CHAPTER

8 METHODS RELATED TO THE NORMAL EQUATIONS

There are a number of techniques for converting a non-symmetric linear system into a symmetric one. One such technique solves the equivalent linear system $A^T Ax = A^T b$, called the normal equations. Often, this approach is avoided in practice because the coefficient matrix $A^T A$ is much worse conditioned than A. However, the normal equations approach may be adequate in some situations. Indeed, there are even applications in which it is preferred to the usual Krylov subspace techniques. This chapter covers iterative methods which are either directly or implicitly related to the normal equations.

8.1 THE NORMAL EQUATIONS

In order to solve the linear system $Ax = b$ when A is nonsymmetric, we can solve the equivalent system

$$A^T A\, x = A^T b \tag{8.1}$$

which is Symmetric Positive Definite. This system is known as the system of the *normal equations* associated with the least-squares problem,

$$\text{minimize} \quad \|b - Ax\|_2. \tag{8.2}$$

Note that (8.1) is typically used to solve the least-squares problem (8.2) for *over-determined* systems, i.e., when A is a rectangular matrix of size $n \times m$, $m < n$.

A similar well known alternative sets $x = A^T u$ and solves the following equation for u:

$$AA^T u = b. \tag{8.3}$$

Once the solution u is computed, the original unknown x could be obtained by multiplying u by A^T. However, most of the algorithms we will see do not invoke the u variable explicitly and work with the original variable x instead. The above system of equations can be used to solve *under-determined* systems, i.e., those systems involving rectangular matrices of size $n \times m$, with $n < m$. It is related to (8.1) in the following way. Assume that $n \leq m$ and that A has full rank. Let x_* be *any* solution to the underdetermined system $Ax = b$. Then (8.3) represents the normal equations for the least-squares problem,

$$\text{minimize} \quad \|x_* - A^T u\|_2. \tag{8.4}$$

Since by definition $A^T u = x$, then (8.4) will find the solution vector x that is closest to x_* in the 2-norm sense. What is interesting is that when $n < m$ there are infinitely many solutions x_* to the system $Ax = b$, but the minimizer u of (8.4) does not depend on the particular x_* used.

The system (8.1) and methods derived from it are often labeled with NR (N for "Normal" and R for "Residual") while (8.3) and related techniques are labeled with NE (N for "Normal" and E for "Error"). If A is square and nonsingular, the coefficient matrices of these systems are both Symmetric Positive Definite, and the simpler methods for symmetric problems, such as the Conjugate Gradient algorithm, can be applied. Thus, CGNE denotes the Conjugate Gradient method applied to the system (8.3) and CGNR the Conjugate Gradient method applied to (8.1).

There are several alternative ways to formulate symmetric linear systems having the same solution as the original system. For instance, the symmetric linear system

$$\begin{pmatrix} I & A \\ A^T & O \end{pmatrix} \begin{pmatrix} r \\ x \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix} \tag{8.5}$$

with $r = b - Ax$, arises from the standard necessary conditions satisfied by the solution of the constrained optimization problem,

$$\text{minimize} \quad \frac{1}{2}\|r - b\|_2^2 \tag{8.6}$$

$$\text{subject to} \quad A^T r = 0. \tag{8.7}$$

The solution x to (8.5) is the vector of Lagrange multipliers for the above problem.

Another equivalent symmetric system is of the form

$$\begin{pmatrix} O & A \\ A^T & O \end{pmatrix} \begin{pmatrix} Ax \\ x \end{pmatrix} = \begin{pmatrix} b \\ A^T b \end{pmatrix}.$$

The eigenvalues of the coefficient matrix for this system are $\pm\sigma_i$, where σ_i is an arbitrary singular value of A. Indefinite systems of this sort are not easier to solve than the original nonsymmetric system in general. Although not obvious immediately, this approach is similar in nature to the approach (8.1) and the corresponding Conjugate Gradient iterations applied to them should behave similarly.

A general consensus is that solving the normal equations can be an inefficient approach in the case when A is poorly conditioned. Indeed, the 2-norm condition number of $A^T A$ is given by

$$\text{Cond}_2(A^T A) = \|A^T A\|_2 \, \|(A^T A)^{-1}\|_2.$$

Now observe that $\|A^T A\|_2 = \sigma_{max}^2(A)$ where $\sigma_{max}(A)$ is the largest singular value of A

which, incidentally, is also equal to the 2-norm of A. Thus, using a similar argument for the inverse $(A^TA)^{-1}$ yields

$$\mathrm{Cond}_2(A^TA) = \|A\|_2^2 \, \|A^{-1}\|_2^2 = \mathrm{Cond}_2^2(A). \tag{8.8}$$

The 2-norm condition number for A^TA is exactly the square of the condition number of A, which could cause difficulties. For example, if originally $\mathrm{Cond}_2(A) = 10^8$, then an iterative method may be able to perform reasonably well. However, a condition number of 10^{16} can be much more difficult to handle by a standard iterative method. That is because any progress made in one step of the iterative procedure may be annihilated by the noise due to numerical errors. On the other hand, if the original matrix has a good 2-norm condition number, then the normal equation approach should not cause any serious difficulties. In the extreme case when A is unitary, i.e., when $A^HA = I$, then the normal equations are clearly the best approach (the Conjugate Gradient method will converge in zero step!).

8.2 ROW PROJECTION METHODS

When implementing a basic relaxation scheme, such as Jacobi or SOR, to solve the linear system

$$A^TAx = A^Tb, \tag{8.9}$$

or

$$AA^Tu = b, \tag{8.10}$$

it is possible to exploit the fact that the matrices A^TA or AA^T need not be formed explicitly. As will be seen, only a row or a column of A at a time is needed at a given relaxation step. These methods are known as *row projection methods* since they are indeed projection methods on rows of A or A^T. Block row projection methods can also be defined similarly.

8.2.1 GAUSS-SEIDEL ON THE NORMAL EQUATIONS

It was stated above that in order to use relaxation schemes on the normal equations, only access to one column of A at a time is needed for (8.9) and one row at a time for (8.10). This is now explained for (8.10) first. Starting from an approximation to the solution of (8.10), a basic relaxation-based iterative procedure modifies its components in a certain order using a succession of relaxation steps of the simple form

$$u_{new} = u + \delta_i e_i \tag{8.11}$$

where e_i is the i-th column of the identity matrix. The scalar δ_i is chosen so that the i-th component of the residual vector for (8.10) becomes zero. Therefore,

$$(b - AA^T(u + \delta_i e_i), e_i) = 0 \tag{8.12}$$

which, setting $r = b - AA^T u$, yields,

$$\delta_i = \frac{(r, e_i)}{\|A^T e_i\|_2^2}. \tag{8.13}$$

Denote by β_i the i-th component of b. Then a basic relaxation step consists of taking

$$\delta_i = \frac{\beta_i - (A^T u, A^T e_i)}{\|A^T e_i\|_2^2}. \tag{8.14}$$

Also, (8.11) can be rewritten in terms of x-variables as follows:

$$x_{new} = x + \delta_i A^T e_i. \tag{8.15}$$

The auxiliary variable u has now been removed from the scene and is replaced by the original variable $x = A^T u$.

Consider the implementation of a forward Gauss-Seidel sweep based on (8.15) and (8.13) for a general sparse matrix. The evaluation of δ_i from (8.13) requires the inner product of the current approximation $x = A^T u$ with $A^T e_i$, the i-th row of A. This inner product is inexpensive to compute because $A^T e_i$ is usually sparse. If an acceleration parameter ω is used, we only need to change δ_i into $\omega\delta_i$. Therefore, a forward SOR sweep would be as follows.

ALGORITHM 8.1: Forward NE-SOR Sweep

```
1. Choose an initial $x$.
2. For $i = 1, 2, \ldots, n$ Do:
3.    $\delta_i = \omega \frac{\beta_i - (A^T e_i, x)}{\|A^T e_i\|_2^2}$
4.    $x := x + \delta_i A^T e_i$
5. EndDo
```

Note that $A^T e_i$ is a vector equal to the transpose of the i-th row of A. All that is needed is the row data structure for A to implement the above algorithm. Denoting by nz_i the number of nonzero elements in the i-th row of A, then each step of the above sweep requires $2nz_i + 2$ operations in line 3, and another $2nz_i$ operations in line 4, bringing the total to $4nz_i + 2$. The total for a whole sweep becomes $4nz + 2n$ operations, where nz represents the total number of nonzero elements of A. Twice as many operations are required for the Symmetric Gauss-Seidel or the SSOR iteration. Storage consists of the right-hand side, the vector x, and possibly an additional vector to store the 2-norms of the rows of A. A better alternative would be to rescale each row by its 2-norm at the start.

Similarly, a Gauss-Seidel sweep for (8.9) would consist of a succession of steps of the form

$$x_{new} = x + \delta_i e_i. \tag{8.16}$$

Again, the scalar δ_i is to be selected so that the i-th component of the residual vector for (8.9) becomes zero, which yields

$$(A^T b - A^T A(x + \delta_i e_i), e_i) = 0. \tag{8.17}$$

With $r \equiv b - Ax$, this becomes $(A^T(r - \delta_i Ae_i), e_i) = 0$, which yields

$$\delta_i = \frac{(r, Ae_i)}{\|Ae_i\|_2^2}. \tag{8.18}$$

Then the following algorithm is obtained.

ALGORITHM 8.2: Forward NR-SOR Sweep

```
1. Choose an initial $x$, compute $r := b - Ax$.
2. For $i = 1, 2, \ldots, n$ Do:
3.    $\delta_i = \omega \frac{(r, Ae_i)}{\|Ae_i\|_2^2}$
4.    $x := x + \delta_i e_i$
5.    $r := r - \delta_i Ae_i$
6. EndDo
```

In contrast with Algorithm 8.1, the column data structure of A is now needed for the implementation instead of its row data structure. Here, the right-hand side b can be overwritten by the residual vector r, so the storage requirement is essentially the same as in the previous case. In the NE version, the scalar $\beta_i - (x, a_i)$ is just the i-th component of the current residual vector $r = b - Ax$. As a result, stopping criteria can be built for both algorithms based on either the residual vector or the variation in the solution. Note that the matrices AA^T and A^TA can be dense or generally much less sparse than A, yet the cost of the above implementations depends only on the nonzero structure of A. This is a significant advantage of relaxation-type preconditioners over incomplete factorization preconditioners when using Conjugate Gradient methods to solve the normal equations.

One question remains concerning the acceleration of the above relaxation schemes by under- or over-relaxation. If the usual acceleration parameter ω is introduced, then we only have to multiply the scalars δ_i in the previous algorithms by ω. One serious difficulty here is to determine the optimal relaxation factor. If nothing in particular is known about the matrix AA^T, then the method will converge for any ω lying strictly between 0 and 2, as was seen in Chapter 4, because the matrix is positive definite. Moreover, another unanswered question is how convergence can be affected by various reorderings of the rows. For general sparse matrices, the answer is not known.

8.2.2 CIMMINO'S METHOD

In a Jacobi iteration for the system (8.9), the components of the new iterate satisfy the following condition:

$$(A^Tb - A^TA(x + \delta_i e_i), e_i) = 0. \tag{8.19}$$

This yields

$$(b - A(x + \delta_i e_i), Ae_i) = 0 \quad \text{or} \quad (r - \delta_i Ae_i, Ae_i) = 0$$

in which r is the old residual $b - Ax$. As a result, the i-component of the new iterate x_{new} is given by

$$x_{new,i} = x_i + \delta_i e_i, \tag{8.20}$$

$$\delta_i = \frac{(r, Ae_i)}{\|Ae_i\|_2^2}. \tag{8.21}$$

Here, be aware that these equations do not result in the same approximation as that produced by Algorithm 8.2, even though the modifications are given by the same formula. Indeed, the vector x is not updated after each step and therefore the scalars δ_i are different for the two algorithms. This algorithm is usually described with an acceleration parameter ω, i.e., all δ_i's are multiplied uniformly by a certain ω. If d denotes the vector with coordinates $\delta_i, i = 1, \ldots, n$, the following algorithm results.

ALGORITHM 8.3: Cimmino-NR

```
1. Choose initial guess $x_0$. Set $x = x_0, r = b - Ax_0$
2. Until convergence Do:
3.    For $i = 1, \ldots, n$ Do:
4.        $\delta_i = \omega \frac{(r, Ae_i)}{\|Ae_i\|_2^2}$
5.    EndDo
6.    $x := x + d$ where $d = \sum_{i=1}^n \delta_i e_i$
7.    $r := r - Ad$
8. EndDo
```

Notice that all the coordinates will use the same residual vector r to compute the updates δ_i. When $\omega = 1$, each instance of the above formulas is mathematically equivalent to performing a projection step for solving $Ax = b$ with $\mathcal{K} = \text{span}\{e_i\}$, and $\mathcal{L} = A\mathcal{K}$. It is also mathematically equivalent to performing an orthogonal projection step for solving $A^T Ax = A^T b$ with $\mathcal{K} = \text{span}\{e_i\}$.

It is interesting to note that when each column Ae_i is normalized by its 2-norm, i.e., if $\|Ae_i\|_2 = 1, i = 1, \ldots, n$, then $\delta_i = \omega(r, Ae_i) = \omega(A^T r, e_i)$. In this situation,

$$d = \omega A^T r = \omega A^T (b - Ax)$$

and the main loop of the algorithm takes the vector form

$$d := \omega A^T r$$
$$x := x + d$$
$$r := r - Ad.$$

Each iteration is therefore equivalent to a step of the form

$$x_{new} = x + \omega \left(A^T b - A^T Ax\right)$$

which is nothing but the Richardson iteration applied to the normal equations (8.1). In particular, as was seen in 4.1, convergence is guaranteed for any ω which satisfies,

$$0 < \omega < \frac{2}{\lambda_{max}} \tag{8.22}$$

where λ_{max} is the largest eigenvalue of $A^T A$. In addition, the best acceleration parameter is given by

$$\omega_{opt} = \frac{2}{\lambda_{min} + \lambda_{max}}$$

in which, similarly, λ_{min} is the smallest eigenvalue of $A^T A$. If the columns are not normalized by their 2-norms, then the procedure is equivalent to a preconditioned Richardson iteration with diagonal preconditioning. The theory regarding convergence is similar but involves the preconditioned matrix or, equivalently, the matrix A' obtained from A by normalizing its columns.

The algorithm can be expressed in terms of projectors. Observe that the new residual satisfies

$$r_{new} = r - \sum_{i=1}^{n} \omega \frac{(r, Ae_i)}{\|Ae_i\|_2^2} Ae_i. \tag{8.23}$$

Each of the operators

$$P_i \ : \quad r \longrightarrow \frac{(r, Ae_i)}{\|Ae_i\|_2^2} Ae_i \equiv P_i r \tag{8.24}$$

is an orthogonal projector onto Ae_i, the i-th column of A. Hence, we can write

$$r_{new} = \left(I - \omega \sum_{i=1}^{n} P_i \right) r. \tag{8.25}$$

There are two important variations to the above scheme. First, because the point Jacobi iteration can be very slow, it may be preferable to work with sets of vectors instead. Let $\pi_1, \pi_2, \ldots, \pi_p$ be a partition of the set $\{1, 2, \ldots, n\}$ and, for each π_j, let E_j be the matrix obtained by extracting the columns of the identity matrix whose indices belong to π_j. Going back to the projection framework, define $A_i = AE_i$. If an orthogonal projection method is used onto E_j to solve (8.1), then the new iterate is given by

$$x_{new} = x + \omega \sum_{i}^{p} E_i d_i \tag{8.26}$$

$$d_i = (E_i^T A^T A E_i)^{-1} E_i^T A^T r = (A_i^T A_i)^{-1} A_i^T r. \tag{8.27}$$

Each individual block-component d_i can be obtained by solving a least-squares problem

$$\min_d \|r - A_i d\|_2.$$

An interpretation of this indicates that each individual substep attempts to reduce the residual as much as possible by taking linear combinations from specific columns of A_i. Similar to the scalar iteration, we also have

$$r_{new} = \left(I - \omega \sum_{i=1}^{n} P_i \right) r$$

where P_i now represents an orthogonal projector onto the span of A_i.

Note that $A_1, A_2, \ldots, A_p$ is a partition of the column-set $\{Ae_i\}_{i=1,\ldots,n}$ and this partition can be arbitrary. Another remark is that the original Cimmino method was formulated for rows instead of columns, i.e., it was based on (8.1) instead of (8.3). The alternative algorithm based on columns rather than rows is easy to derive.

8.3 CONJUGATE GRADIENT AND NORMAL EQUATIONS

A popular combination to solve nonsymmetric linear systems applies the Conjugate Gradient algorithm to solve either (8.1) or (8.3). As is shown next, the resulting algorithms can be rearranged because of the particular nature of the coefficient matrices.

8.3.1 CGNR

We begin with the Conjugate Gradient algorithm applied to (8.1). Applying CG directly to the system and denoting by z_i the residual vector at step i (instead of r_i) results in the following sequence of operations:

- $\alpha_j := (z_j, z_j)/(A^T A p_j, p_j) = (z_j, z_j)/(Ap_j, Ap_j)$
- $x_{j+1} := x_j + \alpha_j p_j$
- $z_{j+1} := z_j - \alpha_j A^T A p_j$
- $\beta_j := (z_{j+1}, z_{j+1})/(z_j, z_j)$
- $p_{j+1} := z_{j+1} + \beta_j p_j$.

If the original residual $r_i = b - Ax_i$ must be available at every step, we may compute the residual z_{i+1} in two parts: $r_{j+1} := r_j - \alpha_j A p_j$ and then $z_{i+1} = A^T r_{i+1}$ which is the residual for the normal equations (8.1). It is also convenient to introduce the vector $w_i = Ap_i$. With these definitions, the algorithm can be cast in the following form.

ALGORITHM 8.4: CGNR

```
 1. Compute $r_0 = b - Ax_0$, $z_0 = A^T r_0$, $p_0 = z_0$.
 2. For $i = 0, \ldots,$ until convergence Do:
 3.    $w_i = Ap_i$
 4.    $\alpha_i = \|z_i\|^2 / \|w_i\|_2^2$
 5.    $x_{i+1} = x_i + \alpha_i p_i$
 6.    $r_{i+1} = r_i - \alpha_i w_i$
 7.    $z_{i+1} = A^T r_{i+1}$
 8.    $\beta_i = \|z_{i+1}\|_2^2 / \|z_i\|_2^2$,
 9.    $p_{i+1} = z_{i+1} + \beta_i p_i$
10. EndDo
```

In Chapter 6, the approximation x_m produced at the m-th step of the Conjugate Gradient algorithm was shown to minimize the energy norm of the error over an affine Krylov subspace. In this case, x_m minimizes the function

$$f(x) \equiv (A^T A(x_* - x), (x_* - x))$$

over all vectors x in the affine Krylov subspace

$$x_0 + \mathcal{K}_m(A^T A, A^T r_0) = x_0 + \text{span}\{A^T r_0, A^T A A^T r_0, \ldots, (A^T A)^{m-1} A^T r_0\},$$

in which $r_0 = b - Ax_0$ is the initial residual with respect to the original equations $Ax = b$, and $A^T r_0$ is the residual with respect to the normal equations $A^T Ax = A^T b$. However, observe that

$$f(x) = (A(x_* - x), A(x_* - x)) = \|b - Ax\|_2^2.$$

Therefore, CGNR produces the approximate solution in the above subspace which has the smallest residual norm with respect to the original linear system $Ax = b$. The difference with the GMRES algorithm seen in Chapter 6, is the subspace in which the residual norm is minimized.

Example 8.1 Table 8.1 shows the results of applying the CGNR algorithm with no preconditioning to three of the test problems described in Section 3.7.

Matrix	Iters	Kflops	Residual	Error
F2DA	300	4847	0.23E+02	0.62E+00
F3D	300	23704	0.42E+00	0.15E+00
ORS	300	5981	0.30E+02	0.60E-02

Table 8.1 *A test run of CGNR with no preconditioning.*

See Example 6.1 for the meaning of the column headers in the table. The method failed to converge in less than 300 steps for all three problems. Failures of this type, characterized by very slow convergence, are rather common for CGNE and CGNR applied to problems arising from partial differential equations. Preconditioning should improve performance somewhat but, as will be seen in Chapter 10, normal equations are also difficult to precondition.

8.3.2 CGNE

A similar reorganization of the CG algorithm is possible for the system (8.3) as well. Applying the CG algorithm directly to (8.3) and denoting by q_i the conjugate directions, the actual CG iteration for the u variable would be as follows:

- $\alpha_j := (r_j, r_j)/(AA^T q_j, q_j) = (r_j, r_j)/(A^T q_j, A^T q_j)$
- $u_{j+1} := u_j + \alpha_j q_j$
- $r_{j+1} := r_j - \alpha_j AA^T q_j$
- $\beta_j := (r_{j+1}, r_{j+1})/(r_j, r_j)$

- $q_{j+1} := r_{j+1} + \beta_j q_j$.

Notice that an iteration can be written with the original variable $x_i = x_0 + A^T(u_i - u_0)$ by introducing the vector $p_i = A^T q_i$. Then, the residual vectors for the vectors x_i and u_i are the same. No longer are the q_i vectors needed because the p_i's can be obtained as $p_{j+1} := A^T r_{j+1} + \beta_j p_j$. The resulting algorithm described below, the Conjugate Gradient for the normal equations (CGNE), is also known as Craig's method.

ALGORITHM 8.5: CGNE (Craig's Method)

```
1. Compute $r_0 = b - Ax_0$, $p_0 = A^T r_0$.
2. For $i = 0, 1, \ldots,$ until convergence Do:
3.    $\alpha_i = (r_i, r_i)/(p_i, p_i)$
4.    $x_{i+1} = x_i + \alpha_i p_i$
5.    $r_{i+1} = r_i - \alpha_i A p_i$
6.    $\beta_i = (r_{i+1}, r_{i+1})/(r_i, r_i)$
7.    $p_{i+1} = A^T r_{i+1} + \beta_i p_i$
8. EndDo
```

We now explore the optimality properties of this algorithm, as was done for CGNR. The approximation u_m related to the variable x_m by $x_m = A^T u_m$ is the actual m-th CG approximation for the linear system (8.3). Therefore, it minimizes the energy norm of the error on the Krylov subspace $\mathcal{K}_m$. In this case, u_m minimizes the function

$$f(u) \equiv (AA^T(u_* - u), (u_* - u))$$

over all vectors u in the affine Krylov subspace,

$$u_0 + \mathcal{K}_m(AA^T, r_0) = u_0 + \text{span}\{r_0, AA^T r_0, \ldots, (AA^T)^{m-1} r_0\}.$$

Notice that $r_0 = b - AA^T u_0 = b - Ax_0$. Also, observe that

$$f(u) = (A^T(u_* - u), A^T(u_* - u)) = \|x_* - x\|_2^2,$$

where $x = A^T u$. Therefore, CGNE produces the approximate solution in the subspace

$$x_0 + A^T \mathcal{K}_m(AA^T, r_0) = x_0 + \mathcal{K}_m(A^T A, A^T r_0)$$

which has the smallest 2-norm of the error. In addition, note that the subspace $x_0 + \mathcal{K}_m(A^T A, A^T r_0)$ is identical with the subspace found for CGNR. Therefore, *the two methods find approximations from the same subspace which achieve different optimality properties: minimal residual for CGNR and minimal error for CGNE.*

8.4 SADDLE-POINT PROBLEMS

Now consider the equivalent system

$$\begin{pmatrix} I & A \\ A^T & O \end{pmatrix} \begin{pmatrix} r \\ x \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix}$$

with $r = b - Ax$. This system can be derived from the necessary conditions applied to the constrained least-squares problem (8.6–8.7). Thus, the 2-norm of $b - r = Ax$ is minimized implicitly under the constraint $A^T r = 0$. Note that A does not have to be a square matrix.

This can be extended into a more general constrained quadratic optimization problem as follows:

$$\text{minimize } f(x) \equiv \frac{1}{2}(Ax, x) - (x, b) \tag{8.28}$$

$$\text{subject to } B^T x = c. \tag{8.29}$$

The necessary conditions for optimality yield the linear system

$$\begin{pmatrix} A & B \\ B^T & O \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} b \\ c \end{pmatrix} \tag{8.30}$$

in which the names of the variables r, x are changed into x, y for notational convenience. It is assumed that the column dimension of B does not exceed its row dimension. The Lagrangian for the above optimization problem is

$$L(x, y) = \frac{1}{2}(Ax, x) - (x, b) + (y, (B^T x - c))$$

and the solution of (8.30) is the saddle point of the above Lagrangian. Optimization problems of the form (8.28–8.29) and the corresponding linear systems (8.30) are important and arise in many applications. Because they are intimately related to the normal equations, we discuss them briefly here.

In the context of fluid dynamics, a well known iteration technique for solving the linear system (8.30) is Uzawa's method, which resembles a relaxed block SOR iteration.

ALGORITHM 8.6: Uzawa's Method

```
1. Choose $x_0, y_0$
2. For $k = 0, 1, \ldots,$ until convergence Do:
3.     $x_{k+1} = A^{-1}(b - By_k)$
4.     $y_{k+1} = y_k + \omega(B^T x_{k+1} - c)$
5. EndDo
```

The algorithm requires the solution of the linear system

$$Ax_{k+1} = b - By_k \tag{8.31}$$

at each iteration. By substituting the result of line 3 into line 4, the x_k iterates can be eliminated to obtain the following relation for the y_k's,

$$y_{k+1} = y_k + \omega \left(B^T A^{-1}(b - By_k) - c \right)$$

which is nothing but a Richardson iteration for solving the linear system

$$B^T A^{-1} B y = B^T A^{-1} b - c. \tag{8.32}$$

Apart from a sign, this system is the reduced system resulting from eliminating the x variable from (8.30). Convergence results can be derived from the analysis of the Richardson iteration.

COROLLARY 8.1 *Let A be a Symmetric Positive Definite matrix and B a matrix of full rank. Then $S = B^T A^{-1} B$ is also Symmetric Positive Definite and Uzawa's algorithm converges, if and only if*

$$0 < \omega < \frac{2}{\lambda_{max}(S)}. \tag{8.33}$$

In addition, the optimal convergence parameter ω is given by

$$\omega_{opt} = \frac{2}{\lambda_{min}(S) + \lambda_{max}(S)}.$$

Proof. The proof of this result is straightforward and is based on the results seen in Example 4.1. ■

It is interesting to observe that when $c = 0$ and A is Symmetric Positive Definite, then the system (8.32) can be regarded as the normal equations for minimizing the A^{-1}-norm of $b - By$. Indeed, the optimality conditions are equivalent to the orthogonality conditions

$$(b - By, Bw)_{A^{-1}} = 0, \quad \forall\, w,$$

which translate into the linear system $B^T A^{-1} B y = B^T A^{-1} b$. As a consequence, the problem will tend to be easier to solve if the columns of B are almost orthogonal with respect to the A^{-1} inner product. This is true when solving the *Stokes problem* where B represents the discretization of the gradient operator while B^T discretizes the divergence operator, and A is the discretization of a Laplacian. In this case, if it were not for the boundary conditions, the matrix $B^T A^{-1} B$ would be the identity. This feature can be exploited in developing preconditioners for solving problems of the form (8.30). Another particular case is when A is the identity matrix and $c = 0$. Then, the linear system (8.32) becomes the system of the normal equations for minimizing the 2-norm of $b - By$. These relations provide insight in understanding that the block form (8.30) is actually a form of normal equations for solving $By = b$ in the least-squares sense. However, a different inner product is used.

In Uzawa's method, a linear system at each step must be solved, namely, the system (8.31). Solving this system is equivalent to finding the minimum of the quadratic function

$$\text{minimize} \quad f_k(x) \;\equiv\; \frac{1}{2}(Ax, x) - (x, b - By_k). \tag{8.34}$$

Apart from constants, $f_k(x)$ is the Lagrangian evaluated at the previous y iterate. The solution of (8.31), or the equivalent optimization problem (8.34), is expensive. A common alternative replaces the x-variable update (8.31) by taking one step in the gradient direction for the quadratic function (8.34), usually with fixed step-length ϵ. The gradient of $f_k(x)$ at the current iterate is $Ax_k - (b - By_k)$. This results in the Arrow-Hurwicz Algorithm.

ALGORITHM 8.7: The Arrow-Hurwicz algorithm

```
1. Select an initial guess $x_0, y_0$ to the system (8.30)
2. For $k = 0, 1, \ldots,$ until convergence Do:
3.    Compute $x_{k+1} = x_k + \epsilon(b - Ax_k - By_k)$
4.    Compute $y_{k+1} = y_k + \omega(B^T x_{k+1} - c)$
5. EndDo
```

The above algorithm is a block-iteration of the form

$$\begin{pmatrix} I & O \\ -\omega B^T & I \end{pmatrix} \begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} I - \epsilon A & -\epsilon B \\ O & I \end{pmatrix} \begin{pmatrix} x_k \\ y_k \end{pmatrix} + \begin{pmatrix} \epsilon b \\ -\omega c \end{pmatrix}.$$

Uzawa's method, and many similar techniques for solving (8.30), are based on solving the reduced system (8.32). An important observation here is that the Schur complement matrix $S \equiv B^T A^{-1} B$ need not be formed explicitly. This can be useful if this reduced system is to be solved by an iterative method. The matrix A is typically factored by a Cholesky-type factorization. The linear systems with the coefficient matrix A can also be solved by a preconditioned Conjugate Gradient method. Of course these systems must then be solved accurately.

Sometimes it is useful to "regularize" the least-squares problem (8.28) by solving the following problem in its place:

$$\begin{aligned} &\text{minimize } f(x) \equiv \frac{1}{2}(Ax, x) - (x, b) + \rho(Cy, y) \\ &\text{subject to } B^T x = c \end{aligned}$$

in which ρ is a scalar parameter. For example, C can be the identity matrix or the matrix $B^T B$. The matrix resulting from the Lagrange multipliers approach then becomes

$$\begin{pmatrix} A & B \\ B^T & \rho C \end{pmatrix}.$$

The new Schur complement matrix is

$$S = \rho C - B^T A^{-1} B.$$

Example 8.2 In the case where $C = B^T B$, the above matrix takes the form

$$S = B^T(\rho I - A^{-1})B.$$

Assuming that A is SPD, S is also positive definite when

$$\rho \geq \frac{1}{\lambda_{min}(A)}.$$

However, it is also *negative definite* for

$$\rho \leq \frac{1}{\lambda_{max}}(A),$$

a condition which may be easier to satisfy on practice.

EXERCISES

1. Derive the linear system (8.5) by expressing the standard necessary conditions for the problem (8.6–8.7).

2. It was stated in Section 8.2.2 that when $\|A^T e_i\|_2 = 1$ for $i = 1, \ldots, n$, the vector d defined in Algorithm 8.3 is equal to $\omega A^T r$.
   - ***a.*** What does this become in the general situation when $\|A^T e_i\|_2 \neq 1$?
   - ***b.*** Is Cimmino's method still equivalent to a Richardson iteration?
   - ***c.*** Show convergence results similar to those of the scaled case.

3. In Section 8.2.2, Cimmino's algorithm was derived based on the Normal Residual formulation, i.e., on (8.1). Derive an "NE" formulation, i.e., an algorithm based on Jacobi's method for (8.3).

4. What are the eigenvalues of the matrix (8.5)? Derive a system whose coefficient matrix has the form
$$B(\alpha) = \begin{pmatrix} 2\alpha I & A \\ A^T & O \end{pmatrix}.$$
and which is also equivalent to the original system $Ax = b$. What are the eigenvalues of $B(\alpha)$? Plot the spectral norm of $B(\alpha)$ as a function of α.

5. It was argued in Section 8.4 that when $c = 0$ the system (8.32) is nothing but the normal equations for minimizing the A^{-1}-norm of the residual $r = b - By$.
   - ***a.*** Write the associated CGNR approach for solving this problem. Find a variant that requires only one linear system solution with the matrix A at each CG step [Hint: Write the CG algorithm for the associated normal equations and see how the resulting procedure can be reorganized to save operations]. Find also a variant that is suitable for the case where the Cholesky factorization of A is available.
   - ***b.*** Derive a method for solving the equivalent system (8.30) for the case when $c = 0$ and then for the general case wjen $c \neq 0$. How does this technique compare with Uzawa's method?

6. Consider the linear system (8.30) in which $c = 0$ and B is of full rank. Define the matrix
$$P = I - B(B^T B)^{-1} B^T.$$
   - ***a.*** Show that P is a projector. Is it an orthogonal projector? What are the range and null spaces of P?
   - ***b.*** Show that the unknown x can be found by solving the linear system
   $$PAPx = Pb, \tag{8.35}$$
   in which the coefficient matrix is singular but the system is consistent, i.e., there is a nontrivial solution because the right-hand side is in the range of the matrix (see Chapter 1).
   - ***c.*** What must be done toadapt the Conjugate Gradient Algorithm for solving the above linear system (which is symmetric, but not positive definite)? In which subspace are the iterates generated from the CG algorithm applied to (8.35)?

d. Assume that the QR factorization of the matrix B is computed. Write an algorithm based on the approach of the previous questions for solving the linear system (8.30).

7. Show that Uzawa's iteration can be formulated as a fixed-point iteration associated with the splitting $C = M - N$ with

$$M = \begin{pmatrix} A & O \\ -\omega B^T & I \end{pmatrix}, \quad N = \begin{pmatrix} O & -B \\ O & I \end{pmatrix}.$$

Derive the convergence result of Corollary 8.1 .

8. Show that each new vector iterate in Cimmino's method is such that

$$x_{new} = x + \omega A^{-1} \sum_i P_i r,$$

where P_i is defined by (8.24).

9. In Uzawa's method a linear system with the matrix A must be solved at each step. Assume that these systems are solved inaccurately by an iterative process. For each linear system the iterative process is applied until the norm of the residual $r_{k+1} = (b - By_k) - Ax_{k+1}$ is less than a certain threshold ϵ_{k+1}.

a. Assume that ω is chosen so that (8.33) is satisfied and that ϵ_k converges to zero as k tends to infinity. Show that the resulting algorithm converges to the solution.

b. Give an explicit upper bound of the error on y_k in the case when ϵ_i is chosen of the form $\epsilon = \alpha^i$, where $\alpha < 1$.

10. Assume $\|b - Ax\|_2$ is to be minimized, in which A is $n \times m$ with $n > m$. Let x_* be the minimizer and $r = b - Ax_*$. What is the minimizer of $\|(b + \alpha r) - Ax\|_2$, where α is an arbitrary scalar?

NOTES AND REFERENCES. Methods based on the normal equations have been among the first to be used for solving nonsymmetric linear systems [130, 58] by iterative methods. The work by Bjork and Elfing [27], and Sameh et al. [131, 37, 36] revived these techniques by showing that they have some advantages from the implementation point of view, and that they can offer good performance for a broad class of problems. In addition, they are also attractive for parallel computers. In [174], a few preconditioning ideas for normal equations were described and these will be covered in Chapter 10. It would be helpful to be able to determine whether or not it is preferable to use the normal equations approach rather than the "direct equations" for a given system, but this may require an eigenvalue/singular value analysis.

It is sometimes argued that the normal equations approach is *always* better, because it has a robust quality which outweighs the additional cost due to the slowness of the method in the generic elliptic case. Unfortunately, this is not true. Although variants of the Kaczmarz and Cimmino algorithms deserve a place in any robust iterative solution package, they cannot be viewed as a panacea. In *most* realistic examples arising from Partial Differential Equations, the normal equations route gives rise to much slower convergence than the Krylov subspace approach for the direct equations. For ill-conditioned problems, these methods will simply fail to converge, unless a good preconditioner is available. ■

CHAPTER

9 PRECONDITIONED ITERATIONS

Although the methods seen in previous chapters are well founded theoretically, they are all likely to suffer from slow convergence for problems which arise from typical applications such as fluid dynamics or electronic device simulation. Preconditioning is a key ingredient for the success of Krylov subspace methods in these applications. This chapter discusses the preconditioned versions of the iterative methods already seen, but without being specific about the particular preconditioners used. The standard preconditioning techniques will be covered in the next chapter.

9.1 INTRODUCTION

Lack of robustness is a widely recognized weakness of iterative solvers, relative to direct solvers. This drawback hampers the acceptance of iterative methods in industrial applications despite their intrinsic appeal for very large linear systems. Both the efficiency and robustness of iterative techniques can be improved by using *preconditioning*. A term introduced in Chapter 4, preconditioning is simply a means of transforming the original linear system into one which has the same solution, but which is likely to be easier to solve with an iterative solver. In general, the reliability of iterative techniques, when dealing with various applications, depends much more on the quality of the preconditioner than on the particular Krylov subspace accelerators used. We will cover some of these preconditioners in detail in the next chapter. This chapter discusses the preconditioned versions of the Krylov subspace algorithms already seen, using a generic preconditioner.

PRECONDITIONED CONJUGATE GRADIENT 9.2

Consider a matrix A that is symmetric and positive definite and assume that a preconditioner M is available. The preconditioner M is a matrix which approximates A in some yet-undefined sense. It is assumed that M is also Symmetric Positive Definite. From a practical point of view, the only requirement for M is that it is inexpensive to solve linear systems $Mx = b$. This is because the preconditioned algorithms will all require a linear system solution with the matrix M at each step. Then, for example, the following preconditioned system could be solved:

$$M^{-1}Ax = M^{-1}b \tag{9.1}$$

or

$$AM^{-1}u = b, \quad x = M^{-1}u. \tag{9.2}$$

Note that these two systems are no longer symmetric in general. The next section considers strategies for preserving symmetry. Then, efficient implementations will be described for particular forms of the preconditioners.

9.2.1 PRESERVING SYMMETRY

When M is available in the form of an incomplete Cholesky factorization, i.e., when

$$M = LL^T,$$

then a simple way to preserve symmetry is to “split” the preconditioner between left and right, i.e., to solve

$$L^{-1}AL^{-T}u = L^{-1}b, \quad x = L^{-T}u, \tag{9.3}$$

which involves a Symmetric Positive Definite matrix.

However, it is not necessary to split the preconditioner in this manner in order to preserve symmetry. Observe that $M^{-1}A$ is self-adjoint for the M-inner product,

$$(x, y)_M \equiv (Mx, y) = (x, My)$$

since

$$(M^{-1}Ax, y)_M = (Ax, y) = (x, Ay) = (x, M(M^{-1}A)y) = (x, M^{-1}Ay)_M.$$

Therefore, an alternative is to replace the usual Euclidean inner product in the Conjugate Gradient algorithm by the M-inner product.

