Errors for basic relaxation methods applied to the model problem of Example* 7.1 *with* $n = N^2 = 225$.

Relaxation method	ω	Error after 2 iterations	Error after 20 iterations
Jacobi	1	7.15e-2	5.41e-2
Gauss–Seidel	1	6.95e-2	3.79e-2
SOR	1.67	5.63e-2	7.60e-4

Assessing SOR and other relaxation possibilities

The beauty of SOR is that it is just as simple as Gauss–Seidel, and yet a significant improvement in convergence speed is obtained. Who said there is no free lunch?!

For parameter values in the range $0 < \omega < 1$, the corresponding ω-Jacobi and ω-Gauss–Seidel iterations are called *under-relaxed* or *damped*. For instance, the damped Jacobi method reads

$$\mathbf{x}_{k+1} = \omega(\mathbf{x}_{Jacobi})_{k+1} + (1-\omega)\mathbf{x}_k = \mathbf{x}_k + \omega D^{-1}\mathbf{r}_k.$$

You will see in Exercise 11 that there is no magic whatsoever in this formula, and the fastest convergence is actually obtained for $\omega = 1$, which is nothing but standard Jacobi. But interestingly,

damped Jacobi does have merit in a different context, as a *smoother*. For more on this, please see Section 7.6 and especially Figure 7.10.

Unfortunately, in more realistic situations than Example 7.7, the precise optimal value of the SOR parameter ω (i.e., the value yielding the fastest convergence) is not known in advance. A common practice then is to try a few iterations with various values of ω and choose the most promising one for the rest of the iteration process. The results typically offer a significant improvement over Gauss–Seidel, but the spectacular SOR performance depicted in Table 7.1 is rarely realized. We may then turn to other options, such as using the methods described in Section 7.4.

Before moving on let us also mention the *symmetric SOR* (SSOR) method. One iteration of SSOR consists of an SOR sweep in one direction followed by another SOR sweep in the reverse direction. The computational cost of each iteration doubles as a result, compared to SOR, but the iteration count may go down significantly, with the advantage that the iteration is less biased in terms of the sweep direction.

Specific exercises for this section: Exercises 4–5.

7.3 Convergence of stationary methods

For the purpose of analyzing the iterative methods introduced in Section 7.2 it is useful to consider their matrix representation. As before, at each iteration $\mathbf{x}_k$ the residual

$$\mathbf{r}_k = \mathbf{b} - A\mathbf{x}_k$$

is known, whereas the error

$$\mathbf{e}_k = \mathbf{x} - \mathbf{x}_k = A^{-1}\mathbf{r}_k$$

is unknown. We wish to be assured that the iteration converges, i.e., that $\mathbf{e}_k \to 0$ as $k \to \infty$.

The iteration matrix and convergence

As in the analysis for the fixed point iteration of Section 3.3, we write

$$\begin{aligned} \mathbf{x}_k &= M^{-1}\mathbf{b} + (I - M^{-1}A)\mathbf{x}_{k-1}, \\ \mathbf{x} &= M^{-1}\mathbf{b} + (I - M^{-1}A)\mathbf{x} \end{aligned}$$

and subtract. Denoting $T = I - M^{-1}A$, this yields

$$\begin{aligned} \mathbf{e}_k &= T\mathbf{e}_{k-1} \\ &= T(T\mathbf{e}_{k-2}) = \cdots = T^k\mathbf{e}_0. \end{aligned}$$

The matrix T is the *iteration matrix*. Since $\mathbf{e}_0$ is a fixed initial error, we have convergence, i.e., $\mathbf{e}_k \to \mathbf{0}$ as $k \to \infty$, if and only if $T^k \to 0$. This is clearly the case if in any induced matrix norm

$$\|T\| < 1,$$

because

$$\|\mathbf{e}_k\| = \|T \cdot T \cdots T\mathbf{e}_0\| \le \|T\|\|T\| \cdots \|T\|\|\mathbf{e}_0\| = \|T\|^k\|\mathbf{e}_0\|.$$

Compare this to the condition of convergence for the fixed point iteration in Section 3.3.

Proceeding more carefully, it turns out that convergence depends on the spectral radius of T (defined on page 77) rather than its norm. The resulting important theorem is given on the following page. Let us see why it holds in the case where T has n linearly independent eigenvectors $\mathbf{v}_i$. Then

Theorem: Stationary Method Convergence.
For the linear problem $A\mathbf{x} = \mathbf{b}$, consider the iterative method

$$\mathbf{x}_{k+1} = \mathbf{x}_k + M^{-1}\mathbf{r}_k, \quad k = 0, 1, \ldots,$$

and define the **iteration matrix** $T = I - M^{-1}A$.

Then the method converges if and only if the spectral radius of the iteration matrix satisfies

$$\rho(T) < 1.$$

The smaller $\rho(T)$ the faster the convergence.

it is possible to write $\mathbf{e}_0 = \sum_{i=1}^{n} \gamma_i \mathbf{v}_i$ for some coefficients $\gamma_1, \ldots, \gamma_n$. So, if the corresponding eigenvalues τ_i of T satisfy

$$T\mathbf{v}_i = \tau_i \mathbf{v}_i, \ |\tau_i| < 1, \quad i = 1, 2, \ldots, n,$$

then

$$T^k \mathbf{e}_0 = \sum_{i=1}^{n} \gamma_i \tau_i^k \mathbf{v}_i \to \mathbf{0}.$$

On the other hand, if there is an eigenvalue of T, say, τ_1, satisfying $|\tau_1| \geq 1$, then for the unlucky initial guess $\mathbf{x}_0 = \mathbf{x} - \mathbf{v}_1$ we get $\mathbf{e}_k = T^k \mathbf{v}_1 = \tau_1^k \mathbf{v}_1$, so there is no convergence. ♦

Of course, if $\|T\| < 1$, then also $\rho(T) < 1$. But the condition on the spectral radius is more telling than the condition on the norm, being necessary and not only sufficient for convergence of the iteration.

Convergence rate of a stationary method

Since T is independent of the iteration counter k, convergence of this general, stationary fixed point iteration is *linear* and not better (in contrast to the quadratic convergence of Newton's method in Section 3.4, for instance). How many iterations are then needed to reduce the error norm by a fixed factor, say, 10, thus reducing the error by an order of magnitude? In Section 3.3 we defined the **rate of convergence** in order to quantify this. Here, writing

$$0.1\|\mathbf{e}_0\| \approx \|\mathbf{e}_k\| \approx \rho(T)^k \|\mathbf{e}_0\|$$

and taking $\log_{10}$ of these approximate relations yields that $-1 \approx k \ \log_{10} \rho(T)$, so

$$k \approx -\frac{1}{\log_{10} \rho(T)}$$

iterations are needed. Thus, define the rate of convergence by

$$rate = -\log_{10} \rho(T).$$

Then $k \approx 1/rate$. The smaller the spectral radius, the larger the rate, and the fewer iterations are needed to achieve the same level of error reduction.

Convergence of the relaxation methods

What can generally be said about the convergence of the basic relaxation methods defined in Section 7.2? They certainly *do not* converge for just any nonsingular matrix A. In particular, they are not even well-defined if $a_{ii} = 0$ for some i. But they converge if A is strictly diagonally dominant.

Example 7.8. Consider again the model problem presented in Example 7.1. The matrix A is symmetric positive definite, but it is not strictly diagonally dominant. Let us consider further the Jacobi iteration. Since $M = D$ we get for the model problem the iteration matrix

$$T = I - D^{-1}A = I - \frac{1}{4}A.$$

Therefore, with $N = \sqrt{n}$ and $h(N+1) = 1$, the eigenvalues are

$$\mu_{l,m} = 1 - \frac{1}{4}\lambda_{l,m} = \frac{1}{2}\left(\cos(l\pi h) + \cos(m\pi h)\right), \quad 1 \le l,m \le N\,.$$

The spectral radius is

$$\rho(T) = \mu_{1,1} = \cos(\pi h) \le 1 - \frac{c}{N^2}$$

for some positive constant c.[28] But this convergence is very slow, because $rate = -\log\rho(T) \sim N^{-2}$. Thus, $\mathcal{O}(N^2) = \mathcal{O}(n)$ iterations are required to reduce the iteration error by a constant factor. Since each sweep (iteration) costs $\mathcal{O}(n)$ flops, the total cost is $\mathcal{O}(n^2)$, comparable to that of banded Gaussian elimination.

For the Gauss–Seidel relaxation it can be shown, for this simple problem, that the spectral radius of T is precisely squared that of T of the Jacobi relaxation. So, the rate is twice as large. But still, $\mathcal{O}(n)$ iterations, therefore $\mathcal{O}(n^2)$ flops, are required for a fixed error reduction factor.

On the other hand, the SOR method demonstrated in Example 7.7 can be shown theoretically to require only $\mathcal{O}(N)$ iterations to reduce the iteration error by a constant factor when the optimal parameter ω is used. As it turns out, for matrices like the discretized Laplacian there is a formula for the optimal parameter. It is given by

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

where ρ_J stands for the spectral radius of the Jacobi iteration matrix. For this problem we have in fact already specified the optimal SOR parameter; see Example 7.7. The spectral radius of the iteration matrix for SOR with ω_{opt} is $\rho = \omega_{opt} - 1$. Taking into account the expense per iteration, the total SOR cost is therefore $\mathcal{O}(n^{3/2})$ flops. ∎

Terminating the iteration

How should we terminate the iteration, for any of these methods, given a convergence error tolerance?

One criterion, often used in practice, is to require that the relative residual be small enough, written as

$$\|\mathbf{r}_k\| \le \mathtt{tol}\|\mathbf{b}\|.$$

[28] Note that $\rho(T) = -\mu_{N,N}$ as well.

Recall from Equation (5.1), however, that this implies

$$\frac{\|\mathbf{x}-\mathbf{x}_k\|}{\|\mathbf{x}\|} \le \kappa(A)\frac{\|\mathbf{r}_k\|}{\|\mathbf{b}\|} \le \kappa(A)\,\texttt{tol}.$$

Thus, the tolerance must be taken suitably small if the condition number of A is large.

If we wish to control the algorithm by measuring the difference between two consecutive iterations, $\|\mathbf{x}_k-\mathbf{x}_{k-1}\|$, then care must be taken when the convergence is slow. For any vector norm and its induced matrix norm we can write

$$\begin{aligned}\|\mathbf{x}-\mathbf{x}_k\| &\le \|T\|\|\mathbf{x}-\mathbf{x}_{k-1}\| = \|T\|\|\mathbf{x}-\mathbf{x}_k+\mathbf{x}_k-\mathbf{x}_{k-1}\| \\ &\le \|T\|\left[\|\mathbf{x}-\mathbf{x}_k\|+\|\mathbf{x}_k-\mathbf{x}_{k-1}\|\right].\end{aligned}$$

Thus we only have for the error that

$$\|\mathbf{e}_k\| = \|\mathbf{x}-\mathbf{x}_k\| \le \frac{\|T\|}{1-\|T\|}\|\mathbf{x}_k-\mathbf{x}_{k-1}\|.$$

So, the tolerance on $\|\mathbf{x}_k-\mathbf{x}_{k-1}\|$ must be adjusted taking $\frac{\|T\|}{1-\|T\|}$ into account. For slowly convergent iterations this number can be quite large, as can be seen for Jacobi and Gauss–Seidel (and even SOR) applied to Example 7.1. Thus, care must be exercised when selecting a stopping criterion for the iterative method.

Specific exercises for this section: Exercises 6–12.

7.4 Conjugate gradient method

A weakness of stationary methods is that information gathered throughout the iteration is not fully utilized. Indeed, the approximate inverse M^{-1} is fixed. We might be able to do better, for example, by setting a varying splitting in each iteration, so $M = M_k$, and requiring a certain optimality property to hold. Such methods are explored in this section and the next one.

Note: We assume throughout this section that A is a symmetric positive definite matrix. Methods for general nonsymmetric matrices are presented in the more advanced Section 7.5.

The methods considered in this section can all be written as

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k,$$

where the vector $\mathbf{p}_k$ is the **search direction** and the scalar α_k is the **step size**. Note that this includes basic stationary methods, since such methods with an associated splitting $A = M - N$ can be written as above with $\alpha_k \equiv 1$ for all k and $\mathbf{p}_k = M^{-1}\mathbf{r}_k$.

Our eventual goal is to introduce the celebrated *conjugate gradient (CG) method*. To get there, though, it is natural to start with the general family of gradient descent methods and, in particular, the method of steepest descent.

Gradient descent methods

The simplest nonstationary scheme based on setting a varying search direction and step size is obtained by setting $\mathbf{p}_k = \mathbf{r}_k$, i.e., $M_k^{-1} = \alpha_k I$, with I the identity matrix. The resulting family of methods is called *gradient descent*.

A popular choice for the step size α_k for gradient descent methods is available by observing that our problem $A\mathbf{x} = \mathbf{b}$ is equivalent to the problem of finding a vector $\mathbf{x}$ that minimizes

$$\phi(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T\mathbf{x}.$$

Let us explain why this is so. In the scalar case of Section 3.4 we obtained critical points by setting the first derivative of the function to be optimized to zero, knowing that such a critical point is a minimizer if the second derivative is positive there. Here, similarly, a critical point is obtained by setting the *gradient*, i.e., the vector of first derivatives of ϕ, to zero. But, reminiscent to the process described in Section 6.1, the gradient of this particular function is $\nabla\phi = A\mathbf{x} - \mathbf{b} = -\mathbf{r}$, so setting this to $\mathbf{0}$ yields our problem

$$A\mathbf{x} = \mathbf{b}.$$

Moreover, the matrix of second derivatives of ϕ is A, and its assumed positive definiteness guarantees that the solution of our problem is the unique minimizer of ϕ.

To summarize our progress so far, we are looking at the gradient descent iteration

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{r}_k$$

and wish to determine the scalar α_k such that $\phi(\mathbf{x}_k + \alpha\mathbf{r}_k)$ is minimized over all values of α. But this is straightforward: we want to minimize over α the expression

$$\frac{1}{2}(\mathbf{x}_k + \alpha\mathbf{r}_k)^T A(\mathbf{x}_k + \alpha\mathbf{r}_k) - \mathbf{b}^T(\mathbf{x}_k + \alpha\mathbf{r}_k),$$

and hence all we need to do is set its derivative with respect to α to 0. This translates into

$$\alpha\mathbf{r}_k^T A\mathbf{r}_k + \mathbf{r}_k^T A\mathbf{x}_k - \mathbf{r}_k^T\mathbf{b} = 0.$$

Since $A\mathbf{x}_k - \mathbf{b} = -\mathbf{r}_k$, it readily follows that the desired minimizer $\alpha = \alpha_k$ is given by

$$\alpha_k = \frac{\mathbf{r}_k^T\mathbf{r}_k}{\mathbf{r}_k^T A\mathbf{r}_k} = \frac{\langle \mathbf{r}_k, \mathbf{r}_k\rangle}{\langle \mathbf{r}_k, A\mathbf{r}_k\rangle},$$

where $\langle\cdot,\cdot\rangle$ stands for the standard inner product: for any two vectors $\mathbf{p}$ and $\mathbf{q}$ of the same length, $\langle \mathbf{p}, \mathbf{q}\rangle = \mathbf{p}^T\mathbf{q}$.