If the CG algorithm is rewritten for this new inner product, denoting by $r_j = b - Ax_j$ the original residual and by $z_j = M^{-1}r_j$ the residual for the preconditioned system, the following sequence of operations is obtained, ignoring the initial step:

1. $\alpha_j := (z_j, z_j)_M / (M^{-1}Ap_j, p_j)_M$
2. $x_{j+1} := x_j + \alpha_j p_j$

3. $r_{j+1} := r_j - \alpha_j A p_j$ and $z_{j+1} := M^{-1} r_{j+1}$

4. $\beta_j := (z_{j+1}, z_{j+1})_M / (z_j, z_j)_M$

5. $p_{j+1} := z_{j+1} + \beta_j p_j$

Since $(z_j, z_j)_M = (r_j, z_j)$ and $(M^{-1} A p_j, p_j)_M = (A p_j, p_j)$, the M-inner products do not have to be computed explicitly. With this observation, the following algorithm is obtained.

ALGORITHM 9.1: Preconditioned Conjugate Gradient

```
1. Compute $r_0 := b - Ax_0$, $z_0 = M^{-1} r_0$, and $p_0 := z_0$
2. For $j = 0, 1, \ldots$, until convergence Do:
3.    $\alpha_j := (r_j, z_j)/(Ap_j, p_j)$
4.    $x_{j+1} := x_j + \alpha_j p_j$
5.    $r_{j+1} := r_j - \alpha_j A p_j$
6.    $z_{j+1} := M^{-1} r_{j+1}$
7.    $\beta_j := (r_{j+1}, z_{j+1})/(r_j, z_j)$
8.    $p_{j+1} := z_{j+1} + \beta_j p_j$
9. EndDo
```

It is interesting to observe that $M^{-1}A$ is also self-adjoint with respect to the A inner-product. Indeed,

$$(M^{-1}Ax, y)_A = (AM^{-1}Ax, y) = (x, AM^{-1}Ay) = (x, M^{-1}Ay)_A$$

and a similar algorithm can be written for this dot product (see Exercise 1).

In the case where M is a Cholesky product $M = LL^T$, two options are available, namely, the split preconditioning option (9.3), or the above algorithm. An immediate question arises about the iterates produced by these two options: Is one better than the other? Surprisingly, the answer is that *the iterates are identical*. To see this, start from Algorithm 9.1 and define the following auxiliary vectors and matrix from it:

$$\begin{aligned} \hat{p}_j &= L^T p_j \\ u_j &= L^T x_j \\ \hat{r}_j &= L^T z_j = L^{-1} r_j \\ \hat{A} &= L^{-1} A L^{-T}. \end{aligned}$$

Observe that

$$(r_j, z_j) = (r_j, L^{-T} L^{-1} r_j) = (L^{-1} r_j, L^{-1} r_j) = (\hat{r}_j, \hat{r}_j).$$

Similarly,

$$(Ap_j, p_j) = (AL^{-T}\hat{p}_j, L^{-T}\hat{p}_j)(L^{-1}AL^{-T}\hat{p}_j, \hat{p}_j) = (\hat{A}\hat{p}_j, \hat{p}_j).$$

All the steps of the algorithm can be rewritten with the new variables, yielding the following sequence of operations:

1. $\alpha_j := (\hat{r}_j, \hat{r}_j)/(\hat{A}\hat{p}_j, \hat{p}_j)$

2. $u_{j+1} := u_j + \alpha_j \hat{p}_j$

3. $\hat{r}_{j+1} := \hat{r}_j - \alpha_j \hat{A} \hat{p}_j$

4. $\beta_j := (\hat{r}_{j+1}, \hat{r}_{j+1}) / (\hat{r}_j, \hat{r}_j)$

5. $\hat{p}_{j+1} := \hat{r}_{j+1} + \beta_j \hat{p}_j$.

This is precisely the Conjugate Gradient algorithm applied to the preconditioned system

$$\hat{A}u = L^{-1}b$$

where $u = L^T x$. It is common when implementing algorithms which involve a right preconditioner to avoid the use of the u variable, since the iteration can be written with the original x variable. If the above steps are rewritten with the original x and p variables, the following algorithm results.

ALGORITHM 9.2: Split Preconditioner Conjugate Gradient

```
1. Compute $r_0 := b - Ax_0$; $\hat{r}_0 = L^{-1} r_0$; and $p_0 := L^{-T} \hat{r}_0$.
2. For $j = 0, 1, \ldots$, until convergence Do:
3.    $\alpha_j := (\hat{r}_j, \hat{r}_j) / (Ap_j, p_j)$
4.    $x_{j+1} := x_j + \alpha_j p_j$
5.    $\hat{r}_{j+1} := \hat{r}_j - \alpha_j L^{-1} A p_j$
6.    $\beta_j := (\hat{r}_{j+1}, \hat{r}_{j+1}) / (\hat{r}_j, \hat{r}_j)$
7.    $p_{j+1} := L^{-T} \hat{r}_{j+1} + \beta_j p_j$
8. EndDo
```

The iterates x_j produced by the above algorithm and Algorithm 9.1 are identical, provided the same initial guess is used.

Consider now the right preconditioned system (9.2). The matrix AM^{-1} is not Hermitian with either the Standard inner product or the M-inner product. However, it is Hermitian with respect to the M^{-1}-inner product. If the CG-algorithm is written with respect to the u-variable and for this new inner product, the following sequence of operations would be obtained, ignoring again the initial step:

1. $\alpha_j := (r_j, r_j)_{M^{-1}} / (AM^{-1} p_j, p_j)_{M^{-1}}$

2. $u_{j+1} := u_j + \alpha_j p_j$

3. $r_{j+1} := r_j - \alpha_j AM^{-1} p_j$

4. $\beta_j := (r_{j+1}, r_{j+1})_{M^{-1}} / (r_j, r_j)_{M^{-1}}$

5. $p_{j+1} := r_{j+1} + \beta_j p_j$.

Recall that the u vectors and the x vectors are related by $x = M^{-1}u$. Since the u vectors are not actually needed, the update for u_{j+1} in the second step can be replaced by $x_{j+1} := x_j + \alpha_j M^{-1} p_j$. Then observe that the whole algorithm can be recast in terms of $q_j = M^{-1} p_j$ and $z_j = M^{-1} r_j$.

1. $\alpha_j := (z_j, r_j) / (Aq_j, q_j)$

2. $x_{j+1} := x_j + \alpha_j q_j$

3. $r_{j+1} := r_j - \alpha_j A q_j$ and $z_{j+1} = M^{-1} r_{j+1}$

4. $\beta_j := (z_{j+1}, r_{j+1}) / (z_j, r_j)$

5. $q_{j+1} := z_{j+1} + \beta_j q_j$.

Notice that the same sequence of computations is obtained as with Algorithm 9.1, the left preconditioned Conjugate Gradient. The implication is that *the left preconditioned CG algorithm with the M-inner product is mathematically equivalent to the right preconditioned CG algorithm with the M^{-1}-inner product.*

9.2.2 EFFICIENT IMPLEMENTATIONS

When applying a Krylov subspace procedure to a preconditioned linear system, an operation of the form

$$v \to w = M^{-1}Av$$

or some similar operation is performed at each step. The most natural way to perform this operation is to multiply the vector v by A and then apply M^{-1} to the result. However, since A and M are related, it is sometimes possible to devise procedures that are more economical than this straightforward approach. For example, it is often the case that

$$M = A - R$$

in which the number of nonzero elements in R is much smaller than in A. In this case, the simplest scheme would be to compute $w = M^{-1}Av$ as

$$w = M^{-1}Av = M^{-1}(M + R)v = v + M^{-1}Rv.$$

This requires that R be stored explicitly. In approximate LU factorization techniques, R is the matrix of the elements that are dropped during the incomplete factorization. An even more efficient variation of the preconditioned Conjugate Gradient algorithm can be derived for some common forms of the preconditioner in the special situation where A is symmetric. Write A in the form

$$A = D_0 - E - E^T \tag{9.4}$$

in which $-E$ is the strict lower triangular part of A and D_0 its diagonal. In many cases, the preconditioner M can be written in the form

$$M = (D - E)D^{-1}(D - E^T) \tag{9.5}$$

in which E is the same as above and D is some diagonal, not necessarily equal to D_0. For example, in the SSOR preconditioner with $\omega = 1$, $D \equiv D_0$. Also, for certain types of matrices, the IC(0) preconditioner can be expressed in this manner, where D can be obtained by a recurrence formula.

Eisenstat's implementation consists of applying the Conjugate Gradient algorithm to the linear system

$$\hat{A}u = (D - E)^{-1}b \tag{9.6}$$

with

$$\hat{A} \equiv (D - E)^{-1}A(D - E^T)^{-1}, \quad x = (D - E^T)^{-1}u. \tag{9.7}$$

This does not quite correspond to a preconditioning with the matrix (9.5). In order to produce the same iterates as Algorithm 9.1, the matrix $\hat{A}$ must be further preconditioned with the diagonal matrix D^{-1}. Thus, the preconditioned CG algorithm, Algorithm 9.1, is actually applied to the system (9.6) in which the preconditioning operation is $M^{-1} = D$. Alternatively, we can initially scale the rows and columns of the linear system and preconditioning to transform the diagonal to the identity. See Exercise 6.

Now note that

$$\begin{aligned}\hat{A} &= (D-E)^{-1}A(D-E^T)^{-1}\\ &= (D-E)^{-1}(D_0 - E - E^T)(D-E^T)^{-1}\\ &= (D-E)^{-1}\left(D_0 - 2D + (D-E) + (D-E^T)\right)(D-E^T)^{-1}\\ &\equiv (D-E)^{-1}D_1(D-E^T)^{-1} + (D-E)^{-1} + (D-E^T)^{-1},\end{aligned}$$

in which $D_1 \equiv D_0 - 2D$. As a result,

$$\hat{A}v = (D-E)^{-1}\left[v + D_1(D-E^T)^{-1}v\right] + (D-E^T)^{-1}v.$$

Thus, the vector $w = \hat{A}v$ can be computed by the following procedure:

$$\begin{aligned}&z := (D-E^T)^{-1}v\\ &w := (D-E)^{-1}(v + D_1 z)\\ &w := w + z.\end{aligned}$$

One product with the diagonal D can be saved if the matrices $D^{-1}E$ and $D^{-1}E^T$ are stored. Indeed, by setting $\hat{D}_1 = D^{-1}D_1$ and $\hat{v} = D^{-1}v$, the above procedure can be reformulated as follows.

ALGORITHM 9.3: Computation of $w = \hat{A}v$

```
1. $\hat{v} := D^{-1}v$
2. $z := (I - D^{-1}E^T)^{-1}\hat{v}$
3. $w := (I - D^{-1}E)^{-1}(\hat{v} + \hat{D}_1 z)$
4. $w := w + z$ .
```

Note that the matrices $D^{-1}E$ and $D^{-1}E^T$ are not the transpose of one another, so we actually need to increase the storage requirement for this formulation if these matrices are stored. However, there is a more economical variant which works with the matrix $D^{-1/2}ED^{-1/2}$ and its transpose. This is left as Exercise 7.

Denoting by $N_z(X)$ the number of nonzero elements of a sparse matrix X, the total number of operations (additions and multiplications) of this procedure is n for (1), $2N_z(E)$ for (2), $2N_z(E^T) + 2n$ for (3), and n for (4). The cost of the preconditioning operation by D^{-1}, i.e., n operations, must be added to this, yielding the total number of operations:

$$\begin{aligned}N_{op} &= n + 2N_z(E) + 2N_z(E^T) + 2n + n + n\\ &= 3n + 2(N_z(E) + N_z(E^T) + n)\\ &= 3n + 2N_z(A).\end{aligned}$$

For the straightforward approach, $2N_z(A)$ operations are needed for the product with A,

$2N_z(E)$ for the forward solve, and $n + 2N_z(E^T)$ for the backward solve giving a total of

$$2N_z(A) + 2N_z(E) + n + 2N_z(E^T) = 4N_z(A) - n.$$

Thus, Eisenstat's scheme is always more economical, when N_z is large enough, although the relative gains depend on the total number of nonzero elements in A. One disadvantage of this scheme is that it is limited to a special form of the preconditioner.

Example 9.1 For a 5-point finite difference matrix, $N_z(A)$ is roughly $5n$, so that with the standard implementation $19n$ operations are performed, while with Eisenstat's implementation only $13n$ operations would be performed, a savings of about $\frac{1}{3}$. However, if the other operations of the Conjugate Gradient algorithm are included, for a total of about $10n$ operations, the relative savings become smaller. Now the original scheme will require $29n$ operations, versus $23n$ operations for Eisenstat's implementation.

9.3 PRECONDITIONED GMRES

In the case of GMRES, or other nonsymmetric iterative solvers, the same three options for applying the preconditioning operation as for the Conjugate Gradient (namely, left, split, and right preconditioning) are available. However, there will be one fundamental difference – the right preconditioning versions will give rise to what is called a *flexible variant*, i.e., a variant in which the preconditioner can change at each step. This capability can be very useful in some applications.

9.3.1 LEFT-PRECONDITIONED GMRES

As before, define the left preconditioned GMRES algorithm, as the GMRES algorithm applied to the system,

$$M^{-1}Ax = M^{-1}b. \tag{9.8}$$

The straightforward application of GMRES to the above linear system yields the following preconditioned version of GMRES.

ALGORITHM 9.4: GMRES with Left Preconditioning

```
1. Compute $r_0 = M^{-1}(b - Ax_0)$, $\beta = \|r_0\|_2$ and $v_1 = r_0/\beta$
2. For $j = 1, \ldots, m$ Do:
3.    Compute $w := M^{-1}Av_j$
4.    For $i = 1, \ldots, j$, Do:
5 .       $h_{i,j} := (w, v_i)$
6.        $w := w - h_{i,j}v_i$
```

```
 7.     EndDo
 8.     Compute $h_{j+1,j} = \|w\|_2$ and $v_{j+1} = w/h_{j+1,j}$
 9.   EndDo
10.   Define $V_m := [v_1, \ldots, v_m]$, $\bar{H}_m = \{h_{i,j}\}_{1\le i\le j+1; 1\le j\le m}$
11.   Compute $y_m = \text{argmin}_y \|\beta e_1 - \bar{H}_m y\|_2$, and $x_m = x_0 + V_m y_m$
12.   If satisfied Stop, else set $x_0 := x_m$ and GoTo 1
```

The Arnoldi loop constructs an orthogonal basis of the left preconditioned Krylov subspace

$$\text{Span}\{r_0, M^{-1}Ar_0, \ldots, (M^{-1}A)^{m-1}r_0\}.$$

It uses a modified Gram-Schmidt process, in which the new vector to be orthogonalized is obtained from the previous vector in the process. All residual vectors and their norms that are computed by the algorithm correspond to the preconditioned residuals, namely, $z_m = M^{-1}(b - Ax_m)$, instead of the original (unpreconditioned) residuals $b - Ax_m$. In addition, there is no easy access to these unpreconditioned residuals, unless they are computed explicitly, e.g., by multiplying the preconditioned residuals by M.This can cause some difficulties if a stopping criterion based on the actual residuals, instead of the preconditioned ones, is desired.

Sometimes a Symmetric Positive Definite preconditioning M for the nonsymmetric matrix A may be available. For example, if A is almost SPD, then (9.8) would not take advantage of this. It would be wiser to compute an approximate factorization to the symmetric part and use GMRES with split preconditioning. This raises the question as to whether or not a version of the preconditioned GMRES can be developed, which is similar to Algorithm 9.1, for the CG algorithm. This version would consist of using GMRES with the M-inner product for the system (9.8).

At step j of the preconditioned GMRES algorithm, the previous v_j is multiplied by A to get a vector

$$w_j = Av_j. \tag{9.9}$$

Then this vector is preconditioned to get

$$z_j = M^{-1}w_j. \tag{9.10}$$

This vector must be M-orthogonalized against all previous v_i's. If the standard Gram-Schmidt process is used, we first compute the inner products

$$h_{ij} = (z_j, v_i)_M = (Mz_j, v_i) = (w_j, v_i),\ i = 1, \ldots, j, \tag{9.11}$$

and then modify the vector z_j into the new vector

$$\hat{z}_j := z_j - \sum_{i=1}^{j} h_{ij}v_i. \tag{9.12}$$

To complete the orthonormalization step, the final $\hat{z}_j$ must be normalized. Because of the orthogonality of $\hat{z}_j$ versus all previous v_i's, observe that

$$(\hat{z}_j, \hat{z}_j)_M = (z_j, \hat{z}_j)_M = (M^{-1}w_j, \hat{z}_j)_M = (w_j, \hat{z}_j). \tag{9.13}$$

Thus, the desired M-norm could be obtained from (9.13), and then we would set

$$h_{j+1,j} := (\hat{z}_j, w_j)^{1/2} \quad \text{and} \quad v_{j+1} = \hat{z}_j / h_{j+1,j}. \tag{9.14}$$

One serious difficulty with the above procedure is that the inner product $(\hat{z}_j, \hat{z}_j)_M$ as computed by (9.13) may be negative in the presence of round-off. There are two remedies. First, this M-norm can be computed explicitly at the expense of an additional matrix-vector multiplication with M. Second, the set of vectors Mv_i can be saved in order to accumulate inexpensively both the vector $\hat{z}_j$ and the vector $M\hat{z}_j$, via the relation

$$M\hat{z}_j = w_j - \sum_{i=1}^{j} h_{ij} M v_i.$$

A modified Gram-Schmidt version of this second approach can be derived easily. The details of the algorithm are left as Exercise 12.

9.3.2 RIGHT-PRECONDITIONED GMRES

The right preconditioned GMRES algorithm is based on solving

$$AM^{-1}u = b, \quad u = Mx. \tag{9.15}$$

As we now show, the new variable u never needs to be invoked explicitly. Indeed, once the initial residual $b - Ax_0 = b - AM^{-1}u_0$ is computed, all subsequent vectors of the Krylov subspace can be obtained without any reference to the u-variables. Note that u_0 is not needed at all. The initial residual for the preconditioned system can be computed from $r_0 = b - Ax_0$, which is the same as $b - AM^{-1}u_0$. In practice, it is usually x_0 that is available, not u_0. At the end, the u-variable approximate solution to (9.15) is given by,

$$u_m = u_0 + \sum_{i=1}^{m} v_i \eta_i$$

with $u_0 = Mx_0$. Multiplying through by M^{-1} yields the desired approximation in terms of the x-variable,

$$x_m = x_0 + M^{-1}\left[\sum_{i=1}^{m} v_i \eta_i\right].$$

Thus, one preconditioning operation is needed at the end of the outer loop, instead of at the beginning in the case of the left preconditioned version.

ALGORITHM 9.5: GMRES with Right Preconditioning

```
1. Compute $r_0 = b - Ax_0$, $\beta = \|r_0\|_2$, and $v_1 = r_0/\beta$
2. For $j = 1, \ldots, m$ Do:
3.    Compute $w := AM^{-1}v_j$
4.    For $i = 1, \ldots, j$, Do:
5.        $h_{i,j} := (w, v_i)$
6.        $w := w - h_{i,j} v_i$
7.    EndDo
```

```
 8.      Compute $h_{j+1,j} = \|w\|_2$ and $v_{j+1} = w/h_{j+1,j}$
 9.      Define $V_m := [v_1, \ldots, v_m]$, $\bar{H}_m = \{h_{i,j}\}_{1 \le i \le j+1; 1 \le j \le m}$
10.   EndDo
11.   Compute $y_m = \text{argmin}_y \|\beta e_1 - \bar{H}_m y\|_2$, and $x_m = x_0 + M^{-1} V_m y_m$.
12.   If satisfied Stop, else set $x_0 := x_m$ and GoTo 1.
```

This time, the Arnoldi loop builds an orthogonal basis of the right preconditioned Krylov subspace

$$\text{Span}\{r_0, AM^{-1}r_0, \ldots, (AM^{-1})^{m-1}r_0\}.$$

Note that the residual norm is now relative to the initial system, i.e., $b - Ax_m$, since the algorithm obtains the residual $b - Ax_m = b - AM^{-1}u_m$, implicitly. This is an essential difference with the left preconditioned GMRES algorithm.

9.3.3 SPLIT PRECONDITIONING

In many cases, M is the result of a factorization of the form

$$M = LU.$$

Then, there is the option of using GMRES on the split-preconditioned system

$$L^{-1}AU^{-1}u = L^{-1}b, \quad x = U^{-1}u.$$

In this situation, it is clear that we need to operate on the initial residual by L^{-1} at the start of the algorithm and by U^{-1} on the linear combination $V_m y_m$ in forming the approximate solution. The residual norm available is that of $L^{-1}(b - Ax_m)$.

A question arises on the differences between the right, left, and split preconditioning options. The fact that different versions of the residuals are available in each case may affect the stopping criterion and may cause the algorithm to stop either prematurely or with delay. This can be particularly damaging in case M is very ill-conditioned. The degree of symmetry, and therefore performance, can also be affected by the way in which the preconditioner is applied. For example, a split preconditioner may be much better if A is nearly symmetric. Other than these two situations, there is little difference generally between the three options. The next section establishes a theoretical connection between left and right preconditioned GMRES.

9.3.4 COMPARISON OF RIGHT AND LEFT PRECONDITIONING

When comparing the left, right, and split preconditioning options, a first observation to make is that the spectra of the three associated operators $M^{-1}A$, AM^{-1}, and $L^{-1}AU^{-1}$ are identical. Therefore, in principle one should expect convergence to be similar, although, as is known, eigenvalues do not always govern convergence. In this section, we compare the optimality properties achieved by left- and right preconditioned GMRES.

For the left preconditioning option, GMRES minimizes the residual norm

$$\|M^{-1}b - M^{-1}Ax\|_2,$$

among all vectors from the affine subspace

$$x_0 + \mathcal{K}_m^L = x_0 + \text{Span}\,\{z_0, M^{-1}Az_0, \ldots, (M^{-1}A)^{m-1}z_0\} \tag{9.16}$$

in which z_0 is the preconditioned initial residual $z_0 = M^{-1}r_0$. Thus, the approximate solution can be expressed as

$$x_m = x_0 + M^{-1}s_{m-1}(M^{-1}A)z_0$$

where s_{m-1} is the polynomial of degree $m-1$ which minimizes the norm

$$\|z_0 - M^{-1}A\; s(M^{-1}A)z_0\|_2$$

among all polynomials s of degree $\leq m-1$. It is also possible to express this optimality condition with respect to the original residual vector r_0. Indeed,

$$z_0 - M^{-1}A\; s(M^{-1}A)z_0 = M^{-1}\left[r_0 - A\; s(M^{-1}A)M^{-1}r_0\right].$$

A simple algebraic manipulation shows that for any polynomial s,

$$s(M^{-1}A)M^{-1}r \;=\; M^{-1}s(AM^{-1})r, \tag{9.17}$$

from which we obtain the relation

$$z_0 - M^{-1}As(M^{-1}A)z_0 = M^{-1}\left[r_0 - AM^{-1}s(AM^{-1})r_0\right]. \tag{9.18}$$

Consider now the situation with the right preconditioned GMRES. Here, it is necessary to distinguish between the original x variable and the transformed variable u related to x by $x = M^{-1}u$. For the u variable, the right preconditioned GMRES process minimizes the 2-norm of $r = b - AM^{-1}u$ where u belongs to

$$u_0 + \mathcal{K}_m^R = u_0 + \text{Span}\,\{r_0, AM^{-1}r_0, \ldots, (AM^{-1})^{m-1}r_0\} \tag{9.19}$$

in which r_0 is the residual $r_0 = b - AM^{-1}u_0$. This residual is identical to the residual associated with the original x variable since $M^{-1}u_0 = x_0$. Multiplying (9.19) through to the left by M^{-1} and exploiting again (9.17), observe that the generic variable x associated with a vector of the subspace (9.19) belongs to the affine subspace

$$M^{-1}u_0 + M^{-1}\mathcal{K}_m^R = x_0 + \text{Span}\,\{z_0, M^{-1}Az_0 \ldots, (M^{-1}A)^{m-1}z_0\}.$$

This is identical to the affine subspace (9.16) invoked in the left preconditioned variant. In other words, for the right preconditioned GMRES, the approximate x-solution can also be expressed as

$$x_m = x_0 + s_{m-1}(AM^{-1})r_0.$$

However, now s_{m-1} is a polynomial of degree $m-1$ which minimizes the norm

$$\|r_0 - AM^{-1}\; s(AM^{-1})r_0\|_2 \tag{9.20}$$

among all polynomials s of degree $\leq m-1$. What is surprising is that the two quantities which are minimized, namely, (9.18) and (9.20), differ only by a multiplication by M^{-1}. Specifically, the left preconditioned GMRES minimizes $M^{-1}r$, whereas the right preconditioned variant minimizes r, where r is taken over the same subspace in both cases.

PROPOSITION 9.1 *The approximate solution obtained by left or right preconditioned GMRES is of the form*

$$x_m = x_0 + s_{m-1}(M^{-1}A)z_0 = x_0 + M^{-1}s_{m-1}(AM^{-1})r_0$$

where $z_0 = M^{-1}r_0$ and s_{m-1} is a polynomial of degree $m-1$. The polynomial s_{m-1} minimizes the residual norm $\|b - Ax_m\|_2$ in the right preconditioning case, and the preconditioned residual norm $\|M^{-1}(b - Ax_m)\|_2$ in the left preconditioning case.

In most practical situations, the difference in the convergence behavior of the two approaches is not significant. The only exception is when M is ill-conditioned which could lead to substantial differences.

9.4 FLEXIBLE VARIANTS

In the discussion of preconditioning techniques so far, it is implicitly assumed that the preconditioning matrix M is fixed, i.e., it does not change from step to step. However, in some cases, no matrix M is available. Instead, the operation $M^{-1}x$ is the result of some unspecified computation, possibly another iterative process. In such cases, it may well happen that M^{-1} is not a constant operator. The previous preconditioned iterative procedures will not converge if M is not constant. There are a number of variants of iterative procedures developed in the literature that can accommodate variations in the preconditioner, i.e., that allow the preconditioner to vary from step to step. Such iterative procedures are called “flexible” iterations. One of these iterations, a flexible variant of the GMRES algorithm, is described next.

9.4.1 FLEXIBLE GMRES

We begin by examining the right preconditioned GMRES algorithm. In line 11 of Algorithm 9.5 the approximate solution x_m is expressed as a linear combination of the preconditioned vectors $z_i = M^{-1}v_i, i = 1, \ldots, m$. These vectors are also computed in line 3, prior to their multiplication by A to obtain the vector w. They are all obtained by applying the same preconditioning matrix M^{-1} to the v_i's. As a result it is not necessary to save them. Instead, we only need to apply M^{-1} to the linear combination of the v_i's, i.e., to $V_m y_m$ in line 11. Suppose now that the preconditioner could change at every step, i.e., that z_j is given by

$$z_j = M_j^{-1}v_j.$$

Then it would be natural to compute the approximate solution as

$$x_m = x_0 + Z_m y_m$$

in which $Z_m = [z_1, \ldots, z_m]$, and y_m is computed as before, as the solution to the least-squares problem in line 11. These are the only changes that lead from the right preconditioned algorithm to the flexible variant, described below.

ALGORITHM 9.6: Flexible GMRES (FGMRES)

```
 1. Compute $r_0 = b - Ax_0$, $\beta = \|r_0\|_2$, and $v_1 = r_0/\beta$
 2. For $j = 1, \ldots, m$ Do:
 3.    Compute $z_j := M_j^{-1} v_j$
 4.    Compute $w := Az_j$
 5.    For $i = 1, \ldots, j$, Do:
 6.       $h_{i,j} := (w, v_i)$
 7.       $w := w - h_{i,j} v_i$
 8.    EndDo
 9.    Compute $h_{j+1,j} = \|w\|_2$ and $v_{j+1} = w/h_{j+1,j}$
10.    Define $Z_m := [z_1, \ldots, z_m]$, $\bar{H}_m = \{h_{i,j}\}_{1 \le i \le j+1; 1 \le j \le m}$
11. EndDo
12. Compute $y_m = \mathrm{argmin}_y \|\beta e_1 - \bar{H}_m y\|_2$, and $x_m = x_0 + Z_m y_m$.
13. If satisfied Stop, else set $x_0 \leftarrow x_m$ and GoTo 1.
```

As can be seen, the main difference with the right preconditioned version, Algorithm 9.5, is that the preconditioned vectors $z_j = M_j^{-1} v_j$ must be saved and the solution updated using these vectors. It is clear that when $M_j = M$ for $j = 1, \ldots, m$, then this method is equivalent mathematically to Algorithm 9.5. It is important to observe that z_j can be defined in line 3 without reference to any preconditioner. That is, any given new vector z_j can be chosen. This added flexibility may cause the algorithm some problems. Indeed, z_j may be so poorly selected that a breakdown could occur, as in the worst-case scenario when z_j is zero.

One difference between FGMRES and the usual GMRES algorithm is that the action of AM_j^{-1} on a vector v of the Krylov subspace is no longer in the span of V_{m+1}. Instead, it is easy to show that

$$AZ_m = V_{m+1} \bar{H}_m \tag{9.21}$$

in replacement of the simpler relation $(AM^{-1})V_m = V_{m+1} \bar{H}_m$ which holds for the standard preconditioned GMRES; see (6.5). As before, H_m denotes the $m \times m$ matrix obtained from $\bar{H}_m$ by deleting its last row and $\hat{v}_{j+1}$ is the vector w which is normalized in line 9 of Algorithm 9.6 to obtain v_{j+1}. Then, the following alternative formulation of (9.21) is valid, even when $h_{m+1,m} = 0$:

$$AZ_m = V_m H_m + \hat{v}_{m+1} e_m^T. \tag{9.22}$$

An optimality property similar to the one which defines GMRES can be proved. Consider the residual vector for an arbitrary vector $z = x_0 + Z_m y$ in the affine space $x_0 + span\{Z_m\}$. This optimality property is based on the relations

$$\begin{aligned} b - Az &= b - A(x_0 + Z_m y) \\ &= r_0 - AZ_m y \end{aligned} \tag{9.23}$$

$$\begin{aligned}&= \beta v_1 - V_{m+1}\bar{H}_m y \\ &= V_{m+1}[\beta e_1 - \bar{H}_m y]. \end{aligned} \tag{9.24}$$

If $J_m(y)$ denotes the function

$$J_m(y) = \|b - A[x_0 + Z_m y]\|_2,$$

observe that by (9.24) and the fact that V_{m+1} is unitary,

$$J_m(y) = \|\beta e_1 - \bar{H}_m y\|_2. \tag{9.25}$$

Since the algorithm minimizes this norm over all vectors u in $\mathbb{R}^m$ to yield y_m, it is clear that the approximate solution $x_m = x_0 + Z_m y_m$ has the smallest residual norm in $x_0 + \text{Span}\{Z_m\}$. Thus, the following result is proved.

PROPOSITION 9.2 *The approximate solution x_m obtained at step m of FGMRES minimizes the residual norm $\|b - Ax_m\|_2$ over $x_0 + \text{Span}\{Z_m\}$.*

Next, consider the possibility of breakdown in FGMRES. A breakdown occurs when the vector v_{j+1} cannot be computed in line 9 of Algorithm 9.6 because $h_{j+1,j} = 0$. For the standard GMRES algorithm, this is not a problem because when this happens then the approximate solution x_j is exact. The situation for FGMRES is slightly different.

PROPOSITION 9.3 *Assume that $\beta = \|r_0\|_2 \neq 0$ and that $j - 1$ steps of FGMRES have been successfully performed, i.e., that $h_{i+1,i} \neq 0$ for $i < j$. In addition, assume that the matrix H_j is nonsingular. Then x_j is exact, if and only if $h_{j+1,j} = 0$.*

Proof. If $h_{j+1,j} = 0$, then $AZ_j = V_j H_j$, and as a result

$$J_j(y) = \|\beta v_1 - AZ_j y_j\|_2 = \|\beta v_1 - V_j H_j y_j\|_2 = \|\beta e_1 - H_j y_j\|_2.$$

If H_j is nonsingular, then the above function is minimized for $y_j = H_j^{-1}(\beta e_1)$ and the corresponding minimum norm reached is zero, i.e., x_j is exact.

Conversely, if x_j is exact, then from (9.22) and (9.23),

$$0 = b - Ax_j = V_j[\beta e_1 - H_j y_j] + \hat{v}_{j+1} e_j^T y_j. \tag{9.26}$$

We must show, by contraction, that $\hat{v}_{j+1} = 0$. Assume that $\hat{v}_{j+1} \neq 0$. Since $\hat{v}_{j+1}$, v_1, $v_2, \ldots, v_m$, form an orthogonal system, then it follows from (9.26) that $\beta e_1 - H_j y_j = 0$ and $e_j^T y_j = 0$. The last component of y_j is equal to zero. A simple back-substitution for the system $H_j y_j = \beta e_1$, starting from the last equation, will show that all components of y_j are zero. Because H_m is nonsingular, this would imply that $\beta = 0$ and contradict the assumption. ■

The only difference between this result and that of Proposition 6.10 for the GMRES algorithm is that the additional assumption must be made that H_j is nonsingular since it is no longer implied by the nonsingularity of A. However, H_m is guaranteed to be nonsingular when all the z_j's are linearly independent and A is nonsingular. This is a consequence of a modification of the first part of Proposition 6.9. That same proof shows that the rank of AZ_m is equal to the rank of the matrix R_m therein. If R_m is nonsingular and $h_{m+1,m} = 0$, then H_m is also nonsingular.

A consequence of the above proposition is that if $Az_j = v_j$, at a certain step, i.e., if the preconditioning is "exact," then the approximation x_j will be exact provided that H_j is nonsingular. This is because $w = Az_j$ would depend linearly on the previous v_i's (it is equal to v_j), and as a result the orthogonalization process would yield $\hat{v}_{j+1} = 0$.

A difficulty with the theory of the new algorithm is that general convergence results, such as those seen in earlier chapters, cannot be proved. That is because the subspace of approximants is no longer a standard Krylov subspace. However, the optimality property of Proposition 9.2 can be exploited in some specific situations. For example, if within each outer iteration *at least one* of the vectors z_j is chosen to be a steepest descent direction vector, e.g., for the function $F(x) = \|b - Ax\|_2^2$, then FGMRES is guaranteed to converge independently of m.

The additional cost of the flexible variant over the standard algorithm is only in the extra memory required to save the set of vectors $\{z_j\}_{j=1,\ldots,m}$. Yet, the added advantage of *flexibility* may be worth this extra cost. A few applications can benefit from this flexibility, especially in developing robust iterative methods or preconditioners on parallel computers. Thus, *any* iterative technique can be used as a preconditioner: block-SOR, SSOR, ADI, Multi-grid, etc. More interestingly, iterative procedures such as GMRES, CGNR, or CGS can also be used as preconditioners. Also, it may be useful to mix two or more preconditioners to solve a given problem. For example, two types of preconditioners can be applied alternatively at each FGMRES step to mix the effects of "local" and "global" couplings in the PDE context.

9.4.2 DQGMRES

Recall that the DQGMRES algorithm presented in Chapter 6 uses an incomplete orthogonalization process instead of the full Arnoldi orthogonalization. At each step, the current vector is orthogonalized only against the k previous ones. The vectors thus generated are "locally" orthogonal to each other, in that $(v_i, v_j) = \delta_{ij}$ for $|i - j| < k$. The matrix $\bar{H}_m$ becomes banded and upper Hessenberg. Therefore, the approximate solution can be updated at step j from the approximate solution at step $j-1$ via the recurrence

$$p_j = \frac{1}{r_{jj}}, \left[v_j - \sum_{i=j-k+1}^{j-1} r_{ij} p_i \right], \quad x_j = x_{j-1} + \gamma_j p_j \tag{9.27}$$

in which the scalars γ_j and r_{ij} are obtained recursively from the Hessenberg matrix $\bar{H}_j$.

An advantage of DQGMRES is that it is also *flexible*. The principle is the same as in FGMRES. In both cases the vectors $z_j = M_j^{-1} v_j$ must be computed. In the case of FGMRES, these vectors must be saved and this requires extra storage. For DQGMRES, it can be observed that the preconditioned vectors z_j only affect the update of the vector p_j in the preconditioned version of the update formula (9.27), yielding

$$p_j = \frac{1}{r_{jj}} \left[M_j^{-1} v_j - \sum_{i=j-k+1}^{j-1} r_{ij} p_i \right].$$

As a result, $M_j^{-1} v_j$ can be discarded immediately after it is used to update p_j. The same memory locations can store this vector and the vector p_j. This contrasts with FGMRES which requires additional vectors of storage.

9.5 PRECONDITIONED CG FOR THE NORMAL EQUATIONS

There are several versions of the preconditioned Conjugate Gradient method applied to the normal equations. Two versions come from the NR/NE options, and three other variations from the right, left, or split preconditioning options. Here, we consider only the left preconditioned variants.

The left preconditioned CGNR algorithm is easily derived from Algorithm 9.1. Denote by r_j the residual for the original system, i.e., $r_j = b - Ax_j$, and by $\tilde{r}_j = A^T r_j$ the residual for the normal equations system. The preconditioned residual z_j is $z_j = M^{-1}\tilde{r}_j$. The scalar α_j in Algorithm 9.1 is now given by

$$\alpha_j = \frac{(\tilde{r}_j, z_j)}{(A^T A p_j, p_j)} = \frac{(\tilde{r}_j, z_j)}{(Ap_j, Ap_j)}.$$

This suggests employing the auxiliary vector $w_j = Ap_j$ in the algorithm which takes the following form.

ALGORITHM 9.7: Left-Preconditioned CGNR

```
1.  Compute $r_0 = b - Ax_0$, $\tilde{r}_0 = A^T r_0$, $z_0 = M^{-1}\tilde{r}_0$, $p_0 = z_0$.
2.  For $j = 0, \ldots,$ until convergence Do:
3.      $w_j = Ap_j$
4.      $\alpha_j = (z_j, \tilde{r}_j)/\|w_j\|_2^2$
5.      $x_{j+1} = x_j + \alpha_j p_j$
6.      $r_{j+1} = r_j - \alpha_j w_j$
7.      $\tilde{r}_{j+1} = A^T r_{j+1}$
8.      $z_{j+1} = M^{-1}\tilde{r}_{j+1}$
9.      $\beta_j = (z_{j+1}, \tilde{r}_{j+1})/(z_j, \tilde{r}_j)$
10.     $p_{j+1} = z_{j+1} + \beta_j p_j$
11. EndDo
```

Similarly, the linear system $AA^T u = b$, with $x = A^T u$, can also be preconditioned from the left, and solved with the preconditioned Conjugate Gradient algorithm. Here, it is observed that the update of the u variable, the associated x variable, and two residuals take the form

$$\alpha_j = \frac{(r_j, z_j)}{(AA^T p_j, p_j)} = \frac{(r_j, z_j)}{(A^T p_j, A^T p_j)}$$
$$u_{j+1} = u_j + \alpha_j p_j \quad \leftrightarrow \quad x_{j+1} = x_j + \alpha_j A^T p_j$$
$$r_{j+1} = r_j - \alpha_j AA^T p_j$$
$$z_{j+1} = M^{-1} r_{j+1}$$

Thus, if the algorithm for the unknown x is to be written, then the vectors $A^T p_j$ can be used instead of the vectors p_j, which are not needed. To update these vectors at the end of the algorithm the relation $p_{j+1} = z_{j+1} + \beta_{j+1} p_j$ in line 8 of Algorithm 9.1 must be multiplied through by A^T. This leads to the left preconditioned version of CGNE, in which the notation has been changed to denote by p_j the vector $A^T p_j$ invoked in the above derivation.

ALGORITHM 9.8: Left-Preconditioned CGNE

```
1.  Compute $r_0 = b - Ax_0$, $z_0 = M^{-1} r_0$, $p_0 = A^T z_0$.
2.  For $j = 0, 1, \ldots$, until convergence Do:
3.      $w_j = Ap_j$
4.      $\alpha_j = (z_j, r_j)/(p_j, p_j)$
5.      $x_{j+1} = x_j + \alpha_j p_j$
6.      $r_{j+1} = r_j - \alpha_j w_j$
7.      $z_{j+1} = M^{-1} r_{j+1}$
8.      $\beta_j = (z_{j+1}, r_{j+1})/(z_j, r_j)$
9.      $p_{j+1} = A^T z_{j+1} + \beta_j p_j$
10. EndDo
```

Not shown here are the right and split preconditioned versions which are considered in Exercise 3.

9.6 THE CGW ALGORITHM

When the matrix is nearly symmetric, we can think of preconditioning the system with the symmetric part of A. This gives rise to a few variants of a method known as the CGW method, from the names of the three authors Concus and Golub [60], and Widlund [225] who proposed this technique in the middle of the 1970s. Originally, the algorithm was not viewed from the angle of preconditioning. Writing $A = M - N$, with $M = \frac{1}{2}(A + A^H)$, the authors observed that the preconditioned matrix

$$M^{-1}A = I - M^{-1}N$$

is equal to the identity matrix, plus a matrix which is skew-Hermitian with respect to the M-inner product. It is not too difficult to show that the tridiagonal matrix corresponding to the Lanczos algorithm, applied to A with the M-inner product, has the form

$$T_m = \begin{pmatrix} 1 & -\beta_2 & & & \\ \beta_2 & 1 & -\beta_3 & & \\ & . & . & . & \\ & & \beta_{m-1} & 1 & -\beta_m \\ & & & \beta_m & 1 \end{pmatrix}. \tag{9.28}$$

As a result, a three-term recurrence in the Arnoldi process is obtained, which results in a solution algorithm that resembles the standard preconditioned CG algorithm (Algorithm 9.1).

A version of the algorithm can be derived easily. From the developments in Section 6.7 relating the Lanczos algorithm to the Conjugate Gradient algorithm, it is known that x_{j+1} can be expressed as

$$x_{j+1} = x_j + \alpha_j p_j .$$

The preconditioned residual vectors must then satisfy the recurrence

$$z_{j+1} = z_j - \alpha_j M^{-1} A p_j$$

and if the z_j's are to be M-orthogonal, then we must have $(z_j - \alpha_j M^{-1} A p_j, z_j)_M = 0$. As a result,

$$\alpha_j = \frac{(z_j, z_j)_M}{(M^{-1} A p_j, z_j)_M} = \frac{(r_j, z_j)}{(A p_j, z_j)} .$$

Also, the next search direction p_{j+1} is a linear combination of z_{j+1} and p_j,

$$p_{j+1} = z_{j+1} + \beta_j p_j .$$

Thus, a first consequence is that

$$(A p_j, z_j)_M = (M^{-1} A p_j, p_j - \beta_{j-1} p_{j-1})_M = (M^{-1} A p_j, p_j)_M = (A p_j, p_j)$$

because $M^{-1} A p_j$ is orthogonal to all vectors in $\mathcal{K}_{j-1}$. In addition, writing that p_{j+1} is M-orthogonal to $M^{-1} A p_j$ yields

$$\beta_j = - \frac{(z_{j+1}, M^{-1} A p_j)_M}{(p_j, M^{-1} A p_j)_M} .$$

Note that $M^{-1} A p_j = -\frac{1}{\alpha_j}(z_{j+1} - z_j)$ and therefore we have, just as in the standard PCG algorithm,

$$\beta_j = \frac{(z_{j+1}, z_{j+1})_M}{(z_j, z_j)_M} = \frac{(z_{j+1}, r_{j+1})}{(z_j, r_j)} .$$

EXERCISES

1. Let a matrix A and its preconditioner M be SPD. Observing that $M^{-1}A$ is self-adjoint with respect to the A inner-product, write an algorithm similar to Algorithm 9.1 for solving the preconditioned linear system $M^{-1}Ax = M^{-1}b$, using the A-inner product. The algorithm should employ only one matrix-by-vector product per CG step.

2. In Section 9.2.1, the split-preconditioned Conjugate Gradient algorithm, Algorithm 9.2, was derived from the Preconditioned Conjugate Gradient Algorithm 9.1. The opposite can also be done. Derive Algorithm 9.1 starting from Algorithm 9.2, providing a different proof of the equivalence of the two algorithms.

3. Six versions of the CG algorithm applied to the normal equations can be defined. Two versions come from the NR/NE options, each of which can be preconditioned from left, right, or on two sides. The left preconditioned variants have been given in Section 9.5. Describe the four other versions: Right P-CGNR, Right P-CGNE, Split P-CGNR, Split P-CGNE. Suitable inner products may be used to preserve symmetry.

4. When preconditioning the normal equations, whether the NE or NR form, two options are available in addition to the left, right and split preconditioners. These are "centered" versions:

$$AM^{-1}A^Tu = b, \quad x = M^{-1}A^Tu$$

for the NE form, and

$$A^TM^{-1}Ax = A^TM^{-1}b$$

for the NR form. The coefficient matrices in the above systems are all symmetric. Write down the adapted versions of the CG algorithm for these options.

5. Let a matrix A and its preconditioner M be SPD. The standard result about the rate of convergence of the CG algorithm is not valid for the Preconditioned Conjugate Gradient algorithm, Algorithm 9.1. Show how to adapt this result by exploiting the M-inner product. Show how to derive the same result by using the equivalence between Algorithm 9.1 and Algorithm 9.2.

6. In Eisenstat's implementation of the PCG algorithm, the operation with the diagonal D causes some difficulties when describing the algorithm. This can be avoided.

a. Assume that the diagonal D of the preconditioning (9.5) is equal to the identity matrix. What are the number of operations needed to perform one step of the PCG algorithm with Eisenstat's implementation? Formulate the PCG scheme for this case carefully.

b. The rows and columns of the preconditioning matrix M can be scaled so that the matrix D of the transformed preconditioner, written in the form (9.5), is equal to the identity matrix. What scaling should be used (the resulting M should also be SPD)?

c. Assume that the same scaling of question b is also applied to the original matrix A. Is the resulting iteration mathematically equivalent to using Algorithm 9.1 to solve the system (9.6) preconditioned with the diagonal D?

7. In order to save operations, the two matrices $D^{-1}E$ and $D^{-1}E^T$ must be stored when computing $\hat{A}v$ by Algorithm 9.3. This exercise considers alternatives.

a. Consider the matrix $B \equiv D\hat{A}D$. Show how to implement an algorithm similar to 9.3 for multiplying a vector v by B. The requirement is that only ED^{-1} must be stored.

b. The matrix B in the previous question is not the proper preconditioned version of A by the preconditioning (9.5). CG is used on an equivalent system involving B but a further preconditioning by a diagonal must be applied. Which one? How does the resulting algorithm compare in terms of cost and storage with an Algorithm based on 9.3?

c. It was mentioned in Section 9.2.2 that $\hat{A}$ needed to be further preconditioned by D^{-1}. Consider the split-preconditioning option: CG is to be applied to the preconditioned system associated with $C = D^{1/2}\hat{A}D^{1/2}$. Defining $\hat{E} = D^{-1/2}ED^{-1/2}$ show that,

$$C = (I - \hat{E})^{-1}D_2(I - \hat{E})^{-T} + (I - \hat{E})^{-1} + (I - \hat{E})^{-T}$$

where D_2 is a certain matrix to be determined. Then write an analogue of Algorithm 9.3 using this formulation. How does the operation count compare with that of Algorithm 9.3?

8. Assume that the number of nonzero elements of a matrix A is parameterized by $Nz(Z) = \alpha n$. How small should α be before it does not pay to use Eisenstat's implementation for the PCG algorithm? What if the matrix A is initially scaled so that D is the identity matrix?

9. Let $M = LU$ be a preconditioner for a matrix A. Show that the left, right, and split preconditioned matrices all have the same eigenvalues. Does this mean that the corresponding preconditioned iterations will converge in (a) exactly the same number of steps? (b) roughly the same number of steps for any matrix? (c) roughly the same number of steps, except for ill-conditioned matrices?

10. Show that the relation (9.17) holds for any polynomial s and any vector r.

11. Write the equivalent of Algorithm 9.1 for the Conjugate Residual method.

12. Assume that a Symmetric Positive Definite matrix M is used to precondition GMRES for solving a nonsymmetric linear system. The main features of the P-GMRES algorithm exploiting this were given in Section 9.2.1. Give a formal description of the algorithm. In particular give a Modified Gram-Schimdt implementation. [Hint: The vectors Mv_i's must be saved in addition to the v_i's.] What optimality property does the approximate solution satisfy? What happens if the original matrix A is also symmetric? What is a potential advantage of the resulting algorithm?

NOTES AND REFERENCES. The preconditioned version of CG described in Algorithm 9.1 is due to Meijerink and van der Vorst [149]. Eisenstat's implementation was developed in [80] and is often referred to as *Eisenstat's trick*. A number of other similar ideas are described in [153]. Several flexible variants of nonsymmetric Krylov subspace methods have been developed by several authors simultaneously; see, e.g., [18], [181], and [211]. There does not seem to exist a similar technique for left preconditioned variants of the Krylov subspace methods. This is because the preconditioned operator $M_j^{-1}A$ now changes at each step. Similarly, no flexible variants have been developed for the BCG-based methods, because the short recurrences of these algorithms rely on the preconditioned operator being constant.

The CGW algorithm can be useful in some instances, such as when the symmetric part of A can be inverted easily, e.g., using fast Poisson solvers. Otherwise, its weakness is that linear systems with the symmetric part must be solved exactly. Inner-outer variations that do not require exact solutions have been described by Golub and Overton [109]. ■

CHAPTER

10 PRECONDITIONING TECHNIQUES

Finding a good preconditioner to solve a given sparse linear system is often viewed as a combination of art and science. Theoretical results are rare and some methods work surprisingly well, often despite expectations. A preconditioner can be defined as any subsidiary approximate solver which is combined with an outer iteration technique, typically one of the Krylov subspace iterations seen in previous chapters. This chapter covers some of the most successful techniques used to precondition a general sparse linear system. Note at the outset that there are virtually no limits to available options for obtaining good preconditioners. For example, preconditioners can be derived from knowledge of the original physical problems from which the linear system arises. However, a common feature of the preconditioners discussed in this chapter is that they are built from the original coefficient matrix.

10.1 INTRODUCTION

Roughly speaking, a preconditioner is any form of implicit or explicit modification of an original linear system which makes it "easier" to solve by a given iterative method. For example, scaling all rows of a linear system to make the diagonal elements equal to one is an explicit form of preconditioning. The resulting system can be solved by a Krylov subspace method and may require fewer steps to converge than with the original system (although this is not guaranteed). As another example, solving the linear system

$$M^{-1}Ax = M^{-1}b$$

where M^{-1} is some complicated mapping that may involve FFT transforms, integral calculations, and subsidiary linear system solutions, may be another form of preconditioning. Here, it is unlikely that the matrix M and $M^{-1}A$ can be computed explicitly. Instead, the iterative processes operate with A and with M^{-1} whenever needed. In practice, the preconditioning operation M^{-1} should be inexpensive to apply to an arbitrary vector.

One of the simplest ways of defining a preconditioner is to perform an *incomplete factorization* of the original matrix A. This entails a decomposition of the form $A = LU - R$ where L and U have the same nonzero structure as the lower and upper parts of A respectively, and R is the *residual* or *error* of the factorization. This incomplete factorization known as ILU(0) is rather easy and inexpensive to compute. On the other hand, it often leads to a crude approximation which may result in the Krylov subspace accelerator requiring many iterations to converge. To remedy this, several alternative incomplete factorizations have been developed by allowing more fill-in in L and U. In general, the more accurate ILU factorizations require fewer iterations to converge, but the preprocessing cost to compute the factors is higher. However, if only because of the improved robustness, these trade-offs generally favor the more accurate factorizations. This is especially true when several systems with the same matrix must be solved because the preprocessing cost can be amortized.

This chapter considers the most common preconditioners used for solving large sparse matrices and compares their performance. It begins with the simplest preconditioners (SOR and SSOR) and then discusses the more accurate variants such as ILUT.

10.2 JACOBI, SOR, AND SSOR PRECONDITIONERS

As was seen in Chapter 4, a fixed-point iteration for solving a linear system

$$Ax = b$$

takes the general form

$$x_{k+1} = M^{-1}Nx_k + M^{-1}b \tag{10.1}$$

where M and N realize the splitting of A into

$$A = M - N. \tag{10.2}$$

The above iteration is of the form

$$x_{k+1} = Gx_k + f \tag{10.3}$$

where $f = M^{-1}b$ and

$$\begin{aligned} G &= M^{-1}N = M^{-1}(M - A) \\ &= I - M^{-1}A. \end{aligned} \tag{10.4}$$

Thus, for Jacobi and Gauss Seidel it has been shown that

$$G_{JA}(A) = I - D^{-1}A \tag{10.5}$$

$$G_{GS}(A) = I - (D - E)^{-1}A, \tag{10.6}$$

where $A = D - E - F$ is the splitting defined in Chapter 4.

The iteration (10.3) is attempting to solve

$$(I - G)x = f \tag{10.7}$$

which, because of the expression (10.4) for G, can be rewritten as

$$M^{-1}Ax = M^{-1}b. \tag{10.8}$$

The above system is the *preconditioned system* associated with the splitting $A = M - N$, and the iteration (10.3) is nothing but a *fixed-point iteration on this preconditioned system.* Similarly, a Krylov subspace method, e.g., GMRES, can be used to solve (10.8), leading to a preconditioned version of the Krylov subspace method, e.g., preconditioned GMRES. The preconditioned versions of some Krylov subspace methods have been discussed in the previous chapter with a generic preconditioner M. In theory, any general splitting in which M is nonsingular can be used. Ideally, M should be close to A in some sense. However, note that a linear system with the matrix M must be solved at each step of the iterative procedure. Therefore, a practical and admittedly somewhat vague requirement is that these solutions steps should be inexpensive.

As was seen in Chapter 4, the SSOR preconditioner is defined by

$$M_{SSOR} = (D - \omega E)D^{-1}(D - \omega F).$$

Typically, when this matrix is used as a preconditioner, it is not necessary to choose ω as carefully as for the underlying fixed-point iteration. Taking $\omega = 1$ leads to the Symmetric Gauss-Seidel (SGS) iteration,

$$M_{SGS} = (D - E)D^{-1}(D - F). \tag{10.9}$$

An interesting observation is that $D - E$ is the lower part of A, including the diagonal, and $D - F$ is, similarly, the upper part of A. Thus,

$$M_{SGS} = LU,$$

with

$$L \equiv (D - E)D^{-1} = I - ED^{-1}, \quad U = D - F.$$

The matrix L is unit lower triangular and U is upper triangular. One question that may arise concerns the implementation of the preconditioning operation. To compute $w = M_{SGS}^{-1}x$, proceed as follows:

$$\begin{aligned} &\text{solve} \quad (I - ED^{-1})z = x, \\ &\text{solve} \quad (D - F)w = z. \end{aligned}$$

A FORTRAN implementation of this preconditioning operation is illustrated in the following code, for matrices stored in the MSR format described in Chapter 3.

FORTRAN CODE

```
      subroutine lusol (n,rhs,sol,luval,lucol,luptr,uptr)
      real*8 sol(n), rhs(n), luval(*)
      integer n, luptr(*), uptr(n)
c-----------------------------------------------------------
c Performs a  forward and  a  backward solve  for an ILU or
c SSOR factorization, i.e., solves (LU) sol  = rhs where LU
c is the ILU  or the SSOR  factorization. For SSOR, L and U
```

```
c should contain the matrices L = I - omega E inv(D), and U
c = D  -  omega  F,  respectively  with  -E =  strict lower
c triangular part of  A,  -F = strict upper triangular part
c of  A, and D = diagonal of A.
c-----------------------------------------------------------
c PARAMETERS:
c n     = Dimension of problem
c rhs   = Right hand side; rhs is unchanged on return
c sol   = Solution of (LU) sol = rhs.
c luval = Values of the LU matrix. L and U are stored
c         together in  CSR format. The diagonal elements of
c         U are inverted. In each row, the L values are
c         followed by the diagonal element (inverted) and
c         then the other U values.
c lucol = Column indices of corresponding elements in luval
c luptr = Contains pointers to the beginning of each row in
c         the LU matrix.
c uptr  = pointer to the diagonal elements in luval, lucol
c------------------------------------------------------------
       integer i,k
c
c     FORWARD SOLVE.  Solve   L . sol = rhs
c
       do i = 1, n
c
c     compute  sol(i) := rhs(i) - sum L(i,j) x sol(j)
c
           sol(i) = rhs(i)
           do k=luptr(i),uptr(i)-1
              sol(i) = sol(i) - luval(k)* sol(lucol(k))
           enddo
      enddo
c
c     BACKWARD SOLVE. Compute  sol := inv(U) sol
c
       do i = n, 1, -1
c
c     compute  sol(i) := sol(i) - sum U(i,j) x sol(j)
c
           do k=uptr(i)+1, luptr(i+1)-1
              sol(i) = sol(i) - luval(k)*sol(lucol(k))
           enddo
c
c     compute  sol(i) := sol(i)/ U(i,i)
c
           sol(i) = luval(uptr(i))*sol(i)
      enddo
      return
      end
```

As was seen above, the SSOR or SGS preconditioning matrix is of the form $M = LU$ where L and U have the same pattern as the L-part and the U-part of A, respectively. Here, L-part means lower triangular part and, similarly, the U-part is the upper triangular part. If the error matrix $A - LU$ is computed, then for SGS, for example, we would find

$$A - LU = D - E - F - (I - ED^{-1})(D - F) = -ED^{-1}F.$$

If L is restricted to have the same structure as the L-part of A and U is to have the same structure as the U-part of A, the question is whether or not it is possible to find L and U that yield an error that is smaller in some sense than the one above. We can, for example, try to find such an incomplete factorization in which the residual matrix $A - LU$ has zero elements in locations where A has nonzero entries. This turns out to be possible in general and yields the ILU(0) factorization to be discussed later. Generally, a pattern for L and U can be specified and L and U may be sought so that they satisfy certain conditions. This leads to the general class of incomplete factorization techniques which are discussed in the next section.

Example 10.1 Table 10.1 shows the results of applying the GMRES algorithm with SGS (SSOR with $\omega = 1$) preconditioning to the five test problems described in Section 3.7.

Matrix	Iters	Kflops	Residual	Error
F2DA	38	1986	0.76E-03	0.82E-04
F3D	20	4870	0.14E-02	0.30E-03
ORS	110	6755	0.31E+00	0.68E-04
F2DB	300	15907	0.23E+02	0.66E+00
FID	300	99070	0.26E+02	0.51E-01

Table 10.1 *A test run of GMRES with SGS preconditioning.*

See Example 6.1 for the meaning of the column headers in the table. Notice here that the method did not converge in 300 steps for the last two problems. The number of iterations for the first three problems is reduced substantially from those required by GMRES without preconditioning shown in Table 6.2. The total number of operations required is also reduced, but not proportionally because each step now costs more due to the preconditioning operation.

10.3 ILU FACTORIZATION PRECONDITIONERS

Consider a general sparse matrix A whose elements are $a_{ij}, i, j = 1, \ldots, n$. A general Incomplete LU (ILU) factorization process computes a sparse lower triangular matrix L and a sparse upper triangular matrix U so the residual matrix $R = LU - A$ satisfies certain constraints, such as having zero entries in some locations. We first describe a general ILU preconditioner geared toward M-matrices. Then we discuss the ILU(0) factorization, the simplest form of the ILU preconditioners. Finally, we will show how to obtain more accurate factorizations.

10.3.1 INCOMPLETE LU FACTORIZATIONS

A general algorithm for building Incomplete LU factorizations can be derived by performing Gaussian elimination and dropping some elements in predetermined nondiagonal positions. To analyze this process and establish existence for M-matrices, the following result of Ky-Fan [86] is needed.

THEOREM 10.1 *Let A be an M-matrix and let A_1 be the matrix obtained from the first step of Gaussian elimination. Then A_1 is an M-matrix.*

Proof. Theorem 1.17 will be used to establish that properties 1, 2, and 3 therein are satisfied. First, consider the off-diagonal elements of A_1:

$$a_{ij}^1 = a_{ij} - \frac{a_{i1}a_{1j}}{a_{11}}.$$

Since a_{ij}, a_{i1}, a_{1j} are nonpositive and a_{11} is positive, it follows that $a_{ij}^1 \leq 0$ for $i \neq j$.

Second, the fact that A_1 is nonsingular is a trivial consequence of the following standard relation of Gaussian elimination

$$A = L_1 A_1 \quad \text{where} \quad L_1 = \left[\frac{A_{*,1}}{a_{11}}, e_2, e_3, \dots e_n \right]. \tag{10.10}$$

Finally, we establish that A_1^{-1} is nonnegative by examining $A_1^{-1}e_j$ for $j = 1, \dots, n$. For $j = 1$, it is clear that $A_1^{-1}e_1 = \frac{1}{a_{11}}e_1$ because of the structure of A_1. For the case $j \neq 1$, (10.10) can be exploited to yield

$$A_1^{-1}e_j = A^{-1}L_1^{-1}e_j = A^{-1}e_j \geq 0.$$

Therefore, all the columns of A_1^{-1} are nonnegative by assumption and this completes the proof. ■

Clearly, the $(n-1) \times (n-1)$ matrix obtained from A_1 by removing its first row and first column is also an M-matrix.

Assume now that some elements are dropped from the result of Gaussian Elimination outside of the main diagonal. Any element that is dropped is a nonpositive element which is transformed into a zero. Therefore, the resulting matrix $\tilde{A}_1$ is such that

$$\tilde{A}_1 = A_1 + R,$$

where the elements of R are such that $r_{ii} = 0, r_{ij} \geq 0$. Thus,

$$A_1 \leq \tilde{A}_1$$

and the off-diagonal elements of $\tilde{A}_1$ are nonpositive. Since A_1 is an M-matrix, theorem 1.18 shows that $\tilde{A}_1$ is also an M-matrix. The process can now be repeated on the matrix $\tilde{A}(2:n, 2:n)$, and then continued until the incomplete factorization of A is obtained. The above arguments shows that at each step of this construction, we obtain an M-matrix and that the process does not break down.

The elements to drop at each step have not yet been specified. This can be done statically, by choosing some non-zero pattern in advance. The only restriction on the zero pattern is that it should exclude diagonal elements because this assumption was used in the above proof. Therefore, for any zero pattern set P, such that

$$P \subset \{(i,j) \mid i \neq j; 1 \leq i,j \leq n\}, \tag{10.11}$$

an Incomplete LU factorization, ILU_P, can be computed as follows.

ALGORITHM 10.1: General Static Pattern ILU

```
1. For $k = 1, \ldots, n-1$ Do:
2.    For $i = k+1, n$ and if $(i,k) \notin P$ Do:
3.       $a_{ik} := a_{ik}/a_{kk}$
4.       For $j = k+1, \ldots, n$ and for $(i,j) \notin P$ Do:
5.          $a_{ij} := a_{ij} - a_{ik} * a_{kj}$
6.       EndDo
7.    EndDo
8. EndDo
```

The *For* loop in line 4 should be interpreted as follows: *For $j = k+1, \ldots, n$ and only for those indices j that are not in P execute the next line.* In practice, it is wasteful to scan j from $k+1$ to n because there is an inexpensive mechanism for identifying those in this set that are in the complement of P.

Using the above arguments, the following result can be proved.

THEOREM 10.2 *Let A be an M-matrix and P a given zero pattern defined as in (10.11). Then Algorithm 10.1 is feasible and produces an incomplete factorization,*

$$A = LU - R \tag{10.12}$$

which is a regular splitting of A.

Proof. At each step of the process, we have

$$\tilde{A}_k = A_k + R_k, \quad A_k = L_k \tilde{A}_{k-1}$$

where, using O_k to denote a zero vector of dimension k, and $A_{m:n,j}$ to denote the vector of components $a_{i,j}, i = m, \ldots, n$,

$$L_k = I - \frac{1}{a_{kk}^{(k)}} \begin{pmatrix} O_k \\ A(k+1:n,k) \end{pmatrix} e_k^T.$$

From this follow the relations

$$\tilde{A}_k = A_k + R_k = L_k \tilde{A}_{k-1} + R_k.$$

Applying this relation recursively, starting from $k = n-1$ up to $k = 1$, it is found that

$$\tilde{A}_{n-1} = L_{n-1} \ldots L_1 A + L_{n-1} \ldots L_2 R_1 + \ldots + L_{n-1} R_{n-2} + R_{n-1}. \tag{10.13}$$

Now define

$$L = (L_{n-1} \ldots L_1)^{-1}, \quad U = \tilde{A}_{n-1}.$$

Then,

$$U = L^{-1}A + S$$

with

$$S = L_{n-1} \ldots L_2 R_1 + \ldots + L_{n-1}R_{n-2} + R_{n-1}.$$

Observe that at stage k, elements are dropped only in the $(n-k) \times (n-k)$ lower part of A_k. Hence, the first k rows and columns of R_k are zero and as a result

$$L_{n-1} \ldots L_{k+1} R_k = L_{n-1} \ldots L_1 R_k$$

so that S can be rewritten as

$$S = L_{n-1} \ldots L_2 (R_1 + R_2 + \ldots + R_{n-1}).$$

If R denotes the matrix

$$R = R_1 + R_2 + \ldots + R_{n-1},$$

then we obtain the factorization

$$A = LU - R,$$

where $(LU)^{-1} = U^{-1}L^{-1}$ is a nonnegative matrix, R is nonnegative. This completes the proof. ■

Now consider a few practical aspects. An ILU factorization based on the form of Algorithm 10.1 is difficult to implement because at each step k, all rows $k+1$ to n are being modified. However, ILU factorizations depend on the implementation of Gaussian elimination which is used. Several variants of Gaussian elimination are known which depend on the order of the three loops associated with the control variables i, j, and k in the algorithm. Thus, Algorithm 10.1 is derived from what is known as the k, i, j variant. In the context of Incomplete LU factorization, the variant that is most commonly used for a row-contiguous data structure is the i, k, j variant, described next for dense matrices.

ALGORITHM 10.2: Gaussian Elimination – IKJ Variant

```
1. For $i = 2, \ldots, n$ Do:
2.    For $k = 1, \ldots, i-1$ Do:
3.       $a_{ik} := a_{ik}/a_{kk}$
4.       For $j = k+1, \ldots, n$ Do:
5.          $a_{ij} := a_{ij} - a_{ik} * a_{kj}$
6.       EndDo
7.    EndDo
8. EndDo
```

The above algorithm is in place meaning that the i-th row of A can be overwritten by the i-th rows of the L and U matrices of the factorization (since L is unit lower triangular, its diagonal entries need not be stored). Each step i of the algorithm generates the i-th row of L and the i-th row of U at the same time. The previous rows $1, 2, \ldots, i-1$ of L and U are accessed at step i but they are not modified. This is illustrated in Figure 10.1.

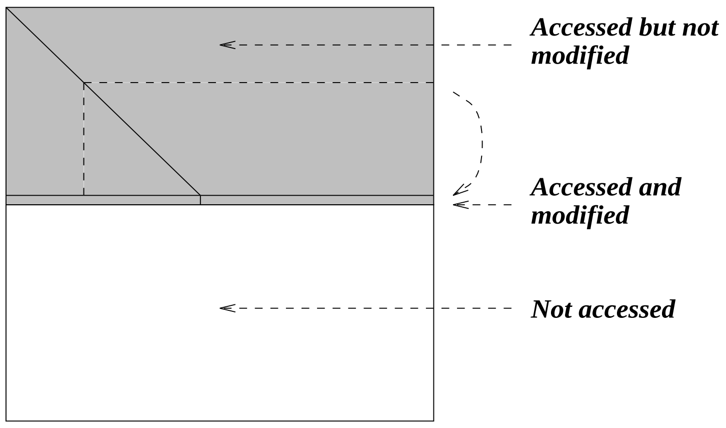


Figure 10.1 *IKJ variant of the LU factorization.*

Adapting this version for sparse matrices is easy because the rows of L and U are generated in succession. These rows can be computed one at a time and accumulated in a row-oriented data structure such as the CSR format. This constitutes an important advantage. Based on this, the general ILU factorization takes the following form.

ALGORITHM 10.3: General ILU Factorization, IKJ Version

```
1. For $i = 2, \ldots, n$ Do:
2.    For $k = 1, \ldots, i-1$ and if $(i,k) \notin P$ Do:
3.          $a_{ik} := a_{ik}/a_{kk}$
4.          For $j = k+1, \ldots, n$ and for $(i,j) \notin P$, Do:
5.               $a_{ij} := a_{ij} - a_{ik}a_{kj}$.
6.          EndDo
7.    EndDo
8. EndDo
```

It is not difficult to see that this more practical IKJ variant of ILU is equivalent to the KIJ version which can be defined from Algorithm 10.1.

PROPOSITION 10.1 *Let P be a zero pattern satisfying the condition (10.11). Then the ILU factors produced by the KIJ-based Algorithm 10.1 and the IKJ-based Algorithm 10.3 are identical if they can both be computed.*

Proof. Algorithm (10.3) is obtained from Algorithm 10.1 by switching the order of the loops k and i. To see that this gives indeed the same result, reformulate the first two loops of Algorithm 10.1 as

```
For $k = 1, n$ Do:
  For $i = 1, n$ Do:
    if $k < i$ and for $(i, k) \notin P$ Do:
      ope(row(i),row(k))
      ......
```

in which *ope(row(i),row(k))* is the operation represented by lines 3 through 6 of both Algorithm 10.1 and Algorithm 10.3. In this form, it is clear that the k and i loops can be safely permuted. Then the resulting algorithm can be reformulated to yield exactly Algorithm 10.3. ■

Note that this is only true for a static pattern ILU. If the pattern is dynamically determined as the Gaussian elimination algorithm proceeds, then the patterns obtained with different versions of GE may be different.

It is helpful to interpret the result of one incomplete elimination step. Denoting by l_{i*}, u_{i*}, and a_{i*} the i-th rows of L, U, and A, respectively, then the k-loop starting at line 2 of Algorithm 10.3 can be interpreted as follows. Initially, we have $u_{i*} = a_{i*}$. Then, each elimination step is an operation of the form

$$u_{i*} := u_{i*} - l_{ik} u_{k*}.$$

However, this operation is performed only on the nonzero pattern, i.e., the complement of P. This means that, in reality, the elimination step takes the form

$$u_{i*} := u_{i*} - l_{ik} u_{k*} + r_{i*}^{(k)},$$

in which $r_{ij}^{(k)}$ is zero when $(i, j) \notin P$ and equals $l_{ik} u_{kj}$ when $(i, j) \in P$. Thus, the row $r_{i*}^{(k)}$ cancels out the terms $l_{ik} u_{kj}$ that would otherwise be introduced in the zero pattern. In the end the following relation is obtained:

$$u_{i*} = a_{i*} - \sum_{k=1}^{i-1} \left(l_{ik} u_{k*} - r_{i*}^{(k)} \right).$$

Note that $l_{ik} = 0$ for $(i, k) \in P$. We now sum up all the $r_{i*}^{(k)}$'s and define

$$r_{i*} = \sum_{k=1}^{i-1} r_{i*}^{(k)}. \tag{10.14}$$

The row r_{i*} contains the elements that fall inside the P pattern at the completion of the k-loop. Using the fact that $l_{ii} = 1$, we obtain the relation,

$$a_{i*} = \sum_{k=1}^{i} l_{ik} u_{k*} - r_{i*}. \tag{10.15}$$

Therefore, the following simple property can be stated.

PROPOSITION 10.2 *Algorithm (10.3) produces factors L and U such that*

$$A = LU - R$$

in which $-R$ is the matrix of the elements that are dropped during the incomplete elimination process. When $(i, j) \in P$, an entry r_{ij} of R is equal to the value of $-a_{ij}$ obtained at *the completion of the k loop in Algorithm 10.3. Otherwise, r_{ij} is zero.*

10.3.2 ZERO FILL-IN ILU (ILU(0))

The *Incomplete LU* factorization technique with no fill-in, denoted by ILU(0), consists of taking the zero pattern P to be precisely the zero pattern of A. In the following, we denote by $b_{i,*}$ the i-th row of a given matrix B, and by $NZ(B)$, the set of pairs $(i,j), 1 \le i,j \le n$ such that $b_{i,j} \neq 0$.

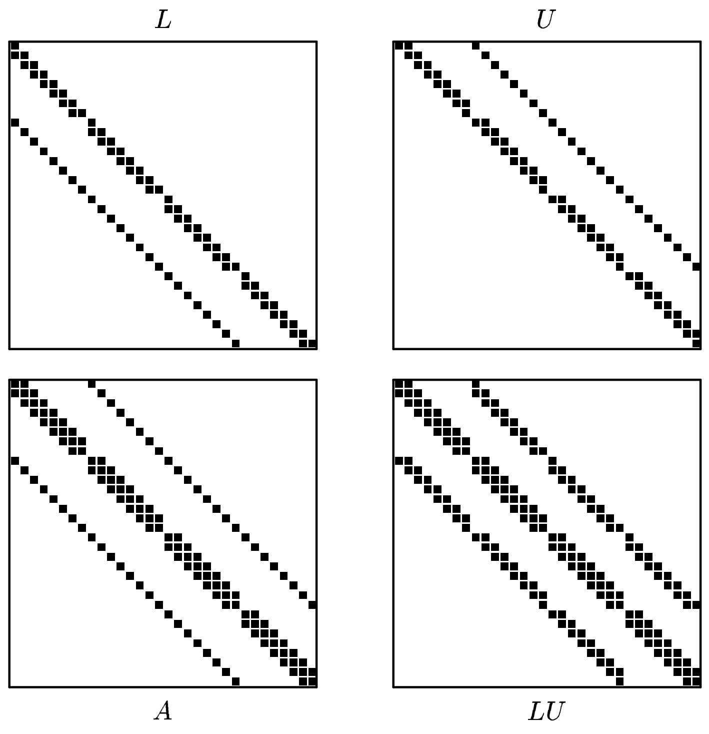


Figure 10.2 *The ILU(0) factorization for a five-point matrix.*

The incomplete factorization ILU(0) factorization is best illustrated by the case for which it was discovered originally, namely, for 5-point and 7-point matrices related to finite difference discretization of PDEs. Consider one such matrix A as illustrated in the bottom left corner of Figure 10.2. The A matrix represented in this figure is a 5-point matrix of size $n = 32$ corresponding to an $n_x \times n_y = 8 \times 4$ mesh. Consider now any lower triangular matrix L which has the same structure as the lower part of A, and any matrix U which has the same structure as that of the upper part of A. Two such matrices are shown at the top of Figure 10.2. If the product LU were performed, the resulting matrix would have the pattern shown in the bottom right part of the figure. It is impossible in general to match A with this product for any L and U. This is due to the extra diagonals in the product, namely, the diagonals with offsets $n_x - 1$ and $-n_x + 1$. The entries in these extra diagonals are called *fill-in elements*. However, if these fill-in elements are ignored, then it is possible to find L and U so that their product is equal to A in the other diagonals. This defines the ILU(0) factorization in general terms: Any pair of matrices L (unit lower triangular) and U (upper triangular) so that the elements of $A - LU$ are zero in the locations of $NZ(A)$. These constraints do not define the ILU(0) factors uniquely since there are, in general, infinitely many pairs of matrices L and U which satisfy these requirements. However, the standard ILU(0) is defined constructively using Algorithm 10.3 with the pattern P equal to the zero pattern of A.

ALGORITHM 10.4: ILU(0)

```
1. For $i = 2, \ldots, n$ Do:
2.    For $k = 1, \ldots, i-1$ and for $(i,k) \in NZ(A)$ Do:
3.        Compute $a_{ik} = a_{ik}/a_{kk}$
4.        For $j = k+1, \ldots, n$ and for $(i,j) \in NZ(A)$, Do:
5.            Compute $a_{ij} := a_{ij} - a_{ik}a_{kj}$.
6.        EndDo
7.    EndDo
8. EndDo
```

In some cases, it is possible to write the ILU(0) factorization in the form

$$M = (D - E)D^{-1}(D - F), \tag{10.16}$$

where $-E$ and $-F$ are the strict lower and strict upper triangular parts of A, and D is a certain diagonal matrix, different from the diagonal of A, in general. In these cases it is sufficient to find a recursive formula for determining the elements in D. A clear advantage is that only an extra diagonal of storage is required. This form of the ILU(0) factorization is equivalent to the incomplete factorizations obtained from Algorithm 10.4 when the product of the *strict-lower part* and the *strict-upper part* of A consists only of diagonal elements and fill-in elements. This is true, for example, for standard 5-point difference approximations to second order partial differential operators; see Exercise 4. In these instances, both the SSOR preconditioner with $\omega = 1$ and the ILU(0) preconditioner can be cast in the form (10.16), but they differ in the way the diagonal matrix D is defined. For SSOR($\omega = 1$), D is the diagonal of the matrix A itself. For ILU(0), it is defined by a recursion so that the diagonal of the product of matrices (10.16) equals the diagonal of A. By definition, together the L and U matrices in ILU(0) have the same number of nonzero elements as the original matrix A.