This completes the description of an important basic method called **steepest descent** for the iterative solution of symmetric positive definite linear systems. In practice, when we set up the iteration, one point to keep in mind is computational efficiency. Here the basic cost of the iteration is just one matrix-vector multiplication: at the kth iteration, knowing $\mathbf{x}_k$ and $\mathbf{r}_k$, we compute $\mathbf{s}_k = A\mathbf{r}_k$, set $\alpha_k = \langle \mathbf{r}_k, \mathbf{r}_k\rangle / \langle \mathbf{r}_k, \mathbf{s}_k\rangle$, and calculate $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{r}_k$ and $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k\mathbf{s}_k$.

Slowness of steepest descent

Unfortunately, the steepest descent method tends to be slow: the iteration count that it takes to reduce the norm of the error by a constant factor grows *linearly* with the condition number. In our prototype Example 7.1, $\kappa(A)$ is proportional to n, which puts the steepest descent method in the same efficiency class as the basic relaxation methods of Section 7.2. This pronouncement is a bit unfair, because steepest descent is still typically faster in terms of the overall iteration count and can be further improved by preconditioning, discussed later on. Also, there are choices of step sizes which for certain problems have been experimentally observed to converge faster; see Exercise 14 for an example. However, better alternatives than any gradient descent method do exist, so we turn to them without further ado.

The CG method

The immensely popular CG method for solving large linear systems $A\mathbf{x} = \mathbf{b}$, where A is symmetric positive definite, is given on the current page.

Algorithm: Conjugate Gradient.
Given an initial guess $\mathbf{x}_0$ and a tolerance `tol`, set at first $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$, $\delta_0 = \langle \mathbf{r}_0, \mathbf{r}_0 \rangle$, $b_\delta = \langle \mathbf{b}, \mathbf{b} \rangle$, $k = 0$ and $\mathbf{p}_0 = \mathbf{r}_0$. Then:

while $\delta_k > \texttt{tol}^2\, b_\delta$
$\mathbf{s}_k = A\mathbf{p}_k$
$\alpha_k = \dfrac{\delta_k}{\langle \mathbf{p}_k, \mathbf{s}_k \rangle}$
$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$
$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{s}_k$
$\delta_{k+1} = \langle \mathbf{r}_{k+1}, \mathbf{r}_{k+1} \rangle$
$\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \dfrac{\delta_{k+1}}{\delta_k} \mathbf{p}_k$
$k = k + 1$
end

earch direction $\mathbf{p}_k$ no longer equals the residual vector $\mathbf{r}_k$, except at the first iteration. Similarly to the simpler looking gradient descent method, only one matrix-vector multiplication involving A is needed per iteration. The matrix A need not be explicitly available: only a procedure to (rapidly) evaluate the matrix-vector product $A\mathbf{v}$ for any given real vector $\mathbf{v}$ of length n is required. The iteration is stopped as soon as the relative residual reaches the value of input tolerance `tol` or goes below it; see the discussion on page 181.

Example 7.9. Much as we would like to return to Example 7.2 yet again, we cannot, because the matrix A there is not symmetric positive definite. So consider another 3×3 problem $A\mathbf{x} = \mathbf{b}$, with

$$A = \begin{pmatrix} 7 & 3 & 1 \\ 3 & 10 & 2 \\ 1 & 2 & 15 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 28 \\ 31 \\ 22 \end{pmatrix}.$$

The exact solution is $\mathbf{x} = A^{-1}\mathbf{b} = (3,2,1)^T$. The point here, as in any 3×3 example, is to demonstrate the algorithm—not to promote it as a suitable method for a trivial problem. We next follow the execution of the CG algorithm. Below we do the calculations to rounding unit but display only a few leading digits.

At first set $\mathbf{x}_0 = (0,0,0)^T$, hence $\mathbf{p}_0 = \mathbf{r}_0 = \mathbf{b}$ and $b_\delta = \delta_0 = \|\mathbf{r}_0\|^2 = 2229$. Next, $\mathbf{s}_0 = A\mathbf{p}_0 = (311, 438, 420)^T$, $\alpha_0 = 2229/(\mathbf{p}_0^T \mathbf{s}_0) = .0707$, $\mathbf{x}_1 = \mathbf{0} + \alpha_0 \mathbf{p}_0 = (1.9797, 2.1918, 1.555)^T$, $\mathbf{r}_1 = \mathbf{b} - \alpha_0 \mathbf{s}_0 = (6.0112, .031847, -7.6955)^T$, $\delta_1 = \|\mathbf{r}_1\|^2 = 95.356$, $\mathbf{p}_1 = (7.209, 1.358, -6.7543)^T$.

On to the second iteration, $\mathbf{s}_1 = (47.783, 21.699, -91.39)^T$, $\alpha_1 = .0962$, $\mathbf{x}_2 = (2.6732, 2.3225, .9057)^T$, $\mathbf{r}_2 = (1.4144, -2.0556, 1.0963)^T$, $\delta_2 = 7.428$, $\mathbf{p}_2 = (1.976, -1.9498, .5702)^T$. Note the decrease in δ from one iteration to the next.

On to the third iteration, $\mathbf{s}_2 = (8.5527, -12.43, 6.6293)^T$, $\alpha_2 = .1654$, $\mathbf{x}_3 = (3,2,1)^T$, $\mathbf{r}_3 = \mathbf{0}$. We have found the exact solution in $n = 3$ CG iterations! This is actually not a coincidence; it is rooted in a well-understood convergence property of CG which will be described soon. ∎

Before developing further insight into the CG algorithm, let us consider a larger numerical example where the potential of this iterative method is demonstrated.

Example 7.10. For our prototype Example 7.1, with $N = 31$ (i.e., a linear system of size $n = N^2 = 961$), we compute the solution using Jacobi, Gauss–Seidel, SOR, and CG. The iteration is stopped when a relative residual norm $\frac{\|\mathbf{r}_k\|}{\|\mathbf{b}\|}$ smaller than 10^{-6} is reached. The convergence history is depicted in two separate plots (see Figure 7.5), where SOR appears in both. This is done because it is difficult to visualize everything on one plot, due to the very different scale of convergence between the slowly convergent Jacobi and Gauss–Seidel schemes, and the rest. Indeed, these two simple relaxation techniques entail an iteration count proportional to N^2 and are clearly not competitive here.

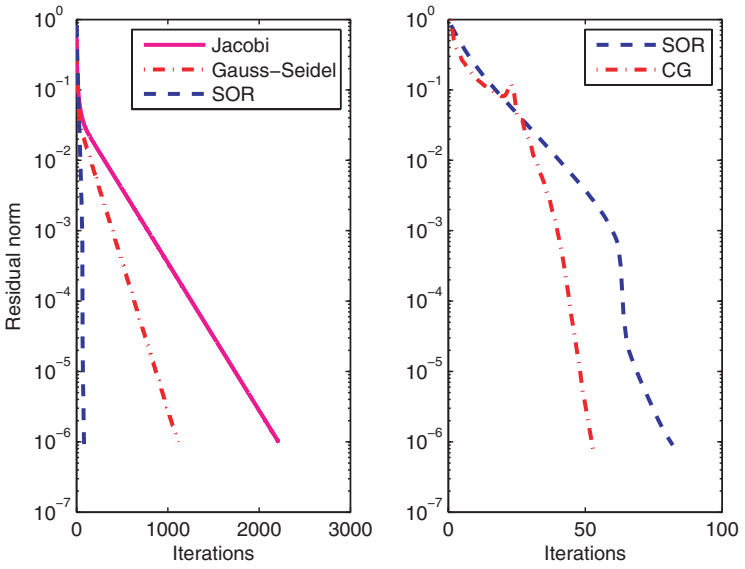


Figure 7.5. *Example* 7.10, *with* $N = 31$*: convergence behavior of various iterative schemes for the discretized Poisson equation.*

The right-hand plot shows that in terms of iteration count CG is better than, yet comparable to, SOR with its parameter set at optimal value. Theory implies that they should both require $\mathcal{O}(N)$ iterations to converge to within a fixed tolerance. But in practice the CG iteration, although more mysterious and cumbersome looking, is cheaper overall than the SOR iteration. CG has another advantage over SOR in that it does not require a special parameter. No wonder it is popular.

We emphasize that only about 50 CG iterations, far fewer than the dimension of the matrix A, are required here to achieve decent accuracy. We also note that in general, the relative residual norm does not necessarily decrease monotonically for CG. ∎

CG magic

The basic idea behind the CG method is to minimize the *energy norm* of the error, defined on the next page, with respect to $B = A$ (or, equivalently, the energy norm of the residual with respect to $B = A^{-1}$) over a *Krylov subspace*, defined on the following page. This subspace grows with each additional iteration k.

Energy Norm.
Given a symmetric positive definite matrix B, its associated energy norm is

$$\|\mathbf{x}\|_B = \sqrt{\mathbf{x}^T B\mathbf{x}} \equiv \sqrt{\langle \mathbf{x}, B\mathbf{x}\rangle}.$$

Krylov Subspace.
For any real, $n \times n$ matrix C and vector $\mathbf{y}$ of the same length, the Krylov Subspace of C with respect to $\mathbf{y}$ is defined by

$$\mathcal{K}_k(C;\mathbf{y}) = \text{span}\{\mathbf{y}, C\mathbf{y}, C^2\mathbf{y}, \ldots, C^{k-1}\mathbf{y}\}.$$

The CG iteration starts by setting the search direction as the direction of gradient descent, $\mathbf{r}_0$, and its first iteration coincides with steepest descent. We then proceed in the kth iteration to minimize the energy norm of the error, seeking a parameter α such that

$$\|\mathbf{x}_k + \alpha\mathbf{p}_k - \mathbf{x}\|_A$$

is minimized. But the real magic lies in the choice of the search directions. In each iteration the method minimizes the energy norm not only for the current iterate but rather for the space spanned by the current search direction and all the ones preceding it. This is achieved by making the search directions A-conjugate, meaning

$$\langle \mathbf{p}_l, A\mathbf{p}_j\rangle = 0, \quad l \neq j.$$

Imposing this requirement does not seem easy at first, but it turns out that such a search direction can be expressed quite simply as a linear combination of the previous one and the current residual. The search directions satisfy

$$\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \frac{\langle \mathbf{r}_{k+1}, \mathbf{r}_{k+1}\rangle}{\langle \mathbf{r}_k, \mathbf{r}_k\rangle}\mathbf{p}_k,$$

and the minimizing parameter turns out to be

$$\alpha_k = \frac{\langle \mathbf{r}_k, \mathbf{r}_k\rangle}{\langle \mathbf{p}_k, A\mathbf{p}_k\rangle}.$$

These formulas both appear in the algorithm definition on page 184. It can be verified that the residual vectors are all orthogonal to one another, i.e., $\langle \mathbf{r}_l, \mathbf{r}_j\rangle = 0$, $l \neq j$.

It can be easily shown (simply trace the recursion back from $\mathbf{x}_k$ to $\mathbf{x}_0$) that the CG iterates satisfy

$$\mathbf{x}_k - \mathbf{x}_0 \in \mathcal{K}_k(A;\mathbf{r}_0),$$

where $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$ is the initial residual.

The minimization property in the energy norm leads to the conclusion that the iteration will terminate at most after n iterations, in the absence of roundoff errors. This explains why in Example 7.9 only $n = 3$ iterations were needed to obtain the exact solution using CG. We note though that in practice the presence of roundoff errors may make things much worse than what this theory predicts.

But waiting n iterations is out of the question, since n is typically large. The practically more important feature of CG is that the first "few" iterations already achieve much progress, as we have seen in Example 7.10 and Figure 7.5. If the eigenvalues of A are located in only a few narrow

clusters, then the CG method requires only a few iterations to converge. Convergence is slower, though, if the eigenvalues are widely spread.

A bound on the error can be expressed in terms of the condition number $\kappa(A)$ of the matrix, as the theorem on the current page shows. Hence, for large κ the number of iterations k required to reduce the initial error by a fixed factor is bounded by $\mathcal{O}(\sqrt{\kappa})$, see Exercise 17. For Example 7.1 we obtain that $\mathcal{O}(\sqrt{n}) = \mathcal{O}(N)$ iterations are required, a rather significant improvement over steepest descent. See also Example 9.7 and Figure 9.6. We note that the error bound given in the CG Convergence Theorem is often quite pessimistic; in practice CG often behaves better than predicted by that bound.

Theorem: CG Convergence.
For the linear problem $A\mathbf{x} = \mathbf{b}$ with A symmetric positive definite, let S_k consist of all n-vectors that can be written as $\mathbf{x}_0 + \mathbf{w}$ with $\mathbf{w} \in \mathcal{K}_k(A; \mathbf{r}_0)$. Then the following hold:

- The kth iterate of the CG method, $\mathbf{x}_k$, minimizes $\|\mathbf{e}_k\|_A$ over S_k.
- In exact arithmetic the solution $\mathbf{x}$ is obtained after at most n iterations. Furthermore, if A has just m distinct eigenvalues then the number of iterations is at most m.
- Only $\mathcal{O}(\sqrt{\kappa(A)})$ iterations are required to reduce the error norm $\|\mathbf{e}_k\|_A$ by a fixed amount, with the error bound

$$\|\mathbf{e}_k\|_A \le 2\left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^k \|\mathbf{e}_0\|_A.$$

While there are several operations in every iteration, it is really the number of matrix-vector multiplications that is typically the dominant factor in the computation. Thus, it is important to observe, yet again, that *one* such multiplication is required per CG iteration.

Preconditioning

If we take $N = \sqrt{n}$ larger in Example 7.10 above, say, $N = 1023$ (corresponding to a realistic screen resolution for instance), then even the CG method requires over 1000 iterations to achieve a decently small convergence error. This is because the number of iterations depends on $\sqrt{\kappa(A)}$, which is $\mathcal{O}(N)$ in that example. In general, if the condition number of A is very large, then CG loses its effectiveness. In such a case it is worthwhile to attempt to use CG to solve a related problem with a matrix that is better conditioned or whose eigenvalues are more tightly clustered. This gives rise to a **preconditioned conjugate gradient** (PCG) method. Let P^{-1} be a symmetric positive definite approximate inverse of A. The CG iteration is then applied to the system

$$P^{-1}A\mathbf{x} = P^{-1}\mathbf{b}.$$

Note that $P^{-1}A$ is not symmetric in general even if P and A are. Strictly speaking, we should reformulate $P^{-1}A\mathbf{x} = P^{-1}\mathbf{b}$ as

$$(P^{-1/2}AP^{-1/2})(P^{1/2}\mathbf{x}) = P^{-1/2}\mathbf{b}$$

to get a linear system with a symmetric positive definite matrix. But this turns out not to be really necessary. The resulting method is given in an algorithm form on the following page. The stopping

criterion is again based on the relative residual, as per the discussion on page 181, except that this time the linear system for which the relative residual is computed is $P^{-1}A\mathbf{x} = P^{-1}\mathbf{b}$ rather than $A\mathbf{x} = \mathbf{b}$.

Algorithm: Preconditioned Conjugate Gradient.
Given an initial guess $\mathbf{x}_0$ and a tolerance `tol`, set at first $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$, $\mathbf{h}_0 = P^{-1}\mathbf{r}_0$, $\delta_0 = \mathbf{r}_0^T\mathbf{h}_0$, $b_\delta = \mathbf{b}^T P^{-1}\mathbf{b}$, $k = 0$ and $\mathbf{p}_0 = \mathbf{h}_0$. Then:

while $\delta_k > \texttt{tol}^2\, b_\delta$

$$\mathbf{s}_k = A\mathbf{p}_k$$
$$\alpha_k = \frac{\delta_k}{\mathbf{p}_k^T\mathbf{s}_k}$$
$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{p}_k$$
$$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k\mathbf{s}_k$$
$$\mathbf{h}_{k+1} = P^{-1}\mathbf{r}_{k+1}$$
$$\delta_{k+1} = \mathbf{r}_{k+1}^T\mathbf{h}_{k+1}$$
$$\mathbf{p}_{k+1} = \mathbf{h}_{k+1} + \frac{\delta_{k+1}}{\delta_k}\mathbf{p}_k$$
$$k = k+1$$

end

To produce an effective method the preconditioner matrix P must be easily invertible. At the same breath it is desirable to have at least one of the following properties hold: $\kappa(P^{-1}A) \ll \kappa(A)$ and/or the eigenvalues of $P^{-1}A$ are much better clustered compared to those of A.

Simple preconditioners can be constructed from the relaxation methods we have seen in Section 7.2. Thus, we can set P as defined in Section 7.2 for different methods. Jacobi's $P = D$, and the preconditioner obtained by applying SSOR relaxation (page 179), are two fairly popular choices among those. But there are much more sophisticated preconditioners, and a member of the particularly popular family of incomplete factorizations is discussed next.

Incomplete Cholesky factorization

Let A be a large, sparse, symmetric positive definite matrix. Among modern classes of preconditioners for such matrices, a very popular family is that of *incomplete Cholesky* (IC) factorizations. The straightforward yet powerful idea behind them conjures memories of Sections 5.5 and 5.7, and we recall in particular that a Cholesky factorization produces factors that are much denser than A in general. The simplest IC factorization, denoted IC(0), constructs a Cholesky decomposition that follows precisely the same steps as the usual decomposition algorithms from Sections 5.2 or 5.5, except a nonzero entry of a factor is generated *only if* the matching entry of A is nonzero! Figure 7.6 illustrates this idea for our good old matrix from Example 7.1. The top left subfigure shows the sparsity pattern of the IC factor F. The top right is $P = FF^T$, effectively the preconditioner. The bottom left subfigure shows the sparsity pattern of the full Cholesky factor G such that $A = GG^T$, which we assume is too expensive to produce and work with in the present context. It is evident that the product FF^T does not restore A precisely, as there are now two additional nonzero diagonals, but the gain in terms of sparsity is significant: the full Cholesky decomposition of A results in much denser factors.

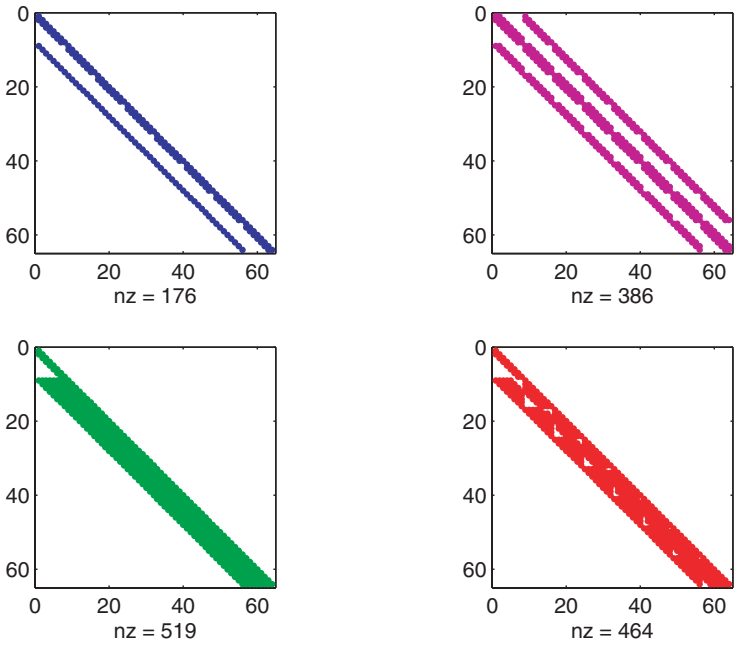


Figure 7.6. *Sparsity patterns for the matrix of Example* 7.1 *with* $N = 8$*: top left, the IC factor* F *with no fill-in (IC(0)); top right, the product* FF^T*; bottom left, the full Cholesky factor* G*; bottom right, the IC factor with drop tolerance* .001*. See Figure* 7.3 *for the sparsity pattern of the original matrix.*

As it turns out, using IC(0) is not always good enough. It may make more sense to construct sparse Cholesky factors in a *dynamic* fashion. That is, drop a constructed nonzero entry based not on where it is in the matrix but rather on whether the generated value is smaller than a threshold value, aptly called *drop tolerance*. (We are actually omitting some subtle details here: this value is also affected by the scaling of the matrix row in question. But let us not get into technicalities of this sort.) Of course, the smaller the drop tolerance, the closer we get to the full Cholesky factor. In addition to the drop tolerance, there is typically another parameter that limits the number of nonzeros per row of the incomplete factors. For a small drop tolerance we expect fewer PCG iterations at a higher cost per iteration.

The bottom right subfigure of Figure 7.6 shows the sparsity pattern that results using IC with a drop tolerance of .001. These figures are a bit misleading because the depicted system is so small: for N large, the number of nonzero entries in IC with the same tolerance is much smaller than in the full factor.

In MATLAB the command `pcg` applies the PCG method, underlining not the complexity of the algorithm but its importance. For example, the application of IC(0) as a preconditioner, with `tol` as the relative residual tolerance for convergence and `maxit` as the maximal allowed number of iterations, can be done by

```
R=cholinc(A,'0');
x=pcg(A,b,tol,maxit,R',R);
```

Example 7.11. Returning to Example 7.10 we now consider comparable experiments with IC preconditioners. Figure 7.7 displays plots of the relative residual norm as a function of the number

of iterations using IC(0) and IC with drop tolerance 0.01. Compared to the CG performance, the iteration count using the IC(0) preconditioner is reduced by a factor of more than two, and using IC with drop tolerance 0.01 further reduces the iteration count by another factor of roughly two.

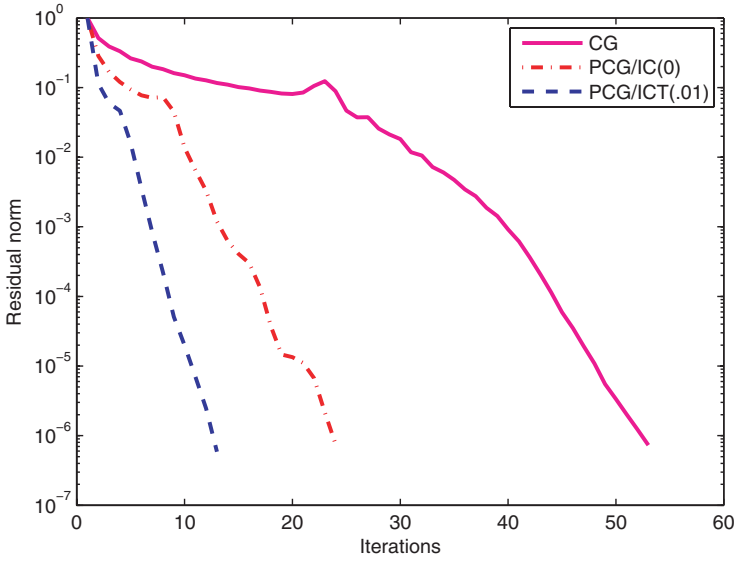


Figure 7.7. *Iteration progress for CG, PCG with the IC*(0) *preconditioner and PCG with the IC preconditioner using drop tolerance* `tol`$= 0.01$.

Note that a CG iteration is cheaper than a PCG iteration with IC(0), which is in turn cheaper than a PCG iteration using IC with a drop tolerance. But for tough problems it is worth paying the extra cost per iteration. ∎

The necessity of preconditioners

The art of preconditioning has been the topic of much recent research. If done properly it may yield great efficiency coupled with flexibility. The latter is due to the fact that unlike the given matrix A, the preconditioner P is ours to choose. However, it is important to also keep an eye on the cost of iteration and to realize that not all preconditioners are effective. While in the plain vanilla nonpreconditioned CG iterations we had to mainly concern ourselves with the cost of vector multiplication by A, here we have to perform *in addition* a vector multiplication by P^{-1}. Of course, we would not form P^{-1} explicitly, and this latter operation amounts to solving a linear system with P.

Despite the higher computational cost of preconditioned iterations compared to an iterative scheme that involves no preconditioning, it is widely agreed today that preconditioning is an inseparable and often crucial part of iterative solvers for many problems. It seems that no general-purpose preconditioner is uniformly effective, and preconditioners that are tailored to a specific class of problems are often the way to go. The construction of very good preconditioners for problems that arise from a discretization of a differential equation often involves aspects of the underlying continuous (differential) problem. There are examples of impressive preconditioning performance in the next two sections.

Specific exercises for this section: Exercises 13–18.

7.5 *Krylov subspace methods

The CG method beefed up by preconditioning as described in Section 7.4 offers significant improvement over the stationary methods considered in the previous sections of this chapter. However, strictly speaking it is applicable only to symmetric positive definite systems. In this section we extend the scope and discuss efficient iterative methods for the general case, where A is a real nonsingular matrix. We introduce the family of Krylov subspace methods, of which CG is a special member.

Note: Let us say it up front: of the practical methods described in this section for general square matrices, none comes close to having the combination of efficiency, robustness, theoretical backing, and elegance that CG has. Indeed, positive definiteness is a big asset, and when it is lost the going gets tough. Nevertheless, these methods can be very efficient in practice, and developing some expertise about them beyond the scope of this section is recommended.

A general recipe with limited applicability

Given a general linear system $A\mathbf{x} = \mathbf{b}$ with A nonsingular, it is actually quite easy to turn it into a symmetric positive definite one: simply multiply both sides by A^T. Thus we consider the linear system

$$B\mathbf{x} = \mathbf{y}, \quad \text{where } B = A^T A,\ \mathbf{y} = A^T \mathbf{b},$$

and this can be solved using the CG method as in Section 7.4 because B is symmetric positive definite. Recalling Section 6.1, the system of equations $B\mathbf{x} = \mathbf{y}$ describes the *normal equations* for the least squares problem

$$\min_{\mathbf{x}} \|\mathbf{b} - A\mathbf{x}\|.$$

Of course, in Section 6.1 we were concerned with overdetermined systems, whose matrix A was of size $m \times n$ with $m > n$. Here we are interested in square matrices ($m = n$), but the principle stays the same. The computation can easily be arranged so that at each iteration only one matrix-vector product involving A and one involving A^T are required. Methods based on applying CG to $A^T A\mathbf{x} = A^T \mathbf{b}$ are known by a few names, for example, *conjugate gradient for least squares* (CGLS).

The technique can certainly be applied to quickly solve toy problems such as Example 7.2, and more generally it is practical when $\kappa(A)$ is not overly large. However, the fact already lamented in Chapter 6, namely, that the condition number of $A^T A$ is squared that of A, has potentially severe repercussions here, because the number of iterations required for a fixed convergence error typically grows linearly with $\sqrt{\kappa(B)} = \kappa(A)$, a rate which we have already found in Section 7.4 to be often unacceptable. See also Exercise 23. The quest for extensions of the CG method therefore has to go deeper than methods based on the normal equations. We next consider Krylov space methods based directly on the given general matrix A, not $B = A^T A$.

Building blocks for Krylov subspace methods

Let us assume no preconditioning for the moment. All the methods described in Section 7.4 can be characterized by

$$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k A\mathbf{p}_k,$$

where $\mathbf{r}_k = \mathbf{b} - A\mathbf{x}_k$ is the residual and $\mathbf{p}_k$ is the search direction in the kth iteration. This includes CG, whereby $\mathbf{p}_k$ depends on the previous search direction $\mathbf{p}_{k-1}$ and $\mathbf{r}_k$. Continuing to write $\mathbf{r}_k$ in

terms of $\mathbf{r}_{k-1}$ and $\mathbf{p}_{k-1}$, and so on, we quickly arrive at the conclusion that there are coefficients $c_1, \ldots, c_k$ such that

$$\mathbf{r}_k = \mathbf{r}_0 + \sum_{j=1}^{k} c_j A^j \mathbf{r}_0.$$

We can write this as

$$\mathbf{r}_k = p_k(A)\mathbf{r}_0,$$

where $p_k(A)$ is a polynomial of degree k in A that satisfies $p_k(0) = 1$.

Moreover, $\sum_{j=1}^{k} c_j A^j \mathbf{r}_0 = \mathbf{r}_k - \mathbf{r}_0 = -A(\mathbf{x}_k - \mathbf{x}_0)$. We get

$$\mathbf{x}_k = \mathbf{x}_0 - \sum_{j=0}^{k-1} c_{j+1} A^j \mathbf{r}_0,$$

or $\mathbf{x}_k - \mathbf{x}_0 \in \mathcal{K}_k(A; \mathbf{r}_0)$, the Krylov subspace whose definition was given on page 186.

This is the essence that we retain when extending to the case where A is no longer symmetric positive definite. We thus compute approximate solutions within the shifted Krylov subspace $\mathbf{x}_0 + \mathcal{K}_k(A; \mathbf{r}_0)$, where

$$\mathcal{K}_k(A; \mathbf{r}_0) = \text{span}\{\mathbf{r}_0, A\mathbf{r}_0, A^2\mathbf{r}_0, \ldots, A^{k-1}\mathbf{r}_0\}.$$

The family of *Krylov subspace solvers* is based on finding a solution within the subspace, which satisfies a certain optimality criterion. These solvers are based on three important building blocks:

1. constructing an orthogonal basis for the Krylov subspace;
2. defining an optimality property;
3. using an effective preconditioner.

Computing an orthogonal basis for the Krylov subspace

Unfortunately, the most obvious choice of vectors to span the Krylov subspace $\mathcal{K}_k(A; \mathbf{r}_0)$, namely, $\{\mathbf{r}_0, A\mathbf{r}_0, A^2\mathbf{r}_0, \ldots, A^{k-1}\mathbf{r}_0\}$, is poorly conditioned for exactly the same reason the *power method*, described in Section 8.1, often works well: as j grows larger, vectors of the form $A^j\mathbf{r}_0$ approach the dominant eigenvector of A and therefore also one another.

Instead, let us construct an *orthonormal* basis for the Krylov subspace.