Example 10.2 Table 10.2 shows the results of applying the GMRES algorithm with ILU(0) preconditioning to the five test problems described in Section 3.7.

Matrix	Iters	Kflops	Residual	Error
F2DA	28	1456	0.12E-02	0.12E-03
F3D	17	4004	0.52E-03	0.30E-03
ORS	20	1228	0.18E+00	0.67E-04
F2DB	300	15907	0.23E+02	0.67E+00
FID	206	67970	0.19E+00	0.11E-03

Table 10.2 *A test run of GMRES with ILU(0) preconditioning.*

See Example 6.1 for the meaning of the column headers in the table. Observe that for the first two problems, the gains compared with the performance of the SSOR preconditioner in Table 10.1 are rather small. For the other three problems, which are a little harder, the gains are more substantial. For the last problem, the algorithm achieves convergence in 205 steps whereas SSOR did not convergence in the 300 steps allowed. The fourth problem (F2DB) is still not solvable by ILU(0) within the maximum number of steps allowed.

For the purpose of illustration, below is a sample FORTRAN code for computing the incomplete L and U factors for general sparse matrices stored in the usual CSR format. The real values of the resulting L, U factors are stored in the array *luval*, except that entries of ones of the main diagonal of the unit lower triangular matrix L are not stored. Thus, one matrix is needed to store these factors together. This matrix is denoted by L/U. Note that since the pattern of L/U is identical with that of A, the other integer arrays of the CSR representation for the LU factors are not needed. Thus, $ja(k)$, which is the column position of the element $a(k)$ in the input matrix, is also the column position of the element $luval(k)$ in the L/U matrix. The code below assumes that the nonzero elements in the input matrix A are sorted by increasing column numbers in each row.

FORTRAN CODE

```
      subroutine ilu0 (n, a, ja, ia, luval, uptr, iw, icode)
      integer n, ja(*), ia(n+1), uptr(n), iw(n)
      real*8 a(*), luval(*)
c-----------------------------------------------------------
c Set-up routine for  ILU(0)  preconditioner.  This routine
c computes the L and  U factors of the ILU(0) factorization
c of a general sparse matrix A  stored in CSR format. Since
c L is unit  triangular, the L and U  factors can be stored
c as a single matrix which occupies  the same storage as A.
c The ja and  ia arrays are not  needed  for the  LU matrix
c since   the  pattern of the  LU  matrix is identical with
c that of A.
c-----------------------------------------------------------
c INPUT:
c ------
c n          = dimension of matrix
c a, ja, ia = sparse matrix in general sparse storage format
c iw         = integer work array of length n
c OUTPUT:
c -------
c luval      = L/U matrices stored together. On return luval,
c              ja, ia is the combined CSR data structure for
c              the LU factors
```

```
c uptr      = pointer to the diagonal elements in the CSR
c             data structure luval, ja, ia
c icode     = integer indicating error code on return
c             icode = 0: normal return
c             icode = k: encountered a zero pivot at step k
c
c-----------------------------------------------------------
c     initialize  work array iw to zero and luval array to a
      do 30 i = 1, ia(n+1)-1
          luval(i) = a(i)
30    continue
      do 31 i=1, n
          iw(i) = 0
 31   continue
c----------------------- Main loop
      do 500 k = 1, n
         j1 = ia(k)
         j2 = ia(k+1)-1
         do 100 j=j1, j2
            iw(ja(j)) = j
 100     continue
         j=j1
 150     jrow = ja(j)
c----------------------- Exit if diagonal element is reached
         if (jrow .ge. k) goto 200
c----------------------- Compute the multiplier for jrow.
         tl = luval(j)*luval(uptr(jrow))
         luval(j) = tl
c----------------------- Perform linear combination
         do 140 jj = uptr(jrow)+1, ia(jrow+1)-1
            jw = iw(ja(jj))
            if (jw .ne. 0) luval(jw)=luval(jw)-tl*luval(jj)
 140     continue
         j=j+1
         if (j .le. j2) goto 150
c----------------------- Store pointer to diagonal element
 200     uptr(k) = j
         if (jrow .ne. k .or. luval(j) .eq. 0.0d0) goto 600
         luval(j) = 1.0d0/luval(j)
c----------------------- Refresh all entries of iw to zero.
         do 201 i = j1, j2
            iw(ja(i)) = 0
 201     continue
 500  continue
c----------------------- Normal return
      icode = 0
      return
c----------------------- Error: zero pivot
 600  icode = k
      return
      end
```

10.3.3 LEVEL OF FILL AND ILU(P)

The accuracy of the ILU(0) incomplete factorization may be insufficient to yield an adequate rate of convergence as shown in Example 10.2. More accurate Incomplete LU factorizations are often more efficient as well as more reliable. These more accurate factorizations will differ from ILU(0) by allowing some fill-in. Thus, ILU(1) keeps the “first order fill-ins,” a term which will be explained shortly.

To illustrate ILU(p) with the same example as before, the ILU(1) factorization results from taking P to be the zero pattern of the product LU of the factors L, U obtained from ILU(0). This pattern is shown at the bottom right of Figure 10.2. Pretend that the original matrix has this “augmented” pattern $NZ_1(A)$. In other words, the fill-in positions created in this product belong to the augmented pattern $NZ_1(A)$, but their actual values are zero. The new pattern of the matrix A is shown at the bottom left part of Figure 10.3. The factors L_1 and U_1 of the ILU(1) factorization are obtained by performing an ILU(0) factorization on this “augmented pattern” matrix. The patterns of L_1 and U_1 are illustrated at the top of Figure 10.3. The new LU matrix shown at the bottom right of the figure has now two additional diagonals in the lower and upper parts.

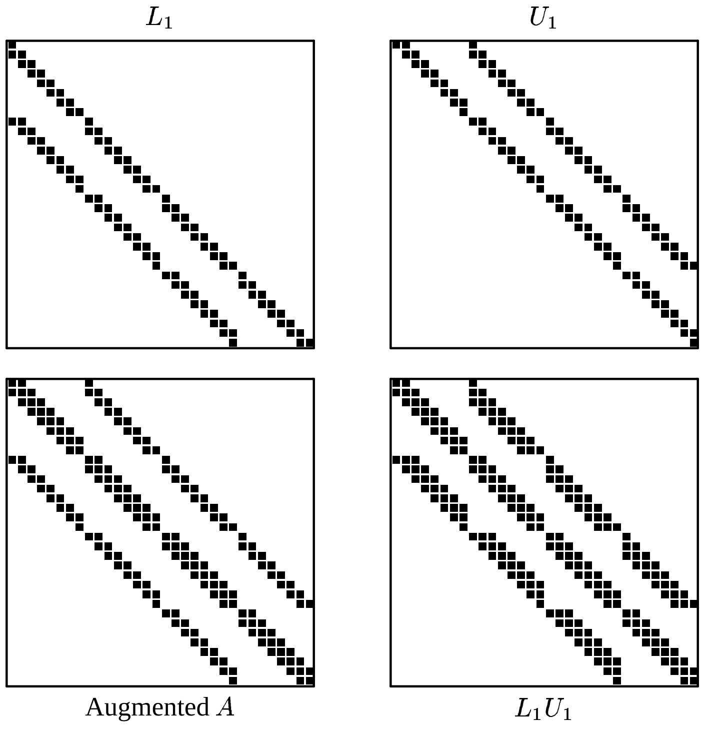


Figure 10.3 *The ILU(1) factorization.*

One problem with the construction defined in this illustration is that it does not extend to general sparse matrices. It can be generalized by introducing the concept of *level of fill*. A level of fill is attributed to each element that is processed by Gaussian elimination, and dropping will be based on the value of the level of fill. Algorithm 10.2 will be used as a model, although any other form of GE can be used. The rationale is that the level of fill should be indicative of the size: the higher the level, the smaller the elements. A very simple model is employed to justify the definition: A size of ϵ^k is attributed to any element whose level of fill is k, where $\epsilon < 1$. Initially, a nonzero element has a level of fill of one (this will be changed later) and a zero element has a level of fill of ∞. An element a_{ij} is updated in line 5 of Algorithm 10.2 by the formula

$$a_{ij} = a_{ij} - a_{ik} \times a_{kj}. \tag{10.17}$$

If lev_{ij} is the current level of the element a_{ij}, then our model tells us that the size of the updated element should be

$$a_{ij} := \epsilon^{lev_{ij}} - \epsilon^{lev_{ik}} \times \epsilon^{lev_{kj}} = \epsilon^{lev_{ij}} - \epsilon^{lev_{ik}+lev_{kj}}.$$

Therefore, roughly speaking, the size of a_{ij} will be the maximum of the two sizes $\epsilon^{lev_{ij}}$ and $\epsilon^{lev_{ik}+lev_{kj}}$, and it is natural to define the new level of fill as,

$$lev_{ij} := \min\{lev_{ij}, lev_{ik} + lev_{kj}\}.$$

In the common definition used in the literature, all the levels of fill are actually shifted by -1 from the definition used above. This is purely for convenience of notation and to conform with the definition used for ILU(0). Thus, initially $lev_{ij} = 0$ if $a_{ij} \neq 0$, and $lev_{ij} = \infty$ otherwise. Thereafter, define recursively

$$lev_{ij} = \min\{lev_{ij}, lev_{ik} + lev_{kj} + 1\}.$$

DEFINITION 10.1 *The initial level of fill of an element a_{ij} of a sparse matrix A is defined by*

$$lev_{ij} = \begin{cases} 0 & \text{if } a_{ij} \neq 0, \text{or } i = j \\ \infty & \text{otherwise.} \end{cases}$$

Each time this element is modified in line 5 of Algorithm 10.2, its level of fill must be updated by

$$lev_{ij} = \min\{lev_{ij}, lev_{ik} + lev_{kj} + 1\}. \tag{10.18}$$

Observe that the level of fill of an element will never increase during the elimination. Thus, if $a_{ij} \neq 0$ in the original matrix A, then the element in location i, j will have a level of fill equal to zero throughout the elimination process. The above systematic definition gives rise to a natural strategy for discarding elements. In ILU(p), all fill-in elements whose level of fill does not exceed p are kept. So using the definition of zero patterns introduced earlier, the zero pattern for ILU(p) is the set

$$P_p = \{(i,j) \mid lev_{ij} > p\},$$

where lev_{ij} is the level of fill value after all updates (10.18) have been performed. The case $p = 0$ coincides with the ILU(0) factorization and is consistent with the earlier definition.

In practical implementations of the ILU(p) factorization it is common to separate the symbolic phase (where the structure of the L and U factors are determined) from the numerical factorization, when the numerical values are computed. Here, a variant is described which does not separate these two phases. In the following description, a_{i*} denotes the i-th row of the matrix A, and a_{ij} the (i, j)-th entry of A.

ALGORITHM 10.5: ILU(p)

```
1. For all nonzero elements $a_{ij}$ define $lev(a_{ij}) = 0$
2. For $i = 2, \ldots, n$ Do:
```

```
3.     For each $k = 1, \ldots, i-1$ and for $lev(a_{ik}) \leq p$ Do:
4.         Compute $a_{ik} := a_{ik}/a_{kk}$
5.         Compute $a_{i*} := a_{i*} - a_{ik}a_{k*}$.
6.         Update the levels of fill of the nonzero $a_{i,j}$'s using (10.18)
7.     EndDo
8.     Replace any element in row $i$ with $lev(a_{ij}) > p$ by zero
9.  EndDo
```

There are a number of drawbacks to the above algorithm. First, the amount of fill-in and computational work for obtaining the ILU(p) factorization is not predictable for $p > 0$. Second, the cost of updating the levels can be quite high. Most importantly, the level of fill-in for indefinite matrices may not be a good indicator of the size of the elements that are being dropped. Thus, the algorithm may drop large elements and result in an inaccurate incomplete factorization, in the sense that $R = LU - A$ is not small. Experience reveals that *on the average* this will lead to a larger number of iterations to achieve convergence, although there are certainly instances where this is not the case. The techniques which will be described in Section 10.4 have been developed to remedy these three difficulties, by producing incomplete factorizations with small error R and a controlled number of fill-ins.

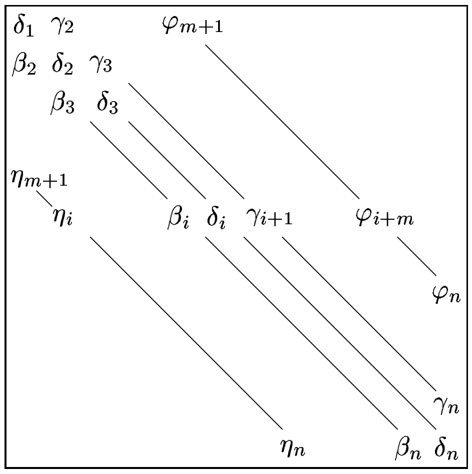

δ_1 γ_2 φ_{m+1}
β_2 δ_2 γ_3
β_3 δ_3
η_{m+1}
η_i β_i δ_i γ_{i+1} φ_{i+m}
φ_n
γ_n
η_n β_n δ_n

Figure 10.4 *Matrix resulting from the discretization of an elliptic problem on a rectangle.*

10.3.4 MATRICES WITH REGULAR STRUCTURE

Often, the original matrix has a regular structure which can be exploited to formulate the ILU preconditioners in a simpler way. Historically, incomplete factorization preconditioners were developed first for such matrices, rather than for general sparse matrices. Here, we call a regularly structured matrix a matrix consisting of a small number of diagonals. As an example, consider the diffusion-convection equation, with Dirichlet boundary conditions

$$-\Delta u + \vec{b}.\nabla u = f \text{ in } \Omega$$
$$u = 0 \text{ on } \partial\Omega$$

where Ω is simply a rectangle. As seen in Chapter 2, if the above problem is discretized using centered differences, a linear system is obtained whose coefficient matrix has the structure shown in Figure 10.4. In terms of the stencils seen in Chapter 4, the representation of this matrix is rather simple. Each row expresses the coupling between unknown i and unknowns $i+1$, $i-1$ which are in the horizontal, or x direction, and the unknowns $i+m$ and $i-m$ which are in the vertical, or y direction. This stencil is represented in Figure 10.5.

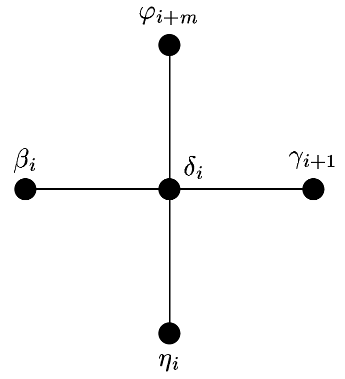


Figure 10.5 *Stencil associated with the 5-point matrix shown in Figure 10.4.*

The desired L and U matrices in the ILU(0) factorization are shown in Figure 10.6.

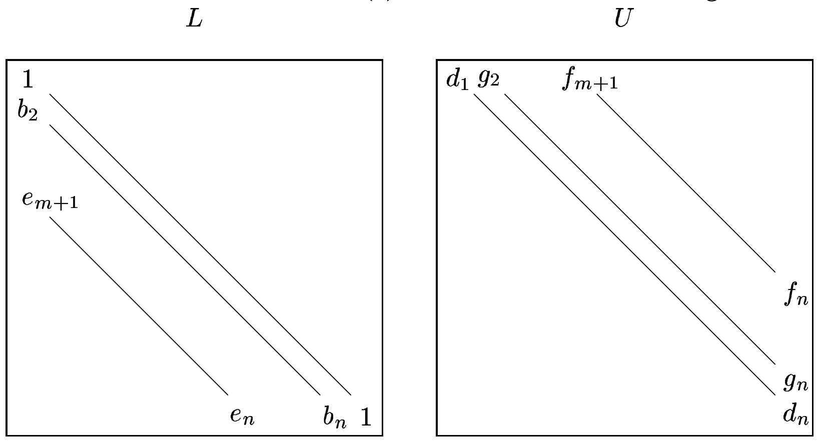


Figure 10.6 *L and U factors of the ILU(0) factorization for the 5-point matrix shown in Figure 10.4.*

Now the respective stencils of these L and U matrices can be represented at a mesh point i as shown in Figure 10.7.

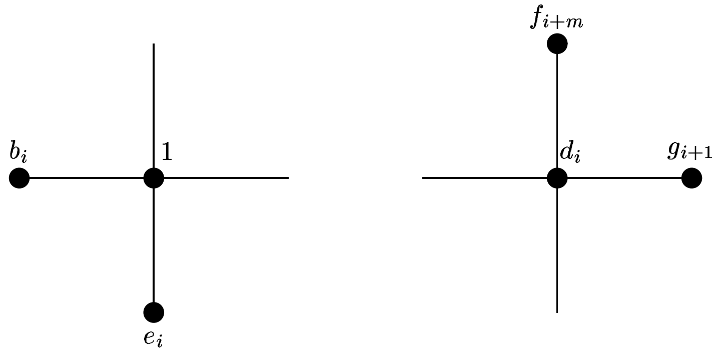


Figure 10.7 *Stencils associated with the L and U factors shown in Figure 10.6.*

The stencil of the product LU can be obtained easily by manipulating stencils directly rather than working with the matrices they represent. Indeed, the i-th row of LU is obtained by performing the following operation:

$$row_i(LU) = 1 \times row_i(U) + b_i \times row_{i-1}(U) + e_i \times row_{i-m}(U).$$

This translates into a combination of the stencils associated with the rows:

$$stencil_i(LU) = 1 \times stencil_i(U) + b_i \times stencil_{i-1}(U) + e_i \times stencil_{i-m}(U)$$

in which $stencil_j(X)$ represents the stencil of the matrix X based at the mesh point labeled j. This gives the stencil for the LU matrix represented in Figure 10.8.

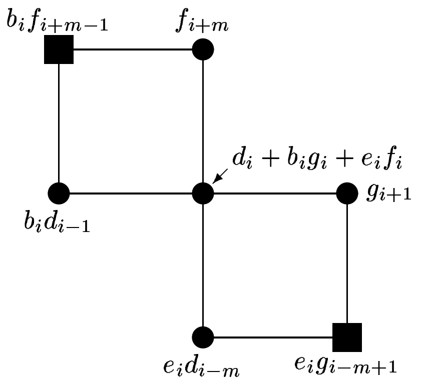


Figure 10.8 *Stencil associated with the product of the L and U factors shown in Figure 10.6.*

In the figure, the fill-in elements are represented by squares and all other nonzero elements of the stencil are filled circles. The ILU(0) process consists of identifying LU with A in locations where the original a_{ij}'s are nonzero. In the Gaussian eliminations process, this is done from $i = 1$ to $i = n$. This provides the following equations obtained directly from comparing the stencils of LU and A (going from lowest to highest indices)

$$e_i d_{i-m} = \eta_i$$

$$\begin{aligned} b_i d_{i-1} &= \beta_i \\ d_i + b_i g_i + e_i f_i &= \delta_i \\ g_{i+1} &= \gamma_{i+1} \\ f_{i+m} &= \varphi_{i+m}. \end{aligned}$$

Observe that the elements g_{i+1} and f_{i+m} are identical with the corresponding elements of the A matrix. The other values are obtained from the following recurrence:

$$\begin{aligned} e_i &= \frac{\eta_i}{d_{i-m}} \\ b_i &= \frac{\beta_i}{d_{i-1}} \\ d_i &= \delta_i - b_i g_i - e_i f_i. \end{aligned}$$

The above recurrence can be simplified further by making the observation that the quantities η_i / d_{i-m} and β_i / d_{i-1} need not be saved since they are scaled versions of the corresponding elements in A. With this observation, *only a recurrence for the diagonal elements d_i is needed*. This recurrence is:

$$d_i = \delta_i - \frac{\beta_i \gamma_i}{d_{i-1}} - \frac{\eta_i \varphi_i}{d_{i-m}}, \quad i = 1, \ldots, n, \tag{10.19}$$

with the convention that any d_j with a non-positive index j is replaced by 1 and any other element with a negative index is zero. The factorization obtained takes the form

$$M = (D - E) D^{-1} (D - F) \tag{10.20}$$

in which $-E$ is the strict lower diagonal of A, $-F$ is the strict upper triangular part of A, and D is the diagonal obtained with the above recurrence. Note that an ILU(0) based on the IKJversion of Gaussian elimination would give the same result.

For a general sparse matrix A with irregular structure, one can also determine a preconditioner in the form (10.20) by requiring only that the diagonal elements of M match those of A (see Exercise 10). However, this will not give the same ILU factorization as the one based on the IKJvariant of Gaussian elimination seen earlier. Why the ILU(0) factorization gives rise to the same factorization as that of (10.20) is simple to understand: The product of L and U does not change the values of the existing elements in the upper part, except for the diagonal. This also can be interpreted on the adjacency graph of the matrix.

This approach can now be extended to determine the ILU(1) factorization as well as factorizations with higher levels of fill. The stencils of the L and U matrices in the ILU(1) factorization are the stencils of the lower part and upper parts of the LU matrix obtained from ILU(0). These are shown in Figure 10.9. In the illustration, the meaning of a given stencil is not in the usual graph theory sense. Instead, all the marked nodes at a stencil based at node i represent those nodes coupled with unknown i by an equation. Thus, all the filled circles in the picture are adjacent to the central node. Proceeding as before and combining stencils to form the stencil associated with the LU matrix, we obtain the stencil shown in Figure 10.10.

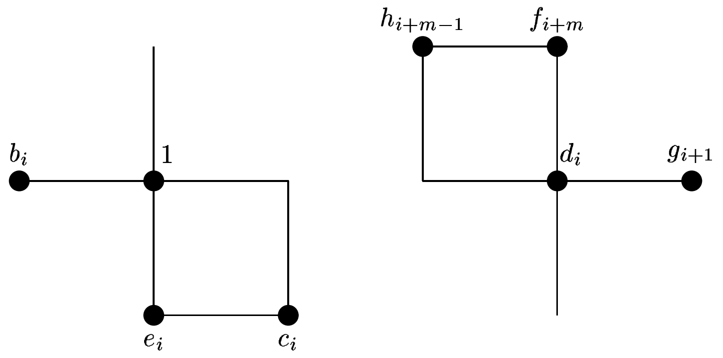


Figure 10.9 *Stencils associated with the L and U factors of the ILU(0) factorization for the matrix associated with the stencil of Figure 10.8.*

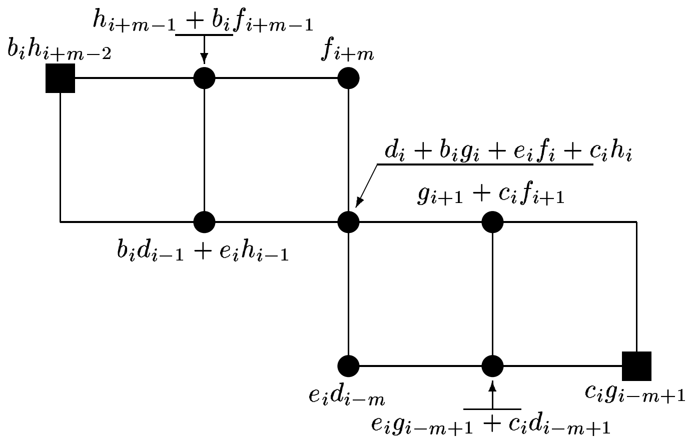

$h_{i+m-1} + b_i f_{i+m-1}$
$b_i h_{i+m-2}$
f_{i+m}
$d_i + b_i g_i + e_i f_i + c_i h_i$
$g_{i+1} + c_i f_{i+1}$
$b_i d_{i-1} + e_i h_{i-1}$
$e_i d_{i-m}$
$c_i g_{i-m+1}$
$e_i g_{i-m+1} + c_i d_{i-m+1}$

Figure 10.10 *Stencil associated with the product of the L and U matrices whose stencils are shown in Figure 10.9.*

As before, the fill-in elements are represented by squares and all other elements are filled circles. A typical row of the matrix associated with the above stencil has nine nonzero elements. Two of these are fill-ins, i.e., elements that fall outside the original structure of the L and U matrices. It is now possible to determine a recurrence relation for obtaining the entries of L and U. There are seven equations in all which, starting from the bottom, are

$$\begin{aligned}
e_i d_{i-m} &= \eta_i \\
e_i g_{i-m+1} + c_i d_{i-m+1} &= 0 \\
b_i d_{i-1} + e_i h_{i-1} &= \beta_i \\
d_i + b_i g_i + e_i f_i + c_i h_i &= \delta_i \\
g_{i+1} + c_i f_{i+1} &= \gamma_{i+1} \\
h_{i+m-1} + b_i f_{i+m-1} &= 0 \\
f_{i+m} &= \varphi_{i+m}.
\end{aligned}$$

This immediately yields the following recurrence relation for the entries of the L and U factors:

$$\begin{aligned} e_i &= \eta_i / d_{i-m} \\ c_i &= -e_i g_{i-m+1} / d_{i-m+1} \\ b_i &= \left(\beta_i - e_i h_{i-1}\right) / d_{i-1} \\ d_i &= \delta_i - b_i g_i - e_i f_i - c_i h_i \\ g_{i+1} &= \gamma_{i+1} - c_i f_{i+1} \\ h_{i+m-1} &= -b_i f_{i+m-1} \\ f_{i+m} &= \varphi_{i+m}. \end{aligned}$$

In proceeding from the nodes of smallest index to those of largest index, we are in effect performing implicitly the IKJversion of Gaussian elimination. The result of the ILU(1) obtained in this manner is therefore identical with that obtained by using Algorithms 10.1 and 10.3.

10.3.5 MODIFIED ILU (MILU)

In all the techniques thus far, the elements that were dropped out during the incomplete elimination process are simply discarded. There are also techniques which attempt to reduce the effect of dropping by *compensating* for the discarded entries. For example, a popular strategy is to add up all the elements that have been dropped at the completion of the k-loop of Algorithm 10.3. Then this sum is subtracted from the diagonal entry in U. This *diagonal compensation* strategy gives rise to the Modified ILU (MILU) factorization.

Thus, in equation (10.14), the final row u_{i*} obtained after completion of the k-loop of Algorithm 10.3 undergoes one more modification, namely,

$$u_{ii} := u_{ii} - (r_{i*}e)$$

in which $e \equiv (1, 1, \ldots, 1)^T$. Note that r_{i*} is a row and $r_{i*}e$ is the sum of the elements in this row, i.e., its *row sum*. The above equation can be rewritten in row form as $u_{i*} := u_{i*} - (r_{i*}e)e_i^T$ and equation (10.15) becomes

$$a_{i*} = \sum_{k=1}^{i} l_{ik} u_{k*} + (r_{i*}e)e_i^T - r_{i*}. \tag{10.21}$$

Observe that

$$a_{i*}e = \sum_{k=1}^{i} l_{ik} u_{k*} e + (r_{i*}e)e_i^T e - r_{i*}e = \sum_{k=1}^{i-1} l_{ik} u_{k*} e = LU\, e.$$

This establishes that $Ae = LUe$. As a result, this strategy guarantees that the row sums of A are equal to those of LU. For PDEs, the vector of all ones represents the discretization of a constant function. This additional constraint forces the ILU factorization to be exact for constant functions in some sense. Therefore, it is not surprising that often the algorithm does well for such problems. For other problems or problems with discontinuous coefficients, MILU algorithms usually are not better than their ILU counterparts, in general.

Example 10.3 For regularly structured matrices there are two elements dropped at the i-th step of ILU(0). These are $b_i f_{i+m-1}$ and $e_i g_{i-m+1}$ located on the north-west and south-east corners of the stencil, respectively. Thus, the row sum $r_{i,*}e$ associated with step i is

$$s_i = \frac{\beta_i \phi_{i+m-1}}{d_{i-1}} + \frac{\eta_i \gamma_{m-i+1}}{d_{i-m}}$$

and the MILU variant of the recurrence (10.19) is

$$s_i = \frac{\beta_i \phi_{i+m-1}}{d_{i-1}} + \frac{\eta_i \gamma_{m-i+1}}{d_{i-m}}$$
$$d_i = \delta_i - \frac{\beta_i \gamma_i}{d_{i-1}} - \frac{\eta_i \varphi_i}{d_{i-m}} - s_i.$$

The new ILU factorization is now such that $A = LU - R$ in which according to (10.21) the i-th row of the new remainder matrix R is given by

$$r_{i,*}^{(new)} = (r_{i*}e)e_i^T - r_{i*}$$

whose row sum is zero.

This generic idea of lumping together all the elements dropped in the elimination process and adding them to the diagonal of U can be used for *any* form of ILU factorization. In addition, there are variants of diagonal compensation in which only a fraction of the dropped elements are added to the diagonal. Thus, the term s_i in the above example would be replaced by ωs_i before being added to u_{ii}, where ω is typically between 0 and 1. Other strategies distribute the sum s_i among nonzero elements of L and U, other than the diagonal.

10.4 THRESHOLD STRATEGIES AND ILUT

Incomplete factorizations which rely on the levels of fill are blind to numerical values because elements that are dropped depend only on the structure of A. This can cause some difficulties for realistic problems that arise in many applications. A few alternative methods are available which are based on dropping elements in the Gaussian elimination process according to their magnitude rather than their locations. With these techniques, the zero pattern P is determined dynamically. The simplest way to obtain an incomplete factorization of this type is to take a sparse direct solver and modify it by adding lines of code which will ignore “small” elements. However, most direct solvers have a complex implementation which involves several layers of data structures that may make this approach ineffective. It is desirable to develop a strategy which is more akin to the ILU(0) approach. This section describes one such technique.

10.4.1 THE ILUT APPROACH

A generic ILU algorithm with threshold can be derived from the IKJversion of Gaussian elimination, Algorithm 10.2, by including a set of rules for dropping small elements. In what follows, *applying a dropping rule to an element* will only mean *replacing the element by zero if it satisfies a set of criteria*. A dropping rule can be applied to a whole row by applying the same rule to all the elements of the row. In the following algorithm, w is a full-length working row which is used to accumulate linear combinations of sparse rows in the elimination and w_k is the k-th entry of this row. As usual, a_{i*} denotes the i-th row of A.

ALGORITHM 10.6: ILUT

```
1.  For $i = 1, \ldots, n$ Do:
2.    $w := a_{i*}$
3.    For $k = 1, \ldots, i-1$ and when $w_k \neq 0$ Do:
4.        $w_k := w_k / a_{kk}$
5.        Apply a dropping rule to $w_k$
6.        If $w_k \neq 0$ then
7.            $w := w - w_k * u_{k*}$
8.        EndIf
9.    EndDo
10.   Apply a dropping rule to row $w$
11.   $l_{i,j} := w_j$ for $j = 1, \ldots, i-1$
12.   $u_{i,j} := w_j$ for $j = i, \ldots, n$
13.   $w := 0$
14. EndDo
```

Now consider the operations involved in the above algorithm. Line 7 is a sparse update operation. A common implementation of this is to use a full vector for w and a companion pointer which points to the positions of its nonzero elements. Similarly, lines 11 and 12 are sparse-vector copy operations. The vector w is filled with a few nonzero elements after the completion of each outer loop i, and therefore it is necessary to zero out those elements at the end of the Gaussian elimination loop as is done in line 13. This is a sparse *set-to-zero* operation.

ILU(0) can be viewed as a particular case of the above algorithm. The dropping rule for ILU(0) is to drop elements that are in positions not belonging to the original structure of the matrix.

In the factorization ILUT(p, τ), the following rule is used.

1. In line 5, an element w_k is dropped (i.e., replaced by zero) if it is less than the relative tolerance τ_i obtained by multiplying τ by the original norm of the i-th row (e.g., the 2-norm).
2. In line 10, a dropping rule of a different type is applied. First, drop again any element in the row with a magnitude that is below the relative tolerance τ_i. Then,

keep only the p largest elements in the L part of the row and the p largest elements in the U part of the row in addition to the diagonal element, which is always kept.

The goal of the second dropping step is to control the number of elements per row. Roughly speaking, p can be viewed as a parameter that helps control memory usage, while τ helps to reduce computational cost. There are several possible variations on the implementation of dropping step 2. For example we can keep a number of elements equal to $nu(i) + p$ in the upper part and $nl(i) + p$ in the lower part of the row, where $nl(i)$ and $nu(i)$ are the number of nonzero elements in the L part and the U part of the i-th row of A, respectively. This variant is adopted in the ILUT code used in the examples.

Note that no pivoting is performed. Partial (column) pivoting may be incorporated at little extra cost and will be discussed later. It is also possible to combine ILUT with one of the many standard reorderings, such as the ordering and the nested dissection ordering, or the reverse Cuthill-McKee ordering. Reordering in the context of incomplete factorizations can also be helpful for improving robustness, *provided enough accuracy is used*. For example, when a red-black ordering is used, ILU(0) may lead to poor performance compared with the natural ordering ILU(0). On the other hand, if ILUT is used by allowing gradually more fill-in, then the performance starts improving again. In fact, in some examples, the performance of ILUT for the red-black ordering *eventually outperforms* that of ILUT for the natural ordering using the same parameters p and τ.

10.4.2 ANALYSIS

Existence theorems for the ILUT factorization are similar to those of other incomplete factorizations. If the diagonal elements of the original matrix are positive while the off-diagonal elements are negative, then under certain conditions of diagonal dominance the matrices generated during the elimination will have the same property. If the original matrix is diagonally dominant, then the transformed matrices will also have the property of being diagonally dominant under certain conditions. These properties are analyzed in detail in this section.

The row vector w resulting from line 4 of Algorithm 10.6 will be denoted by $u_{i,*}^{k+1}$. Note that $u_{i,j}^{k+1} = 0$ for $j \le k$. Lines 3 to 10 in the algorithm involve a sequence of operations of the form

$$l_{ik} := u_{ik}^k / u_{kk} \tag{10.22}$$

$$\text{if } |l_{ik}| \quad \text{small enough set} \quad l_{ik} = 0$$

$$\text{else:}$$

$$u_{i,j}^{k+1} := u_{i,j}^k - l_{ik} u_{k,j} - r_{ij}^k \quad j = k+1, \ldots, n \tag{10.23}$$

for $k = 1, \ldots, i-1$, in which initially $u_{i,*}^1 := a_{i,*}$ and where r_{ij}^k is an element subtracted from a fill-in element which is being dropped. It should be equal either to zero (no dropping) or to $u_{ij}^k - l_{ik} u_{kj}$ when the element $u_{i,j}^{k+1}$ is being dropped. At the end of the i-th step of Gaussian elimination (outer loop in Algorithm 10.6), we obtain the i-th row of U,

$$u_{i,*} \equiv u_{i-1,*}^i \tag{10.24}$$

and the following relation is satisfied:

$$a_{i,*} = \sum_{k=1}^{i} l_{k,j} u_{i,*}^{k} + r_{i,*},$$

where $r_{i,*}$ is the row containing all the fill-ins.

The existence result which will be proved is valid only for certain modifications of the basic ILUT(p, τ) strategy. We consider an ILUT strategy which uses the following modification:

- **Drop Strategy Modification.** For any $i < n$, let a_{i,j_i} be the element of largest modulus among the elements $a_{i,j},\ j = i+1, \dots n$, in the original matrix. Then elements generated in position (i, j_i) during the ILUT procedure are not subject to the dropping rule.

This modification prevents elements generated in position (i, j_i) from ever being dropped. Of course, there are many alternative strategies that can lead to the same effect.

A matrix H whose entries h_{ij} satisfy the following three conditions:

$$h_{ii} > 0 \quad \text{for} \quad 1 \le i < n \quad \text{and} \quad h_{nn} \ge 0 \tag{10.25}$$

$$h_{ij} \le 0 \quad \text{for} \quad i, j = 1, \dots, n \quad \text{and} \quad i \ne j; \tag{10.26}$$

$$\sum_{j=i+1}^{n} h_{ij} < 0, \quad \text{for} \quad 1 \le i < n \tag{10.27}$$

will be referred to as an $\hat{M}$ matrix. The third condition is a requirement that there be at least one nonzero element to the right of the diagonal element, in each row except the last. The row sum for the i-th row is defined by

$$rs(h_{i,*}) = h_{i,*} e = \sum_{j=1}^{n} h_{i,j}.$$

A given row of an $\hat{M}$ matrix H is *diagonally dominant*, if its row sum is nonnegative. An $\hat{M}$ matrix H is said to be diagonally dominant if all its rows are diagonally dominant. The following theorem is an existence result for ILUT. The underlying assumption is that an ILUT strategy is used with the modification mentioned above.

THEOREM 10.3 *If the matrix A is a diagonally dominant $\hat{M}$ matrix, then the rows $u_{i,*}^{k}, k = 0, 1, 2, \dots, i$ defined by (10.23) starting with $u_{i,*}^{0} = 0$ and $u_{i,*}^{1} = a_{i,*}$ satisfy the following relations for $k = 1, \dots, l$*

$$u_{ij}^{k} \le 0 \quad j \ne i \tag{10.28}$$

$$rs(u_{i,*}^{k}) \ge rs(u_{i,*}^{k-1}) \ge 0, \tag{10.29}$$

$$u_{ii}^{k} > 0 \quad \textit{when} \quad i < n \quad \textit{and} \quad u_{nn}^{k} \ge 0. \tag{10.30}$$

Proof. The result can be proved by induction on k. It is trivially true for $k = 0$. To prove that the relation (10.28) is satisfied, start from the relation

$$u_{i,*}^{k+1} := u_{i,*}^{k} - l_{ik} u_{k,*} - r_{i*}^{k}$$

in which $l_{ik} \leq 0, u_{k,j} \leq 0$. Either r_{ij}^k is zero which yields $u_{ij}^{k+1} \leq u_{ij}^k \leq 0$, or r_{ij}^k is nonzero which means that u_{ij}^{k+1} is being dropped, i.e., replaced by zero, and therefore again $u_{ij}^{k+1} \leq 0$. This establishes (10.28). Note that by this argument $r_{ij}^k = 0$ except when the j-th element in the row is dropped, in which case $u_{ij}^{k+1} = 0$ and $r_{ij}^k = u_{ij}^k - l_{ik}u_{k,j} \leq 0$. Therefore, $r_{ij}^k \leq 0$, always. Moreover, when an element in position (i,j) is not dropped, then

$$u_{i,j}^{k+1} := u_{i,j}^k - l_{ik}u_{k,j} \leq u_{i,j}^k$$

and in particular by the rule in the modification of the basic scheme described above, for $i < n$, we will always have for $j = j_i$,

$$u_{i,j_i}^{k+1} \leq u_{i,j_i}^k \tag{10.31}$$

in which j_i is defined in the statement of the modification.