The Arnoldi process

The first vector in the Krylov subspace is $\mathbf{r}_0$, and hence to get the first basis vector all we need is to take this vector and normalize it, yielding

$$\mathbf{q}_1 = \mathbf{r}_0 / \|\mathbf{r}_0\|.$$

Next we want to find a vector $\mathbf{q}_2$ orthogonal to $\mathbf{q}_1$ such that the pair $\{\mathbf{q}_1, \mathbf{q}_2\}$ spans the same subspace that $\{\mathbf{r}_0, A\mathbf{r}_0\}$ spans. Since $A\mathbf{q}_1$ is in the same direction as $A\mathbf{r}_0$, we require

$$A\mathbf{q}_1 = h_{11}\mathbf{q}_1 + h_{21}\mathbf{q}_2.$$

For convenience, the coefficients h_{ij} are endowed with a double subscript: the first subscript is a running index signifying the coefficient of a basis vector, and the second signifies the current step. Multiplying both sides of the above equation by $\mathbf{q}_1^T$ and observing that by orthogonality $\langle \mathbf{q}_1, \mathbf{q}_2 \rangle = 0$,

we obtain $h_{11} = \langle \mathbf{q}_1, A\mathbf{q}_1 \rangle$. Next, taking norms and using the fact that $\mathbf{q}_2$ is a unit vector gives $h_{21} = \|A\mathbf{q}_1 - h_{11}\mathbf{q}_1\|$, and then

$$\mathbf{q}_2 = \frac{A\mathbf{q}_1 - h_{11}\mathbf{q}_1}{h_{21}}.$$

Generalizing the above derivation, at the jth step we have

$$A\mathbf{q}_j = h_{1j}\mathbf{q}_1 + h_{2j}\mathbf{q}_2 + \cdots + h_{j+1,j}\mathbf{q}_{j+1} = \sum_{i=1}^{j+1} h_{ij}\mathbf{q}_i.$$

By the orthonormality achieved thus far, for $1 \le m \le j$, $\langle \mathbf{q}_m, \mathbf{q}_j \rangle = 0$ if $m \ne j$ and $\langle \mathbf{q}_m, \mathbf{q}_j \rangle = 1$ if $m = j$. This gives

$$\langle \mathbf{q}_m, A\mathbf{q}_j \rangle = \sum_{i=1}^{j+1} h_{ij} \langle \mathbf{q}_m, \mathbf{q}_i \rangle = h_{mj}.$$

For the current unknowns $h_{j+1,j}$ and $\mathbf{q}_{j+1}$, we get $h_{j+1,j}\mathbf{q}_{j+1} = A\mathbf{q}_j - \sum_{i=1}^{j} h_{ij}\mathbf{q}_i$. Taking norms and using the fact that $\|\mathbf{q}_{j+1}\| = 1$ yields

$$h_{j+1,j} = \left\| A\mathbf{q}_j - \sum_{i=1}^{j} h_{ij}\mathbf{q}_i \right\|,$$

and then

$$\mathbf{q}_{j+1} = \frac{A\mathbf{q}_j - \sum_{i=1}^{j} h_{ij}\mathbf{q}_i}{h_{j+1,j}}.$$

This incremental process can be concisely encapsulated in terms of the following matrix decomposition: for any given $k \ge 1$ there is a relation of the form

$$A[\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_k] = [\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_k, \mathbf{q}_{k+1}] \cdot \begin{pmatrix} h_{11} & \cdots & \cdots & \cdots & h_{1k} \\ h_{21} & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \ldots & 0 & h_{k,k-1} & h_{kk} \\ 0 & \ldots & 0 & 0 & h_{k+1,k} \end{pmatrix},$$

which can be written as

$$A\,Q_k = Q_{k+1} H_{k+1,k},$$

where Q_{k+1} is the matrix containing the $k+1$ vectors of the orthogonal basis for the Krylov subspace, Q_k is the same matrix but with the first k columns only, and $H_{k+1,k}$ is a matrix of size $(k+1) \times k$. It is easy to show that

$$Q_k^T A\,Q_k = H_{k,k},$$

where $H_{k,k}$ is the $k \times k$ square matrix containing the first k rows of $H_{k+1,k}$. This matrix is in upper Hessenberg form (see page 139). All these relations will see use soon when we derive solution methods.

The Arnoldi algorithm is given on the next page. It is not necessary to provide A explicitly; only a routine that returns $\mathbf{w} = A\mathbf{v}$, given a vector $\mathbf{v}$ as input, is required.

Algorithm: Arnoldi Process.
Input: initial unit vector $\mathbf{q}_1$ (possibly $\mathbf{r}_0/\|\mathbf{r}_0\|$), matrix A, and number of steps k.

```
for j = 1 to k
  z = A q_j
  for i = 1 to j
    h_{i,j} = ⟨q_i, z⟩
    z ← z − h_{i,j} q_i
  end
  h_{j+1,j} = ‖z‖
  if h_{j+1,j} = 0, quit
  q_{j+1} = z / h_{j+1,j}
end
```

Example 7.12. Consider the toy system introduced in Example 7.2, given by

$$\begin{aligned} 7x_1 + 3x_2 + x_3 &= 3, \\ -3x_1 + 10x_2 + 2x_3 &= 4, \\ x_1 + 7x_2 - 15x_3 &= 2\,. \end{aligned}$$

We use it to demonstrate how the Arnoldi algorithm is carried out. Suppose the initial guess is $\mathbf{x}_0 = \mathbf{0}$. Then, $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0 = \mathbf{b}$, and we choose $\mathbf{q}_1 = \mathbf{r}_0/\|\mathbf{r}_0\| = \mathbf{b}/\|\mathbf{b}\| = \frac{1}{\sqrt{29}}(3,4,2)^T = (0.55709, 0.74278, 0.37139)^T$.

Next we compute $\mathbf{q}_2$. The procedure outlined above yields $h_{11} = \mathbf{q}_1^T A\mathbf{q}_1 = 8.5172$. We can now compute $h_{21} = \|A\mathbf{q}_1 - h_{11}\mathbf{q}_1\| = 3.4603$ and $\mathbf{q}_2 = (A\mathbf{q}_1 - h_{11}\mathbf{q}_1)/h_{21} = (0.50703, 0.049963, -0.86048)^T$. Proceeding in the same fashion to compute the second column of the upper Hessenberg matrix leads to $h_{12} = \mathbf{q}_1^T A\mathbf{q}_2 = 4.6561$, $h_{22} = \mathbf{q}_2^T A\mathbf{q}_2 = -10.541$, and $h_{32} = \|A\mathbf{q}_2 - h_{12}\mathbf{q}_1 - h_{22}\mathbf{q}_2\| = 8.4986$.

The next basis vector turns out to be $\mathbf{q}_3 = (0.6577, -0.66767, 0.34878)^T$. Thus, we have constructed a decomposition of the form

$$AQ_2 = Q_3 H_{3,2},$$

where

$$Q_3 = \begin{pmatrix} 0.55709 & 0.50703 & 0.6577 \\ 0.74278 & 0.049963 & -0.66767 \\ 0.37139 & -0.86048 & 0.34878 \end{pmatrix},$$

Q_2 is the 3×2 matrix comprised of the first two columns of Q_3, and

$$H_{3,2} = \begin{pmatrix} 8.5172 & 4.6561 \\ 3.4603 & -10.541 \\ 0 & 8.4986 \end{pmatrix}.$$

It is easy to confirm that $Q_2^T A Q_2 = H_{2,2}$, where $H_{2,2}$ is the 2×2 matrix composed of the upper two rows of $H_{3,2}$, and that Q_3 is an orthogonal matrix. ∎

Of course, for large problems we will never proceed with Arnoldi for a number of steps equal or nearly equal to the dimensions of the matrix. Therefore a square Q_k like Q_3 in Example 7.12 never happens in practice. Typically the $n \times k$ matrix Q_k (or Q_{k+1}, which is "thicker" by one column) is a tall and skinny rectangular matrix, since $k \ll n$.

The Lanczos method

When A is symmetric, $A = A^T$ implies that $H_{k,k}$ must be symmetric.[29] Therefore, it must be tridiagonal; let us denote it by $T_{k,k}$. In this case the Arnoldi process reduces to the well-known *Lanczos* method. The fact that we have a tridiagonal matrix simplifies the calculations and yields significant computational savings. Specifically, when dealing with the jth column we have to consider three terms only. Let us write

$$A[\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_k] = [\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_k, \mathbf{q}_{k+1}] \cdot \begin{pmatrix} \gamma_1 & \beta_1 & 0 & \cdots & 0 \\ \beta_1 & \gamma_2 & \beta_2 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \beta_{k-1} \\ 0 & \ldots & 0 & \beta_{k-1} & \gamma_k \\ 0 & \ldots & 0 & 0 & \beta_k \end{pmatrix}.$$

From this it follows, for a given j, that

$$A\mathbf{q}_j = \beta_{j-1}\mathbf{q}_{j-1} + \gamma_j \mathbf{q}_j + \beta_j \mathbf{q}_{j+1}.$$

Just like for the Arnoldi algorithm, we now use orthogonality and obtain

$$\gamma_j = \langle \mathbf{q}_j, A\mathbf{q}_j \rangle.$$

When dealing with the jth column, β_{j-1} is already known, so for computing β_j we can use

$$\beta_j \mathbf{q}_{j+1} = A\mathbf{q}_j - \beta_{j-1}\mathbf{q}_{j-1} - \gamma_j \mathbf{q}_j,$$

with the value of γ_j just computed. Taking norms yields

$$\beta_j = \| A\mathbf{q}_j - \beta_{j-1}\mathbf{q}_{j-1} - \gamma_j \mathbf{q}_j \|.$$

Once β_j has been computed, $\mathbf{q}_{j+1}$ is readily available.

This outlines the *Lanczos algorithm* for computing the orthogonal basis for $\mathcal{K}_k(A; \mathbf{r}_0)$ in cases when A is symmetric. The matrix $T_{k+1,k}$ associated with the construction

$$AQ_k = Q_{k+1}T_{k+1,k}$$

[29] We are not assuming here that A is also positive definite, so we are generally not in CG territory. There are of course many real matrices that are symmetric and not definite, for instance, $\left(\begin{smallmatrix} 1 & \mu \\ \mu & 0 \end{smallmatrix}\right)$ for any real scalar μ.

is indeed tridiagonal. A MATLAB script for the Lanczos algorithm follows. We offer here the modified Gram–Schmidt version for Lanczos, which is preferred over classical Gram–Schmidt, see Section 6.3.

```
function [Q,T] = lanczos(A,Q,k)

% preallocate for speed
alpha=zeros(k,1);
beta=zeros(k,1);

Q(:,1) = Q(:,1)/norm(Q(:,1));
beta(1,1)=0;

for j=1:k
  w=A*Q(:,j);
  if j>1
     w=A*Q(:,j)-beta(j,1)*Q(:,j-1);
  end
  alpha(j,1)=Q(:,j)'*w;
  w=w-alpha(j,1)*Q(:,j);
  beta(j+1,1)=norm(w);

  if abs(beta(j+1,1))<1e-10
     disp('Zero beta --- returning.');
     T=spdiags([beta(2:j+1) alpha(1:j) beta(1:j)],-1:1,j+1,j);
     return
  end
  Q(:,j+1)=w/beta(j+1,1);
end
T=spdiags([beta(2:end) alpha beta(1:end-1)],-1:1,k+1,k);
```

Arnoldi, Lanczos, and eigenvalue computations

A remarkable property of the Arnoldi and Lanczos methods is that $H_{k,k}$ or $T_{k,k}$ typically does very well in approximating the eigenvalues of the original matrix. The eigenvectors can also be approximated, using the orthogonal basis for the Krylov subspace, Q_k. Of course, given that $k \ll n$, we cannot expect *all* the eigenvalues of A to be approximated, since $H_{k,k}$ or $T_{k,k}$ is much smaller. *Extremal* eigenvalues of the original matrix are well approximated, and typically it is the largest ones that are approximated in the best fashion. This fact comes in handy in advanced iterative methods for computing eigenvalues of large and sparse matrices. The eigenvalues of $H_{k,k}$ or $T_{k,k}$ are known as **Ritz values**.

Optimality criteria: CG and FOM, MINRES and GMRES

Now that the orthogonal basis construction is taken care of, we turn to discuss the second building block mentioned on page 192: optimality criteria. There are various alternatives for deciding on the type of solution we are looking for within the Krylov subspace at iteration k. Two particularly popular criteria are the following:

- force the residual $\mathbf{r}_k$ to be orthogonal to the Krylov subspace $\mathcal{K}_k(A;\mathbf{r}_0)$;
- seek the residual with minimum ℓ_2-norm within the Krylov subspace.

Galerkin orthogonalization

The first alternative above is often referred to as a Galerkin orthogonalization approach, and it leads to the *full orthogonalization method* (FOM) for nonsymmetric matrices and to no other than CG (after doing some further fancy manipulations) for symmetric positive definite matrices.

Since the columns of Q_k are the orthonormal basis vectors of the Krylov subspace, this optimality criterion amounts to requiring

$$Q_k^T(\mathbf{b} - A\mathbf{x}_k) = 0.$$

But since $\mathbf{x}_k - \mathbf{x}_0 \in \mathcal{K}_k(A;\mathbf{r}_0)$ we can write $\mathbf{x}_k = \mathbf{x}_0 + Q_k\mathbf{y}$, and the criterion simplifies to

$$Q_k^T A Q_k \mathbf{y} = Q_k^T \mathbf{r}_0.$$

This is good news, because $Q_k^T A Q_k$ is nothing but $H_{k,k}$ if A is nonsymmetric or $T_{k,k}$ if A is symmetric! Furthermore, the right-hand side of this system can also be simplified. Since $\mathbf{q}_1$ is just the normalized $\mathbf{r}_0$ and all other columns in Q_k are orthogonal to it we have

$$Q_k^T \mathbf{r}_0 = \|\mathbf{r}_0\|\mathbf{e}_1,$$

where $\mathbf{e}_1 = (1,0,\ldots,0)^T$.

From this it follows that a method based on orthogonalization amounts to solving

$$H_{k,k}\mathbf{y} = \|\mathbf{r}_0\|\mathbf{e}_1$$

(with $T_{k,k}$ replacing $H_{k,k}$ if A is symmetric) for $\mathbf{y}$ and then setting $\mathbf{x}_k = \mathbf{x}_0 + Q_k\mathbf{y}$.

When A is symmetric positive definite, some elegant manipulations can be performed in the process of inverting $T_{k,k}$. The $k \times k$ linear system can be solved by exploiting certain algebraic connections to the linear system in the previous, $(k-1)$st step, and the resulting algorithm can be written using short recurrences. Indeed, the CG method can be derived in many ways, not only from an optimization point of view that utilizes downhill search directions, but also from an algebraic point of view based on the decomposition of a symmetric positive definite tridiagonal matrix.

GMRES for general matrices

The second optimality criterion that we have mentioned, namely, that of minimizing $\|\mathbf{r}_k\|$ within $\mathcal{K}_k(A;\mathbf{r}_0)$ in the ℓ_2-norm, leads to the popular methods *generalized minimum residual* (GMRES) for nonsymmetric matrices and *minimum residual* (MINRES) for symmetric but not necessarily positive definite matrices.

By the Arnoldi algorithm, we have that $AQ_k = Q_{k+1}H_{k+1,k}$. We can write

$$\mathbf{x}_k = \mathbf{x}_0 + Q_k\mathbf{z}$$

and aim to find $\mathbf{x}_k$ which minimizes $\|\mathbf{b} - A\mathbf{x}_k\|$. Recall that using CG we are minimizing $\|\mathbf{r}_k\|_{A^{-1}}$ instead, but that would make sense only if A is symmetric positive definite as otherwise we have no energy norm. Hence we look at minimizing $\|\mathbf{r}_k\|$ here.

We have

$$\|\mathbf{b} - A\mathbf{x}_k\| = \|\mathbf{b} - A\mathbf{x}_0 - AQ_k\mathbf{z}\| = \|\mathbf{r}_0 - Q_{k+1}H_{k+1,k}\mathbf{z}\| = \|Q_{k+1}^T\mathbf{r}_0 - H_{k+1,k}\mathbf{z}\|.$$

Now the expression on the right involves a small $(k+1)\times k$ matrix! Moreover, $Q_{k+1}^T\mathbf{r}_0 = \|\mathbf{r}_0\|\mathbf{e}_1$. So, the minimization problem $\min_{\mathbf{x}_k}\|\mathbf{b} - A\mathbf{x}_k\|$ is equivalent to $\min_{\mathbf{z}}\|\rho\mathbf{e}_1 - H_{k+1,k}\mathbf{z}\|$, where $\rho = \|\mathbf{r}_0\|$.