Consider the row sum of u_{i*}^{k+1}. We have

$$\begin{aligned} rs(u_{i,*}^{k+1}) &= rs(u_{i,*}^k) - l_{ik} \quad rs(u_{k,*}) - rs(r_{i*}^k) \\ &\geq rs(u_{i,*}^k) - l_{ik} \quad rs(u_{k,*}) & (10.32) \\ &\geq rs(u_{i,*}^k) & (10.33) \end{aligned}$$

which establishes (10.29) for $k+1$.

It remains to prove (10.30). From (10.29) we have, for $i < n$,

$$\begin{aligned} u_{ii}^{k+1} &\geq \sum_{j=k+1,n} - u_{i,j}^{k+1} = \sum_{j=k+1,n} |u_{i,j}^{k+1}| & (10.34) \\ &\geq |u_{i,j_i}^{k+1}| \geq |u_{i,j_i}^k| \geq \ldots & (10.35) \\ &\geq |u_{i,j_i}^1| = |a_{i,j_i}|. & (10.36) \end{aligned}$$

Note that the inequalities in (10.35) are true because u_{i,j_i}^k is never dropped by assumption and, as a result, (10.31) applies. By the condition (10.27), which defines $\hat{M}$ matrices, $|a_{i,j_i}|$ is positive for $i < n$. Clearly, when $i = n$, we have by (10.34) $u_{nn} \geq 0$. This completes the proof. ∎

The theorem does not mean that the factorization is effective only when its conditions are satisfied. In practice, the preconditioner is efficient under fairly general conditions.

10.4.3 IMPLEMENTATION DETAILS

A poor implementation of ILUT may well lead to an expensive factorization phase, and possibly an impractical algorithm. The following is a list of the potential difficulties that may cause inefficiencies in the implementation of ILUT.

1. Generation of the linear combination of rows of A (Line 7 in Algorithm 10.6).
2. Selection of the p largest elements in L and U.
3. Need to access the elements of L in increasing order of columns (in line 3 of Algorithm 10.6).

For (1), the usual technique is to generate a full row and accumulate the linear combination of the previous rows in it. The row is zeroed again after the whole loop is finished using a sparse set-to-zero operation. A variation on this technique uses only a full integer array $jr(1:n)$, the values of which are zero except when there is a nonzero element. With this full row, a short real vector $w(1:maxw)$ must be maintained which contains the real values of the row, as well as a corresponding short integer array $jw(1:maxw)$ which points to the column position of the real values in the row. When a nonzero element resides in position j of the row, then $jr(j)$ is set to the address k in w, jw where the nonzero element is stored. Thus, $jw(k)$ points to $jr(j)$, and $jr(j)$ points to $jw(k)$ and $w(k)$. This is illustrated in Figure 10.11.

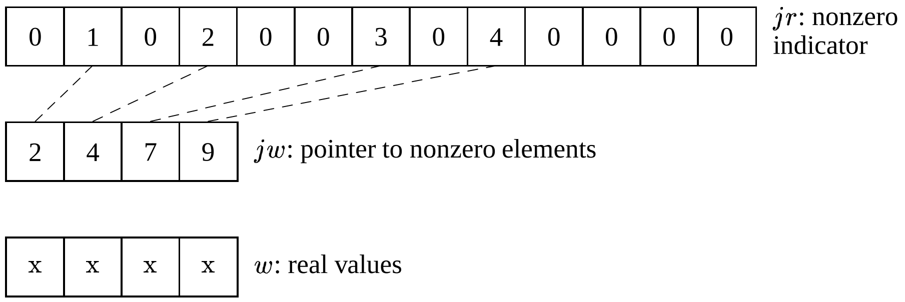


Figure 10.11 *Illustration of data structure used for the working row in ILUT.*

Note that jr holds the information on the row consisting of both the L part and the U part of the LU factorization. When the linear combinations of the rows are performed, first determine the pivot. Then, unless it is small enough to be dropped according to the dropping rule being used, proceed with the elimination. If a new element in the linear combination is not a fill-in, i.e., if $jr(j) = k \neq 0$, then update the real value $w(k)$. If it is a fill-in ($jr(j) = 0$), then append an element to the arrays w, jw and update jr accordingly.

For (2), the natural technique is to employ a heap-sort strategy. The cost of this implementation would be $O(m+p \times \log_2 m)$, i.e., $O(m)$ for the heap construction and $O(\log_2 m)$ for each extraction. Another implementation is to use a modified quick-sort strategy based on the fact that sorting the array is not necessary. Only the largest p elements must be extracted. This is a *quick-split* technique to distinguish it from the full quick-sort. The method consists of choosing an element, e.g., $x = w(1)$, in the array $w(1:m)$, then permuting the data so that $|w(k)| \leq |x|$ if $k \leq mid$ and $|w(k)| \geq |x|$ if $k \geq mid$, where mid is some split point. If $mid = p$, then exit. Otherwise, split *one of the left or right sub-arrays* recursively, depending on whether mid is smaller or larger than p. The cost of this strategy *on the average* is $O(m)$. The savings relative to the simpler bubble sort or insertion sort schemes are small for small values of p, but they become rather significant for large p and m.

The next implementation difficulty is that the elements in the L part of the row being built are not in an increasing order of columns. Since these elements must be accessed from left to right in the elimination process, all elements in the row after those already eliminated must be scanned. The one with smallest column number is then picked as the next element to eliminate. This operation can be efficiently organized as a binary search tree which allows easy insertions and searches. However, this improvement is rather complex to implement and is likely to yield moderate gains.

Example 10.4 Tables 10.3 and 10.4 show the results of applying GMRES(10) preconditioned with ILUT($1, 10^{-4}$) and ILUT($5, 10^{-4}$), respectively, to the five test problems described in Section 3.7. See Example 6.1 for the meaning of the column headers in the table. As shown, all linear systems are now solved in a relatively small number of iterations, with the exception of F2DB which still takes 130 steps to converge with *lfil = 1* (but only 10 with *lfil = 5*.) In addition, observe a marked improvement in the operation count and error norms. Note that the operation counts shown in the column Kflops do not account for the operations required in the set-up phase to build the preconditioners. For large values of *lfil* , this may be large.

Matrix	Iters	Kflops	Residual	Error
F2DA	18	964	0.47E-03	0.41E-04
F3D	14	3414	0.11E-02	0.39E-03
ORS	6	341	0.13E+00	0.60E-04
F2DB	130	7167	0.45E-02	0.51E-03
FID	59	19112	0.19E+00	0.11E-03

Table 10.3 *A test run of GMRES(10)-ILUT(1*$, 10^{-4}$*) preconditioning.*

If the total time to solve one linear system with A is considered, a typical curve of the total time required to solve a linear system when the *lfil* parameter varies would look like the plot shown in Figure 10.12. As *lfil* increases, a critical value is reached where the preprocessing time and the iteration time are equal. Beyond this critical point, the preprocessing time dominates the total time. If there are several linear systems to solve with the same matrix A, then it is advantageous to use a more accurate factorization, since the cost of the factorization will be amortized. Otherwise, a smaller value of *lfil* will be more efficient.

Matrix	Iters	Kflops	Residual	Error
F2DA	7	478	0.13E-02	0.90E-04
F3D	9	2855	0.58E-03	0.35E-03
ORS	4	270	0.92E-01	0.43E-04
F2DB	10	724	0.62E-03	0.26E-03
FID	40	14862	0.11E+00	0.11E-03

Table 10.4 *A test run of GMRES(10)-ILUT(5*$, 10^{-4}$*) preconditioning.*

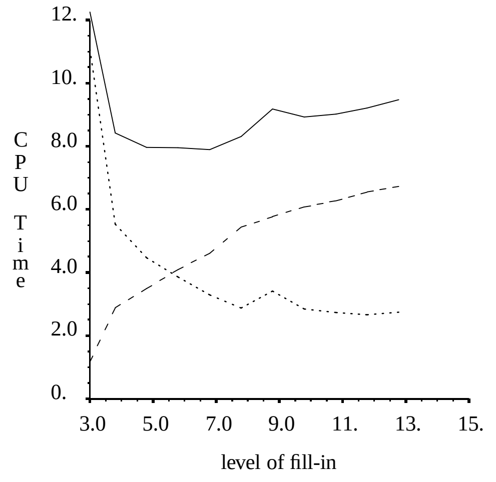


Figure 10.12 *Typical CPU time as a function of lfil The dashed line is the ILUT time, the dotted line is the GMRES time, and the solid line shows the total.*

10.4.4 THE ILUTP APPROACH

The ILUT approach may fail for many of the matrices that arise from real applications, for one of the following reasons.

1. The ILUT procedure encounters a zero pivot;
2. The ILUT procedure encounters an overflow or underflow condition, because of an exponential growth of the entries of the factors;
3. The ILUT preconditioner terminates normally but the incomplete factorization preconditioner which is computed is *unstable*.

An unstable ILU factorization is one for which $M^{-1} = U^{-1}L^{-1}$ has a very large norm leading to poor convergence or divergence of the outer iteration. The case (1) can be overcome to a certain degree by assigning an arbitrary nonzero value to a zero diagonal element that is encountered. Clearly, this is not a satisfactory remedy because of the loss in accuracy in the preconditioner. The ideal solution in this case is to use pivoting. However, a form of pivoting is desired which leads to an algorithm with similar cost and complexity to ILUT. Because of the data structure used in ILUT, row pivoting is not practical. Instead, column pivoting can be implemented rather easily.

Here are a few of the features that characterize the new algorithm which is termed ILUTP ("P" stands for pivoting). ILUTP uses a permutation array $perm$ to hold the new orderings of the variables, along with the reverse permutation array. At step i of the elimination process the largest entry in a row is selected and is defined to be the new i-th variable. The two permutation arrays are then updated accordingly. The matrix elements of L and U are kept in their original numbering. However, when expanding the L-U row which corresponds to the i-th outer step of Gaussian elimination, the elements are loaded with respect to the new labeling, using the array $perm$ for the translation. At the end of the process, there are two options. The first is to leave all elements labeled with respect to the original labeling. No additional work is required since the variables are already in this form in the algorithm, but the variables must then be permuted at each preconditioning step. The second solution is to apply the permutation to all elements of A as well as L/U. This does not require applying a permutation at each step, but rather produces a permuted solution which must be permuted back at the end of the iteration phase. The complexity of the ILUTP procedure is virtually identical to that of ILUT. A few additional options can be provided. A tolerance parameter called $permtol$ may be included to help determine whether or not to permute variables: A nondiagonal element a_{ij} is candidate for a permutation only when $tol \times |a_{ij}| > |a_{ii}|$. Furthermore, pivoting may be restricted to take place only within diagonal blocks of a fixed size. The size $mbloc$ of these blocks must be provided. A value of $mbloc \geq n$ indicates that there are no restrictions on the pivoting.

For difficult matrices, the following strategy seems to work well:

1. Always apply a scaling to all the rows (or columns) e.g., so that their 1-norms are all equal to 1; then apply a scaling of the columns (or rows).
2. Use a small drop tolerance (e.g., $\epsilon = 10^{-4}$ or $\epsilon = 10^{-5}$).
3. Take a large fill-in parameter (e.g., $lfil = 20$).
4. Do not take a small value for $permtol$. Reasonable values are between 0.5 and 0.01, with 0.5 being the best in many cases.
5. Take $mbloc = n$ unless there are reasons why a given block size is justifiable.

Example 10.5 Table 10.5 shows the results of applying the GMRES algorithm with ILUTP($1, 10^{-4}$) preconditioning to the five test problems described in Section 3.7. The *permtol* parameter is set to 1.0 in this case.

Matrix	Iters	Kflops	Residual	Error
F2DA	18	964	0.47E-03	0.41E-04
F3D	14	3414	0.11E-02	0.39E-03
ORS	6	341	0.13E+00	0.61E-04
F2DB	130	7167	0.45E-02	0.51E-03
FID	50	16224	0.17E+00	0.18E-03

Table 10.5 *A test run of GMRES with ILUTP(1) preconditioning.*

See Example 6.1 for the meaning of the column headers in the table. The results are identical with those of ILUT($1, 10^{-4}$) shown in Table 10.3, for the first four problems, but there is an improvement for the fifth problem.

10.4.5 THE ILUS APPROACH

The ILU preconditioners discussed so far are based mainly on the the IKJvariant of Gaussian elimination. Different types of ILUs can be derived using other forms of Gaussian elimination. The main motivation for the version to be described next is that ILUT does not take advantage of symmetry. If A is symmetric, then the resulting $M = LU$ is nonsymmetric in general. Another motivation is that in many applications including computational fluid dynamics and structural engineering, the resulting matrices are stored in a *sparse skyline* (SSK) format rather than the standard Compressed Sparse Row format.

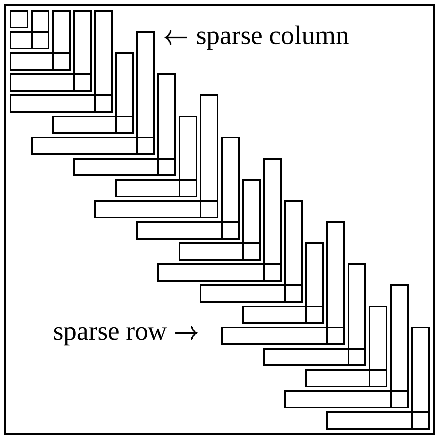


Figure 10.13 *Illustration of the sparse skyline format.*

In this format, the matrix A is decomposed as

$$A = D + L_1 + L_2^T$$

in which D is a diagonal of A and L_1, L_2 are strictly lower triangular matrices. Then a sparse representation of L_1 and L_2 is used in which, typically, L_1 and L_2 are stored in the CSR format and D is stored separately.

Incomplete Factorization techniques may be developed for matrices in this format without having to convert them into the CSR format. Two notable advantages of this approach are (1) the savings in storage for structurally symmetric matrices, and (2) the fact that the algorithm gives a symmetric preconditioner when the original matrix is symmetric.

Consider the sequence of matrices

$$A_{k+1} = \begin{pmatrix} A_k & v_k \\ w_k & \alpha_{k+1} \end{pmatrix},$$

where $A_n = A$. If A_k is nonsingular and its LDU factorization

$$A_k = L_k D_k U_k$$

is already available, then the LDU factorization of A_{k+1} is

$$A_{k+1} = \begin{pmatrix} L_k & 0 \\ y_k & 1 \end{pmatrix} \begin{pmatrix} D_k & 0 \\ 0 & d_{k+1} \end{pmatrix} \begin{pmatrix} U_k & z_k \\ 0 & 1 \end{pmatrix}$$

in which

$$z_k = D_k^{-1} L_k^{-1} v_k \tag{10.37}$$

$$y_k = w_k U_k^{-1} D_k^{-1} \tag{10.38}$$

$$d_{k+1} = \alpha_{k+1} - y_k D_k z_k. \tag{10.39}$$

Hence, the last row/column pairs of the factorization can be obtained by solving two unit lower triangular systems and computing a scaled dot product. This can be exploited for sparse matrices provided an appropriate data structure is used to take advantage of the sparsity of the matrices L_k, U_k as well as the vectors v_k, w_k, y_k, and z_k. A convenient data structure for this is to store the rows/columns pairs w_k, v_k^T as a single row in sparse mode. All these pairs are stored in sequence. The diagonal elements are stored separately. This is called the Unsymmetric Sparse Skyline (USS) format. Each step of the ILU factorization based on this approach will consist of two approximate sparse linear system solutions and a sparse dot product. The question that arises is: How can a sparse triangular system be solved inexpensively? It would seem natural to solve the triangular systems (10.37) and (10.38) exactly and then drop small terms at the end, using a numerical dropping strategy. However, the total cost of computing the ILU factorization with this strategy would be $O(n^2)$ operations at least, which is not acceptable for very large problems. Since only an approximate solution is required, the first idea that comes to mind is the truncated Neumann series,

$$z_k = D_k^{-1} L_k^{-1} v_k = D_k^{-1}(I + E_k + E_k^2 + \ldots + E_k^p) v_k \tag{10.40}$$

in which $E_k \equiv I - L_k$. In fact, by analogy with ILU(p), it is interesting to note that the powers of E_k will also tend to become smaller as p increases. A close look at the structure of $E_k^p v_k$ shows that there is indeed a strong relation between this approach and ILU(p) in the symmetric case. Now we make another important observation, namely, that the vector $E_k^j v_k$ can be computed in *sparse-sparse mode*, i.e., in terms of operations involving products of *sparse matrices by sparse vectors*. Without exploiting this, the total cost would still be $O(n^2)$. When multiplying a sparse matrix A by a sparse vector v, the operation can best be done by accumulating the linear combinations of the columns of A. A sketch of the resulting ILUS algorithm is as follows.

ALGORITHM 10.7: ILUS(ϵ, p)

```
1. Set $A_1 = D_1 = a_{11}$, $L_1 = U_1 = 1$
2. For $i = 1, \ldots, n-1$ Do:
3.     Compute $z_k$ by (10.40) in sparse-sparse mode
4.     Compute $y_k$ in a similar way
5.     Apply numerical dropping to $y_k$ and $z_k$
6.     Compute $d_{k+1}$ via (10.39)
7. EndDo
```

If there are only i nonzero components in the vector v and an average of ν nonzero elements per column, then the total cost per step will be $2 \times i \times \nu$ on the average. Note that the computation of d_k via (10.39) involves the inner product of two sparse vectors which is often implemented by expanding one of the vectors into a full vector and computing the inner product of a sparse vector by this full vector. As mentioned before, in the symmetric case ILUS yields the Incomplete Cholesky factorization. Here, the work can be halved since the generation of y_k is not necessary.

Also note that a simple iterative procedure such as MR or GMRES(m) can be used to solve the triangular systems in sparse-sparse mode. Similar techniques will be seen in Section 10.5. Experience shows that these alternatives are not much better than the Neumann series approach [53].

10.5 APPROXIMATE INVERSE PRECONDITIONERS

The Incomplete LU factorization techniques were developed originally for M-matrices which arise from the discretization of Partial Differential Equations of elliptic type, usually in one variable. For the common situation where A is indefinite, standard ILU factorizations may face several difficulties, and the best known is the fatal breakdown due to the encounter of a zero pivot. However, there are other problems that are just as serious. Consider an incomplete factorization of the form

$$A = LU + E \tag{10.41}$$

where E is the error. The preconditioned matrices associated with the different forms of preconditioning are similar to

$$L^{-1}AU^{-1} = I + L^{-1}EU^{-1}. \tag{10.42}$$

What is sometimes missed is the fact that the error matrix E in (10.41) is not as important as the "preconditioned" error matrix $L^{-1}EU^{-1}$ shown in (10.42) above. When the matrix A is diagonally dominant, then L and U are well conditioned, and the size of $L^{-1}EU^{-1}$ remains confined within reasonable limits, typically with a nice clustering of its eigenvalues around the origin. On the other hand, when the original matrix is not diagonally dominant, L^{-1} or U^{-1} may have very large norms, causing the error $L^{-1}EU^{-1}$ to be very large and thus adding large perturbations to the identity matrix. It can be observed experimentally that ILU preconditioners can be very poor in these situations which often arise when the matrices are indefinite, or have large nonsymmetric parts.

One possible remedy is to try to find a preconditioner that does not require solving a linear system. For example, the original system can be preconditioned by a matrix M which is a direct approximation to the inverse of A.

10.5.1 APPROXIMATING THE INVERSE OF A SPARSE MATRIX

A simple technique for finding approximate inverses of arbitrary sparse matrices is to attempt to find a sparse matrix M which minimizes the Frobenius norm of the residual matrix $I - AM$,

$$F(M) = \|I - AM\|_F^2. \tag{10.43}$$

A matrix M whose value $F(M)$ is small would be a right-approximate inverse of A. Similarly, a left-approximate inverse can be defined by using the objective function

$$||I - MA||_F^2. \tag{10.44}$$

Finally, a left-right pair L, U can be sought to minimize

$$||I - LAU||_F^2. \tag{10.45}$$

In the following, only (10.43) and(10.45) are considered. The case (10.44) is very similar to the right preconditioner case (10.43). The objective function (10.43) decouples into the sum of the squares of the 2-norms of the individual columns of the residual matrix $I - AM$,

$$F(M) = ||I - AM||_F^2 = \sum_{j=1}^{n} ||e_j - Am_j||_2^2 \tag{10.46}$$

in which e_j and m_j are the j-th columns of the identity matrix and of the matrix M, respectively. There are two different ways to proceed in order to minimize (10.46). The function (10.43) can be minimized globally as a function of the sparse matrix M, e.g., by a gradient-type method. Alternatively, the individual functions

$$f_j(m) = ||e_j - Am||_2^2, \quad j = 1, 2, \ldots, n \tag{10.47}$$

can be minimized. The second approach is appealing for parallel computers, although there is also parallelism to be exploited in the first approach. These two approaches will be discussed in turn.

10.5.2 GLOBAL ITERATION

The *global iteration* approach consists of treating M as an unknown sparse matrix and using a descent-type method to minimize the objective function (10.43). This function is a quadratic function on the space of $n \times n$ matrices, viewed as objects in $\mathbb{R}^{n^2}$. The proper inner product on the space of matrices, to which the squared norm (10.46) is associated, is

$$\langle X, Y \rangle = \text{tr}(Y^T X). \tag{10.48}$$

In the following, an *array representation* of an n^2 vector X means the $n \times n$ matrix whose column vectors are the successive n-vectors of X.

In a descent algorithm, a new iterate M_{new} is defined by taking a step along a selected direction G, i.e.,

$$M_{new} = M + \alpha G$$

in which α is selected to minimize the objective function $F(M_{new})$. From results seen in Chapter 5, minimizing the residual norm is equivalent to imposing the condition that $R - \alpha AG$ be orthogonal to AG with respect to the $\langle \cdot, \cdot \rangle$ inner product. Thus, the optimal α is given by

$$\alpha = \frac{\langle R, AG \rangle}{\langle AG, AG \rangle} = \frac{\text{tr}(R^T AG)}{\text{tr}\left((AG)^T AG\right)}. \tag{10.49}$$

The denominator may be computed as $\|AG\|_F^2$. The resulting matrix M will tend to become denser after each descent step and it is therefore essential to apply a numerical dropping strategy to the resulting M. However, the descent property of the step is now lost, i.e., it is no longer guaranteed that $F(M_{new}) \leq F(M)$. An alternative would be to apply numerical dropping to the direction of search G before taking the descent step. In this case, the amount of fill-in in the matrix M cannot be controlled.

The simplest choice for the descent direction G is to take it to be equal to the residual matrix $R = I - AM$, where M is the new iterate. Except for the numerical dropping step, the corresponding descent algorithm is nothing but the Minimal Residual (MR) algorithm, seen in Section 5.3.2, on the $n^2 \times n^2$ linear system $AM = I$. The global Minimal Residual algorithm will have the following form.

ALGORITHM 10.8: Global Minimal Residual Descent Algorithm

```
1. Select an initial $M$
2. Until convergence Do:
3.    Compute $C := AM$ and $G := I - C$
4.    Compute $\alpha = \mathrm{tr}(G^T AG)/\|C\|_F^2$
5.    Compute $M := M + \alpha G$
6.    Apply numerical dropping to $M$
7. EndDo
```

A second choice is to take G to be equal to the direction of steepest descent, i.e., the direction opposite to the gradient of the function (10.43) with respect to M. If all vectors as represented as 2-dimensional $n \times n$ arrays, then the gradient can be viewed as a matrix G, which satisfies the following relation for small perturbations E,

$$F(M + E) = F(M) + \langle G, E \rangle + o(\|E\|). \tag{10.50}$$

This provides a way of expressing the gradient as an operator on arrays, rather than n^2 vectors.

PROPOSITION 10.3 *The array representation of the gradient of F with respect to M is the matrix*

$$G = -2A^T R$$

in which R is the residual matrix $R = I - AM$.

Proof. For any matrix E we have

$$\begin{aligned} F(M+E) - F(M) &= \mathrm{tr}\left[(I - A(M+E))^T (I - A(M+E))\right] \\ &\qquad - \mathrm{tr}\left[(I - A(M)^T (I - A(M)\right] \\ &= \mathrm{tr}\left[(R - AE)^T (R - AE) - R^T R\right] \\ &= -\mathrm{tr}\left[(AE)^T R + R^T AE - (AE)^T (AE)\right] \\ &= -2\mathrm{tr}(R^T AE) + \mathrm{tr}\left[(AE)^T (AE)\right] \\ &= -2\left\langle A^T R, E\right\rangle + \langle AE, AE \rangle . \end{aligned}$$

Comparing this with (10.50) yields the desired result. ■

Thus, the steepest descent algorithm will consist of replacing G in line 3 of Algorithm 10.8 by $G = A^T R = A^T(I - AM)$. As is expected with steepest descent techniques, the algorithm can be quite slow.

ALGORITHM 10.9: Global Steepest Descent Algorithm

```
1. Select an initial $M$
2. Until convergence Do:
3.    Compute $R = I - AM$, and $G := A^T R$ ;
4.    Compute $\alpha = \|G\|_F^2 / \|AG\|_F^2$
5.    Compute $M := M + \alpha G$
6.    Apply numerical dropping to $M$
7. EndDo
```

In either steepest descent or minimal residual, the G matrix must be stored explicitly. The scalars $\|AG\|_F^2$ and $\mathrm{tr}(G^T AG)$ needed to obtain α in these algorithms can be computed from the successive columns of AG, which can be generated, used, and discarded. As a result, the matrix AG need not be stored.

10.5.3 COLUMN-ORIENTED ALGORITHMS

Column-oriented algorithms consist of minimizing the individual objective functions (10.47) separately. Each minimization can be performed by taking a sparse initial guess and solving approximately the n parallel linear subproblems

$$Am_j = e_j, \qquad j = 1, 2, \ldots, n \tag{10.51}$$

with a few steps of a nonsymmetric descent-type method, such as MR or GMRES. If these linear systems were solved (approximately) without taking advantage of sparsity, the cost of constructing the preconditioner would be of order n^2. That is because each of the n columns would require $O(n)$ operations. Such a cost would become unacceptable for large linear systems. To avoid this, the iterations must be performed in *sparse-sparse mode*, a term which was already introduced in Section 10.4.5. The column m_j and the subsequent iterates in the MR algorithm must be stored and operated on as sparse vectors. The Arnoldi basis in the GMRES algorithm are now to be kept in sparse format. Inner products and vector updates involve pairs of sparse vectors.

In the following MR algorithm, n_i iterations are used to solve (10.51) approximately for each column, giving an approximation to the j-th column of the inverse of A. Each initial m_j is taken from the columns of an initial guess, M_0.

ALGORITHM 10.10: Approximate Inverse via MR Iteration

```
1.  Start: set $M = M_0$
2.  For each column $j = 1, \ldots, n$ Do:
3.      Define $m_j = Me_j$
4.      For $i = 1, \ldots, n_i$ Do:
5.          $r_j := e_j - Am_j$
6.          $\alpha_j := \frac{(r_j, Ar_j)}{(Ar_j, Ar_j)}$
7.          $m_j := m_j + \alpha_j r_j$
8.          Apply numerical dropping to $m_j$
9.      EndDo
10. EndDo
```

The algorithm computes the current residual r_j and then minimizes the residual norm $\|e_j - A(m_j + \alpha r_j)\|_2$, with respect to α. The resulting column is then pruned by applying the numerical dropping step in line 8.

In the sparse implementation of MR and GMRES, the matrix-vector product, SAXPY, and dot product kernels now all involve sparse vectors. The matrix-vector product is much more efficient if the sparse matrix is stored by columns, since all the entries do not need to be traversed. Efficient codes for all these kernels may be constructed which utilize a full n-length work vector.

Columns from an initial guess M_0 for the approximate inverse are used as the initial guesses for the iterative solution of the linear subproblems. There are two obvious choices: $M_0 = \alpha I$ and $M_0 = \alpha A^T$. The scale factor α is chosen to minimize the norm of $I - AM_0$. Thus, the initial guess is of the form $M_0 = \alpha G$ where G is either the identity or A^T. The optimal α can be computed using the formula (10.49), in which R is to be replaced by the identity, so $\alpha = \mathrm{tr}(AG)/\mathrm{tr}(AG(AG)^T)$. The identity initial guess is less expensive to use but $M_0 = \alpha A^T$ is sometimes a much better initial guess. For this choice, the initial preconditioned system AM_0 is SPD.

The linear systems needed to solve when generating each column of the approximate inverse may themselves be preconditioned with the most recent version of the preconditioning matrix M. Thus, each system (10.51) for approximating column j may be preconditioned with M_0' where the first $j-1$ columns of M_0' are the m_k that already have been computed, $1 \leq k < j$, and the remaining columns are the initial guesses for the m_k, $j \leq k \leq n$. Thus, *outer* iterations can be defined which sweep over the matrix, as well as *inner* iterations which compute each column. At each outer iteration, the initial guess for each column is taken to be the previous result for that column.

10.5.4 THEORETICAL CONSIDERATIONS

The first theoretical question which arises is whether or not the approximate inverses obtained by the approximations described earlier can be singular. It cannot be proved that M is nonsingular unless the approximation is accurate enough. This requirement may be in conflict with the requirement of keeping the approximation sparse.

PROPOSITION 10.4 *Assume that A is nonsingular and that the residual of the approximate inverse M satisfies the relation*

$$\|I - AM\| < 1 \tag{10.52}$$

where $\|.\|$ is any consistent matrix norm. Then M is nonsingular.

Proof. The result follows immediately from the equality

$$AM = I - (I - AM) \equiv I - N. \tag{10.53}$$

Since $\|N\| < 1$, Theorem 1.5 seen in Chapter 1 implies that $I - N$ is nonsingular. ■

The result is true in particular for the Frobenius norm which is consistent (see Chapter 1).

It may sometimes be the case that AM is poorly balanced and as a result R can be large. Then balancing AM can yield a smaller norm and possibly a less restrictive condition for the nonsingularity of M. It is easy to extend the previous result as follows. If A is nonsingular and two nonsingular diagonal matrices D_1, D_2 exist such that

$$\|I - D_1 AMD_2\| < 1 \tag{10.54}$$

where $\|.\|$ is any consistent matrix norm, then M is nonsingular.

Each column is obtained independently by requiring a condition on the residual norm of the form

$$\|e_j - Am_j\| \le \tau, \tag{10.55}$$

for some vector norm $\|.\|$. From a practical point of view the 2-norm is preferable since it is related to the objective function which is used, namely, the Frobenius norm of the residual $I - AM$. However, the 1-norm is of particular interest since it leads to a number of simple theoretical results. In the following, it is assumed that a condition of the form

$$\|e_j - Am_j\|_1 \le \tau_j \tag{10.56}$$

is required for each column.

The above proposition does not reveal anything about the degree of sparsity of the resulting approximate inverse M. It may well be the case that in order to guarantee nonsingularity, M must be dense, or nearly dense. In fact, in the particular case where the norm in the proposition is the 1-norm, it is known that the approximate inverse may be *structurally dense*, in that it is always possible to find a sparse matrix A for which M will be dense if $\|I - AM\|_1 < 1$.

Next, we examine the sparsity of M and prove a simple result for the case where an assumption of the form (10.56) is made.

PROPOSITION 10.5 *Let $B = A^{-1}$ and assume that a given element b_{ij} of B satisfies the inequality*

$$|b_{ij}| > \tau_j \max_{k=1,n} |b_{ik}|, \tag{10.57}$$

then the element m_{ij} is nonzero.

Proof. From the equality $AM = I - R$ we have $M = A^{-1} - A^{-1}R$, and hence

$$m_{ij} = b_{ij} - \sum_{k=1}^{n} b_{ik} r_{kj}.$$

Therefore,

$$\begin{aligned} |m_{ij}| &\geq |b_{ij}| - \sum_{k=1}^{n} |b_{ik} r_{kj}| \\ &\geq |b_{ij}| - \max_{k=1,n} |b_{ik}| \, \|r_j\|_1 \\ &\geq |b_{ij}| - \max_{k=1,n} |b_{ik}| \tau_j. \end{aligned}$$

Now the condition (10.57) implies that $|m_{ij}| > 0$. ■

The proposition implies that if R is small enough, then the nonzero elements of M are located in positions corresponding to the larger elements in the inverse of A. The following negative result is an immediate corollary.

COROLLARY 10.1 *Let $\tau = \max_{j=1,\ldots,n} \tau_j$. If the nonzero elements of $B = A^{-1}$ are τ-equimodular in that*

$$|b_{ij}| \; > \; \tau \max_{k=1,n,\; l=1,n} |b_{lk}|,$$

then the nonzero sparsity pattern of M includes the nonzero sparsity pattern of A^{-1}. In particular, if A^{-1} is dense and its elements are τ-equimodular, then M is also dense.

The smaller the value of τ, the more likely the condition of the corollary will be satisfied. Another way of stating the corollary is that *accurate* and *sparse* approximate inverses may be computed only if the elements of the actual inverse have variations in size. Unfortunately, this is difficult to verify in advance and it is known to be true only for certain types of matrices.

10.5.5 CONVERGENCE OF SELF PRECONDITIONED MR

We now examine the convergence of the MR algorithm in the case where self preconditioning is used, but no numerical dropping is applied. The column-oriented algorithm is considered first. Let M be the current approximate inverse at a given substep. The self preconditioned MR iteration for computing the j-th column of the next approximate inverse is obtained by the following sequence of operations:

```
1.   $r_j := e_j - Am_j = e_j - AMe_j$
2.   $t_j := Mr_j$
3.   $\alpha_j := \frac{(r_j, At_j)}{(At_j, At_j)}$
4.   $m_j := m_j + \alpha_j t_j$ .
```

Note that α_j can be written as

$$\alpha_j = \frac{(r_j, AMr_j)}{(AMr_j, AMr_j)} \equiv \frac{(r_j, Cr_j)}{(Cr_j, Cr_j)}$$

where

$$C = AM$$

is the preconditioned matrix at the given substep. The subscript j is now dropped to simplify the notation. The new residual associated with the current column is given by

$$r^{new} = r - \alpha At = r - \alpha AMr \equiv r - \alpha Cr.$$

The orthogonality of the new residual against AMr can be used to obtain

$$\|r^{new}\|_2^2 = \|r\|_2^2 - \alpha^2 \|Cr\|_2^2.$$

Replacing α by its value defined above we get

$$\|r^{new}\|_2^2 = \|r\|_2^2 \left[1 - \left(\frac{(Cr, r)}{\|Cr\|_2 \|r\|_2} \right)^2 \right].$$

Thus, at each inner iteration, the residual norm for the j-th column is reduced according to the formula

$$\|r^{new}\|_2 = \|r\|_2 \sin \angle(r, Cr) \tag{10.58}$$

in which $\angle(u, v)$ denotes the acute angle between the vectors u and v. Assume that each column converges. Then, the preconditioned matrix C converges to the identity. As a result of this, the angle $\angle(r, Cr)$ will tend to $\angle(r, r) = 0$, and therefore the convergence ratio $\sin \angle(r, Cr)$ will also tend to zero, showing superlinear convergence.

Now consider equation (10.58) more carefully. Denote by R the residual matrix $R = I - AM$ and observe that

$$\begin{aligned}\sin \angle(r, Cr) &= \min_{\alpha} \frac{\|r - \alpha\, Cr\|_2}{\|r\|_2} \\ &\le \frac{\|r - Cr\|_2}{\|r\|_2} \equiv \frac{\|Rr\|_2}{\|r\|_2} \\ &\le \|R\|_2.\end{aligned}$$

This results in the following statement.

PROPOSITION 10.6 *Assume that the self preconditioned MR algorithm is employed with one inner step per iteration and no numerical dropping. Then the 2-norm of each residual $e_j - Am_j$ of the j-th column is reduced by a factor of at least $\|I - AM\|_2$, where M is the approximate inverse before the current step, i.e.,*

$$\|r_j^{new}\|_2 \le \|I - AM\|_2 \, \|r_j\|_2. \tag{10.59}$$

In addition, the residual matrices $R_k = I - AM_k$ obtained after each outer iteration satisfy

$$\|R_{k+1}\|_F \le \|R_k\|_F^2. \tag{10.60}$$

As a result, when the algorithm converges, it does so quadratically.

Proof. Inequality (10.59) was proved above. To prove quadratic convergence, first use the inequality $\|X\|_2 \le \|X\|_F$ and (10.59) to obtain

$$\|r_j^{new}\|_2 \;\le\; \|R_{k,j}\|_F \; \|r_j\|_2 .$$

Here, the k index corresponds to the outer iteration and the j-index to the column. Note that the Frobenius norm is reduced for each of the inner steps corresponding to the columns, and therefore,

$$\|R_{k,j}\|_F \le \|R_k\|_F .$$

This yields

$$\|r_j^{new}\|_2^2 \;\le\; \|R_k\|_F^2 \; \|r_j\|_2^2$$

which, upon summation over j, gives

$$\|R_{k+1}\|_F \le \|R_k\|_F^2 .$$

This completes the proof. ■

Note that the above theorem does not prove convergence. It only states that when the algorithm converges, it does so quadratically at the limit. In addition, the result ceases to be valid in the presence of dropping.

Consider now the case of the global iteration. When self preconditioning is incorporated into the global MR algorithm (Algorithm 10.8), the search direction becomes $Z_k = M_k R_k$, where R_k is the current residual matrix. Then, the main steps of the algorithm (without dropping) are as follows.