Recall from Section 6.2 that a linear least squares problem can be solved using the economy size QR factorization. Writing

$$H_{k+1,k} = U_{k+1,k} R_{k,k},$$

where $R_{k,k}$ is an upper triangular matrix and $U_{k+1,k}$ consists of k orthonormal vectors, yields

$$\|\rho \mathbf{e}_1 - H_{k+1,k}\mathbf{z}\| = \|\rho \mathbf{e}_1 - U_{k+1,k} R_{k,k}\mathbf{z}\| = \|\rho U_{k+1,k}^T \mathbf{e}_1 - R_{k,k}\mathbf{z}\|.$$

We therefore have

$$\mathbf{z} = R_{k,k}^{-1} U_{k+1,k}^T \|\mathbf{r}_0\| \mathbf{e}_1.$$

Once $\mathbf{z}$ is recovered, the kth iterate is simply obtained by setting $\mathbf{x}_k = \mathbf{x}_0 + Q_k \mathbf{z}$.

The procedure outlined above is known as the GMRES method. There are a few more details in its specification that we have omitted, since they are quite technical, but the essence is here. (If you will not rest until we hint at what those omitted details are, let us just say that they are related to an efficient utilization of Givens rotations or other means for solving the least squares problem involving the upper Hessenberg matrix.) The GMRES algorithm is more involved than CG, and while it is not really very long, the various details make it less concise than the algorithms we have seen so far in this chapter. Its main steps are recaptured on the current page.

GMRES Steps.
The main components of a single iteration of GMRES are

1. perform a step of the Arnoldi process;
2. update the QR factorization of the updated upper Hessenberg matrix;
3. solve the resulting least squares problem.

MINRES for symmetric matrices

For symmetric matrices A, Arnoldi is replaced by Lanczos and the upper Hessenberg matrix is really just tridiagonal. The same mechanism can be applied but the resulting iterative method is simpler and, like CG, requires short three-term recurrence relations. It is called MINRES.

In each iteration the residual for GMRES or MINRES is minimized over the current Krylov subspace, and since a $(k+1)$-dimensional Krylov subspace contains the preceding k-dimensional subspace, we are assured (at least in the absence of roundoff errors) that the residual decreases monotonically. Moreover, like CG these two schemes converge to the exact solution in n iterations. In the special case of nonsingular symmetric matrices that are indefinite (i.e., their eigenvalues are neither all positive nor all negative), MINRES is a very popular method.

CG and MINRES both work on symmetric matrices, but MINRES minimizes $\|\mathbf{r}_k\|$, whereas CG minimizes the energy norm $\|\mathbf{r}_k\|_{A^{-1}}$ on the same subspace. MINRES is applicable to a wider class of problems. However, for positive definite matrices, for which both methods work, CG is more economical and is better understood theoretically, and hence it is the preferred method.

Limited memory GMRES

For nonsymmetric matrices, where Arnoldi cannot be replaced by Lanczos, there is a significant price to pay when using GMRES. As we proceed with iterations, we accumulate more and more basis vectors of the Krylov subspace. We need to have those around since the solution is given as their linear combination. As a result, the storage requirements keep creeping up as we iterate. This

may become a burden if the matrix is very large and if convergence is not very fast, which often occurs in realistic scenarios.

One way of dealing with this difficulty is by using **restarted GMRES**. We run GMRES as described above, but once a certain maximum number of vectors, say, m, is reached, we take the current iterate as our initial guess, compute the residual, and start again. The resulting method is denoted by GMRES(m). Thus, m is the number of vectors that are stored, and we may have $m \ll n$. Of course, restarted GMRES is almost certain to converge more slowly than the full GMRES. Moreover, we do not really have an optimality criterion anymore, although we still have monotonicity in residual norm reduction. But despite these drawbacks of restarted GMRES, in practice it is usually preferred over full GMRES, since the memory requirement issue is critical in large scale settings. The full GMRES is just "too good to be practical" for really large problems.

MATLAB provides commands for several Krylov subspace solvers. Check out `gmres`, `pcg`, and `minres`. A typical command looks like this:

```
[x,flag,relres,iter,resvec]=gmres(A,b,m,tol,maxit,M1,M2);
```

The input parameters here are set so that the iteration is terminated when either $\|\mathbf{r}_k\|/\|\mathbf{b}\| <$ `tol` or a maximum iteration count of `maxit` has been reached. The input value `m` is the restart parameter of GMRES(m); the other commands such as `pcg` or `minres` have a similar calling sequence except for this parameter which is not required by any other method. The (optional) input parameters `M1` and `M2` are factors of a preconditioner, to be discussed further below.

Example 7.13. There are many situations in which mathematical models lead to linear systems where the matrix is nonsymmetric. One such example is the numerical discretization of the steady-state **convection-diffusion equation**. This equation describes physical phenomena that involve interaction among particles of a certain material (diffusion), as well as a form of motion (convection). An example here would be the manner in which a pollutant, say, spreads as it moves in a stream of water. It moves along with the water and other objects, and at the same time it changes its shape and chemical structure and spreads as its molecules interact with each other in a certain manner that can be described mathematically.

The description that follows bears several similarities to Example 7.1, and you are urged to review that example (page 168) before proceeding. Here we highlight mainly what is *different* in the current example. In its simplest form, the convection-diffusion equation is defined on the open *unit square*, $0 < x, y < 1$, and reads

$$-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + \sigma \frac{\partial u}{\partial x} + \tau \frac{\partial u}{\partial y} = g(x, y).$$

The parameters σ and τ are associated with the *convection*: when they both vanish, we are back to the Poisson equation of Example 7.1. Assume also the same homogeneous Dirichlet boundary conditions as before.

Discretizing using centered differences for both the first and the second partial derivatives (see Section 14.1 or trust us), and using the notation of Example 7.1 and in particular Figure 7.2, we obtain a linear system of equations given by

$$\begin{aligned} 4u_{i,j} - \tilde{\beta} u_{i+1,j} - \hat{\beta} u_{i-1,j} - \tilde{\gamma} u_{i,j+1} - \hat{\gamma} u_{i,j-1} = b_{i,j}, \;\; 1 \le i, j \le N, \\ u_{i,j} = 0 \;\; \text{otherwise}, \end{aligned}$$

where $\beta = \frac{\sigma h}{2}$, $\hat{\beta} = 1 + \beta$, $\tilde{\beta} = 1 - \beta$, and $\gamma = \frac{\tau h}{2}$, $\hat{\gamma} = 1 + \gamma$, $\tilde{\gamma} = 1 - \gamma$. The associated linear system of equations is written as

$$A\mathbf{u} = \mathbf{b},$$

where $\mathbf{u}$ consists of the $n = N^2$ unknowns $\{u_{i,j}\}$ organized as a vector, and $\mathbf{b}$ is composed likewise from the values $\{b_{i,j}\}$. The specific nonzero structure of the matrix is the same as that discussed in Example 7.1; see Figure 7.3. However, the resulting blocks are more lively. We have

$$A = \begin{pmatrix} J & L & & & \\ K & J & L & & \\ & \ddots & \ddots & \ddots & \\ & & K & J & L \\ & & & K & J \end{pmatrix}, \quad J = \begin{pmatrix} 4 & -\tilde{\beta} & & & \\ -\hat{\beta} & 4 & -\tilde{\beta} & & \\ & \ddots & \ddots & \ddots & \\ & & -\hat{\beta} & 4 & -\tilde{\beta} \\ & & & -\hat{\beta} & 4 \end{pmatrix},$$

and also $K = -\hat{\gamma} I_N$, $L = -\tilde{\gamma} I_N$, where I_N denotes the identity matrix of size N. We can see that the matrix A is diagonally dominant if $|\beta| < 1$ and $|\gamma| < 1$. Let us stay within this range of values; there is more than one good reason for this.

To generate the matrix A in sparse form with MATLAB, we can use a nice feature that comes in handy when forming highly structured matrices: the *Kronecker product*. Given two matrices of appropriate sizes, C and B, their Kronecker product is defined as

$$C \otimes B = \begin{pmatrix} c_{11}B & c_{12}B & \cdots & c_{1N}B \\ c_{21}B & c_{22}B & \cdots & c_{2N}B \\ \vdots & \vdots & \ldots & \vdots \\ c_{N1}B & c_{N2}B & \cdots & c_{NN}B \end{pmatrix}.$$

For notational convenience, let us denote an $N \times N$ tridiagonal matrix with the first subdiagonal, main diagonal, and first superdiagonal having constant values a, b, and c, respectively, by $\mathrm{tri}_N(a,b,c)$. Then our two-dimensional convection-diffusion matrix can be written as

$$A = I_N \otimes T_1 + T_2 \otimes I_N,$$

where $T_1 = \mathrm{tri}_N(-\hat{\beta}, 4, -\tilde{\beta})$ and $T_2 = \mathrm{tri}_N(-\hat{\gamma}, 0, -\tilde{\gamma})$. We can thus define our A using the following MATLAB script:

```
function A=kron_conv_diff(beta,gamma,N);
ee=ones(N,1);
a=4; b=-1-gamma; c=-1-beta; d=-1+beta; e=-1+gamma;
t1=spdiags([c*ee,a*ee,d*ee],-1:1,N,N);
t2=spdiags([b*ee,zeros(N,1),e*ee],-1:1,N,N);
A=kron(speye(N),t1)+kron(t2,speye(N));
```

Note the utilization of two useful commands: `spdiags` generates a matrix in sparse form by putting the input values along the diagonals, and `kron` performs the Kronecker product.

To demonstrate the performance of a solver for this problem, we set $N = 100$, $\beta = \gamma = 0.1$, and construct the convection-diffusion matrix. For the right-hand side we used an artificially generated vector, so that the solution is a vector of all 1's. After generating the matrix and right-hand side, the linear system was solved using restarted GMRES. Here is the MATLAB command that we ran:

```
[x,flag,relres,iter,resvec]=gmres(A,b,20,1e-8,1000);
```

The input parameters were set to stop the iteration when either $\|\mathbf{r}_k\|/\|\mathbf{b}\| < 10^{-8}$ or the iteration count exceeds 1000. In this example the former happened first, and the program terminated with `flag`=0.

Figure 7.8 shows the relative residual history for two runs: the preconditioned one will be explained later, so let us concentrate for now on the solid blue curve. Within fewer than 400 iterations, indeed far fewer than $n = 10,000$, the relative residual norm is about 10^{-7}. Also, the residual norm goes down monotonically, as expected. ∎

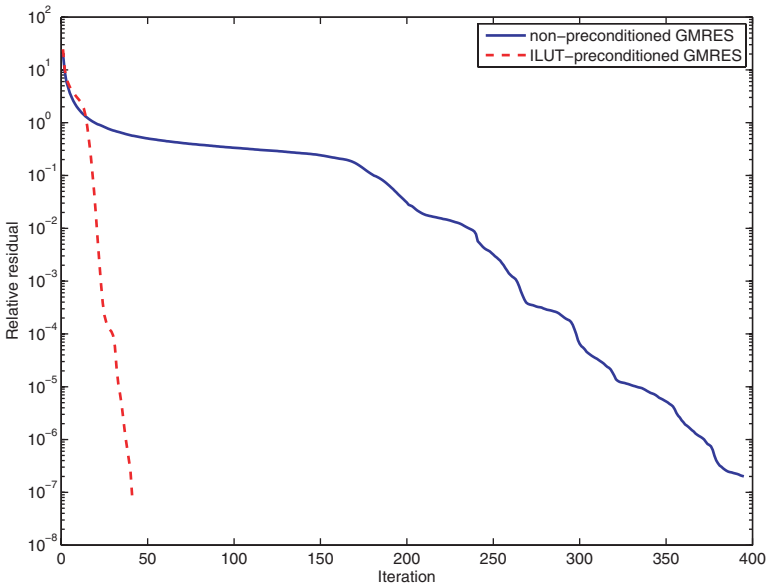


Figure 7.8. *Convergence behavior of restarted GMRES with $m = 20$, for a $10,000 \times 10,000$ matrix that corresponds to the convection-diffusion equation on a 100×100 uniform mesh.*

Krylov subspace solvers and min-max polynomials

Suppose that a given matrix A (not necessarily symmetric positive definite) has a complete set of n eigenpairs $\{\lambda_i, \mathbf{v}_i\}$, and suppose the initial residual satisfies

$$\mathbf{r}_0 = \sum_{i=1}^{n} \alpha_i \mathbf{v}_i .$$

Then

$$\mathbf{r}_k = p_k(A)\mathbf{r}_0 = \sum_{i=1}^{n} \alpha_i\, p_k(A)\mathbf{v}_i = \sum_{i=1}^{n} \alpha_i\, p_k(\lambda_i)\mathbf{v}_i .$$

This shows that the residual reduction depends on how well the polynomial dampens the eigenvalues. For the exact solution $\mathbf{x}$, denote the convergence error as before by $\mathbf{e}_k = \mathbf{x} - \mathbf{x}_k$. Since $A\mathbf{e}_k = \mathbf{r}_k$, we have

$$A\mathbf{e}_k = \mathbf{r}_k = p_k(A)\mathbf{r}_0 = p_k(A)A\mathbf{e}_0,$$

and since A commutes with powers of A, multiplying both sides by A^{-1} gives $\mathbf{e}_k = p_k(A)\mathbf{e}_0$.

The basic idea of Krylov subspace methods is to construct a "good" polynomial in this sense, and this may be posed as a problem in approximation theory: seek polynomials that satisfy the min-max property

$$\min_{\substack{p_k \in \pi_k \\ p_k(0)=1}} \max_{\lambda \in \sigma(A)} |p_k(\lambda)|,$$

where π_k denotes the space of all polynomials of degree up to k and $\sigma(A)$ signifies the spectrum of A. All this is done *implicitly*: in practice our iterative solvers never really directly solve a min-max problem.

Example 7.14. Back to the 3×3 linear system of Example 7.9, we look at the polynomial $p_k(A)$ that is obtained throughout the iteration process. Constructing the polynomial can be done, quite simply, by unrolling the CG iteration and expressing the residuals $\mathbf{r}_k, k = 1,2,3$, in terms of the initial residual $\mathbf{r}_0$. Figure 7.9 shows graphs of the polynomials p_1, p_2 and p_3. As evident, the values of the polynomials at the eigenvalues of A are dampened as we iterate, until they vanish for the cubic polynomial p_3. ∎

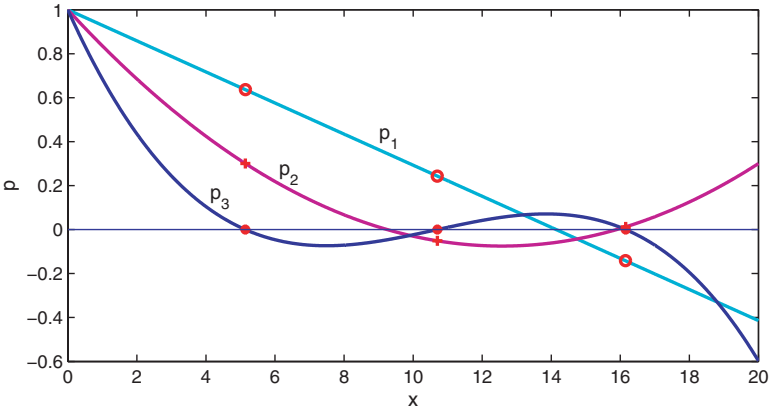


Figure 7.9. *The polynomials that are constructed in the course of three CG iterations for the small linear system of Examples* 7.9 *and* 7.14. *The values of the polynomials at the eigenvalues of the matrix are marked on the linear, quadratic, and cubic curves.*

Other Krylov solvers

There are other practical Krylov subspace methods whose detailed description belongs in a more advanced text. An important class of methods are ones that try to find short recurrence relations for nonsymmetric systems. This can be accomplished only if desirable optimality and orthogonality properties are given up. The notion of *bi-orthogonalization* is important here. This approach requires the availability of A^T for some of the schemes.