1. $R_k := I - AM_k$
2. $Z_k := M_k R_k$
3. $\alpha_k := \frac{\langle R_k, AZ_k \rangle}{\langle AZ_k, AZ_k \rangle}$
4. $M_{k+1} := M_k + \alpha_k Z_k$

At each step the new residual matrix R_{k+1} satisfies the relation

$$R_{k+1} = I - AM_{k+1} = I - A(M_k + \alpha_k Z_k) = R_k - \alpha_k A Z_k .$$

An important observation is that R_k is a polynomial in R_0. This is because, from the above relation,

$$R_{k+1} = R_k - \alpha_k A M_k R_k = R_k - \alpha_k (I - R_k) R_k = (1 - \alpha_k) R_k + \alpha_k R_k^2 . \qquad (10.61)$$

Therefore, induction shows that $R_{k+1} = p_{2^k}(R_0)$ where p_j is a polynomial of degree j. Now define the preconditioned matrices,

$$B_k \equiv AM_k = I - R_k . \qquad (10.62)$$

Then, the following recurrence follows from (10.61),

$$B_{k+1} = B_k + \alpha_k B_k (I - B_k) \qquad (10.63)$$

and shows that B_{k+1} is also a polynomial of degree 2^k in B_0. In particular, *if the initial B_0 is symmetric, then so are all subsequent B_k's*. This is achieved when the initial M is a multiple of A^T, namely if $M_0 = \alpha_0 A^T$.

Similar to the column oriented case, when the algorithm converges it does so quadratically.

PROPOSITION 10.7 *Assume that the self preconditioned global MR algorithm is used without dropping. Then, the residual matrices obtained at each iteration satisfy*

$$\|R_{k+1}\|_F \leq \|R_k^2\|_F. \tag{10.64}$$

As a result, when the algorithm converges, then it does so quadratically.

Proof. Define for any α,

$$R(\alpha) = (1-\alpha)R_k + \alpha R_k^2$$

Recall that α_k achieves the minimum of $\|R(\alpha)\|_F$ over all α's. In particular,

$$\begin{aligned} \|R_{k+1}\|_F &= \min_\alpha \|R(\alpha)\|_F \\ &\leq \|R(1)\|_F = \|R_k^2\|_F \\ &\leq \|R_k\|_F^2. \end{aligned} \tag{10.65}$$

This proves quadratic convergence at the limit. ■

For further properties see Exercise 16.

10.5.6 FACTORED APPROXIMATE INVERSES

A notable disadvantage of the right or left preconditioning approach method is that it is difficult to assess in advance whether or not the resulting approximate inverse M is nonsingular. An alternative would be to seek a two-sided approximation, i.e., a pair L, U, with L lower triangular and U upper triangular, which attempts to minimize the objective function (10.45). The techniques developed in the previous sections can be exploited for this purpose.

In the factored approach, two matrices L and U which are *unit* lower and upper triangular matrices are sought such that

$$LAU \approx D$$

where D is some unknown diagonal matrix. When D is nonsingular and $LAU = D$, then L, U are called *inverse LU factors* of A since in this case $A^{-1} = UD^{-1}L$. Once more, the matrices are built one column or row at a time. Assume as in Section 10.4.5 that we have the sequence of matrices

$$A_{k+1} = \begin{pmatrix} A_k & v_k \\ w_k & \alpha_{k+1} \end{pmatrix}$$

in which $A_n \equiv A$. If the inverse factors L_k, U_k are available for A_k, i.e.,

$$L_k A_k U_k = D_k,$$

then the inverse factors L_{k+1}, U_{k+1} for A_{k+1} are easily obtained by writing

$$\begin{pmatrix} L_k & 0 \\ -y_k & 1 \end{pmatrix} \begin{pmatrix} A_k & v_k \\ w_k & \alpha_{k+1} \end{pmatrix} \begin{pmatrix} U_k & -z_k \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} D_k & 0 \\ 0 & \delta_{k+1} \end{pmatrix} \tag{10.66}$$

in which z_k, y_k, and δ_{k+1} are such that

$$A_k z_k = v_k \tag{10.67}$$

$$y_k A_k = w_k \tag{10.68}$$

$$\delta_{k+1} = \alpha_{k+1} - w_k z_k = \alpha_{k+1} - y_k v_k . \tag{10.69}$$

Note that the formula (10.69) exploits the fact that either the system (10.67) is solved exactly (middle expression) or the system (10.68) is solved exactly (second expression) or both systems are solved exactly (either expression). In the realistic situation where neither of these two systems is solved exactly, then this formula should be replaced by

$$\delta_{k+1} = \alpha_{k+1} - w_k z_k - y_k v_k + y_k A_k z_k . \tag{10.70}$$

The last row/column pairs of the approximate factored inverse can be obtained by solving two sparse systems and computing a few dot products. It is interesting to note that the only difference with the ILUS factorization seen in Section 10.4.5 is that *the coefficient matrices for these systems are not the triangular factors of A_k, but the matrix A_k itself.*

To obtain an approximate factorization, simply exploit the fact that the A_k matrices are sparse and then employ iterative solvers in sparse-sparse mode. In this situation, formula (10.70) should be used for δ_{k+1}. The algorithm would be as follows.

ALGORITHM 10.11: Approximate Inverse Factors Algorithm

```
1. For $k = 1, \ldots, n$ Do:
2.     Solve (10.67) approximately;
3.     Solve (10.68) approximately;
4.     Compute $\delta_{k+1} = \alpha_{k+1} - w_k z_k - y_k v_k + y_k A_k z_k$
5. EndDo
```

A linear system must be solved with A_k in line 2 and a linear system with A_k^T in line 3. This is a good scenario for the Biconjugate Gradient algorithm or its equivalent two-sided Lanczos algorithm. In addition, the most current approximate inverse factors can be used to precondition the linear systems to be solved in steps 2 and 3. This was termed "self preconditioning" earlier. All the linear systems in the above algorithm can be solved in parallel since they are independent of one another. The diagonal D can then be obtained at the end of the process.

This approach is particularly suitable in the symmetric case. Since there is only one factor, the amount of work is halved. In addition, there is no problem with the existence in the positive definite case as is shown in the following lemma which states that δ_{k+1} is always > 0 when A is SPD, independently of the accuracy with which the system (10.67) is solved.

LEMMA 10.1 *Let A be SPD. Then, the scalar δ_{k+1} as computed by (10.70) is positive.*

Proof. In the symmetric case, $w_k = v_k^T$. Note that δ_{k+1} as computed by formula (10.70) is the $(k+1, k+1)$ element of the matrix $L_{k+1}A_{k+1}L_{k+1}^T$. It is positive because A_{k+1} is SPD. This is independent of the accuracy for solving the system to obtain z_k. ■

In the general nonsymmetric case, there is no guarantee that δ_{k+1} will be nonzero, unless the systems (10.67) and (10.68) are solved accurately enough. There is no practical problem here, since δ_{k+1} is computable. The only question remaining is a theoretical one: Can δ_{k+1} be guaranteed to be nonzero if the systems are solved with enough accuracy? Intuitively, if the system is solved exactly, then the D matrix must be nonzero since it is equal to the D matrix of the exact inverse factors in this case. The minimal assumption to make is that each A_k is nonsingular. Let δ_{k+1}^* be the value that would be obtained if at least one of the systems (10.67) or (10.68) is solved exactly. According to equation (10.69), in this situation this value is given by

$$\delta_{k+1}^* = \alpha_{k+1} - w_k A_k^{-1} v_k. \tag{10.71}$$

If A_{k+1} is nonsingular, then $\delta_{k+1}^* \neq 0$. To see this refer to the defining equation (10.66) and compute the product $L_{k+1}A_{k+1}U_{k+1}$ in the general case. Let r_k and s_k be the residuals obtained for these linear systems, i.e.,

$$r_k = v_k - A_k z_k, \quad s_k = w_k - y_k A_k. \tag{10.72}$$

Then a little calculation yields

$$L_{k+1}A_{k+1}U_{k+1} = \begin{pmatrix} L_k A_k U_k & L_k r_k \\ s_k U_k & \delta_{k+1} \end{pmatrix}. \tag{10.73}$$

If one of r_k or s_k is zero, then it is clear that the term δ_{k+1} in the above relation becomes δ_{k+1}^* and it must be nonzero since the matrix on the left-hand side is nonsingular. Incidentally, this relation shows the structure of the last matrix $L_n A_n U_n \equiv LAU$. The components 1 to $j-1$ of column j consist of the vector $L_j r_j$, the components 1 to $j-1$ of row i make up the vector $s_k U_k$, and the diagonal elements are the δ_i's. Consider now the expression for δ_{k+1} from (10.70).

$$\begin{aligned}
\delta_{k+1} &= \alpha_{k+1} - w_k z_k - y_k v_k + y_k A_k z_k \\
&= \alpha_{k+1} - w_k A_k^{-1}(v_k - r_k) - (w_k - s_k)A_k^{-1} v_k + (v_k - r_k)A_k^{-1}(w_k - s_k) \\
&= \alpha_{k+1} - v_k A_k^{-1} w_k + r_k A_k^{-1} s_k \\
&= \delta_{k+1}^* + r_k A_k^{-1} s_k.
\end{aligned}$$

This perturbation formula is of a second order in the sense that $|\delta_{k+1} - \delta_{k+1}^*| = O(\|r_k\|\,\|s_k\|)$. It guarantees that δ_{k+1} is nonzero whenever $|r_k A_k^{-1} s_k| < |\delta_{k+1}^*|$.

10.5.7 IMPROVING A PRECONDITIONER

After a computed ILU factorization results in an unsatisfactory convergence, it is difficult to improve it by modifying the L and U factors. One solution would be to discard this factorization and attempt to recompute a fresh one possibly with more fill-in. Clearly, this may be a wasteful process. A better alternative is to use approximate inverse techniques. Assume a (sparse) matrix M is a preconditioner to the original matrix A, so the preconditioned matrix is

$$C = M^{-1}A.$$

A sparse matrix S is sought to approximate the inverse of $M^{-1}A$. This matrix is then to be used as a preconditioner to $M^{-1}A$. Unfortunately, the matrix C is usually dense. However, observe that all that is needed is a matrix S such that

$$AS \approx M.$$

Recall that the columns of A and M are sparse. One approach is to compute a least-squares approximation in the Frobenius norm sense. This approach was used already in Section 10.5.1 when M is the identity matrix. Then the columns of S were obtained by approximately solving the linear systems $As_i \approx e_i$. The same idea can be applied here. Now, the systems

$$As_i = m_i$$

must be solved instead, where m_i is the i-th column of M which is sparse. Thus, the coefficient matrix and the right-hand side are sparse, as before.

10.6 BLOCK PRECONDITIONERS

Block preconditioning is a popular technique for block-tridiagonal matrices arising from the discretization of elliptic problems. It can also be generalized to other sparse matrices. We begin with a discussion of the block-tridiagonal case.

10.6.1 BLOCK-TRIDIAGONAL MATRICES

Consider a block-tridiagonal matrix blocked in the form

$$A = \begin{pmatrix} D_1 & E_2 & & & \\ F_2 & D_2 & E_3 & & \\ & \ddots & \ddots & \ddots & \\ & & F_{m-1} & D_{m-1} & E_m \\ & & & F_m & D_m \end{pmatrix}. \qquad (10.74)$$

One of the most popular block preconditioners used in the context of PDEs is based on this block-tridiagonal form of the coefficient matrix A. Let D be the block-diagonal matrix consisting of the diagonal blocks D_i, L the block strictly-lower triangular matrix consisting of the sub-diagonal blocks F_i, and U the block strictly-upper triangular matrix consisting of the super-diagonal blocks E_i. Then, the above matrix has the form

$$A = L + D + U.$$

A block ILU preconditioner is defined by

$$M = (L + \Delta)\Delta^{-1}(\Delta + U), \tag{10.75}$$

where L and U are the same as above, and Δ is a block-diagonal matrix whose blocks Δ_i are defined by the recurrence:

$$\Delta_i = D_i - F_i \Omega_{i-1} E_i, \tag{10.76}$$

in which Ω_j is some sparse approximation to Δ_j^{-1}. Thus, to obtain a block factorization, approximations to the inverses of the blocks Δ_i must be found. This clearly will lead to difficulties if explicit inverses are used.

An important particular case is when the diagonal blocks D_i of the original matrix are tridiagonal, while the co-diagonal blocks E_i and F_i are diagonal. Then, a simple recurrence formula for computing the inverse of a tridiagonal matrix can be exploited. Only the tridiagonal part of the inverse must be kept in the recurrence (10.76). Thus,

$$\Delta_1 = D_1, \tag{10.77}$$

$$\Delta_i = D_i - F_i \Omega_{i-1}^{(3)} E_i, \quad i = 1, \ldots, m, \tag{10.78}$$

where $\Omega_k^{(3)}$ is the tridiagonal part of Δ_k^{-1}.

$$(\Omega_k^{(3)})_{i,j} = (\Delta_k^{-1})_{i,j} \quad \text{for} \quad |i - j| \leq 1.$$

The following theorem can be shown.

THEOREM 10.4 *Let A be Symmetric Positive Definite and such that*

- $a_{ii} > 0,\ i = 1, \ldots, n,$ *and* $a_{ij} \leq 0$ *for all* $j \neq i$.
- *The matrices D_i are all (strict) diagonally dominant.*

Then each block Δ_i computed by the recurrence (10.77), (10.78) is a symmetric M-matrix. In particular, M is also a positive definite matrix.

We now show how the inverse of a tridiagonal matrix can be obtained. Let a tridiagonal matrix Δ of dimension l be given in the form

$$\Delta = \begin{pmatrix} \alpha_1 & -\beta_2 & & & \\ -\beta_2 & \alpha_2 & -\beta_3 & & \\ & \ddots & \ddots & \ddots & \\ & & -\beta_{l-1} & \alpha_{l-1} & -\beta_l \\ & & & -\beta_l & \alpha_l \end{pmatrix},$$

and let its Cholesky factorization be

$$\Delta = LDL^T,$$

with

$$D = \text{diag}\ \{\delta_i\}$$

and

$$L = \begin{pmatrix} 1 & & & & \\ -\gamma_2 & 1 & & & \\ & \ddots & \ddots & & \\ & & -\gamma_{l-1} & 1 & \\ & & & -\gamma_l & 1 \end{pmatrix}.$$

The inverse of Δ is $L^{-T}D^{-1}L^{-1}$. Start by observing that the inverse of L^T is a unit upper triangular matrix whose coefficients u_{ij} are given by

$$u_{ij} = \gamma_{i+1}\gamma_{i+2}\dots\gamma_{j-1}\gamma_j \quad \text{for} \quad 1 \le i < j < l.$$

As a result, the j-th column c_j of L^{-T} is related to the $(j-1)$-st column c_{j-1} by the very simple recurrence,

$$c_j = e_j + \gamma_j c_{j-1}, \quad \text{for} \quad j \ge 2$$

starting with the first column $c_1 = e_1$. The inverse of Δ becomes

$$\Delta^{-1} = L^{-T}D^{-1}L^{-1} = \sum_{j=1}^{l} \frac{1}{\delta_j} c_j c_j^T. \tag{10.79}$$

See Exercise 12 for a proof of the above equality. As noted, the recurrence formulas for computing Δ^{-1} can be unstable and lead to numerical difficulties for large values of l.

10.6.2 GENERAL MATRICES

A general sparse matrix can often be put in the form (10.74) where the blocking is either natural as provided by the physical problem, or artificial when obtained as a result of RCMK ordering and some block partitioning. In such cases, a recurrence such as (10.76) can still be used to obtain a block factorization defined by (10.75). A 2-level preconditioner can be defined by using sparse inverse approximate techniques to approximate Ω_i. These are sometimes termed implicit-explicit preconditioners, the implicit part referring to the block-factorization and the explicit part to the approximate inverses used to explicitly approximate Δ_i^{-1}.

10.7 PRECONDITIONERS FOR THE NORMAL EQUATIONS

When the original matrix is strongly indefinite, i.e., when it has eigenvalues spread on both sides of the imaginary axis, the usual Krylov subspace methods may fail. The Conjugate Gradient approach applied to the normal equations may then become a good alternative. Choosing to use this alternative over the standard methods may involve inspecting the spectrum of a Hessenberg matrix obtained from a small run of an unpreconditioned GMRES algorithm.

If the normal equations approach is chosen, the question becomes how to precondition the resulting iteration. An ILU preconditioner can be computed for A and the preconditioned normal equations,

$$A^T(LU)^{-T}(LU)^{-1}Ax = A^T(LU)^{-T}(LU)^{-1}b,$$

can be solved. However, when A is not diagonally dominant the ILU factorization process may encounter a zero pivot. Even when this does not happen, the resulting preconditioner may be of poor quality. An incomplete factorization routine with pivoting, such as ILUTP, may constitute a good choice. ILUTP can be used to precondition either the original equations or the normal equations shown above. This section explores a few other options available for preconditioning the normal equations.

10.7.1 JACOBI, SOR, AND VARIANTS

There are several ways to exploit the relaxation schemes for the Normal Equations seen in Chapter 8 as preconditioners for the CG method applied to either (8.1) or (8.3). Consider (8.3), for example, which requires a procedure delivering an approximation to $(AA^T)^{-1}v$ for any vector v. One such procedure is to perform one step of SSOR to solve the system $(AA^T)w = v$. Denote by M^{-1} the linear operator that transforms v into the vector resulting from this procedure, then the usual Conjugate Gradient method applied to (8.3) can be recast in the same form as Algorithm 8.5. This algorithm is known as CGNE/SSOR. Similarly, it is possible to incorporate the SSOR preconditioning in Algorithm 8.4, which is associated with the Normal Equations (8.1), by defining M^{-1} to be the linear transformation that maps a vector v into a vector w resulting from the forward sweep of Algorithm 8.2 followed by a backward sweep. We will refer to this algorithm as CGNR/SSOR.

The CGNE/SSOR and CGNR/SSOR algorithms will not break down if A is nonsingular, since then the matrices AA^T and A^TA are Symmetric Positive Definite, as are the preconditioning matrices M. There are several variations to these algorithms. The standard alternatives based on the same formulation (8.1) are either to use the preconditioner on the right, solving the system $A^TAM^{-1}y = b$, or to split the preconditioner into a forward SOR sweep on the left and a backward SOR sweep on the right of the matrix A^TA. Similar options can also be written for the Normal Equations (8.3) again with three different ways of preconditioning. Thus, at least six different algorithms can be defined.

10.7.2 IC(0) FOR THE NORMAL EQUATIONS

The Incomplete Cholesky IC(0) factorization can be used to precondition the Normal Equations (8.1) or (8.3). This approach may seem attractive because of the success of incomplete factorization preconditioners. However, a major problem is that the Incomplete Cholesky factorization is not guaranteed to exist for an arbitrary Symmetric Positive Definite matrix B. All the results that guarantee existence rely on some form of diagonal dominance. One of the first ideas suggested to handle this difficulty was to use an Incomplete Cholesky factorization on the "shifted" matrix $B + \alpha I$. We refer to IC(0) applied to $B = A^TA$ as ICNR(0), and likewise IC(0) applied to $B = AA^T$

as ICNE(0). Shifted variants correspond to applying IC(0) to the shifted B matrix.

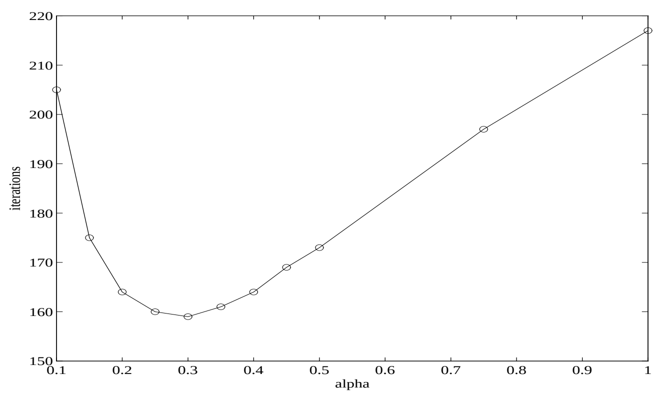


Figure 10.14 *Iteration count as a function of the shift α.*

One issue often debated is how to find good values for the shift α. There is no easy and well-founded solution to this problem for irregularly structured symmetric sparse matrices. One idea is to select the smallest possible α that makes the shifted matrix diagonally dominant. However, this shift tends to be too large in general because IC(0) may exist for much smaller values of α. Another approach is to determine the smallest α for which the IC(0) factorization exists. Unfortunately, this is not a viable alternative. As is often observed, the number of steps required for convergence starts decreasing as α increases, and then increases again. An illustration of this is shown in Figure 10.14. This plot suggests that there is an optimal value for α which is far from the smallest admissible one. For small α, the diagonal dominance of $B + \alpha I$ is weak and, as a result, the computed IC factorization is a poor approximation to the matrix $B(\alpha) \equiv B + \alpha I$. In other words, $B(\alpha)$ is close to the original matrix B, but the IC(0) factorization is far from $B(\alpha)$. For large α, the opposite is true. The matrix $B(\alpha)$ has a large deviation from $B(0)$, but its IC(0) factorization may be quite good. Therefore, the general shape of the curve shown in the figure is not too surprising.

To implement the algorithm, the matrix $B = AA^T$ need not be formed explicitly. All that is required is to be able to access one row of B at a time. This row can be computed, used, and then discarded. In the following, the i-th row $e_i^T A$ of A is denoted by a_i. The algorithm is row-oriented and all vectors denote row vectors. It is adapted from the ILU(0) factorization of a sparse matrix, i.e., Algorithm 10.4, but it actually computes the LDL^T factorization instead of an LU or LL^T factorization. The main difference with Algorithm 10.4 is that the loop in line 7 is now restricted to $j \leq i$ because of symmetry. If only the l_{ij} elements are stored row-wise, then the rows of $U = L^T$ which are needed in this loop are not directly available. Denote the j-th row of $U = L^T$ by u_j. These rows are accessible by adding a column data structure for the L matrix which is updated dynamically. A linked list data structure can be used for this purpose. With this in mind, the IC(0) algorithm will have the following structure.

ALGORITHM 10.12: Shifted ICNE(0)

```
1.  Initial step: Set $d_1 := a_{11}$ , $l_{11} = 1$
2.  For $i = 2, 3, \ldots, n$ Do:
3.     Obtain all the nonzero inner products
4.        $l_{ij} = (a_j, a_i), j = 1, 2, \ldots, i-1$, and $l_{ii} := \|a_i\|^2 + \alpha$
5.     Set $NZ(i) \equiv \{j \mid l_{ij} \neq 0\}$
6.     For $k = 1, \ldots, i-1$ and if $k \in NZ(i)$ Do:
7.        Extract row $u_k = (Le_k)^T$
8.        Compute $l_{ik} := l_{ik}/d_k$
9.        For $j = k+1, \ldots, i$ and if $(i, j) \in NZ(i)$ Do:
10.          Compute $l_{ik} := l_{ik} - l_{ij} u_{kj}$
11.       EndDo
12.    EndDo
13.    Set $d_i := l_{ii}, l_{ii} := 1$
14. EndDo
```

Note that initially the row u_1 in the algorithm is defined as the first row of A. All vectors in the algorithm are row vectors.

The step represented by lines 3 and 4, which computes the inner products of row number i with all previous rows, needs particular attention. If the inner products

$$a_1^T a_i, \; a_2^T a_i, \ldots, \; a_{i-1}^T a_i$$

are computed separately, the total cost of the incomplete factorization would be of the order of n^2 steps and the algorithm would be of little practical value. However, most of these inner products are equal to zero because of sparsity. This indicates that it may be possible to compute only those nonzero inner products at a much lower cost. Indeed, if c is the column of the $i-1$ inner products c_{ij}, then c is the product of the rectangular $(i-1) \times n$ matrix A_{i-1} whose rows are $a_1^T, \ldots, a_{i-1}^T$ by the vector a_i, i.e.,

$$c = A_{i-1} a_i. \tag{10.80}$$

This is a sparse matrix-by-sparse vector product which was discussed in Section 10.5. It is best performed as a linear combination of the columns of A_{i-1} which are sparse. The only difficulty with this implementation is that it requires both the row data structure of A and of its transpose. A standard way to handle this problem is by building a linked-list data structure for the transpose. There is a similar problem for accessing the transpose of L, as mentioned earlier. Therefore, two linked lists are needed: one for the L matrix and the other for the A matrix. These linked lists avoid the storage of an additional real array for the matrices involved and simplify the process of updating the matrix A when new rows are obtained. It is important to note that these linked lists are used only in the preprocessing phase and are discarded once the incomplete factorization terminates.

10.7.3 INCOMPLETE GRAM-SCHMIDT AND ILQ

Consider a general sparse matrix A and denote its rows by $a_1, a_2, \ldots, a_n$. The (complete) LQ factorization of A is defined by

$$A = LQ,$$

where L is a lower triangular matrix and Q is unitary, i.e., $Q^T Q = I$. The L factor in the above factorization is identical with the Cholesky factor of the matrix $B = AA^T$. Indeed, if $A = LQ$ where L is a lower triangular matrix having positive diagonal elements, then

$$B = AA^T = LQQ^T L^T = LL^T.$$

The uniqueness of the Cholesky factorization with a factor L having positive diagonal elements shows that L is equal to the Cholesky factor of B. This relationship can be exploited to obtain preconditioners for the Normal Equations.

Thus, there are two ways to obtain the matrix L. The first is to form the matrix B explicitly and use a sparse Cholesky factorization. This requires forming the data structure of the matrix AA^T, which may be much denser than A. However, reordering techniques can be used to reduce the amount of work required to compute L. This approach is known as *symmetric squaring.*

A second approach is to use the Gram-Schmidt process. This idea may seem undesirable at first because of its poor numerical properties when orthogonalizing a large number of vectors. However, because the rows remain very sparse in the incomplete LQ factorization (to be described shortly), any given row of A will be orthogonal typically to most of the previous rows of Q. As a result, the Gram-Schmidt process is much less prone to numerical difficulties. From the data structure point of view, Gram-Schmidt is optimal because it does not require allocating more space than is necessary, as is the case with approaches based on symmetric squaring. Another advantage over symmetric squaring is the simplicity of the orthogonalization process and its strong similarity with the LU factorization. At every step, a given row is combined with previous rows and then normalized. The incomplete Gram-Schmidt procedure is modeled after the following algorithm.

ALGORITHM 10.13: LQ Factorization of A

```
1. For $i = 1, \ldots, n$ Do:
2.    Compute $l_{ij} := (a_i, q_j)$ , for $j = 1, 2, \ldots, i-1$,
3.    Compute $q_i := a_i - \sum_{j=1}^{i-1} l_{ij} q_j$ , and $l_{ii} = \|q_i\|_2$
4.    If $l_{ii} := 0$ then Stop; else Compute $q_i := q_i / l_{ii}$.
5. EndDo
```

If the algorithm completes, then it will result in the factorization $A = LQ$ where the rows of Q and L are the rows defined in the algorithm. To define an *incomplete* factorization, a *dropping* strategy similar to those defined for Incomplete LU factorizations must be incorporated. This can be done in very general terms as follows. Let P_L and P_Q be the chosen zero patterns for the matrices L, and Q, respectively. The only restriction on P_L is that

$$P_L \subset \{(i, j) \mid i \neq j\}.$$

As for P_Q, for each row there must be at least one nonzero element, i.e.,

$$\{j \,|(i,j) \in P_Q\} \neq \{1,2,\ldots,n\}, \qquad \text{for } i = 1,\ldots,n.$$

These two sets can be selected in various ways. For example, similar to ILUT, they can be determined dynamically by using a drop strategy based on the magnitude of the elements generated. As before, x_i denotes the i-th row of a matrix X and x_{ij} its (i,j)-th entry.

ALGORITHM 10.14: Incomplete Gram-Schmidt

```
1. For $i = 1,\ldots,n$ Do:
2.    Compute $l_{ij} := (a_i, q_j)$ , for $j = 1,2,\ldots,i-1$,
3.    Replace $l_{ij}$ by zero if $(i,j) \in P_L$
4.    Compute $q_i := a_i - \sum_{j=1}^{i-1} l_{ij} q_j$ ,
5.    Replace each $q_{ij}, j = 1,\ldots,n$ by zero if $(i,j) \in P_Q$
6.    $l_{ii} := \|q_i\|_2$
7.    If $l_{ii} = 0$ then Stop; else compute $q_i := q_i / l_{ii}$.
8. EndDo
```

We recognize in line 2 the same practical problem encountered in the previous section for IC(0) for the Normal Equations. It can be handled in the same manner. Therefore, the row structures of A, L, and Q are needed, as well as a linked list for the column structure of Q.

After the i-th step is performed, the following relation holds:

$$q_i = l_{ii} q_i + r_i = a_i - \sum_{j=1}^{j-1} l_{ij} q_j$$

or

$$a_i = \sum_{j=1}^{j} l_{ij} q_j + r_i \tag{10.81}$$

where r_i is the row of elements that have been dropped from the row q_i in line 5. The above equation translates into

$$A = LQ + R \tag{10.82}$$

where R is the matrix whose i-th row is r_i, and the notation for L and Q is as before.

The case where the elements in Q are not dropped, i.e., the case when P_Q is the empty set, is of particular interest. Indeed, in this situation, $R = 0$ and we have the exact relation $A = LQ$. However, Q is not unitary in general because elements are dropped from L. If at a given step $l_{ii} = 0$, then (10.81) implies that a_i is a linear combination of the rows q_1, …, q_{j-1}. Each of these q_k is, inductively, a linear combination of $a_1, \ldots a_k$. Therefore, a_i would be a linear combination of the previous rows, $a_1, \ldots, a_{i-1}$ which cannot be true if A is nonsingular. As a result, the following proposition can be stated.

PROPOSITION 10.8 *If A is nonsingular and $P_Q = \phi$, then the Algorithm 10.14 completes and computes an incomplete LQ factorization $A = LQ$, in which Q is nonsingular and L is a lower triangular matrix with positive elements.*

A major problem with the decomposition (10.82) is that the matrix Q is not orthogonal in general. In fact, nothing guarantees that it is even nonsingular unless Q is not dropped or the dropping strategy is made tight enough.

Because the matrix L of the *complete* LQ factorization of A is identical with the Cholesky factor of B, one might wonder why the IC(0) factorization of B does not always exist while the ILQ factorization seems to always exist. In fact, the relationship between ILQ and ICNE, i.e., the Incomplete Cholesky for $B = AA^T$, can lead to a more rigorous way of choosing a good pattern for ICNE, as is explained next.

We turn our attention to Modified Gram-Schmidt. The only difference is that the row q_j is updated immediately after an inner product is computed. The algorithm is described without dropping for Q for simplicity.

ALGORITHM 10.15: Incomplete Modified Gram-Schmidt

```
1. For $i = 1, \ldots, n$ Do:
2.    $q_i := a_i$
3.    For $j = 1, \ldots, i-1$, Do:
4.       Compute $l_{ij} := \begin{cases} 0 & \text{if } (i,j) \in P_L \\ (q_i, q_j) & \text{otherwise} \end{cases}$
5.       Compute $q_i := q_i - l_{ij} q_j$.
6.    EndDo
7.    $l_{ii} := \|q_i\|_2$
8.    If $l_{ii} = 0$ then Stop; else Compute $q_i := q_i / l_{ii}$.
9. EndDo
```

When A is nonsingular, the same result as before is obtained if no dropping is used on Q, namely, that the factorization will exist and be exact in that $A = LQ$. Regarding the implementation, if the zero pattern P_L is known in advance, the computation of the inner products in line 4 does not pose a particular problem. Without any dropping in Q, this algorithm may be too costly in terms of storage. It is interesting to see that this algorithm has a connection with ICNE, the incomplete Cholesky applied to the matrix AA^T. The following result is stated without proof.

THEOREM 10.5 *Let A be an $n \times m$ matrix and let $B = AA^T$. Consider a zero-pattern set P_L which is such that for any $1 \leq i, j, k \leq n$, with $i < j$ and $i < k$, the following holds:*

$$(i,j) \in P_L \text{ and } (i,k) \notin P_L \rightarrow (j,k) \in P_L.$$

Then the matrix L obtained from Algorithm 10.15 with the zero-pattern set P_L is identical with the L factor that would be obtained from the Incomplete Cholesky factorization applied to B with the zero-pattern set P_L.

For a proof, see [222]. This result shows how a zero-pattern can be defined which guarantees the existence of an Incomplete Cholesky factorization on AA^T.

EXERCISES

1. Assume that A is the Symmetric Positive Definite matrix arising from the 5-point finite difference discretization of the Laplacean on a given mesh. We reorder the matrix using the red-black ordering and obtain the reordered matrix

$$B = \begin{pmatrix} D_1 & E \\ E^T & D_2 \end{pmatrix}.$$

We then form the Incomplete Cholesky factorization on this matrix.

a. Show the fill-in pattern for the IC(0) factorization for a matrix of size $n = 12$ associated with a 4×3 mesh.

b. Show the nodes associated with these fill-ins on the 5-point stencil in the finite difference mesh.

c. Give an approximate count of the total number of fill-ins when the original mesh is square, with the same number of mesh points in each direction. How does this compare with the natural ordering? Any conclusions?

2. Consider a 6×6 tridiagonal nonsingular matrix A.

a. What can be said about its ILU(0) factorization (when it exists)?

b. Suppose that the matrix is permuted (symmetrically, i.e., both rows and columns) using the permutation

$$\pi = [1, 3, 5, 2, 4, 6].$$

i. Show the pattern of the permuted matrix.

ii. Show the locations of the fill-in elements in the ILU(0) factorization.

iii. Show the pattern of the ILU(1) factorization as well as the fill-ins generated.

iv. Show the level of fill of each element at the end of the ILU(1) process (including the fill-ins).

v. What can be said of the ILU(2) factorization for this permuted matrix?

3. Assume that A is the matrix arising from the 5-point finite difference discretization of an elliptic operator on a given mesh. We reorder the original linear system using the red-black ordering and obtain the reordered linear system

$$\begin{pmatrix} D_1 & E \\ F & D_2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}.$$

a. Show how to obtain a system (called the *reduced system*) which involves the variable x_2 only.

b. Show that this reduced system is also a sparse matrix. Show the stencil associated with the reduced system matrix on the original finite difference mesh and give a graph-theory interpretation of the reduction process. What is the maximum number of nonzero elements in each row of the reduced system.

4. It was stated in Section 10.3.2 that for some specific matrices the ILU(0) factorization of A can be put in the form

$$M = (D - E)D^{-1}(D - F)$$

in which $-E$ and $-F$ are the strict-lower and -upper parts of A, respectively.

a. Characterize these matrices carefully and give an interpretation with respect to their adjacency graphs.

b. Verify that this is true for standard 5-point matrices associated with any domain Ω.

c. Is it true for 9-point matrices?

d. Is it true for the higher level ILU factorizations?

5. Let A be a pentadiagonal matrix having diagonals in offset positions $-m, -1, 0, 1, m$. The coefficients in these diagonals are all constants: a for the main diagonal and -1 for all others. It is assumed that $a \geq \sqrt{8}$. Consider the ILU(0) factorization of A as given in the form (10.20). The elements d_i of the diagonal D are determined by a recurrence of the form (10.19).

a. Show that $\frac{a}{2} \leq d_i \leq a$ for $i = 1, \ldots, n$.

b. Show that d_i is a decreasing sequence. [Hint: Use induction].

c. Prove that the formal (infinite) sequence defined by the recurrence converges. What is its limit?

6. Consider a matrix A which is split in the form $A = D_0 - E - F$, where D_0 is a block diagonal matrix whose block-diagonal entries are the same as those of A, and where $-E$ is strictly lower triangular and $-F$ is strictly upper triangular. In some cases the block form of the ILU(0) factorization can be put in the form (Section 10.3.2):

$$M = (D - E)D^{-1}(D - F).$$

The block entries of D can be defined by a simple matrix recurrence. Find this recurrence relation. The algorithm may be expressed in terms of the block entries the matrix A.

7. Generalize the formulas developed at the end of Section 10.6.1 for the inverses of symmetric tridiagonal matrices, to the nonsymmetric case.

8. Develop recurrence relations for Incomplete Cholesky with no fill-in (IC(0)), for 5-point matrices, similar to those seen in Section 10.3.4 for ILU(0). Same question for IC(1).

9. What becomes of the formulas seen in Section 10.3.4 in the case of a 7-point matrix (for three-dimensional problems)? In particular, can the ILU(0) factorization be cast in the form (10.20) in which $-E$ is the strict-lower diagonal of A and $-F$ is the strict upper triangular part of A, and D is a certain diagonal?

10. Consider an arbitrary matrix A which is split in the usual manner as $A = D_0 - E - F$, in which $-E$ and $-F$ are the strict-lower and -upper parts of A, respectively, and define, for any diagonal matrix D, the approximate factorization of A given by

$$M = (D - E)D^{-1}(D - F).$$

Show how a diagonal D can be determined such that A and M have the same diagonal elements. Find a recurrence relation for the elements of D. Consider now the symmetric case and assume that the matrix D which is positive can be found. Write M in the form

$$M = (D^{1/2} - ED^{-1/2})(D^{1/2} - ED^{-1/2})^T \equiv L_1 L_1^T.$$

What is the relation between this matrix and the matrix of the SSOR(ω) preconditioning, in the particular case when $D^{-1/2} = \omega I$? Conclude that this form of ILU factorization is in effect an SSOR preconditioning with a different relaxation factor ω for each equation.

11. Consider a general sparse matrix A (irregularly structured). We seek an approximate LU factorization of the form

$$M = (D - E)D^{-1}(D - F)$$

in which $-E$ and $-F$ are the strict-lower and -upper parts of A, respectively. It is assumed that A is such that

$$a_{ii} > 0, \quad a_{ij}a_{ji} \geq 0 \quad \text{for} \quad i, j = 1, \ldots, n.$$

a. By identifying the diagonal elements of A with those of M, derive an algorithm for generating the elements of the diagonal matrix D recursively.

b. Establish that if $d_j > 0$ for $j < i$ then $d_i \leq a_{ii}$. Is it true in general that $d_j > 0$ for all j?

c. Assume that for $i = 1, \ldots, j-1$ we have $d_i \geq \alpha > 0$. Show a sufficient condition under which $d_j \geq \alpha$. Are there cases in which this condition cannot be satisfied for any α?

d. Assume now that all diagonal elements of A are equal to a constant, i.e., $a_{jj} = a$ for $j = 1, \ldots, n$. Define $\alpha \equiv \frac{a}{2}$ and let

$$S_j \equiv \sum_{i=1}^{j-1} a_{ij}a_{ji}, \quad \sigma \equiv \max_{j=1,\ldots,n} S_j.$$

Show a condition on σ under which $d_j \geq \alpha, \ j = 1, 2, \ldots, n$.