Let us just mention two favorite methods: BiCGSTAB and QMR. It is not difficult to find examples for which these methods, as well as restarted GMRES, are each better than the other ones, and other examples where they each fail. But they are worthy of your consideration nonetheless, because for many large problems that arise in practice, each of these methods can provide a powerful solution engine, especially when the system is properly preconditioned. Practical choice among these methods requires some experience or experimentation.

The importance and popularity of BiCGSTAB and QMR is reflected by the fact that, just like for CG, MINRES, and GMRES, there are built-in MATLAB commands for them: `bicgstab` and `qmr`.

Preconditioning

As mentioned in Section 7.4, preconditioning is often a necessity in the application of CG, and this is in fact true for the whole family of Krylov subspace solution methods. The purpose of the preconditioner is to cluster the eigenvalues more tightly, or reduce the condition number of the matrix $P^{-1}A$. Unlike the situation for CG, and to a lesser extent MINRES or full GMRES, for which these goals are backed up by a reasonably sound theoretical justification, practical methods such as BiCGSTAB or restarted GMRES have a less complete theory to back them up. However, there is an overwhelming practical evidence that the above-stated goals in the design of a preconditioner is the right thing to do in many generic situations.

There are several basic approaches for preconditioning. Some are based on general algebraic considerations; others make use of properties of the problem underlying the linear system.

There are a few ways to perform a preconditioning operation. *Left preconditioning* is simply the operation of multiplying the system $A\mathbf{x} = \mathbf{b}$ by P^{-1} on the left, to obtain $P^{-1}A\mathbf{x} = P^{-1}\mathbf{b}$. *Right preconditioning* is based on solving $AP^{-1}\tilde{\mathbf{x}} = \mathbf{b}$, then recovering $\mathbf{x} = P^{-1}\tilde{\mathbf{x}}$. *Split preconditioning* is a combination of sorts of left and right preconditioning: it is based on solving $P_1^{-1}AP_2^{-1}\tilde{\mathbf{x}} = P_1^{-1}\mathbf{b}$, then computing $\mathbf{x} = P_2^{-1}\tilde{\mathbf{x}}$. There are a few differences between these approaches. For example, for the same preconditioner P, the Krylov subspace associated with left preconditioning is different from the Krylov subspace associated with right preconditioning in a meaningful way. (Why?) If the matrix A is symmetric, then split preconditioning with $P_1 = P_2$ is the approach that most transparently preserves symmetry, but in practice just left or right preconditioning can be used, too.

Constructing preconditioned iterates amounts in principle to replacing the original matrix for the Arnoldi or Lanczos process, as well as the other components of the solver, by the preconditioned one. But there are a few details that need to be ironed out to guarantee that there are no redundant preconditioner inversions and other basic linear algebra operations in the process. Therefore, just as we saw for PCG vs. CG in Section 7.4, iterates are computed in a slightly different fashion compared to a scheme we would obtain by simply replacing the matrix by its preconditioned counterpart.

ILU preconditioner

In Section 7.4 we described incomplete Cholesky (IC) methods. The analogous factorization for general matrices is called *incomplete LU* (ILU). The factorization ILU(0) is based on the same principle as IC(0), i.e., a complete avoidance of fill-in.

For ILU, too, we can use instead a drop tolerance `tol` to decide whether or not to implement a step in the decomposition, just like IC with a drop tolerance in the symmetric positive definite case. The resulting method is called ILUT. Pivoting issues inevitably arise here, not shared by the IC case, but a detailed description is beyond the scope of this text.

Example 7.15. Let us return to Example 7.13 and add a preconditioner to the GMRES(20) run, using MATLAB's incomplete LU built-in command with a drop tolerance `tol` $= 0.01$:

```
[L,U]=luinc(A,0.01);
[x,flag,relres,iter,resvec]=gmres(A,b,20,1e-8,1000,L,U);
```

The red dotted curve in Figure 7.8 shows the relative residual norms obtained using preconditioned GMRES(20) with the ILUT setting described above. As you can see, preconditioning makes

a significant difference in this case: for a residual value of 10^{-6} the number of iterations required is roughly 40, about one-tenth that of the nonpreconditioned case. We have to be cautious here, since the cost of each iteration is higher when preconditioning is used. But it turns out that the overall computational expense is indeed reduced. The cost of a matrix-vector product can be roughly estimated by the number of nonzeros in the matrix. Here the number of nonzeros of A is 49,600, and the number of nonzeros of L plus the number of nonzeros of U, with the main diagonal counted only once, is 97,227, and hence approximately twice as large. But the saving in iterations is impressive, and it indicates that the preconditioner cost is a price well worth paying, despite all the forward and backward substitutions that are now required. Note also that the residual decreases monotonically, as indeed expected. ∎

For symmetric matrices that are not positive definite, there is a hitch. The most obvious method to use, MINRES, unfortunately requires the preconditioner to be symmetric positive definite, strictly speaking. Thus, a straightforward symmetric version of the ILU approach for constructing a preconditioner may be prohibited by MATLAB in this case.

Specific exercises for this section: Exercises 19–24.

7.6 *Multigrid methods

To motivate the methods briefly introduced in this section, let us consider again the Poisson problem and its discretization (7.1) (page 169) as described in Example 7.1. The condition number of the resulting matrix is $\kappa(A) = \mathcal{O}(n = N^2)$, and thus the number of CG iterations required to reduce the error by a constant amount is $\mathcal{O}(N)$. IC preconditioning helps, as we have seen, but it turns out that the number of iterations required still grows with N (see Figure 7.12(a), coming up soon). Is there an effective method that requires a small, *fixed* number of iterations independent of N?

The answer is affirmative, and to describe such a method we have to dive into the particulars of the problem that is being solved. Since there are many practical problems for which this example may serve as a simple prototype, this excursion turns out to be worthwhile.

Error components with different frequencies

Consider the same Poisson problem discretized twice: once on a finer grid with step size h such that $(N+1)h = 1$, and once on a coarser grid with the larger step size $h_c = 2h$ for which the number of unknowns per spatial variable is $N_c = 1/h_c - 1 \approx N/2$. The coarse grid matrix, A_c, has the eigenvalues

$$\begin{aligned}\lambda^c_{l,m} &= 4 - 2(\cos(l\pi h_c) + \cos(m\pi h_c)) \\ &= 4 - 2(\cos(2l\pi h) + \cos(2m\pi h)), \quad 1 \le l,m \le N_c,\end{aligned}$$

forming a subset of the eigenvalues of the fine grid matrix. We require here that h_c be such that N_c is an integer number. For simplicity, we may consider $N = 2^j - 1$, with j a positive integer.

To the eigenvalues that appear only in the fine set and not in the coarse set correspond eigenvectors that are, at least in one of the grid directions, *highly oscillatory*. Thus, if there is an effective way to deal with the error or residual components corresponding to just the highly oscillatory eigenvectors on the fine grid,[30] then the other components of the error or residual may be dealt with on the coarser grid, where the same operations are roughly 4 times cheaper. This rationale can obviously be repeated, leading to a recursive, multigrid method.

[30] These are commonly referred to as *high-frequency* components of the error or residual.

Smoothing by relaxation

The essential observation that has led to multigrid methods is that there exist simple and cheap *relaxation* schemes such as *Gauss–Seidel*, *damped Jacobi*, or the ILU family of methods that reduce the highly oscillatory components of the residual (or the iteration error) in a given iteration much faster than they reduce the low frequency ones. What this means is that the error, or the residual $\mathbf{r} = \mathbf{b} - A\mathbf{x}$, becomes smoother upon applying such a relaxation, when considered as a grid function, well before it decreases significantly in magnitude.

Example 7.16. Figure 7.10 illustrates the smoothing effect of a simple relaxation scheme. At the top panel, the one-dimensional counterpart of the discretized Poisson equation of Example 7.1[31] with $N = 99$ has the highly oscillatory residual $\mathbf{r} = \sin(5\mathbf{u}) + .1\sin(100\mathbf{u})$, where $\mathbf{u}$ is the vector of grid point values. The bottom panel depicts the same residual after applying four damped Jacobi iterations with $\omega = 0.8$. It has only a slightly smaller magnitude but is much smoother. ∎

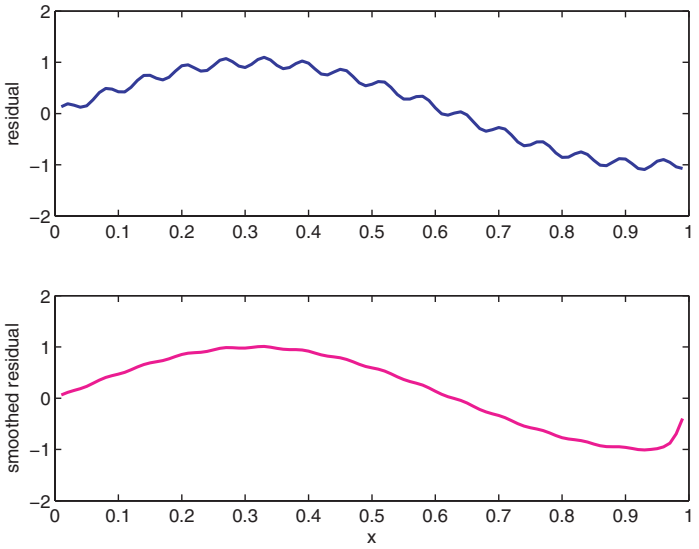


Figure 7.10. *An illustration of the smoothing effect, using damped Jacobi with $\omega = 0.8$ applied to the Poisson equation in one dimension.*

The multigrid cycle

The relaxation operator is thus a **smoother**. Now, it is possible to represent the smoothed residual error well on a coarser grid, thereby obtaining a smaller problem to solve, $A_c\mathbf{v}_c = \mathbf{r}_c$. The obtained correction $\mathbf{v}_c$ is then prolongated (interpolated) back to the finer grid and added to the current solution $\mathbf{x}$, and this is followed by additional relaxation. The process is repeated recursively: for the coarse grid problem the same idea is applied utilizing an even coarser grid. At the coarsest level the problem size is so small that it can be rapidly solved "exactly," say, using a direct method.

The multigrid algorithm given on the following page applies first ν_1 relaxations at the current finer level, then calculates the residual, coarsens it as well as the operator A, solves the coarse grid correction problem approximately γ times starting from the zero correction, prolongates (interpolates) the correction back to the finer grid, and applies ν_2 more relaxations. The whole thing

[31] Specifically, the ODE $-\frac{d^2u}{dx^2} = g(x)$, $0 < x < 1$, $u(0) = u(1) = 0$, is discretized on a uniform grid with $h = .01$.

Algorithm: Multigrid Method.
Given three positive integer parameters ν_1, ν_2, and γ, start at *level* = *finest*:

function $\mathbf{x} =$ `multigrid`$(A, \mathbf{b}, \mathbf{x}, level)$
if level $=$ coarsest then
  solve exactly $\mathbf{x} = A^{-1}\mathbf{b}$
else
  for $j = 1 : \nu_1$
    $\mathbf{x} =$ `relax`$(A, \mathbf{b}, \mathbf{x})$
  end
  $\mathbf{r} = \mathbf{b} - A\mathbf{x}$
  $[A_c, \mathbf{r}_c] =$ `restrict`$(A, \mathbf{r})$
  $\mathbf{v}_c = 0$
  for $l = 1 : \gamma$
    $\mathbf{v}_c =$ `multigrid`$(A_c, \mathbf{r}_c, \mathbf{v}_c, level - 1)$
  end
  $\mathbf{x} = \mathbf{x} +$ `prolongate`$(\mathbf{v}_c)$
  for $j = 1 : \nu_2$
    $\mathbf{x} =$ `relax`$(A, \mathbf{b}, \mathbf{x})$
  end
end

constitutes one multigrid iteration, or a *cycle*, costing $\mathcal{O}(n)$ operations. If $\gamma = 1$, then we have a V-cycle, while if $\gamma = 2$ we have a W-cycle. Typical relaxation values for ν_1 and ν_2 are 1 or 2. Although the W-cycle is better theoretically, in practice the cheaper V-cycle is more popular.

Example 7.17. This is the last time we consider the Poisson problem of Example 7.1: promise. But as befits a grand finale, let's do it in style.

At first, here is our script for carrying out one multigrid V-cycle for this problem. We use damped Jacobi with $\omega = .8$ for relaxation, because it is the simplest effective smoother. The prolongation from coarse to fine grid uses bilinear interpolation. (This is a "broken line" interpolation, as MATLAB does automatically when plotting a curve, in each dimension x and y; for more on this see Sections 11.2 and 11.6.) The restriction is the transpose of the prolongation, and the coarse grid operator is the same five-point formula (7.1) on the coarser grid.

```
function [x,res] = poismg(A,b,x,level);
% multigrid V-cycle to solve simplest Poisson on a square
% The uniform grid is N by N, N = 2^l-1 some l > 2,
% b is the right hand side; homogeneous Dirichlet
% A has been created by   A = delsq(numgrid('S',N+2));

coarsest = 3;              % coarsest grid
nu1 = 2;                   % relaxations before coarsening grid
nu2 = 2;                   % relaxations after return to finer level
omeg = .8;                 % relaxation damping parameter
```

```
if level == coarsest
  x = A\b;               % solve exactly on coarsest level
  r = b - A*x;

else % begin multigrid cycle

  % relax using damped Jacobi
  Dv = diag(A);          % diagonal part of A as vector
  for i=1:nu1
    r = b - A*x;
    x = x + omeg*r./Dv;
  end

  % restrict residual from r to rc on coarser grid
  r = b - A*x;
  N = sqrt(length(b));
  r = reshape(r,N,N);
  Nc = (N+1)/2 - 1; nc = Nc^2;    % coarser grid dimensions
  Ac = delsq(numgrid('S',Nc+2));  % coarser grid operator
  rc = r(2:2:N-1,2:2:N-1) + .5*(r(3:2:N,2:2:N-1)+...
      r(1:2:N-2,2:2:N-1) + r(2:2:N-1,3:2:N)+...
      r(2:2:N-1,1:2:N-2)) + .25*(r(3:2:N,3:2:N)+...
      r(3:2:N,1:2:N-2) + r(1:2:N-2,3:2:N) + r(1:2:N-2,1:2:N-2));
  rc = reshape(rc,nc,1);

  % descend level. Use V-cycle
  vc = zeros(size(rc));            % initialize correction to 0
  [vc,r] = poismg(Ac,rc,vc,level-1); % same on coarser grid

  % prolongate correction from vc to v on finer grid
  v = zeros(N,N);
  vc = reshape(vc,Nc,Nc);
  v(2:2:N-1,2:2:N-1) = vc;
  vz = [zeros(1,N);v;zeros(1,N)];   % embed v with a ring of 0s
  vz = [zeros(N+2,1),vz,zeros(N+2,1)];
  v(1:2:N,2:2:N-1) = .5*(vz(1:2:N,3:2:N)+vz(3:2:N+2,3:2:N));
  v(2:2:N-1,1:2:N) = .5*(vz(3:2:N,1:2:N)+vz(3:2:N,3:2:N+2));
  v(1:2:N,1:2:N) = .25*(vz(1:2:N,1:2:N)+...
      vz(1:2:N,3:2:N+2)+...
      vz(3:2:N+2,3:2:N+2)+vz(3:2:N+2,1:2:N));

  % add to current solution
  n = N^2;
  x = x + reshape(v,n,1);

  % relax using damped Jacobi
  for i=1:nu2
    r = b - A*x;
    x = x + omeg*r./Dv;
  end

end
res = norm(b - A*x);
```

We invoke this *recursive* function for a relative residual tolerance of `tol` = 1.e-6 using something like this:

```
A = delsq(numgrid('S',N+2));
b = A*ones(size(A,1),1);
for itermg = 1:100
  [xmg,rmg(itermg)] = poismg(A,b,xmg,flevel);
  if rmg(itermg)/bb < tol , break, end
end
```

For both a 255×255 grid ($n = 65,025$, $flevel = 8$) and a 511×511 grid ($n = 261,121$, $flevel = 9$), convergence is obtained after just 12 iterations!