12. Show the second part of (10.79). [Hint: Exploit the formula $AB^T = \sum_{j=1}^n a_j b_j^T$ where a_j, b_j are the j-th columns of A and B, respectively].

13. Let a preconditioning matrix M be related to the original matrix A by $M = A + E$, in which E is a matrix of rank k.

a. Assume that both A and M are Symmetric Positive Definite. How many steps at most are required for the preconditioned Conjugate Gradient method to converge when M is used as a preconditioner?

b. Answer the same question for the case when A and M are nonsymmetric and the full GMRES is used on the preconditioned system.

14. Formulate the problem for finding an approximate inverse M to a matrix A as a large $n^2 \times n^2$ linear system. What is the Frobenius norm in the space in which you formulate this problem?

15. The concept of *mask* is useful in the global iteration technique. For a sparsity pattern S, i.e., a set of pairs (i, j) and a matrix B, we define the product $C = B \odot S$ to be the matrix whose elements c_{ij} are zero if (i, j) does not belong to S, and b_{ij} otherwise. This is called a mask operation since its effect is to ignore every value not in the pattern S. Consider a global minimization of the function $F_S(M) \equiv \|S \odot (I - AM)\|_F$.

a. What does the result of Proposition 10.3 become for this new objective function?

b. Formulate an algorithm based on a global masked iteration, in which the mask is fixed and equal to the pattern of A.

c. Formulate an algorithm in which the mask is adapted at each outer step. What criteria would you use to select the mask?

16. Consider the global self preconditioned MR iteration algorithm seen in Section 10.5.5. Define the acute angle between two matrices as

$$\cos \angle(X, Y) \equiv \frac{\langle X, Y \rangle}{\|X\|_F \|Y\|_F}.$$

a. Following what was done for the (standard) Minimal Residual algorithm seen in Chapter 5, establish that the matrices $B_k = AM_k$ and $R_k = I - B_k$ produced by global MR without dropping are such that

$$\|R_{k+1}\|_F \leq \|R_k\|_F \ \sin \angle(R_k, B_k R_k).$$

b. Let now $M_0 = \alpha A^T$ so that B_k is symmetric for all k (see Section 10.5.5). Assume that, at a given step k the matrix B_k is positive definite. Show that

$$\cos \angle(R_k, B_k R_k) \geq \frac{\lambda_{min}(B_k)}{\lambda_{max}(B_k)}$$

in which $\lambda_{min}(B_k)$ and $\lambda_{max}(B_k)$ are, respectively, the smallest and largest eigenvalues of B_k.

17. In the two-sided version of approximate inverse preconditioners, the option of minimizing

$$f(L, U) = \|I - LAU\|_F^2$$

was mentioned, where L is unit lower triangular and U is upper triangular.

a. What is the gradient of $f(L, U)$?

b. Formulate an algorithm based on minimizing this function globally.

18. Consider the two-sided version of approximate inverse preconditioners, in which a unit lower triangular L and an upper triangular U are sought so that $LAU \approx I$. One idea is to use an alternating procedure in which the first half-step computes a right approximate inverse U to LA, which is restricted to be upper triangular, and the second half-step computes a left approximate inverse L to AU, which is restricted to be lower triangular.

a. Consider the first half-step. Since the candidate matrix U is restricted to be upper triangular, special care must be exercised when writing a column-oriented approximate inverse algorithm. What are the differences with the standard MR approach described by Algorithm 10.10?

b. Now consider seeking an upper triangular matrix U such that the matrix $(LA)U$ is close to the identity only in its upper triangular part. A similar approach is to be taken for the second half-step. Formulate an algorithm based on this approach.

19. Write all six variants of the preconditioned Conjugate Gradient algorithm applied to the Normal Equations, mentioned at the end of Section 10.7.1.

20. With the standard splitting $A = D - E - F$, in which D is the diagonal of A and $-E, -F$ its lower- and upper triangular parts, respectively, we associate the factored approximate inverse factorization,

$$(I + ED^{-1})A(I + D^{-1}F) = D + R. \tag{10.83}$$

a. Determine R and show that it consists of second order terms, i.e., terms involving products of at least two matrices from the pair E, F.

b. Now use the previous approximation for $D + R \equiv D_1 - E_1 - F_1$,

$$(I + E_1 D_1^{-1})(D + R)(I + D_1^{-1} F_1) = D_1 + R_1.$$

Show how the approximate inverse factorization (10.83) can be improved using this new approximation. What is the order of the resulting approximation?

NOTES AND REFERENCES. A breakthrough paper on preconditioners is the article [149] by Meijerink and van der Vorst who established existence of the incomplete factorization for M-matrices and showed that preconditioning the Conjugate Gradient by using an ILU factorization can result in an extremely efficient combination. The idea of preconditioning was exploited in many earlier papers. For example, in [11, 12] Axelsson discusses SSOR iteration, "accelerated" by either the Conjugate

Gradient or Chebyshev acceleration. Incomplete factorizations were also discussed in early papers, for example, by Varga [212] and Buleev [45]. Thus, Meijerink and van der Vorst's paper played an essential role in directing the attention of researchers and practitioners to a rather important topic and marked a turning point. Many of the early techniques were developed for regularly structured matrices. The generalization, using the definition of level of fill for high-order Incomplete LU factorizations for unstructured matrices, was introduced by Watts [223] for petroleum engineering problems.

Recent research on iterative techniques has been devoted in great part to the development of better iterative accelerators, while "robust" preconditioners have by and large been neglected. This is certainly caused by the inherent lack of theory to support such methods. Yet these techniques are vital to the success of iterative methods in real-life applications. A general approach based on modifying a given direct solver by including a drop-off rule was one of the first in this category [151, 157, 235, 98]. More economical alternatives, akin to ILU(p), were developed later [179, 183, 68, 67, 226, 233]. ILUT and ILUTP, are inexpensive general purpose preconditioners which are fairly robust and efficient. However, many of these preconditioners, including ILUT and ILUTP, can fail. Occasionally, a more accurate ILUT factorization leads to a larger number of steps needed for convergence. One source of failure is the instability of the preconditioning operation. These phenomena of instability have been studied by Elman [81] who proposed a detailed analysis of ILU and MILU preconditioners for model problems. The theoretical analysis on ILUT stated as Theorem 10.3 is modeled after Theorem 1.14 in Axelsson and Barker [16] for ILU(0).

Some theory for block preconditioners is discussed in Axelsson's book [15]. Different forms of block preconditioners were developed independently by Axelsson, Brinkkemper, and Il'in [17] and by Concus, Golub, and Meurant [61], initially for block matrices arising from PDEs in two dimensions. Later, some generalizations were proposed [137]. Thus, the 2-level implicit-explicit preconditioning introduced in [137] consists of using sparse inverse approximations to Δ_i^{-1} for obtaining Ω_i.

The current rebirth of approximate inverse preconditioners [112, 62, 137, 54] is spurred by both parallel processing and robustness considerations. Other preconditioners which are not covered here are those based on domain decomposition techniques. Some of these techniques will be reviewed in Chapter 13.

On another front, there is also increased interest in methods that utilize Normal Equations in one way or another. Earlier, ideas revolved around shifting the matrix $B = A^T A$ before applying the IC(0) factorization as was suggested by Kershaw [134] in 1977. Manteuffel [148] also made some suggestions on how to select a good α in the context of the CGW algorithm. Currently, new ways of exploiting the relationship with the QR (or LQ) factorization to define IC(0) more rigorously are being explored; see the recent work in [222]. ■

CHAPTER

11 PARALLEL IMPLEMENTATIONS

Parallel computing is fast becoming an inexpensive alternative to the standard supercomputer approach for solving large scale problems that arise in scientific and engineering applications. Since iterative methods are appealing for large linear systems of equations, it is no surprise that they are the prime candidates for implementations on parallel architectures. There have been two traditional approaches for developing parallel iterative techniques thus far. The first extracts parallelism whenever possible from standard algorithms. The advantage of this viewpoint is that it is easier to understand in general since the underlying method has not changed from its sequential equivalent. The second approach is to develop alternative algorithms which have enhanced parallelism. This chapter will give an overview of implementations and will emphasize methods in the first category. The later chapters will consider alternative algorithms that have been developed specifically for parallel computing environments.

11.1 INTRODUCTION

The remaining chapters of this book will examine the impact of high performance computing on the design of iterative methods for solving large linear systems of equations. Because of the increased importance of three-dimensional models combined with the high cost associated with sparse direct methods for solving these problems, iterative techniques are starting to play a major role in many application areas. The main appeal of iterative methods is their low storage requirement. Another advantage is that they are far easier to implement on parallel computers than sparse direct methods because they only require a rather small set of computational kernels. Increasingly, direct solvers are being used in conjunction with iterative solvers to develop robust preconditioners.

The first considerations for high-performance implementations of iterative methods involved implementations on vector computers. These efforts started in the mid 1970s when the first vector computers appeared. Currently, there is a larger effort to develop new practical iterative methods that are not only efficient in a parallel environment, but also robust. Often, however, these two requirements seem to be in conflict.

This chapter begins with a short overview of the various ways in which parallelism has been exploited in the past and a description of the current architectural models for existing commercial parallel computers. Then, the basic computations required in Krylov subspace methods will be discussed along with their implementations.

11.2 FORMS OF PARALLELISM

Parallelism has been exploited in a number of different forms since the first computers were built. The six major forms of parallelism are: (1) multiple functional units; (2) pipelining; (3) vector processing; (4) multiple vector pipelines; (5) multiprocessing; and (6) distributed computing. Next is a brief description of each of these approaches.

11.2.1 MULTIPLE FUNCTIONAL UNITS

This is one of the earliest forms of parallelism. It consists of multiplying the number of functional units such as adders and multipliers. Thus, the control units and the registers are shared by the functional units. The detection of parallelism is done at compilation time with a "Dependence Analysis Graph," an example of which is shown in Figure 11.1.

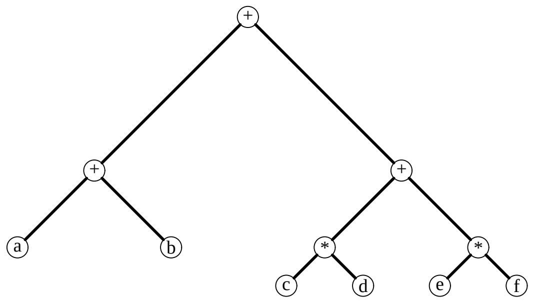
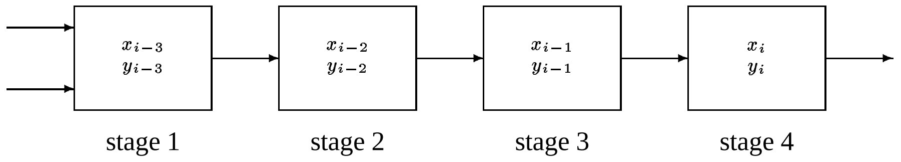


Figure 11.1 *Dependence analysis for arithmetic expression:* $(a + b) + (c * d + d * e)$.

In the example of Figure 11.1, the two multiplications can be performed simultaneously, then the two additions in the middle are performed simultaneously. Finally, the addition at the root is performed.

11.2.2 PIPELINING

The pipelining concept is essentially the same as that of an assembly line used in car manufacturing. Assume that an operation takes s stages to complete. Then the operands can be passed through the s stages instead of waiting for all stages to be completed for the first two operands.



If each stage takes a time τ to complete, then an operation with n numbers will take the time $s\tau + (n-1)\tau = (n+s-1)\tau$. The speed-up would be the ratio of the time to complete the s stages in a non-pipelined unit versus, i.e., $s \times n \times \tau$, over the above obtained time,

$$S = \frac{ns}{n+s-1}.$$

For large n, this would be close to s.

11.2.3 VECTOR PROCESSORS

Vector computers appeared in the beginning of the 1970s with the CDC Star 100 and then the CRAY-1 and Cyber 205. These are computers which are equipped with vector pipelines, i.e., pipelined functional units, such as a pipelined floating-point adder, or a pipelined floating-point multiplier. In addition, they incorporate vector instructions explicitly as part of their instruction sets. Typical vector instructions are, for example:

`VLOAD`	To load a vector from memory to a vector register
`VADD`	To add the content of two vector registers
`VMUL`	To multiply the content of two vector registers.

Similar to the case of multiple functional units for scalar machines, vector pipelines can be duplicated to take advantage of any fine grain parallelism available in loops. For example, the Fujitsu and NEC computers tend to obtain a substantial portion of their performance in this fashion. There are many vector operations that can take advantage of *multiple vector pipelines*.

11.2.4 MULTIPROCESSING AND DISTRIBUTED COMPUTING

A multiprocessor system is a computer, or a set of several computers, consisting of several processing elements (PEs), each consisting of a CPU, a memory, an I/O subsystem, etc. These PEs are connected to one another with some communication medium, either a bus or some multistage network. There are numerous possible configurations, some of which will be covered in the next section.

Distributed computing is a more general form of multiprocessing, in which the processors are actually computers linked by some Local Area Network. Currently, there are a number of libraries that offer communication mechanisms for exchanging messages between Unix-based systems. The best known of these are the Parallel Virtual Machine (PVM) and the Message Passing Interface (MPI). In heterogeneous networks of computers, the processors are separated by relatively large distances and that has a negative impact on the performance of distributed applications. In fact, this approach is cost-effective only for large applications, in which a high volume of computation can be performed before more data is to be exchanged.

11.3 TYPES OF PARALLEL ARCHITECTURES

There are currently three leading architecture models. These are:

- The shared memory model.
- SIMD or data parallel models.
- The distributed memory message passing model.

A brief overview of the characteristics of each of the three groups follows. Emphasis is on the possible effects these characteristics have on the implementations of iterative methods.

11.3.1 SHARED MEMORY COMPUTERS

A shared memory computer has the processors connected to a large global memory with the same global view, meaning the address space is the same for all processors. One of the main benefits of shared memory models is that access to data depends very little on its location in memory. In a shared memory environment, transparent data access facilitates programming to a great extent. From the user's point of view, data are stored in a large global memory that is readily accessible to any processor. However, memory conflicts as well as the necessity to maintain data coherence can lead to degraded performance. In addition, shared memory computers cannot easily take advantage of data locality in problems which have an intrinsically local nature, as is the case with most discretized PDEs. Some current machines have a physically distributed memory but they are logically shared, i.e., each processor has the same view of the global address space.

There are two possible implementations of shared memory machines: (1) bus-based architectures, and (2) switch-based architecture. These two model architectures are illustrated in Figure 11.2 and Figure 11.3, respectively. So far, shared memory computers have been implemented more often with buses than with switching networks.

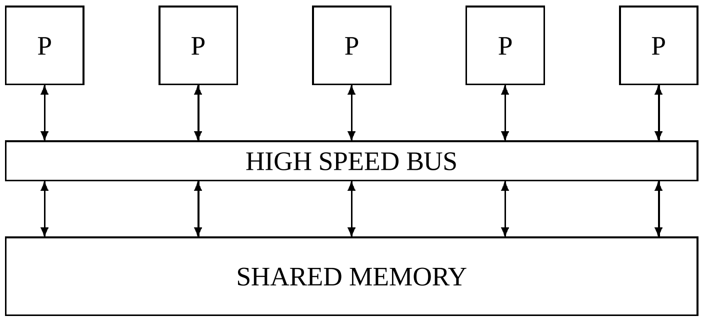


Figure 11.2 *A bus-based shared memory computer.*

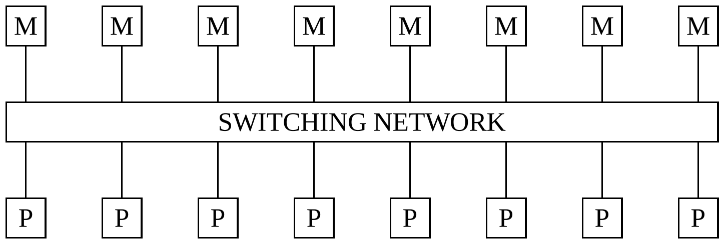


Figure 11.3 *A switch-based shared memory computer.*

Buses are the backbone for communication between the different units of most computers. Physically, a bus is nothing but a bundle of wires, made of either fiber or copper. These wires carry information consisting of data, control signals, and error correction bits. The speed of a bus, often measured in Megabytes per second and called the *bandwidth* of the bus, is determined by the number of lines in the bus and the clock rate. Often, the limiting factor for parallel computers based on bus architectures is the bus bandwidth rather than the CPU speed.

The primary reason why bus-based multiprocessors are more common than switch-based ones is that the hardware involved in such implementations is simple. On the other hand, the difficulty with bus-based machines is that the number of processors which can be connected to the memory will be small in general. Typically, the bus is timeshared, meaning slices of time are allocated to the different clients (processors, IO processors, etc.) that request its use.

In a multiprocessor environment, the bus can easily be saturated. Several remedies are possible. The first, and most common, remedy is to attempt to reduce traffic by adding *local memories* or *caches* attached to each processor. Since a data item used by a given processor is likely to be reused by the same processor in the next instructions, storing the data item in local memory will help reduce traffic in general. However, this strategy causes some difficulties due to the requirement to maintain *data coherence*. If Processor (A) reads some data from the shared memory, and Processor (B) modifies the same data in shared memory, immediately after, the result is two copies of the same data that have different values. A mechanism should be put in place to ensure that the most recent update of the data is always used. The additional overhead incurred by such memory coherence operations may well offset the savings involving memory traffic.

The main features here are the switching network and the fact that a global memory is shared by all processors through the switch. There can be p processors on one side connected to p memory units or banks on the other side. Alternative designs based on switches connect p processors to each other instead of p memory banks. The switching network can be a crossbar switch when the number of processors is small. A crossbar switch is analogous to a telephone switch board and allows p inputs to be connected to m outputs without conflict. Since crossbar switches for large numbers of processors are typically expensive they are replaced by multistage networks. Signals travel across a small number of stages consisting of an array of elementary switches, e.g., 2×2 or 4×4 switches.

There have been two ways of exploiting multistage networks. In *circuit switching* networks, the elementary switches are set up by sending electronic signals across all of the switches. The circuit is set up once in much the same way that telephone circuits are switched in a switchboard. Once the switch has been set up, communication between processors $P_1, \ldots, P_n$ is open to the memories

$$M_{\pi_1}, M_{\pi_2}, \ldots, M_{\pi_n},$$

in which π represents the desired permutation. This communication will remain functional for as long as it is not reset. Setting up the switch can be costly, but once it is done, communication can be quite fast. In *packet switching* networks, a packet of data will be given an address token and the switching within the different stages will be determined based on this address token. The elementary switches have to provide for buffering capabilities, since messages may have to be queued at different stages.

11.3.2 DISTRIBUTED MEMORY ARCHITECTURES

The *distributed memory* model refers to the distributed memory *message passing* architectures as well as to distributed memory SIMD computers. A typical distributed memory system consists of a large number of identical processors which have their own memories and which are interconnected in a regular topology. Examples are depicted in Figures 11.4 and 11.5. In these diagrams, each processor unit can be viewed actually as a complete processor with its own memory, CPU, I/O subsystem, control unit, etc. These processors are linked to a number of “neighboring” processors which in turn are linked to other neighboring processors, etc. In “Message Passing” models there is no global synchronization of the parallel tasks. Instead, computations are *data driven* because a processor performs a given task only when the operands it requires become available. The programmer must program all the data exchanges explicitly between processors.

In SIMD designs, a different approach is used. A host processor stores the program and each slave processor holds different data. The host then broadcasts instructions to processors which execute them simultaneously. One advantage of this approach is that there is no need for large memories in each node to store large programs since the instructions are broadcast one by one to all processors.

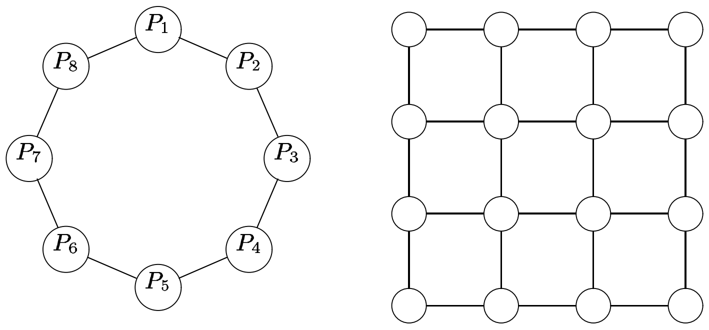


Figure 11.4 *An eight-processor ring (left) and a* 4×4 *multi-processor mesh (right).*

An important advantage of distributed memory computers is their ability to exploit locality of data in order to keep communication costs to a minimum. Thus, a two-dimensional processor grid such as the one depicted in Figure 11.4 is perfectly suitable for solving discretized elliptic Partial Differential Equations (e.g., by assigning each grid point to a corresponding processor) because some iterative methods for solving the resulting linear systems will require only interchange of data between adjacent grid points.

A good general purpose multiprocessor must have powerful *mapping capabilities* because it should be capable of easily emulating many of the common topologies such as 2-D and 3-D grids or linear arrays, FFT-butterflies, finite element meshes, etc.

Three-dimensional configurations are also popular. A massively parallel commercial computer based on a 3-D mesh, called T3D, is marketed by CRAY Research, Inc. For 2-D and 3-D grids of processors, it is common that processors on each side of the grid are connected to those on the opposite side. When these "wrap-around" connections are included, the topology is sometimes referred to as a torus.

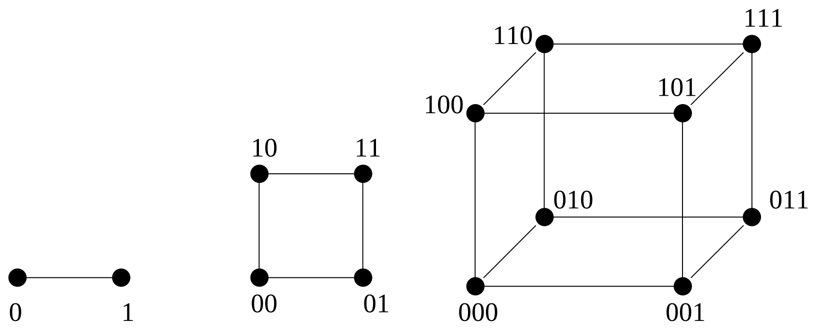


Figure 11.5 *The* n*-cubes of dimensions* $n = 1, 2, 3$.

Hypercubes are highly concurrent multiprocessors based on the binary n-cube topology which is well known for its rich interconnection capabilities. A parallel processor based on the n-cube topology, called a *hypercube* hereafter, consists of 2^n identical processors, interconnected with n neighbors. A 3-cube can be represented as an ordinary cube in three dimensions where the vertices are the $8 = 2^3$ nodes of the 3-cube; see Figure 11.5. More generally, one can construct an n-cube as follows: First, the 2^n nodes are labeled by the 2^n binary numbers from 0 to $2^n - 1$. Then a link between two nodes is drawn if and only if their binary numbers differ by one (and only one) bit.

The first property of an n-cube graph is that it can be constructed recursively from lower dimensional cubes. More precisely, consider two identical $(n-1)$-cubes whose vertices are labeled likewise from 0 to 2^{n-1}. By joining every vertex of the first $(n-1)$-cube to the vertex of the second having the same number, one obtains an n-cube. Indeed, it suffices to renumber the nodes of the first cube as $0 \wedge a_i$ and those of the second as $1 \ \wedge \ a_i$ where a_i is a binary number representing the two similar nodes of the $(n-1)$-cubes and where $\wedge$ denotes the concatenation of binary numbers.

Separating an n-cube into the subgraph of all the nodes whose leading bit is 0 and the subgraph of all the nodes whose leading bit is 1, the two subgraphs are such that each node of the first is connected to one node of the second. If the edges between these two graphs is removed, the result is 2 disjoint $(n-1)$-cubes. Moreover, generally, for a given numbering, the graph can be separated into two subgraphs obtained by considering all the nodes whose i^{th} bit is 0 and those whose i^{th} bit is 1. This will be called tearing along the i^{th} direction. Since there are n bits, there are n directions. One important consequence of this is that arbitrary meshes with dimension $\leq n$ can be mapped on hypercubes. However, the hardware cost for building a hypercube is high, because each node becomes difficult to design for larger dimensions. For this reason, recent commercial vendors have tended to prefer simpler solutions based on two- or three-dimensional meshes.

Distributed memory computers come in two different designs, namely, SIMD and MIMD. Many of the early projects have adopted the SIMD organization. For example, the historical ILLIAC IV Project of the University of Illinois was a machine based on a mesh topology where all processors execute the same instructions.

SIMD distributed processors are sometimes called array processors because of the regular arrays that they constitute. In this category, systolic arrays can be classified as an example of distributed computing. Systolic arrays are distributed memory computers in which each processor is a cell which is programmed (possibly micro-coded) to perform only one of a few operations. All the cells are synchronized and perform the same task. Systolic arrays are designed in VLSI technology and are meant to be used for special purpose applications, primarily in signal processing.

11.4 TYPES OF OPERATIONS

Now consider two prototype Krylov subspace techniques, namely, the preconditioned Conjugate Gradient method for the symmetric case and the preconditioned GMRES algorithm for the nonsymmetric case. For each of these two techniques, we analyze the types of operations that are performed. It should be emphasized that other Krylov subspace techniques require similar operations.

11.4.1 PRECONDITIONED CG

Consider Algorithm 9.1. The first step when implementing this algorithm on a high-performance computer is identifying the main operations that it requires. We distinguish five types of operations, which are:

1. Preconditioner setup.
2. Matrix vector multiplications.
3. Vector updates.
4. Dot products.
5. Preconditioning operations.

In the above list the potential bottlenecks are (1), setting up the preconditioner and (5), solving linear systems with M, i.e., the preconditioning operation. Section 11.6 discusses the implementation of traditional preconditioners, and the last two chapters are devoted to preconditioners that are specialized to parallel environments. Next come the matrix-by-vector products which deserve particular attention. The rest of the algorithm consists essentially of dot products and vector updates which do not cause significant difficulties in parallel machines, although inner products can lead to some loss of efficiency on certain types of computers with large numbers of processors.

11.4.2 GMRES

The only new operation here with respect to the Conjugate Gradient method is the orthogonalization of the vector Av_i against the previous v's. The usual way to accomplish this is via the modified Gram-Schmidt process, which is basically a sequence of subprocesses of the form:

- Compute $\alpha = (y, v)$.
- Compute $\hat{y} := y - \alpha v$.

This orthogonalizes a vector y against another vector v of norm one. Thus, the outer loop of the modified Gram-Schmidt is sequential, but the inner loop, i.e., each subprocess, can be parallelized by dividing the inner product and SAXPY operations among processors. Although this constitutes a perfectly acceptable approach for a small number of processors, the elementary subtasks may be too small to be efficient on a large number of processors. An alternative for this case is to use a standard Gram-Schmidt process with reorthogonalization. This replaces the previous sequential orthogonalization process by a matrix operation of the form $\hat{y} = y - VV^T y$, i.e., BLAS-1 kernels are replaced by BLAS-2 kernels.

Recall that the next level of BLAS, i.e., level 3 BLAS, exploits blocking in dense matrix operations in order to obtain performance on machines with hierarchical memories. Unfortunately, level 3 BLAS kernels cannot be exploited here because at every step, there is only one vector to orthogonalize against all previous ones. This may be remedied by using block Krylov methods.

11.4.3 VECTOR OPERATIONS

These are usually the simplest operations to implement on any computer. In many cases, compilers are capable of recognizing them and invoking the appropriate machine instructions, possibly vector instructions. In the specific case of CG-like algorithms, there are two types of operations: vector updates and dot products.

Vector Updates Operations of the form

```
y(1:n) = y(1:n) + a * x(1:n),
```

where a is a scalar and y and x two vectors, are known as *vector updates* or SAXPY operations. They are typically straightforward to implement in all three machine models discussed earlier. On an SIMD computer, the above code segment can be used on many of the recent systems and the compiler will translate it into the proper parallel version. The above line of code is written in FORTRAN 90, which is the prototype programming language for this type of computers. On shared memory computers, we can simply write the usual FORTRAN loop, possibly in the above FORTRAN 90 style on some computers, and the compiler will translate it again in the appropriate parallel executable code.

On distributed memory computers, some assumptions must be made about the way in which the vectors are distributed. The main assumption is that the vectors x and y are distributed in the same manner among the processors, meaning the indices of the components of any vector that are mapped to a given processor are the same. In this case, the vector-update operation will be translated into p independent vector updates, requiring no communication. Specifically, if $nloc$ is the number of variables local to a given processor, this processor will simply execute a vector loop of the form

```
y(1:nloc) = y(1:nloc) + a * x(1:nloc)
```

and all processors will execute a similar operation simultaneously.

Dot products A number of operations use all the components of a given vector to compute a single floating-point result which is then needed by all processors. These are termed *Reduction Operations* and the dot product is the prototype example. A distributed version of the dot-product is needed to compute the inner product of two vectors x and y that are distributed the same way across the processors. In fact, to be more specific, this distributed dot-product operation should compute the inner product $t = x^T y$ of these two vectors and then make the result t available in each processor. Typically, this result is needed to perform vector updates or other operations in each node. For a large number of processors, this sort of operation can be demanding in terms of communication costs. On the other hand, parallel computer designers have become aware of their importance and are starting to provide hardware and software support for performing *global reduction operations* efficiently. Reduction operations that can be useful include global sums, global max/min calculations, etc. A commonly adopted convention provides a single subroutine for all these operations, and passes the type of operation to be performed (add, max, min, multiply,. . .) as one of the arguments. With this in mind, a distributed dot-product function can be programmed roughly as follows.

```
      function distdot(nloc, x, incx, y, incy)
      integer nloc
      real*8 x(nloc), y(nloc)
      tloc = DDOT (nloc, x, incx, y, incy)
      distdot = REDUCE(tloc,'add')
      end
```

The function DDOT performs the usual BLAS-1 dot product of *x* and y with strides *incx* and *incy*, respectively. The REDUCE operation, which is called with “add” as the operation-type parameter, sums all the variables “tloc” from each processor and put the resulting global sum in the variable $distdot$ in each processor.

11.4.4 REVERSE COMMUNICATION

To conclude this section, the following important observation can be made regarding the practical implementation of Krylov subspace accelerators, such as PCG or GMRES. The only operations that involve communication are the dot product, the matrix-by-vector product, and, potentially, the preconditioning operation. There is a mechanism for delegating the last two operations to a calling program, outside of the Krylov accelerator. The result of this is that the Krylov acceleration routine will be free of any matrix data structures as well as communication calls. This makes the Krylov routines portable, except for the possible redefinition of the inner product *distdot*.

This mechanism, particular to FORTRAN programming, is known as *reverse communication*. Whenever a matrix-by-vector product or a preconditioning operation is needed, the subroutine is exited and the calling program unit performs the desired operation. Then the subroutine is called again, after placing the desired result in one of its vector arguments.

A typical execution of a flexible GMRES routine with reverse communication is shown in the code segment below. The integer parameter *icode* indicates the type of operation needed by the subroutine. When *icode* is set to one, then a preconditioning operation must be applied to the vector $wk1$. The result is copied in $wk2$ and FGMRES is called again. If it is equal to two, then the vector $wk1$ must be multiplied by the matrix A. The result is then copied in $wk2$ and FGMRES is called again.

```
    icode = 0
 1  continue
    call fgmres (n,im,rhs,sol,i,vv,w,wk1, wk2,eps,
   *             maxits,iout,icode)
    if (icode .eq. 1) then
       call  precon(n, wk1, wk2) ! user's preconditioner
       goto 1
    else if (icode .eq. 2) then
       call  matvec (n,wk1, wk2) ! user's matvec
       goto 1
    endif
```

Reverse communication enhances the flexibility of the FGMRES routine substantially. For example, when changing preconditioners, we can iterate on a coarse mesh and do the necessary interpolations to get the result in $wk2$ in a given step and then iterate on the fine mesh in the following step. This can be done without having to pass any data regarding the matrix or the preconditioner to the FGMRES accelerator.

Note that the purpose of reverse communication simply is to avoid passing data structures related to the matrices, to the accelerator FGMRES. The problem is that these data structures are not fixed. For example, it may be desirable to use different storage formats for different architectures. A more elegant solution to this problem is *Object-Oriented Programming*. In Object-Oriented Programming languages such as C++, a class can be declared, e.g., a class of sparse matrices, and operators can be defined on them. Data structures are not passed to these operators. Instead, the implementation will recognize the types of the operands and invoke the proper functions. This is similar to what exists currently for arithmetic. For operation $s = z + y$, the compiler will recognize what type of operand is involved and invoke the proper operation, either integer, double real, or complex, etc.

11.5 MATRIX-BY-VECTOR PRODUCTS

Matrix-by-vector multiplications (sometimes called “Matvecs” for short) are relatively easy to implement efficiently on high performance computers. For a description of storage formats for sparse matrices, see Chapter 3. We will first discuss matrix-by-vector algorithms without consideration of sparsity. Then we will cover sparse Matvec operations for a few different storage formats.

11.5.1 THE CASE OF DENSE MATRICES

The computational kernels for performing sparse matrix operations such as matrix-by--vector products are intimately associated with the data structures used. However, there are a few general approaches that are common to different algorithms for matrix-by-vector products which can be described for dense matrices. Two popular ways of performing these operations are (1) the inner product form described in Algorithm 11.1, and (2) the SAXPY form described by Algorithm 11.2.

ALGORITHM 11.1: Dot Product Form – Dense Case

```
1. Do i = 1, n
2.     y(i) = dotproduct(a(i,1:n),x(1:n))
3. EndDo
```

The dot product operation *dotproduct(v(1:n),w(1:n))* computes the dot product of the two vectors *v* and *w* of length n each. If there is no ambiguity on the bounds, we simply write *dotproduct(v,w)*. The above algorithm proceeds by rows. It computes the dot-product of row i of the matrix A with the vector x and assigns the result to $y(i)$. The next algorithm uses columns instead and results in the use of the SAXPY operations.

ALGORITHM 11.2: SAXPY Form – Dense Case

```
1. y(1:n) = 0.0
2. Do j = 1, n
3.    y(1:n) = y(1:n) + x(j) * a(1:n,j)
4. EndDo
```

The SAXPY form of the Matvec operation computes the result $y = Ax$ as a linear combination of the columns of the matrix A. A third possibility consists of performing the product by diagonals. This option bears no interest in the dense case, but it is at the basis of many important matrix-by-vector algorithms in the sparse case.

ALGORITHM 11.3: DIA Form – Dense Case

```
1. y(1:n) = 0
2. Do k = – n+1, n – 1
3.    Do i = 1 – min(k,0), n – max(k,0)
4.         y(i) = y(i) + a(i,k+i)*x(k+i)
5.    EndDo
6. EndDo
```

The product is performed by diagonals, starting from the leftmost diagonal whose offset is $-n + 1$ to the rightmost diagonal whose offset is $n - 1$.

11.5.2 THE CSR AND CSC FORMATS

One of the most general schemes for storing sparse matrices is the Compressed Sparse Row storage format described in Chapter 3. Recall that the data structure consists of three arrays: a real array *A(1:nnz)* to store the nonzero elements of the matrix row-wise, an integer array *JA(1:nnz)* to store the column positions of the elements in the real array *A*, and, finally, a pointer array *IA(1:n+1)*, the i-th entry of which points to the beginning of the i-th row in the arrays *A* and *JA*. To perform the matrix-by-vector product $y = Ax$ in parallel using this format, note that each component of the resulting vector y can be computed independently as the dot product of the i-th row of the matrix with the vector x. This yields the following sparse version of Algorithm 11.1 for the case where the matrix is stored in CSR format.

ALGORITHM 11.4: CSR Format – Dot Product Form

```
1. Do i = 1, n
2.    k1 = ia(i)
3.    k2 = ia(i+1)-1
4.    y(i) = dotproduct(a(k1:k2),x(ja(k1:k2)))
5. EndDo
```

Line 4 of the above algorithm computes the dot product of the vector with components *a(k1), a(k1+1), $\cdots$, a(k2)* with the vector with components *x(ja(k1)), x(ja(k1+1)), $\cdots$, x(ja(k2))*.

The fact that the outer loop can be performed in parallel can be exploited on any parallel platform. On some shared-memory machines, the synchronization of this outer loop is inexpensive and the performance of the above program can be excellent. On distributed memory machines, the outer loop can be split in a number of steps to be executed on each processor. Thus, each processor will handle a few rows that are assigned to it. It is common to assign a certain number of rows (often contiguous) to each processor and to also assign the component of each of the vectors similarly. The part of the matrix that is needed is loaded in each processor initially. When performing a matrix-by-vector product, interprocessor communication will be necessary to get the needed components of the vector x that do not reside in a given processor. This important case will return in Section 11.5.6.

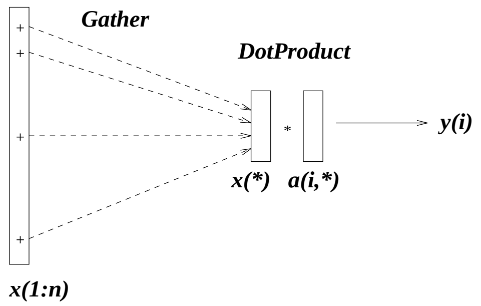


Figure 11.6 *Illustration of the row-oriented matrix-by-vector multiplication.*

The indirect addressing involved in the second vector in the dot product is called a *gather* operation. The vector *x(ja(k1:k2))* is first "gathered" from memory into a vector of contiguous elements. The dot product is then carried out as a standard dot-product operation between two dense vectors. This is illustrated in Figure 11.6.

Example 11.1 This example illustrates the use of scientific libraries for performing sparse matrix operations. If the pseudo-code for Algorithm 11.4 is compiled as it is on the Connection Machine, in CM-FORTRAN (Thinking Machine's early version of FORTRAN 90), the resulting computations will be executed on the front-end host of the CM-2 or the Control Processor (CP) of the CM-5, rather than on the PEs. This is due to the fact that the code does not involve any vector constructs. The scientific library (CMSSL) provides *gather* and *scatter* operations as well as *scan_add* operations which can be exploited to implement this algorithm more efficiently as is show in the following code segment:

```
y = 0.0
call sparse_util_gather ( tmp, x, gather_trace, . . .)
tmp = a*tmp
call cmf_scan_add (tmp, tmp, cmf_upward, cmf_inclusive, . . .)
call sparse_util_scatter ( y, scatter_pointer, tmp,
         scatter_trace, . . . )
```

The *sparse_util_gather* routine is first called to gather the corresponding entries from the vector x into a temporary array tmp, then the multiplications are carried out element-by-element in parallel. The *cmf_scan_add* routine from the CM Fortran Utility Library is used to perform the summation for each row. Finally, the call to *sparse_util_scatter* copies the results. Segmented scan-adds are particularly useful for implementing sparse matrix-by-vector products when they are provided as part of the libraries. Note that the *sparse_util-_gather_setup* and *sparse_util_scatter_setup* routines must be called to compute the communication patterns, *gather_trace* and *scatter_trace*, before this algorithm is called. These tend to be expensive operations.

Now assume that the matrix is stored by columns (CSC format). The matrix-by-vector product can be performed by the following algorithm which is a sparse version of Algorithm 11.2.

ALGORITHM 11.5: CSC Format – SAXPY Form

```
1. y(1:n) = 0.0
2. Do i = 1, n
3.    k1 = ia(i)
4.    k2 = ia(i + 1)-1
5.         y(ja(k1:k2)) = y(ja(k1:k2)) + x(j) * a(k1:k2)
6. EndDo
```

The above code initializes y to zero and then adds the vectors $x(j) \times a(1:n,j)$ for $j = 1,\ldots,n$ to it. It can also be used to compute the product of the *transpose* of a matrix by a vector, when the matrix is stored (row-wise) in the CSR format. Normally, the vector *y(ja(k1:k2))* is gathered and the SAXPY operation is performed in vector mode. Then the resulting vector is "scattered" back into the positions *ja(*)*, by what is called a *Scatter* operation. This is illustrated in Figure 11.7.

A major difficulty with the above FORTRAN program is that it is intrinsically sequential. First, the outer loop is not parallelizable as it is, but this may be remedied as will be seen shortly. Second, the inner loop involves writing back results of the right-hand side into memory positions that are determined by the indirect address function *ja*. To be correct, *y(ja(1))* must be copied first, followed by *y(ja(2))*, etc. However, if it is known that the mapping *ja(i)* is one-to-one, then the order of the assignments no longer matters. Since compilers are not capable of deciding whether this is the case, a compiler directive from the user is necessary for the Scatter to be invoked.

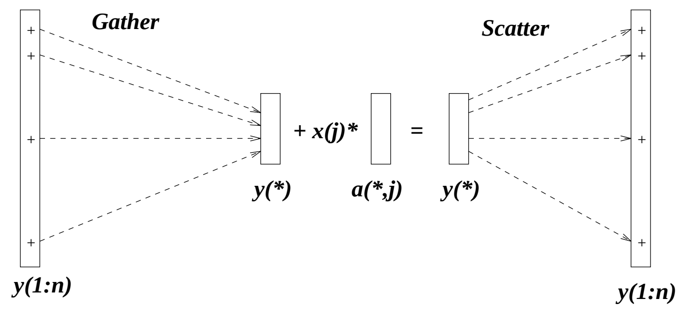


Figure 11.7 *Illustration of the column-oriented matrix-by-vector multiplication.*

Going back to the outer loop, p subsums can be computed (independently) into p separate temporary vectors and then these p subsums can be added at the completion of all these partial sums to obtain the final result. For example, an optimized version of the previous algorithm can be implemented as follows:

ALGORITHM 11.6: CSC Format – Split SAXPY Form

```
1. tmp(1:n,1:p) = 0.0
2. Do m=1, p
3.    Do j = m, n, p
4.       k1 = ia(j)
5.       k2 = ia(j + 1)-1
6.       tmp(ja(k1:k2),j) = tmp(ja(k1:k2),j) + x(j) * a(k1:k2)
7.    EndDo
8. EndDo
9. y(1:n) = SUM(tmp,dim=2)
```

The *SUM* across the second dimension at the end of the algorithm constitutes additional work, but it is highly vectorizable and parallelizable.

11.5.3 MATVECS IN THE DIAGONAL FORMAT

The above storage schemes are general but they do not exploit any special structure of the matrix. The *diagonal storage format* was one of the first data structures used in the context of high performance computing to take advantage of special sparse structures. Often, sparse matrices consist of a small number of diagonals in which case the matrix-by-vector product can be performed by diagonals as in Algorithm 11.3. For sparse matrices, most of the $2n-1$ diagonals invoked in the outer loop of Algorithm 11.3 are zero. There are again different variants of Matvec algorithms for the diagonal format, related to different orderings of the loops in the basic FORTRAN program. Recall that the matrix is stored in a rectangular array *diag(1:n,1:ndiag)* and the offsets of these diagonals from the main diagonal may be stored in a small integer array *offset(1:ndiag)*. Consider a “dot-product” variant first.

ALGORITHM 11.7: DIA Format – Dot Product Form

```
1. Do i = 1, n
2.    tmp = 0.0d0
3.    Do j = 1, ndiag
4.       tmp = tmp + diag(i,j)*x(i+offset(j))
5.    EndDo
6.    y(i) = tmp
7. EndDo
```

In a second variant, the vector y is initialized to zero, and then x is multiplied by each of the diagonals and the separate results are added to y. The innermost loop in this computation is sometimes called a *Triad* operation.

ALGORITHM 11.8: Matvec in Triad Form

```
1. y = 0.0d0
2. Do j = 1, ndiag
3.    joff = offset(j)
4.    i1 = max(1, 1-offset(j))
5.    i2 = min(n, n-offset(j))
6.    y(i1:i2) = y(i1:i2) + diag(i1:i2,j)*x(i1+joff:i2+joff)
7. EndDo
```

Good speeds can be reached on vector machines when the matrix is large enough.

One drawback with diagonal storage is that it is not general enough. For general sparse matrices, we can either generalize the diagonal storage scheme or reorder the matrix in order to obtain a diagonal structure. The simplest generalization is the Ellpack-Itpack Format.

11.5.4 THE ELLPACK-ITPACK FORMAT

The Ellpack-Itpack (or Ellpack) format is of interest only for matrices whose maximum number of nonzeros per row, *jmax,* is small. The nonzero entries are stored in a real array *ae(1:n,1:jmax)*. Along with this is integer array *jae(1:n,1:jmax)* which stores the column indices of each corresponding entry in *ae*. Similar to the diagonal scheme, there are also two basic ways of implementing a matrix-by-vector product when using the Ellpack format. We begin with an analogue of Algorithm 11.7.

ALGORITHM 11.9: Ellpack Format – Dot-Product Form

```
1. Do i = 1, n
2.    yi = 0
3.    Do j = 1, ncol
4.       yi = yi + ae(j,i) * x(jae(j,i))
5.    EndDo
```

```
6.      y(i) = yi
7.   EndDo
```

In data-parallel mode, the above algorithm can be implemented by using a temporary two-dimensional array to store the values $x(ja(j,i))$, and then performing a pointwise array product of a and this two-dimensional array. The result is then summed along the rows

forall (i=1:n, j=1:ncol) tmp(i,j) = x(jae(i,j))
*y = SUM(ae*tmp, dim=2).*

The FORTRAN *forall* construct performs the operations as controlled by the loop heading, in parallel. Alternatively, use of the temporary array can be avoided by recoding the above lines as follows:

*forall (i = 1:n) y(i) = SUM(ae(i,1:ncol)*x(jae(i,1:ncol)))* .

The main difference between these loops and the previous ones for the diagonal format is the presence of indirect addressing in the innermost computation. A disadvantage of the Ellpack format is that if the number of nonzero elements per row varies substantially, many zero elements must be stored unnecessarily. Then the scheme becomes inefficient. As an extreme example, if all rows are very sparse except for one of them which is full, then the arrays *ae, jae* must be full $n \times n$ arrays, containing mostly zeros. This is remedied by a variant of the format which is called the *jagged diagonal format*.

11.5.5 THE JAGGED DIAGONAL FORMAT

A more general alternative to the diagonal or Ellpack format is the Jagged Diagonal (JAD) format. This can be viewed as a generalization of the Ellpack-Itpack format which removes the assumption on the fixed length rows. To build the jagged diagonal structure, start from the CSR data structure and sort the rows of the matrix by decreasing number of nonzero elements. To build the first "jagged diagonal" (j-diagonal), extract the first element from each row of the CSR data structure. The second jagged diagonal consists of the second elements of each row in the CSR data structure. The third, fourth, . . ., jagged diagonals can then be extracted in the same fashion. The lengths of the successive j-diagonals decreases. The number of j-diagonals that can be extracted is equal to the number of nonzero elements of the first row of the permuted matrix, i.e., to the largest number of nonzero elements per row. To store this data structure, three arrays are needed: a real array *DJ* to store the values of the jagged diagonals, the associated array *JDIAG* which stores the column positions of these values, and a pointer array *IDIAG* which points to the beginning of each j-diagonal in the *DJ, JDIAG* arrays.

Example 11.2 Consider the following matrix and its sorted version PA:

$$A = \begin{pmatrix} 1. & 0. & 2. & 0. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 0. & 6. & 7. & 0. & 8. \\ 0. & 0. & 9. & 10. & 0. \\ 0. & 0. & 0. & 11. & 12. \end{pmatrix} \quad \rightarrow \quad PA = \begin{pmatrix} 3. & 4. & 0. & 5. & 0. \\ 0. & 6. & 7. & 0. & 8. \\ 1. & 0. & 2. & 0. & 0. \\ 0. & 0. & 9. & 10. & 0. \\ 0. & 0. & 0. & 11. & 12. \end{pmatrix}$$

The rows of PA have been obtained from those of A by sorting them by number of nonzero elements, from the largest to the smallest number. Then the JAD data structure for A is as follows:

DJ	3.	6.	1.	9.	11.	4.	7.	2.	10.	12.	5.	8.
JDIAG	1	2	1	3	4	2	3	3	4	5	4	5
IDIAG	1	6	11	13								

Thus, there are two j-diagonals of full length (five) and one of length two.

A matrix-by-vector product with this storage scheme can be performed by the following code segment.

```
1. Do j=1, ndiag
2.    k1 = idiag(j)
3.    k2 = idiag(j+1) – 1
4.    len = idiag(j+1) – k1
5.    y(1:len) = y(1:len) + dj(k1:k2)*x(jdiag(k1:k2))
6. EndDo
```

Since the rows of the matrix A have been permuted, the above code will compute PAx, a permutation of the vector Ax, rather than the desired Ax. It is possible to permute the result back to the original ordering after the execution of the above program. This operation can also be performed until the final solution has been computed, so that only two permutations on the solution vector are needed, one at the beginning and one at the end. For preconditioning operations, it may be necessary to perform a permutation before or within each call to the preconditioning subroutines. There are many possible variants of the jagged diagonal format. One variant which does not require permuting the rows is described in Exercise 8.

11.5.6 THE CASE OF DISTRIBUTED SPARSE MATRICES

Given a sparse linear system to be solved on a distributed memory environment, it is natural to map pairs of equations-unknowns to the same processor in a certain predetermined way. This mapping can be determined automatically by a graph partitioner or it can be assigned ad hoc from knowledge of the problem. Assume that there is a convenient partitioning of the adjacency graph. Without any loss of generality, the matrix under consideration can be viewed as originating from the discretization of a Partial Differential Equation on a certain domain. This is illustrated in Figure 11.8. Initially, assume that each subgraph (or subdomain, in the PDE literature) is assigned to a different processor, although this restriction can be relaxed, i.e., a processor can hold several subgraphs to increase parallelism.

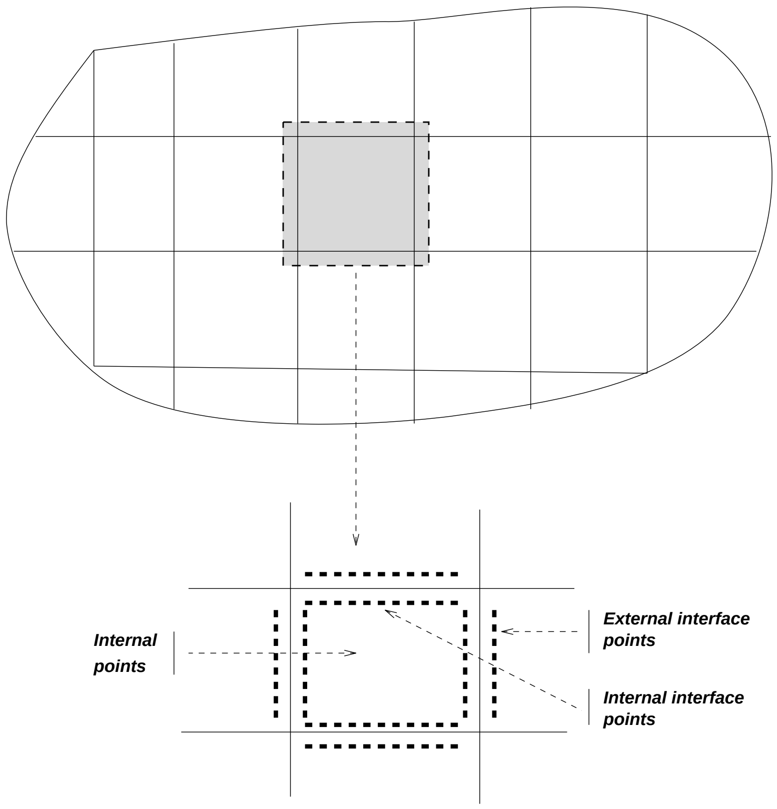


Figure 11.8 *Decomposition of physical domain or adjacency graph and the local data structure.*

A local data structure must be set up in each processor (or subdomain, or subgraph) which will allow the basic operations such as (global) matrix-by-vector products and preconditioning operations to be performed efficiently. The only assumption to make regarding the mapping is that if row number i is mapped into processor p, then so is the unknown i, i.e., the matrix is distributed row-wise across the processors according to the distribution of the variables. The graph is assumed to be undirected, i.e., the matrix has a symmetric pattern.

It is important to "preprocess the data" in order to facilitate the implementation of the communication tasks and to gain efficiency during the iterative process. The preprocessing requires setting up the following: information *in each processor*.

1. List of processors with which communication will take place. These are called "neighboring processors" although they may not be physically nearest neighbors.
2. List of local nodes that are coupled with external nodes. These are the *local interface nodes*.
3. Local representation of the distributed matrix in each processor.

In order to perform a matrix-by-vector product with a distributed sparse matrix, the matrix consisting of rows that are local to a given processor must be multiplied by some global vector v. Some components of this vector will be local, and some components must be brought from external processors. These external variables correspond to interface points belonging to adjacent subdomains. When performing a matrix-by-vector product, neighboring processors must exchange values of their adjacent interface nodes.

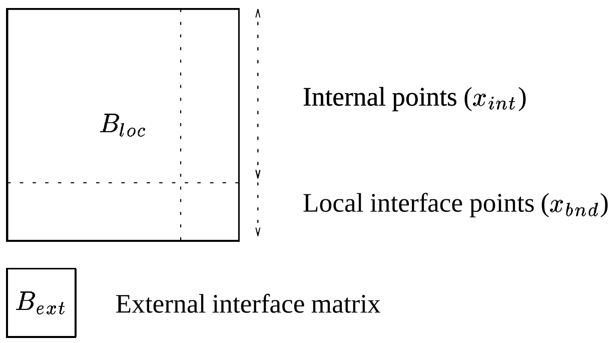

B_{loc}
Internal points (x_{int})
Local interface points (x_{bnd})
B_{ext}
External interface matrix

Figure 11.9 *The local matrices and data structure associated with each subdomain.*

Let A_{loc} be the local part of the matrix, i.e., the (rectangular) matrix consisting of all the rows that are mapped to *myproc*. Call B_{loc} the "diagonal block" of A located in A_{loc}, i.e., the submatrix of A_{loc} whose nonzero elements a_{ij} are such that j is a local variable. Similarly, call B_{ext} the "offdiagonal" block, i.e., the submatrix of A_{loc} whose nonzero elements a_{ij} are such that j is *not* a local variable. To perform a matrix-by-vector product, start multiplying the diagonal block B_{loc} by the local variables. Then, multiply the external variables by the sparse matrix B_{ext}. Notice that since the external interface points are not coupled with local internal points, only the rows $n_{int} + 1$ to n_{nloc} in the matrix B_{ext} will have nonzero elements. Thus, the matrix-by-vector product can be separated into two such operations, one involving only the local variables and the other involving external variables. It is necessary to construct these two matrices and define a local numbering of the local variables in order to perform the two matrix-by-vector products efficiently each time.

To perform a global matrix-by-vector product, with the distributed data structure described above, each processor must perform the following operations. First, multiply the local variables by the matrix B_{loc}. Second, obtain the external variables from the neighboring processors in a certain order. Third, multiply these by the matrix B_{ext} and add the resulting vector to the one obtained from the first multiplication by B_{loc}. Note that the first and second steps can be done in parallel. With this decomposition, the global matrix-by-vector product can be implemented as indicated in Algorithm 11.10 below. In what follows, x_{loc} is a vector of variables that are local to a given processor. The components corresponding to the local interface points (ordered to be the last components in x_{loc} for convenience) are called x_{bnd}. The external interface points, listed in a certain order, constitute a vector which is called x_{ext}. The matrix B_{loc} is a sparse $nloc \times nloc$ matrix which represents the restriction of A to the local variables x_{loc}. The matrix B_{ext} operates on the external variables x_{ext} to give the correction which must be added to the vector $B_{loc}x_{loc}$ in order to obtain the desired result $(Ax)_{loc}$.

ALGORITHM 11.10: Distributed Sparse Matrix Product Kernel

```
1. Exchange interface data, i.e.,
2.     Scatter $x_{bnd}$ to neighbors and
3.     Gather $x_{ext}$ from neighbors
4. Do Local Matvec: $y = B_{loc}x_{loc}$
5. Do External Matvec: $y = y + B_{ext}x_{ext}$
```

An important observation is that the matrix-by-vector products in lines 4 and 5 can use any convenient data structure that will improve efficiency by exploiting knowledge on the local architecture. An example of the implementation of this operation is illustrated next:

```
call bdxchg(nloc,x,y,nproc,proc,ix,ipr,type,xlen,iout)
y(1:nloc) = 0.0
call amux1 (nloc,x,y,aloc,jaloc,ialoc)
nrow = nloc – nbnd + 1
call amux1(nrow,x,y(nbnd),aloc,jaloc,ialoc(nloc+1))
```

In the above code segment, *bdxchg* is the only routine requiring communication. Its purpose is to exchange interface values between nearest neighbor processors. The first call to *amux1* performs the operation $y := y + B_{loc}x_{loc}$, where y has been initialized to zero prior to the call. The second call to *amux1* performs $y := y + B_{ext}x_{ext}$. Notice that the data for the matrix B_{ext} is simply appended to that of B_{loc}, a standard technique used for storing a succession of sparse matrices. The B_{ext} matrix acts only on the subvector of x which starts at location $nbnd$ of x. The size of the B_{ext} matrix is $nrow = nloc - nbnd + 1$.

11.6 STANDARD PRECONDITIONING OPERATIONS

Each step of a preconditioned iterative method requires the solution of a linear system of equations

$$Mz = y.$$

This section only considers those traditional preconditioners, such as ILU or SOR or SSOR, in which the solution with M is the result of solving triangular systems. Since these are commonly used, it is important to explore ways to implement them efficiently in a parallel environment. We only consider lower triangular systems of the form

$$Lx = b. \tag{11.1}$$

Without loss of generality, it is assumed that L is unit lower triangular.

11.6.1 PARALLELISM IN FORWARD SWEEPS

Typically in solving a lower triangular system, the solution is overwritten onto the right-hand side on return. In other words, there is one array x for both the solution and the right-hand side. Therefore, the forward sweep for solving a lower triangular system with coefficients $al(i,j)$ and right-hand-side x is as follows.

ALGORITHM 11.11: Sparse Forward Elimination

```
1. Do i=2, n
2.    For (all j such that al(i,j) is nonzero) Do:
3.       x(i) := x(i) – al(i,j) * x(j)
4.    EndDo
5. EndDo
```

Assume that the matrix is stored row wise in the general Compressed Sparse Row (CSR) format, except that the diagonal elements (ones) are not stored. Then the above algorithm translates into the following code segment:

```
1. Do i=2, n
2.    Do j=ial(i), ial(i+1) – 1
3.       x(i)=x(i) – al(j) * x(jal(j))
4.    EndDo
5. EndDo
```

The outer loop corresponding to the variable i is sequential. The j loop is a sparse dot product of the i^{th} row of L and the (dense) vector x. This dot product may be split among the processors and the partial results may be added at the end. However, the length of the vector involved in the dot product is typically short. So, this approach is quite inefficient in general. We examine next a few alternative approaches. The regularly structured and the irregularly structured cases are treated separately.

11.6.2 LEVEL SCHEDULING: THE CASE OF 5-POINT MATRICES

First, consider an example which consists of a 5-point matrix associated with a 4×3 mesh as represented in Figure 11.10. The lower triangular matrix associated with this mesh is represented in the left side of Figure 11.10. The stencil represented in the right side of Figure 11.10 establishes the data dependence between the unknowns in the lower triangular system solution when considered from the point of view of a grid of unknowns. It tells us that in order to compute the unknown in position (i,j), only the two unknowns in positions $(i-1,j)$ and $(i,j-1)$ are needed . The unknown x_{11} does not depend on any other variable and can be computed first. Then the value of x_{11} can be used to get $x_{1,2}$ and $x_{2,1}$ simultaneously. Then these two values will in turn enable $x_{3,1}$, $x_{2,2}$ and $x_{1,3}$ to be obtained simultaneously, and so on. Thus, the computation can proceed in wavefronts. The steps for this wavefront algorithm are shown with dashed lines in Figure 11.10. Observe that the maximum degree of parallelism (or vector length, in the case of vector processing) that can be reached is the minimum of n_x, n_y, the number of mesh points in the x and y directions, respectively, for 2-D problems. For 3-D problems, the parallelism is of the order of the maximum size of the sets of domain points $x_{i,j,k}$, where $i + j + k = lev$, a constant level lev. It is important to note that there is little parallelism or vectorization at the beginning and at the end of the sweep. The degree of parallelism is equal to one initially, and then increases by one for each wave reaching its maximum, and then decreasing back down to one at the end of the sweep. For example, for a 4×3 grid, the levels (sets of equations that can be solved in parallel) are $\{1\}$, $\{2, 5\}$, $\{3, 6, 9\}$, $\{4, 7, 10\}$, $\{8, 11\}$, and finally $\{12\}$. The first and last few steps may take a heavy toll on achievable speed-ups.

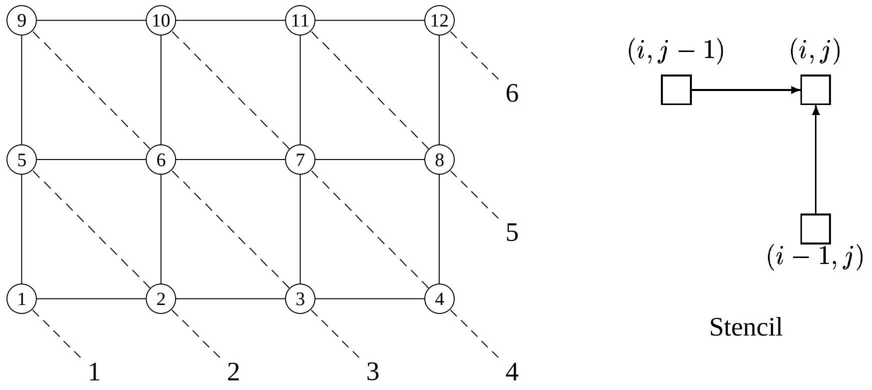


Figure 11.10 *Level scheduling for a* 4×3 *grid problem.*

The idea of proceeding by *levels* or *wavefronts* is a natural one for finite difference matrices on rectangles. Discussed next is the more general case of irregular matrices, a textbook example of scheduling, or *topological sorting*, and is well known in different forms to computer scientists.

11.6.3 LEVEL SCHEDULING FOR IRREGULAR GRAPHS

The simple scheme described above can be generalized for irregular grids. The objective of the technique, called *level scheduling*, is to group the unknowns in subsets so that they can be determined simultaneously. To explain the idea, consider again Algorithm 11.11 for solving a unit lower triangular system. The i-th unknown can be determined once all the other ones that participate in equation i become available. In the i-th step, all unknowns j that $al(i, j) \neq 0$ must be known. To use graph terminology, these unknowns are *adjacent* to unknown number i. Since L is lower triangular, the adjacency graph is a directed acyclic graph. The edge $j \rightarrow i$ in the graph simply indicates that x_j must be known before x_i can be determined. It is possible and quite easy to find a labeling of the nodes that satisfy the property that if $label(j) < label(i)$, then task j must be executed before task i. This is called a topological sorting of the unknowns.

The first step computes x_1 and any other unknowns for which there are no predecessors in the graph, i.e., all those unknowns x_i for which the offdiagonal elements of row i are zero. These unknowns will constitute the elements of the first level. The next step computes in parallel all those unknowns that will have the nodes of the first level as their (only) predecessors in the graph. The following steps can be defined similarly: The unknowns that can be determined at step l are all those that have as predecessors equations that have been determined in steps $1, 2, \ldots, l-1$. This leads naturally to the definition of a *depth* for each unknown. The *depth* of a vertex is defined by performing the following loop for $= 1, 2, \ldots, n$, after initializing $depth(j)$ to zero for all j.

$$depth(i) = 1 + \max_j \{depth(j), \text{ for all } j \text{ such that } al(i,j) \neq 0\}.$$

By definition, a *level* of the graph is the set of nodes with the same depth. A data structure for the levels can be defined: A permutation $q(1:n)$ defines the new ordering and $level(i), i = 1, \cdots, nlev+1$ points to the beginning of the i-th level in that array.

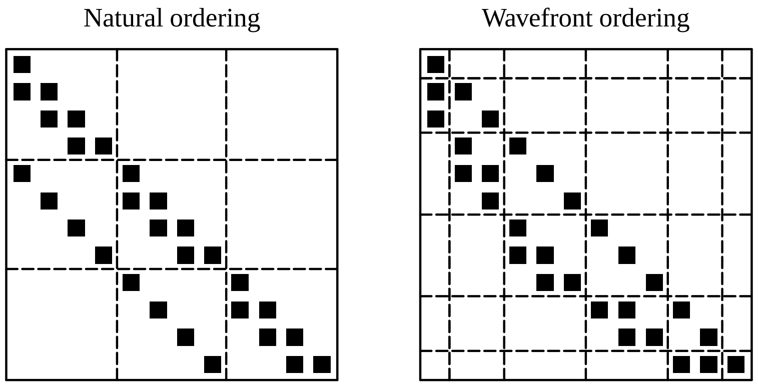


Figure 11.11 *Lower triangular matrix associated with mesh of Figure 11.10.*

Once these level sets are found, there are two different ways to proceed. The permutation vector q can be used to permute the matrix according to the new order. In the 4×3 example mentioned in the previous subsection, this means renumbering the variables $\{1\}$, $\{2,5\}$, $\{3,6,9\}, \ldots$, consecutively, i.e., as $\{1, 2, 3, \ldots\}$. The resulting matrix after the permutation is shown in the right side of Figure 11.11. An alternative is simply to keep the permutation array and use it to identify unknowns that correspond to a given level in the solution. Then the algorithm for solving the triangular systems can be written as follows, assuming that the matrix is stored in the usual row sparse matrix format.

ALGORITHM 11.12: Forward Elimination with Level Scheduling

```
1. Do lev=1, nlev
2.    j1 = level(lev)
3.    j2 = level(lev+1) – 1
4.    Do k = j1, j2
5.         i = q(k)
6.         Do j= ial(i), ial(i+1) – 1
```

```
 7.            x(i) = x(i) – al(j) * x(jal(j))
 8.         EndDo
 9.      EndDo
10.   EndDo
```

An important observation here is that the outer loop, which corresponds to a level, performs an operation of the form

$$x := x - Bx$$

where B is a submatrix consisting only of the rows of level lev, and excluding the diagonal elements. This operation can in turn be optimized by using a proper data structure for these submatrices. For example, the JAD data structure can be used. The resulting performance can be quite good. On the other hand, implementation can be quite involved since two embedded data structures are required.

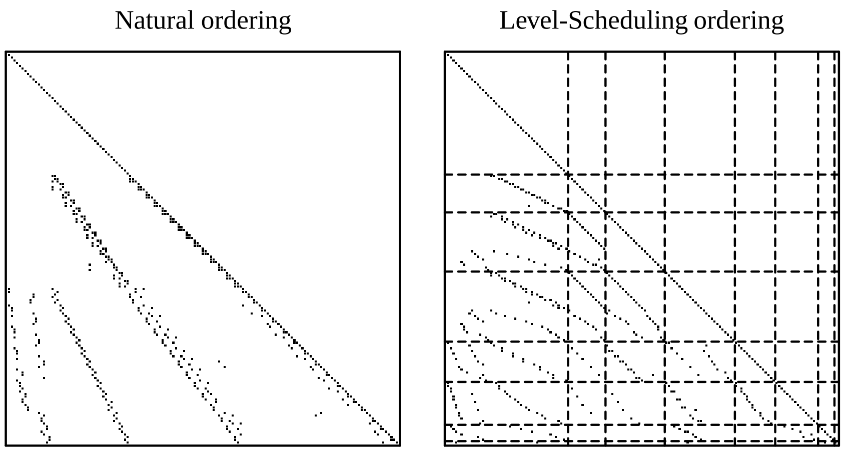


Figure 11.12 *Lower-triangular matrix associated with a finite element matrix and its level-ordered version.*

Example 11.3 Consider a finite element matrix obtained from the example shown in Figure 3.1. After an additional level of refinement, done in the same way as was described in Chapter 3, the resulting matrix, shown in the left part of Figure 11.12, is of size $n = 145$. In this case, 8 levels are obtained. If the matrix is reordered by levels, the matrix shown in the right side of the figure results. The last level consists of only one element.

EXERCISES

1. Give a short answer to each of the following questions:

a. What is the main disadvantage of shared memory computers based on a bus architecture?

b. What is the main factor in yielding the speed-up in pipelined processors?

c. Related to the previous question: What is the main limitation of pipelined processors in regards to their potential for providing high speed-ups?

2. Show that the number of edges in a binary n-cube is $n2^{n-1}$.

3. Show that a binary 4-cube is identical with a *torus* which is a 4×4 mesh with wrap-around connections. Are there hypercubes of any other dimensions that are equivalent topologically to toruses?

4. A Gray code of length $k = 2^n$ is a sequence $a_0, \ldots, a_{k-1}$ of n-bit binary numbers such that (a) any two successive numbers in the sequence differ by one and only one bit; (b) all n-bit binary numbers are represented in the sequence; and (c) a_0 and a_{k-1} differ by one bit.

a. Find a Gray code sequence of length $k = 8$ and show the (closed) path defined by the sequence of nodes of a 3-cube, whose labels are the elements of the Gray code sequence. What type of paths does a Gray code define in a hypercube?

b. To build a "binary reflected" Gray code, start with the trivial Gray code sequence consisting of the two one-bit numbers 0 and 1. To build a two-bit Gray code, take the same sequence and insert a zero in front of each number, then take the sequence in *reverse order* and insert a one in front of each number. This gives $G_2 = \{00, 01, 11, 10\}$. The process is repeated until an n-bit sequence is generated. Show the binary reflected Gray code sequences of length 2, 4, 8, and 16. Prove (by induction) that this process does indeed produce a valid Gray code sequence.

c. Let an n-bit Gray code be given and consider the sub-sequence of all elements whose first bit is constant (e.g., zero). Is this an $n-1$ bit Gray code sequence? Generalize this to any of the n-bit positions. Generalize further to any set of $k < n$ bit positions.

d. Use the previous question to find a strategy to map a $2^{n_1} \times 2^{n_2}$ mesh into an (n_1+n_2)-cube.

5. Consider a ring of k processors which are characterized by the following communication performance characteristics. Each processor can communicate with its two neighbors *simultaneously*, i.e., it can send or receive a message while sending or receiving another message. The time for a message of length m to be transmitted between two nearest neighbors is of the form

$$\beta + m\tau.$$

a. A message of length m is "broadcast" to all processors by sending it from P_1 to P_2 and then from P_2 to P_3, etc., until it reaches all destinations, i.e., until it reaches P_k. How much time does it take for the message to complete this process?

b. Now split the message into packets of equal size and pipeline the data transfer. Typically, each processor will receive packet number i from the previous processor, while sending packet $i-1$ it has already received to the next processor. The packets will travel in chain from P_1 to $P_2, \ldots,$ to P_k. In other words, each processor executes a program that is described roughly as follows:

```
Do i=1, Num_packets
```

```
        Receive Packet number i from Previous Processor
        Send Packet number i to Next Processor
    EndDo
```

There are a few additional conditionals. Assume that the number of packets is equal to $k-1$. How much time does it take for all packets to reach all k processors? How does this compare with the simple method in (a)?

6. (a) Write a short FORTRAN routine (or C function) which sets up the level number of each unknown of an upper triangular matrix. The input matrix is in CSR format and the output should be an array of length n containing the level number of each node. (b) What data structure should be used to represent levels? Without writing the code, show how to determine this data structure from the output of your routine. (c) Assuming the data structure of the levels has been determined, write a short FORTRAN routine (or C function) to solve an upper triangular system using the data structure resulting in the previous question. Show clearly which loop should be executed in parallel.

7. In the jagged diagonal format described in Section 11.5.5, it is necessary to preprocess the matrix by sorting its rows by decreasing number of rows. What type of sorting should be used for this purpose?

8. In the jagged diagonal format described in Section 11.5.5, the matrix had to be preprocessed by sorting it by rows of decreasing number of elements.

a. What is the main reason it is necessary to reorder the rows?

b. Assume that the same process of extracting one element per row is used. At some point the extraction process will come to a stop and the remainder of the matrix can be put into a CSR data structure. Write down a good data structure to store the two pieces of data and a corresponding algorithm for matrix-by-vector products.

c. This scheme is efficient in many situations but can lead to problems if the first row is very short. Suggest how to remedy the situation by padding with zero elements, as is done for the Ellpack format.

9. Many matrices that arise in PDE applications have a structure that consists of a few diagonals and a small number of nonzero elements scattered irregularly in the matrix. In such cases, it is advantageous to extract the diagonal part and put the rest in a general sparse (e.g., CSR) format. Write a pseudo-code to extract the main diagonals and the sparse part. As input parameter, the number of diagonals desired must be specified.

NOTES AND REFERENCES. Kai Hwang's book [124] is recommended for an overview of parallel architectures. More general recommended reading on parallel computing are the book by Bertsekas and Tsitsiklis [25] and a more recent volume by Kumar et al. [139]. One characteristic of high-performance architectures is that trends come and go rapidly. A few years ago, it seemed that massive parallelism was synonymous with distributed memory computing, specifically of the hypercube type. Currently, many computer vendors are mixing message-passing paradigms with "global address space," i.e., shared memory viewpoint. This is illustrated in the recent T3D machine built by CRAY Research. This machine is configured as a three-dimensional torus and allows all three programming paradigms discussed in this chapter, namely, data-parallel, shared memory, and message-passing. It is likely that the T3D will set a certain trend. However, another recent development is the advent of network supercomputing which is motivated by astounding gains both in workstation performance and in high-speed networks. It is possible to solve large problems on clusters of workstations and to obtain excellent performance at a fraction of the cost of a massively parallel computer.

Regarding parallel algorithms, the survey paper of Ortega and Voigt [156] gives an exhaustive bibliography for research done before 1985 in the general area of solution of Partial Differential Equations on supercomputers. An updated bibliography by Ortega, Voigt, and Romine is available in [99]. See also the survey [178] and the monograph [71]. Until the advent of supercomputing in the mid 1970s, storage schemes for sparse matrices were chosen mostly for convenience as performance was not an issue, in general. The first paper showing the advantage of diagonal storage schemes in sparse matrix computations is probably [133]. The first discovery by supercomputer manufacturers of the specificity of sparse matrix computations was the painful realization that without hardware support, vector computers could be inefficient. Indeed, the early CRAY machines did not have hardware instructions for gather and scatter operations but this was soon remedied in the second-generation machines. For a detailed account of the beneficial impact of hardware for “scatter” and “gather” on vector machines, see [146].

Level scheduling is a textbook example of topological sorting in graph theory and was discussed from this viewpoint in, e.g., [8, 190, 228]. For the special case of finite difference matrices on rectangular domains, the idea was suggested by several authors independently, [208, 209, 111, 186, 10]. In fact, the level scheduling approach described in this chapter is a “greedy” approach and is unlikely to be optimal. There is no reason why an equation should be solved as soon as it is possible. For example, it may be preferable to use a *backward scheduling* [7] which consists of defining the levels from bottom up in the graph. Thus, the last level consists of the leaves of the graph, the previous level consists of their predecessors, etc. Dynamic scheduling can also be used as opposed to static scheduling. The main difference is that the level structure is not preset; rather, the order of the computation is determined at run-time. The advantage over pre-scheduled triangular solutions is that it allows processors to always execute a task as soon as its predecessors have been completed, which reduces idle time. On loosely coupled distributed memory machines, this approach may be the most viable since it will adjust dynamically to irregularities in the execution and communication times that can cause a lock-step technique to become inefficient. However, for those shared memory machines in which hardware synchronization is available and inexpensive, dynamic scheduling would have some disadvantages since it requires managing queues and generates explicitly busy waits. Both approaches have been tested and compared in [22, 189] where it was concluded that on the Encore Multimax dynamic scheduling is usually preferable except for problems with few synchronization points and a large degree of parallelism. In [118], a combination of prescheduling and dynamic scheduling was found to be the best approach on a Sequent balance 21000. There seems to have been no comparison of these two approaches on distributed memory machines or on shared memory machines with microtasking or hardware synchronization features. In [22, 24] and [7, 8], a number of experiments are presented to study the performance of level scheduling within the context of preconditioned Conjugate Gradient methods. Experiments on an Alliant FX-8 indicated that a speed-up of around 4 to 5 can be achieved easily. These techniques have also been tested for problems in Computational Fluid Dynamics [214, 216]. ■