We have run the same example also using the CG method described in Section 7.4, as well as CG with two preconditioners: IC(.01) as before, and the multigrid V-cycle as a preconditioner for CG. The progress of the iterations for a given finest grid is depicted in Figure 7.11. For better visualization, we exclude the graph that describes the convergence history of nonpreconditioned CG, which took a bit more than 400 iterations to converge.

This figure is in the spirit of Figures 7.5 and 7.7, but for a much larger matrix dimension n.

Next, let us vary N, i.e., the discretization parameter $h = 1/(N+1)$ in Example 7.1, and observe the convergence behavior for different resolutions. The results are depicted in Figure 7.12. The superiority of the multigrid-based methods for this particular example, especially for large N, is very clear. In particular, the number of iterations required to achieve convergence to within a fixed tolerance is independent of N. ∎

Multigrid method assessment

The stunning performance of the multigrid method demonstrated in Example 7.17 can be extended to many other PDE problems, but not without effort. More generally, the matrix A need not be symmetric positive definite, in fact not even symmetric, although some structure is mandatory if an effective smoother is to be found. See Exercise 27. Also, the coarse grid operator and the prolongation operator may be defined differently for more challenging problems.

There are several other variants that apply also for *nonlinear* PDE problems. The big disadvantage of the multigrid approach is that the resulting methods are fussy, and much fine-tuning is required to work with them. Essentially for this reason the option of using a multigrid iteration as a preconditioner for a Krylov subspace solver is attractive, as the resulting solver is often more robust. The fact that, at least for problems close enough to our model problem, the number of iterations is independent of the matrix size means that as a preconditioner one V-cycle moves the eigenvalues in a way that makes the interval that contains all the eigenvalues virtually independent of the grid size. Hence, also the preconditioned method requires only a fixed number of iterations. This is evident in Figure 7.12(a).

It is interesting to note that even if a "pure" multigrid method fails to converge, one cycle may still be an effective preconditioner. This is because what a Krylov subspace method requires is that (the vast majority of) the eigenvalues be nicely clustered, not that their magnitude be below some stability bound.

Specific exercises for this section: Exercises 25–27.

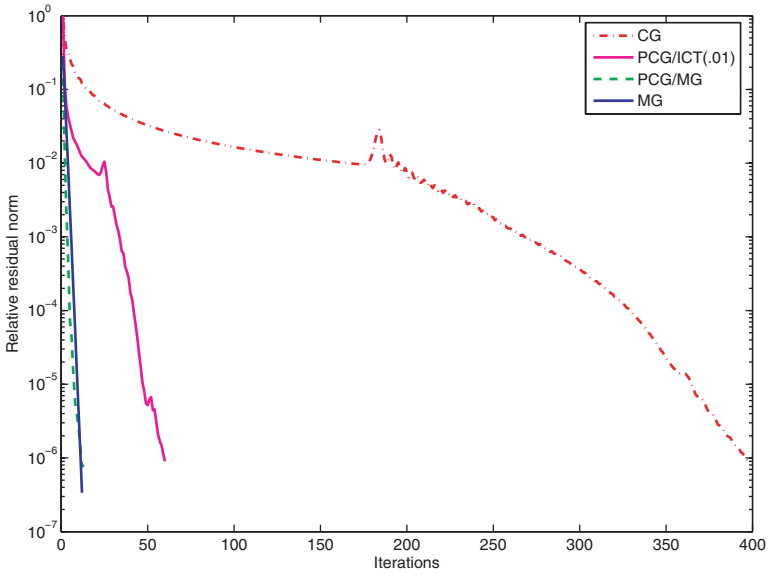


Figure 7.11. *Convergence behavior of various iterative schemes for the Poisson equation (see Example* 7.17*) with* $n = 255^2$.

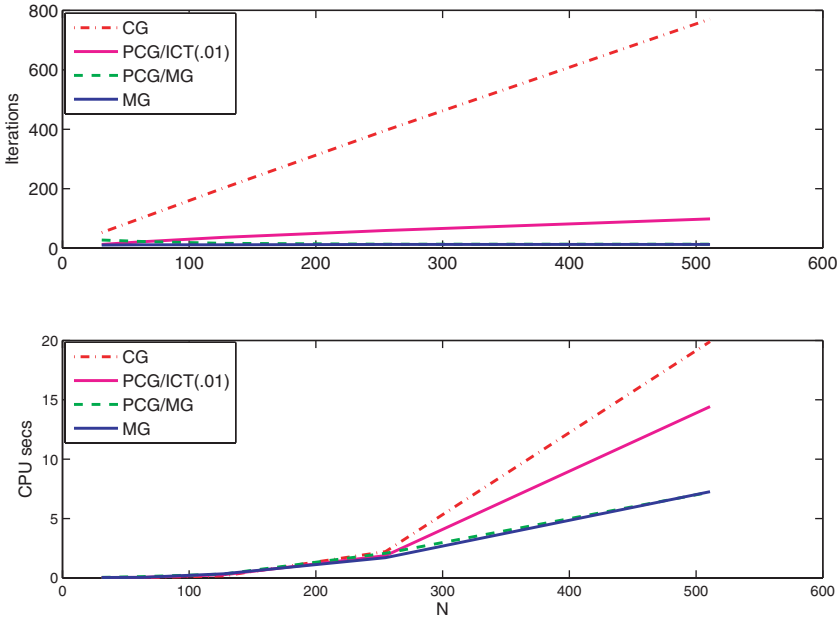


Figure 7.12. *Example* 7.17: *number of iterations (top panel) and CPU times (bottom panel) required to achieve convergence to* `tol`= 1.e-6 *for the Poisson problem of Example* 7.1 *(page* 168*) with* $N = 2^l - 1, l = 5,6,7,8,9$.

7.7 Exercises

0. **Review questions**

   (a) State three disadvantages of direct solution methods that are well addressed by an iterative approach.

   (b) What is a splitting?

   (c) What properties should the matrix that characterizes a splitting have?

   (d) Suppose $A = M - N$. Show that the following schemes are equivalent:

$$M\mathbf{x}_{k+1} = N\mathbf{x}_k + \mathbf{b};$$

$$\mathbf{x}_{k+1} = (I - M^{-1}A)\mathbf{x}_k + M^{-1}\mathbf{b};$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + M^{-1}\mathbf{r}_k, \text{ where } \mathbf{r}_k = \mathbf{b} - A\mathbf{x}_k.$$

   (e) What is an iteration matrix?

   (f) What is a necessary and sufficient condition for convergence of a stationary scheme for any initial guess?

   (g) State one advantage and one disadvantage of the Jacobi relaxation scheme over Gauss–Seidel.

   (h) Write down the iteration matrix corresponding to SOR.

   (i) What special case of the SOR scheme corresponds to Gauss–Seidel?

   (j) What are nonstationary methods?

   (k) How are search direction and step size (or step length) related to the methods of Section 7.4?

   (l) For what type of matrices is the CG method suitable?

   (m) Show that the first CG iteration coincides with that of the steepest descent method.

   (n) What is an energy norm and what is its connection to CG?

   (o) What is a preconditioner?

   (p) What makes incomplete factorizations potentially good preconditioners?

   (q) What disadvantages do iterative methods potentially have that may lead one to occasionally prefer the direct methods of Chapter 5?

1. Let A be a symmetric positive definite $n \times n$ matrix with entries a_{ij} that are nonzero only if one of the following holds: $i = 1$, or $i = n$, or $j = 1$, or $j = n$, or $i = j$. Otherwise, $a_{ij} = 0$.

   (a) Show that only $5n - 6$ of the n^2 elements of A are possibly nonzero.

   (b) Plot the zero-structure of A. (In MATLAB you can invent such a matrix for $n = 20$, say, and use `spy(A)`.)

   (c) Explain why for $n = 100{,}000$ using `chol` (see Section 5.5) to solve $A\mathbf{x} = \mathbf{b}$ for a given right-hand-side vector would be problematic.

2. Consider the problem described in Example 7.1, where the boundary condition $u(0, y) = 0$ is replaced by

$$\frac{\partial u}{\partial x}(0, y) = 0.$$

(Example 4.17 considers such a change in one variable, but here life is harder.) Correspondingly, we change the conditions $u_{0,j} = 0,\ j = 1,\ldots,N$, into

$$4u_{0,j} - 2u_{1,j} - u_{0,j+1} - u_{0,j-1} = b_{0,j}, \quad 1 \le j \le N,$$

where still $u_{0,0} = u_{0,N+1} = 0$.

You don't need to know for the purposes of this exercise why these new linear relations make sense, only that N new unknowns and N new conditions have been introduced.

(a) What is the vector of unknowns $\mathbf{u}$ now? What is $\mathbf{b}$? What is the dimension of the system?

(b) What does A look like?

[Hint: This exercise may not be easy; do it for the case $N = 3$ first!]

3. The linear system

$$\begin{aligned} 10x_1 + x_2 + x_3 &= 12, \\ x_1 + 10x_2 + x_3 &= 12, \\ x_1 + x_2 + 10x_3 &= 12 \end{aligned}$$

has the unique solution $x_1 = x_2 = x_3 = 1$. Starting from $\mathbf{x}_0 = (0,0,0)^T$ perform

- two iterations using Jacobi;
- one iteration using Gauss–Seidel.

Calculate error norms for the three iterations in ℓ_1. Which method seems to converge faster?

4. Consider the 2×2 matrix

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

and suppose we are required to solve $A\mathbf{x} = \mathbf{b}$.

(a) Write down explicitly the iteration matrices corresponding to the Jacobi, Gauss–Seidel, and SOR schemes.

(b) Find the spectral radius of the Jacobi and Gauss–Seidel iteration matrices and the asymptotic rates of convergence for these two schemes.

(c) Plot a graph of the spectral radius of the SOR iteration matrix vs. the relaxation parameter ω for $0 \le \omega \le 2$.

(d) Find the optimal SOR parameter, ω^*. What is the spectral radius of the corresponding iteration matrix? Approximately how much faster would SOR with ω^* converge compared to Jacobi?

5. Consider the class of $n \times n$ matrices A defined in Example 7.1. Denote $N = \sqrt{n}$ and $h = 1/(N+1)$.

(a) Derive a formula for computing the condition number of A in the 2-norm without actually forming this matrix.

How does $\kappa(A)$ depend on n?

(b) Write a program that solves the system of equations $A\mathbf{x} = \mathbf{b}$ for a given N, where the right-hand-side $\mathbf{b}$ is defined to have the same value h^2 in all its n locations. Your program should apply Jacobi iterations without ever storing A or any other $n \times n$ matrix, terminating when the residual $\mathbf{r} = \mathbf{b} - A\mathbf{x}$ satisfies

$$\|\mathbf{r}\| \leq \texttt{tol}\|\mathbf{b}\|.$$

In addition, your program should compute the condition number.

Test the program on a small example, say, $N = 3$, before proceeding further.

(c) Apply your program for the problem instances $N = 2^l - 1$, $l = 2,3,4,5$, using $\texttt{tol} = 10^{-5}$. For each l, record n, the number of Jacobi iterations required to reduce the relative residual to below the tolerance, and the condition number. What relationship do you observe between n and the iteration count? Confirm the theoretical relationship between n and $\kappa(A)$ obtained in Part (a). Explain your observations. (You may wish to plot $\kappa(A)$ vs. n, for instance, to help you focus this observation.)

6. Continuing Exercise 3:

(a) Show that Jacobi's method will converge for this matrix regardless of the starting vector $\mathbf{x}_0$.

(b) Now apply two Jacobi iterations for the problem

$$\begin{aligned} 2x_1 + 5x_2 + 5x_3 &= 12, \\ 5x_1 + 2x_2 + 5x_3 &= 12, \\ 5x_1 + 5x_2 + 2x_3 &= 12, \end{aligned}$$

starting from $\mathbf{x}_0 = (0,0,0)^T$. Does the method appear to converge? Explain why.

7. (a) Show that if the square matrix A is strictly diagonally dominant, then the Jacobi relaxation yields an iteration matrix that satisfies

$$\|T\|_\infty < 1.$$

(b) Show that if A is a 2×2 symmetric positive definite matrix, the Jacobi scheme converges for any initial guess.

8. Consider the matrix

$$A = \begin{pmatrix} 1 & a & a \\ a & 1 & a \\ a & a & 1 \end{pmatrix}.$$

Find the values of a for which A is symmetric positive definite but the Jacobi iteration does not converge.

9. Let α be a scalar, and consider the iterative scheme

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha(\mathbf{b} - A\mathbf{x}_k).$$

This is the gradient descent method with a fixed step size α.

(a) If $A = M - N$ is the splitting associated with this method, state what M and the iteration matrix T are.

(b) Suppose A is symmetric positive definite and its eigenvalues are $\lambda_1 > \lambda_2 > \cdots > \lambda_n > 0$.

i. Derive a condition on α that guarantees convergence of the scheme to the solution $\mathbf{x}$ for any initial guess.

ii. Show that the best value for the step size in terms of maximizing the speed of convergence is

$$\alpha = \frac{2}{\lambda_1 + \lambda_n}.$$

Find the spectral radius of the iteration matrix in this case, and express it in terms of the condition number of A.

(c) Determine whether the following statement is true or false. Justify your answer.

"If A is strictly diagonally dominant and $\alpha = 1$, then the iterative scheme converges to the solution for any initial guess $\mathbf{x}_0$."

10. Consider the two-dimensional partial differential equation

$$-\Delta u + \omega^2 u = f,$$

where $\Delta u = (\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2})$ and ω is a given real scalar, on the unit square $\Omega = (0,1) \times (0,1)$, subject to homogeneous Dirichlet boundary conditions: $u = 0$ on $\partial\Omega$. The matrix that corresponds to a straightforward extension of (7.1) for this differential problem is

$$A(\omega) = A + (\omega h)^2 I,$$

where A is the block tridiagonal matrix from Example 7.1, I is the identity matrix, and h is the grid width, given by $h = \frac{1}{N+1}$. Recall that the eigenvalues of $A(0)$ are

$$\lambda_{\ell,m} = 4 - 2(\cos(l\pi h) + \cos(m\pi h)) \ , \quad 1 \le l, m \le N \ .$$

(a) For a fixed ω, find $\kappa(A) = \kappa_2(A)$ and show that $\kappa(A) = \mathcal{O}(N^2)$.

(b) Explain why for any real ω the matrix $A(\omega)$ is symmetric positive definite.

(c) For $\omega \neq 0$, which of $A(\omega)$ and $A(0)$ is better conditioned?

(d) What is the spectral radius of the Jacobi iteration matrix associated with $A(\omega)$?

11. The *damped Jacobi* (or *under-relaxed Jacobi*) method is briefly described in Sections 7.2 and 7.6. Consider the system $A\mathbf{x} = \mathbf{b}$, and let D be the diagonal part of A.

(a) Write down the corresponding splitting in terms of D, A, and ω.

(b) Suppose A is again the same block tridiagonal matrix arising from discretization of the two-dimensional Poisson equation.

i. Find the eigenvalues of the iteration matrix of the damped Jacobi scheme.

ii. Find the values of $\omega > 0$ for which the scheme converges.

iii. Determine whether there are values $\omega \neq 1$ for which the performance is better than the performance of standard Jacobi, i.e., with $\omega = 1$.

12. Consider the linear system $A\mathbf{x} = \mathbf{b}$, where A is a symmetric matrix. Suppose that $M - N$ is a splitting of A, where M is symmetric positive definite and N is symmetric. Show that if $\lambda_{min}(M) > \rho(N)$, then the iterative scheme $M\mathbf{x}_{k+1} = N\mathbf{x}_k + \mathbf{b}$ converges to $\mathbf{x}$ for any initial guess $\mathbf{x}_0$.

13. Repeat parts (b) and (c) of Exercise 5 with the CG method (without preconditioning) replacing the Jacobi method, and running for the problem instances $N = 2^l - 1$, $l = 2,3,4,5,6$. All other details and requirements remain the same. Explain your observations.

14. Repeat parts (b) and (c) of Exercise 5 with the gradient descent method (without preconditioning) replacing the Jacobi method, and running for the problem instances $N = 2^l - 1$, $l = 2,3,4,5,6$. All other details and requirements remain the same. Try two choices for the step size:

    (a) Steepest descent, given by

    $$\alpha_k = \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{r}_k^T A \mathbf{r}_k} = \frac{\langle \mathbf{r}_k, \mathbf{r}_k \rangle}{\langle \mathbf{r}_k, A\mathbf{r}_k \rangle}.$$

    (b) The same formula applied only every second step, i.e., for k even use the steepest descent formula and for k odd use the already calculated α_{k-1}.

    What are your observations?

15. Show that the energy norm is indeed a norm when the associated matrix is symmetric positive definite.

16. Let A be symmetric positive definite and consider the CG method. Show that for $\mathbf{r}_k$ the residual in the kth iteration and $\mathbf{e}_k$ the error in the kth iteration, the following energy norm identities hold:

    (a) $\|\mathbf{r}_k\|_{A^{-1}} = \|\mathbf{e}_k\|_A$.

    (b) If $\mathbf{x}_k$ minimizes the quadratic function $\phi(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{x}^T\mathbf{b}$ (note that $\mathbf{x}$ here is an argument vector, not the exact solution) over a subspace S, then the same $\mathbf{x}_k$ minimizes the error $\|\mathbf{e}_k\|_A$ over S.

17. Show that the error bound on $\|\mathbf{e}_k\|_A$ given in the CG Convergence Theorem on page 187 implies that the number of iterations k required to achieve an error reduction by a constant amount is bounded by $\sqrt{\kappa(A)}$ times a modest constant.

    [Hint: Recall the discussion of convergence rate from Section 7.3. Also, by Taylor's expansion we can write

    $$\ln(1 \pm \varepsilon) \approx \pm\varepsilon,$$

    for $0 \le \varepsilon \ll 1$.]

18. Write a program that solves the problem described in Exercise 10, with a right-hand-side vector defined so that the solution is a vector of all 1s. For example, for $\omega = 0$ the matrix and right-hand side can be generated by

    ```
    A = delsq(numgrid('S',N+2));
    b = A*ones(N^2,1);
    ```

    Your program will find the numerical solution using the Jacobi, Gauss–Seidel, SOR, and CG methods. For CG use the MATLAB command `pcg` and run it once without preconditioning and once preconditioned with incomplete Cholesky IC(0). For SOR, use the formula for finding the optimal ω^* and apply the scheme only for this value. As a stopping criterion use $\|\mathbf{r}_k\|/\|\mathbf{r}_0\| < 10^{-6}$. Also, impose an upper bound of 2000 iterations. That is, if a scheme fails to satisfy the accuracy requirement on the relative norm of the residual after 2000 iterations, it should be stopped. For each of the methods, start with a zero initial guess. Your program should print out the following:

- Iteration counts for the five cases (Jacobi, Gauss–Seidel, SOR, CG, PCG).
- Plots of the relative residual norms $\frac{\|\mathbf{r}_k\|}{\|\mathbf{b}\|}$ vs. iterations. Use the MATLAB command `semilogy` for plotting.

Use two grids with $N = 31$ and $N = 63$, and repeat your experiments for three values of ω: $\omega = 0$, $\omega^2 = 10$, and $\omega^2 = 1000$. Use your conclusions from Exercise 16 to explain the observed differences in speed of convergence.

19. Suppose CG is applied to a symmetric positive definite linear system $A\mathbf{x} = \mathbf{b}$ where the right-hand-side vector $\mathbf{b}$ happens to be an eigenvector of the matrix A. How many iterations will it take to converge to the solution? Does your answer change if A is not SPD and instead of CG we apply GMRES?

20. (a) Write a program for solving the linear least squares problems that arise throughout the iterations of the GMRES method, using Givens rotations, where the matrix is a nonsquare $(k+1) \times k$ upper Hessenberg matrix. Specifically, solve

$$\min_{\mathbf{z}} \|\rho e_1 - H_{k+1,k}\mathbf{z}\|.$$

Provide a detailed explanation of how this is done in your program, and state what Q and R in the associated QR factorization are.

(b) Given $H_{k+1,k}$, suppose we perform a step of the Arnoldi process. As a result, we now have a new upper Hessenberg matrix $H_{k+2,k+1}$. Describe the relationship between the old and the new upper Hessenberg matrices and explain how this relationship can be used to solve the new least squares problem in an economical fashion.

(c) The least squares problems throughout the iterations can be solved using a QR decomposition approach. Show that the upper triangular factor cannot be singular unless $\mathbf{x}_k = \mathbf{x}$, the exact solution.

21. Consider the saddle point linear system

$$\underbrace{\begin{pmatrix} A & B^T \\ B & 0 \end{pmatrix}}_{\mathcal{K}} \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} = \begin{pmatrix} \mathbf{d} \\ \mathbf{b} \end{pmatrix},$$

where A is $n \times n$, symmetric positive definite and B is $m \times n$ with $m < n$. Consider the preconditioner

$$\mathcal{M} = \begin{pmatrix} A & 0 \\ 0 & BA^{-1}B^T \end{pmatrix}.$$

(a) Show that if B has full row rank, the matrix $\mathcal{K}$ is nonsingular.

(b) Show that $\mathcal{K}$ is symmetric indefinite.

(c) How many iterations does it take for a preconditioned minimum residual scheme to converge in this case, if roundoff errors are ignored? To answer this question, it is recommended to find the eigenvalues of $\mathcal{M}^{-1}\mathcal{K}$.

(d) Practical preconditioners based on this methodology approximate the matrix $BA^{-1}B^T$ (or its inverse), rather than use it as is. Give two good reasons for this.

22. A *skew-symmetric* matrix is a matrix S that satisfies

$$S^T = -S.$$

(a) Show that for any general matrix A, the matrix $(A - A^T)/2$ is skew-symmetric. (This matrix is in fact referred to as the "skew-symmetric part" of A.)

(b) Show that the diagonal of a skew-symmetric matrix S must be zero component-wise.

(c) Show that the eigenvalues of S must be purely imaginary.

(d) If S is $n \times n$ with n an odd number, then it is necessarily singular. Why?

(e) Suppose that the skew-symmetric S is nonsingular and sparse. In the process of solving a linear system associated with S, a procedure equivalent to Arnoldi or Lanczos is applied to form an orthogonal basis for the corresponding Krylov subspace. Suppose the resulting matrices satisfy the relation

$$SQ_k = Q_{k+1}U_{k+1,k},$$

where Q_k is an $n \times k$ matrix whose orthonormal columns form the basis for the Krylov subspace, Q_{k+1} is the matrix of basis vectors containing also the $(k+1)$st basis vector, and $U_{k+1,k}$ is a $(k+1) \times k$ matrix.

i. Determine the nonzero structure of $U_{k+1,k}$. Specifically, state whether it is tridiagonal or upper Hessenberg, and explain what can be said about the values along the main diagonal.

ii. Preconditioners for systems with a dominant skew-symmetric part often deal with the possibility of singularity by solving a *shifted* skew-symmetric system, where instead of solving for S one solves for $S + \beta_k I$ with β_k a scalar. Suppose we have the same right-hand-side, but we need to solve the system for several values of β_k. Can the Arnoldi or Lanczos type procedure outlined above be applied once and for all and then be easily adapted?

iii. Describe the main steps of a MINRES-like method for solving a skew-symmetric linear system.

23. Define a linear problem with $n = 500$ using the script

```
A = randn(500,500); xt = randn(500,1); b = A * xt;
```

Now we save **xt** away and solve $A\mathbf{x} = \mathbf{b}$. Set `tol` = 1.e-6 and maximum iteration limit of 2000. Run three solvers for this problem:

(a) CG on the normal equations: $A^T A\mathbf{x} = A^T\mathbf{b}$.

(b) GMRES(500).

(c) GMRES(100), i.e., restarted GMRES with $m = 100$.

Record residual norm and solution error norm for each run.

What are your conclusions?

24. Let A be a general nonsymmetric nonsingular square matrix, and consider the following two alternatives. The first is applying GMRES to solve the linear system $A\mathbf{x} = \mathbf{b}$; the second is applying CG to the normal equations

$$A^T A\mathbf{x} = A^T\mathbf{b}.$$

We briefly discussed this in Section 7.5; the method we mentioned in that context was CGLS.

(a) Suppose your matrix A is nearly orthogonal. Which of the two solvers is expected to converge faster?

(b) Suppose your matrix is block diagonal relative to 2×2 blocks, where the jth block is given by

$$\begin{pmatrix} 1 & j-1 \\ 0 & 1 \end{pmatrix}$$

with $j = 1, \ldots, n/2$. Which of the two solvers is expected to converge faster?

[Hint: Consider the eigenvalues and the singular values of the matrices.]

25. Consider the Helmholtz equation

$$-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \omega^2 u = g(x, y),$$

defined in the unit square with homogeneous Dirichlet boundary conditions.

(a) Suppose this problem is discretized on a uniform grid with step size $h = 1/(N+1)$ using a five-point scheme as in Example 7.1 plus an additional term. Write down the resulting difference method.

(b) Call the resulting matrix A. Find a value ω_c such that for $\omega^2 < \omega_c^2$ and h arbitrarily small, A is still positive definite, but for $\omega^2 > \omega_c^2$ the positive definiteness is lost.

(c) Solve the problem for $\omega = 1$ and $\omega = 10$ for $N = 2^7 - 1$ using an appropriate preconditioned Krylov subspace method of your choice, or a multigrid method. Use `tol` = 1.e-6. Verify that the last residual norm is below `tol` and tell us how many iterations it took to get there.

26. The *smoothing factor* μ^* for a discrete operator is defined as the worst (i.e., smallest) factor by which high frequency components are reduced in a single relaxation step. For the two-dimensional Laplacian we have discussed throughout this chapter and a basic relaxation scheme, this can be stated as follows. Suppose $\mathbf{e}_0$ is the error before a relaxation step associated with a stationary iteration matrix T and $\mathbf{e}_1$ the error after that step, and write

$$\mathbf{e}_0 = \sum_{l,m=1}^{N} \alpha_{l,m} \mathbf{v}_{l,m},$$

where $\{\mathbf{v}_{l,m}\}_{l,m=1}^{N}$ are the eigenvectors of the iteration matrix. Then

$$\mathbf{e}_1 = \sum_{l,m=1}^{N} \alpha_{l,m} \mu_{l,m} \mathbf{v}_{l,m},$$

where $\{\mu_{l,m}\}_{l,m=1}^{N}$ are eigenvalues of the iteration matrix. The smoothing factor is thus given by

$$\mu^* = \max\left\{|\mu_{l,m}| : \ \frac{N+1}{2} \le l \le N, 1 \le m \le N\right\}.$$

(a) Denote the discrete Laplacian by A and the iteration matrix for damped Jacobi by T_ω. Confirm that the eigenvectors of A are the same as the eigenvectors of T_ω for this scheme. (If you have already worked on Exercise 11, this should be old news.)

(b) Show that the optimal ω that gives the smallest smoothing factor over $0 \le \omega \le 1$ for the two-dimensional Laplacian is $\omega^* = \frac{4}{5}$, and find the smoothing factor $\mu^* = \mu^*(\omega^*)$ in this case. Note: μ^* should not depend on the mesh size.

(c) Show that Jacobi (i.e., the case $\omega = 1$) is not an effective smoother.

27. Write a program that solves the problem of Example 7.13 for $N = 127$ and $N = 255$ using a multigrid method. The script in that example and the code for a V-cycle in Section 7.6 should prove useful with appropriate modification. Set your coarsest level (where the problem is solved exactly, say, by a direct solver) to $N = 31$.

    You should be able to obtain considerably faster convergence than in Example 7.13 at the price of a considerable headache.

7.8 Additional notes

The basic relaxation methods of Section 7.2 are described by just about any text that includes a section on iterative methods for linear systems. They are analyzed in more detail, it almost seems, the older the text is.

The CG method forms the basis of modern day solution techniques for large, sparse, symmetric positive definite systems. See LeVeque [50], Saad [62], and Greenbaum [32] for nice presentations of this method from different angles.

As described in Section 7.5, various Krylov subspace methods have been developed in recent years which extend the CG method to general nonsingular matrices. The book of Saad [62] provides a thorough description, along with a nice framework in the form of projection methods. We have briefly described only a few of those methods. A popular family of methods that we have not discussed are bi-orthogonalization techniques, of which BiCGSTAB is a member; see van der Vorst [72].

While CG is the clear "winner" for symmetric positive definite systems, declaring a winner for general systems is not easy, since different solvers rely on different optimality and spectral properties. A paper by Nachtigal, Reddy, and Trefethen [55] makes this point by showing that for a small collection of methods, there are different problems for which a given method is the winner and another is the loser.

Modern methods for large, sparse systems of equations are rarely as simple as the relaxation methods we have described in Sections 7.2 and 7.3. However, the simple relaxation methods are occasionally used as building blocks for more complex methods. We saw this in Section 7.6: both Gauss–Seidel and damped Jacobi turn out to be effective *smoothers*. Much more on this and other aspects of multigrid methods (including the important class of *algebraic multigrid methods*) can be found in Trottenberg, Oosterlee, and Schuller [71].

For a given practical problem, the challenge usually boils down to finding a good preconditioner. Indeed, these days the focus has shifted from attempting to find new iterative solvers to attempting to find effective preconditioners. Incomplete factorizations are very popular as preconditioners but are by no means the only alternative. If the linear system arises from partial differential equations, then often very effective preconditioners are based on properties of the underlying differential operators. Similarly, if the matrix has a special block structure, preconditioners that exploit that structure are sought. The book by Elman, Silvester, and Wathen [23] illustrates these concepts very well for fluid flow problems.